AUTOMATIC TRANSLATION OF MOORE FSM INTO TDES SUPERVISORS

AUTOMATIC TRANSLATION OF MOORE FINITE STATE MACHINES INTO TIMED DISCRETE EVENT SYSTEM SUPERVISORS

BY
HINA MAHMOOD, M.Sc.

A Thesis
submitted to the Department of Computing and Software
and the School of Graduate Studies
of McMaster University
in partial fulfilment of the requirements
for the degree of
Doctor of Philosophy

Doctor of Philosophy (2023) McMaster University
(Computing and Software) Hamilton, Ontario, Canada

TITLE:	Automatic Translation of Moore Finite State Machines into Timed Discrete Event System Supervisors
AUTHOR:	Hina Mahmood B.Sc., M.Sc. (Software Engineering), International Islamic University, Islamabad, Pakistan.
SUPERVISOR:	Dr. Ryan J. Leduc
NUMBER OF PAGES:	xx, 336

Abstract

In the area of Discrete Event Systems (DES), formal verification techniques are important in examining a variety of system properties including controllability and nonblocking. Nonetheless, in reality, most software and hardware practitioners are not proficient in formal methods which holds them back from the formal representation and verification of their systems. Alternatively, it is a common observation that control engineers are typically familiar with Moore synchronous Finite State Machines (FSM) and use them to express their controllers' behaviour.

Taking this into consideration, we devise a generic and structured approach to automatically translate Moore synchronous FSM into timed DES (TDES) supervisors. In this thesis, we describe our FSM-TDES translation method, present a set of algorithms to realize the translation steps and rules, and demonstrate the application and correctness of our translation approach with the help of an example.

In order to develop our automatic FSM-TDES translation approach, we exploit the structural similarity created by the sampled-data (SD) supervisory control theory between the two models. To build upon the SD framework, first we address a related issue of disabling the *tick* event in order to force an eligible prohibitable event in the SD framework. To do this, we introduce a new synchronization operator called the *SD synchronous product* ($||_{SD}$), adapt the existing TDES and SD properties, and devise our $||_{SD}$ setting. We formally verify the controllability and nonblocking properties of our $||_{SD}$ setting by establishing logical equivalence between the existing SD setting and our $||_{SD}$ setting. We present algorithms to implement our $||_{SD}$ setting in the DES research tool, DESpot (2023).

The formulation of the $||_{SD}$ operator provides twofold benefits. First, it simplifies the design logic of the TDES supervisors that are modelled in the SD framework. This results in improving the ease of manually designing SD controllable TDES supervisors, and reduced verification time of the closed-loop system. We demonstrate these benefits by applying our $||_{SD}$ setting to an example system. Second, it bridges the gap between theoretical supervisors and physical controllers with respect to event forcing. This makes our FSM-TDES translation approach relatively uncomplicated. Our automatic FSM-TDES translation approach enables the designers to obtain a formal representation of their controllers without designing TDES supervisors by hand and without requiring formal methods expertise.

Overall, this work should increase the adoption of the SD supervisory control theory in particular, and formal methods in general, in the industry by facilitating software and hardware practitioners in the formal representation and verification of their control systems.

To my beloved parents
for their unconditional love, care, affection, support and prayers

Acknowledgements

In the name of **ALLAH**, the most Gracious, the most Merciful, whose blessings made it possible for me to complete and defend this thesis with flying colours. Each moment during the course of my Ph.D., I experienced the Grace of ALLAH, who enlightened my thoughts with His wisdom, opened before me unexpected avenues and inspired me to move forward even at the moments of despair.

I am thankful to my supervisor, Dr. Ryan Leduc, for providing me with this thesis topic and his guidance. I am grateful to the chair of my Ph.D. supervisory committee, Dr. Frantisek Franek, for stepping in as the reviewer of my thesis, and for his immense support and encouragement all through my degree. I am also thankful to my committee member, Dr. Emil Sekerinski, for his support and advice.

I would like to gratefully thank all those members of the CAS department who made this Ph.D. journey a pleasant and memorable experience for me. First, I would like to express my sincere gratitude to the Department Chair, Dr. Mark Lawford, whose constant support and cooperation over the past several months made it possible for me to successfully complete my Ph.D. I am grateful to him for patiently listening to all my issues and resolving them in the best possible way. I am thankful to him for providing me with department funding towards the end of my degree. I am also thankful to him for all the time he gave to our meetings and promptly replying to all my messages despite of his busy schedule, for calmly answering all my technical and non-technical questions, and for providing valuable guidance related to the thesis and defense; learned a lot from him in a short period of time.

I am thankful to the Graduate Advisor for Software Engineering, Dr. Spencer Smith, for providing his useful advice and support in the hour of need, for staying in touch with me and telling me about the Ombuds office. I am also thankful to the Acting Associate Chair for Graduate Studies, Dr. Fei Chiang, for reaching out to me herself during the challenging times and offering me her help and assistance.

Thanks to the Department Manager, Laurie LeBlanc, for giving me her time to assess my situation and guiding me in the right direction to seek to resolve all my issues quickly and effectively. Thanks to the Graduate Administrator, Stefanie Bittcher, for always being so cordial and helpful, and especially for her tremendous cooperation towards the end of my degree.

Last but not least, I would like to extend my deepest gratitude to my cohesive

family and wonderful friends. I am indebted to my parents for their encouragement and motivation throughout the years that made me so able that I am at this position today. Thanks are due to my dependable and caring brothers, Dr. Salman, Imran and Irfan, who always stand by me through thick and thin. Thanks to my sisterly sister-in-laws, Aysha and Leena, for their friendly and enjoyable conversations. A very special thanks to my adorable nephews, Ibrahim and Abdullah, and my sweet niece, Alishba, for always making me forget my worries, and inspiring me to be fearless and enjoy life from the unique perspective of being a kid again. I also thank Dr. Saba Amin for her cheerful company and late night chats, after which I always feel fresh and relaxed; blessed to have such amazing people in my life.

Contents

List of Figures

List of Tables

List of XML Input Files

Abbreviations and Notation

ALF	Activity Loop Free
BDD	Binary Decision Diagram
CB	Concurrent Behaviour
CSCE	Concurrent Supervisory Control Equivalent
CS Deterministic	Concurrent String Deterministic
DC	Don't Care
DEF	Default
DES	Discrete Event System
FMS	Flexible Manufacturing System
FSM	Finite State Machine
GDC	Global Don't Care
IEC	International Electrotechnical Commission
IL	Instruction List
IO	Input-Output
LD	Ladder Diagram
L.H.S	Left Hand Side
MBSE	Model-Based System Engineering
NSL	Next State Logic
PLC	Programmable Logic Controller
POS	Product-of-Sums

R.H.S	Right Hand Side
SCT	Supervisory Control Theory
SD	Sampled-Data
SOP	Sum-of-Products
$\|\|_{SD}$	Sampled-Data Synchronous Product
TDES	Timed Discrete Event System
TA	Timed Automata

Chapter 1

Introduction

The goal of the work presented in this thesis is to facilitate software and hardware designers and practitioners in the formal representation and verification of their control systems, thereby increasing the adoption of formal methods in industry. We achieve these goals by reducing the complexity and improving the ease of formal design and verification process for designers, especially those who are not proficient in formal methods. To do this, we propose a generic and structured approach for the automatic translation of Moore synchronous Finite State Machines (FSM) into Timed Discrete Event System (TDES) supervisors. We base our work on the sampled-data (SD) supervisory control theory (Wang, 2009; Wang and Leduc, 2012; Leduc *et al.*, 2014).

In this thesis, first, as a stepping stone, we present an approach to automate the mechanism of forcing eligible prohibitable events in the SD supervisory control framework. We formally verify our proposed setting, and provide algorithms to implement our approach in the DES research tool, DESpot (2023). We demonstrate the benefits of our approach through its application to an illustrative example. Then, we describe our approach to automatically convert a Moore synchronous FSM into a TDES supervisor. We design a set of algorithms to realize our FSM-TDES translation approach, and show the correctness of our translation method with the help of an example.

In this chapter, first we introduce the area of Discrete Event Systems (DES) and related terminology. Next, we explain our motivation for choosing this research topic, and discuss some prior work related to ours. This is followed by the identification of the research gap, and a description of our proposal and methodology to address this gap. After that, we state the research questions that this thesis aims to answer, and summarize the contributions of this thesis. Finally, we close this chapter by providing an outline of how the rest of this thesis is organized.

1.1 Introduction to Discrete Event Systems

Discrete Event Systems (DES) are dynamic systems that encompass processes that are discrete in time and state space, often asynchronous, and typically non-deterministic. These systems evolve by changing state in accordance with the instantaneous occurrence of physical *events*. DES are quite common in industry and include a variety of man-made systems namely manufacturing systems, computer and communication networks, transport and logistic systems, and traffic control systems. These applications generally require some degree of control and coordination to ensure the orderly flow of events according to the given specifications and/or to prevent the occurrence of undesired chains of events.

In order to solve the control problem of DES and extend the control theory concepts of continuous systems to DES, Wonham and Ramadge (1987); Ramadge and Wonham (1989) introduced *Supervisory Control Theory (SCT)*. This theoretical approach is based on automata theory and formal language models (Hopcroft and Ullman, 1979). SCT provides algorithms and methods for the analysis and control of DES.

In SCT, uncontrolled behaviour of the DES is modelled by an automaton and is referred to as the *plant*. In order to achieve desired behaviour as per the given specifications, a *supervisor* is introduced that alters the unrestricted behaviour of the plant by operating synchronously with it and using a feedback control mechanism. SCT partitions the set of events into *controllable* and *uncontrollable* events, the former being amenable to disablement by a supervisor. In SCT, a system is desired to have two properties, *controllability* (undesired actions do not occur) and *nonblocking* (no deadlock or livelock).

A *timed DES (TDES)* model adds timing information to an untimed DES in order to deal with temporal specifications. The TDES modelling framework, proposed by Brandin (1993); Brandin and Wonham (1994), extends the untimed DES by introducing a new *tick* event. The *tick* event represents the passage of one time unit and corresponds to the tick of a global clock to which the system is assumed to be synchronized. It also introduces a new class of non-*tick* events called *forcible events* that can preempt the occurrence of a *tick* event, when needed. Non-*tick* controllable events are referred to as *prohibitable events*.

1.2 Motivation

With the increasing complexity of control systems, formal verification techniques have become an important tool to check a variety of system properties including controllability and nonblocking. In order to design a new system that needs to be formally verified, it must be expressed as a formal model. Likewise, one of the ways to formally verify an existing system is to translate the software into a formal representation. To

be able to formalize new and existing systems, software designers and practitioners are required to have expertise in formal methods. They need to manually design and express their controllers as theoretical supervisors before providing them as an input to a formal verification tool to check the desired system properties.

However, in reality, most of the software and hardware designers in industry are not trained in formal methods, hence unfamiliar with formal modelling and verification strategies (Vogel-Heuser *et al.*, 2014). This does not only hinder them from expressing their controllers using a formal representation, but also hold them back from applying formal verification techniques to examine various properties of their theoretical models. This is believed to be one of the primary reasons for the limited adoption of formal methods in industry. At the same time, it is a common observation that many software and hardware designers are typically familiar with Moore synchronous Finite State Machines (FSM) and use them to express their system controllers.

The sampled-data (SD) supervisory control theory (Wang, 2009; Wang and Leduc, 2012; Leduc *et al.*, 2014) is a generic, formal implementation approach that proposes to implement TDES supervisors as SD controllers, a good example of which is a Moore synchronous FSM (or Moore FSM, for short) (Brown and Vranesic, 2013). One of the properties of the SD methodology is that it makes the design of TDES supervisors consistent with the SD controllers by preventing software designers from expressing logic in theoretical models that cannot be physically implemented. This structural similarity created by the SD methodology between the TDES supervisors and Moore FSM motivated us to go the opposite way, i.e. devise an approach to automatically convert Moore FSM into TDES supervisors. In other words, we thought of exploiting this structural similarity between the two models to do the automatic reverse translation in order to facilitate the designers and practitioners in the formal representation and verification of their control systems without them being proficient in formal methods.

1.3 Related Work

This thesis aims at developing a generic, well-defined and automated approach to translate Moore FSM into TDES supervisors in order to assist software designers, especially non-specialists with little or no knowledge of formal methods, in the formal representation and verification of their TDES systems. In this section, we review some notable related prior research.

Specifically, we examine the existing formal approaches for the implementation of SCT supervisors, and discuss real-world applications of SCT that have been reported in the literature. Based on this review, we present our argument of why we have selected the SD supervisory control theory (Leduc *et al.*, 2014) as the basis of our work. Then, we analyze some significant previous DES research on the formal verification

of existing systems.

1.3.1 Formal Implementation Approaches

Although theoretical aspects of SCT have received substantial attention in academia, the implementation of DES supervisors is still an open issue (Vieira *et al.*, 2017; Zaytoon and Riera, 2017). This is due to the clear interpretation gap between the roles a supervisor is assumed to play within the SCT modelling framework and the roles a controller has to play in the real-world control systems (Zaytoon and Carre-Meneatrier, 2001). Fabian and Hellgren (1998) discuss the DES supervisor implementation issues by highlighting discrepancy between the abstract SCT supervisors and resulting control realization.

Below, we classify the existing DES implementation approaches based on untimed and timed DES.

Untimed DES Approaches

SCT is primarily based on automata theory and regular languages (Hopcroft and Ullman, 1979). Thus, a significant body of work uses automata as the primary means of modelling DES.

Some of the early contributions to the implementation of automata-based SCT supervisors on a programmable logic controller (PLC) (Bolton, 2015) using Ladder Diagram (LD) (Antonsen, 2021) are presented in Arinez *et al.* (1993); Leduc (1996); Lauzon *et al.* (1997). These studies allowed only one event to occur per cycle. Later on, some significant approaches and algorithms for the PLC implementation of monolithic and modular supervisors are discussed in de Queiroz and Cury (2002); Vieira *et al.* (2006); Hasdemir *et al.* (2008); Silva *et al.* (2011); Vieira *et al.* (2017); Prenzel and Provost (2018); James *et al.* (2019).

Based on the local modular supervisory control approach of de Queiroz and Cury (2000), Leal *et al.* (2012) presented a methodology to turn a PLC into a state transition system, whereas Alves *et al.* (2022) discussed an implementation scheme for a networked control system. Cantarelli (2006) represented supervisors as a Mealy machine and addressed concurrency among the events. Dietrich *et al.* (2002); Gouyon *et al.* (2004) proposed different rules to choose from multiple events that are possible in a single scan cycle. The signal-interpreted approach of Fouquet and Provost (2017) handled I/O signals instead of events, thereby avoiding the need to adapt an event-based approach to a signal-based real environment.

An alternative approach to the automaton framework of SCT is a petri net based approach to supervisory control design (Seatzu *et al.*, 2013). Using petri net formalism, the implementation of DES supervisors is discussed in Crockett *et al.* (1987); Zhou *et al.* (1992); Hellgren *et al.* (2002); Thapa *et al.* (2005); Basile *et al.* (2013); Feio *et al.* (2017).

Timed DES Approaches

Unlike untimed DES, only a few studies have focused on the implementation of SCT supervisors for time-sensitive systems. This is due to the added complexity of incorporating time in the DES modelling and control, where enabled events may only occur within certain time bounds.

Brandin (1996) described an automata-based real-time supervisory control approach for automated manufacturing systems implemented in LD. Uzam (2012) proposed an adhoc technique for PLC-based implementation of supervisors with time delay functions. This technique is based on the LD implementation approach of Uzam *et al.* (2009b) and the concept of postponed events from Gelen *et al.* (2010). Recently, Szpak *et al.* (2020) presented an architecture for the implementation of timed modular supervisors on a PLC and applied it to a real-world test-bench.

For the implementation of petri net controllers into LD, Uzam and Jones (1998) developed a heuristic-based method, whereas Jimenez *et al.* (2001); Moreira and Basilio (2014) provided a set of translation rules. Uzam and Wonham (2006) presented a hybrid approach that coupled SCT supervisors to petri net plant models. This approach was extended to monolithic supervisors in Uzam and Gelen (2009) and modular supervisors in Gelen and Uzam (2014). Azkarate *et al.* (2021) described a systematic way to implement a petri net controller using PLC. Besides PLC-based software implementations, hard-wired implementations of petri net controllers are reported in Chang *et al.* (1998) and Uzam *et al.* (2009a).

1.3.2 Real-World Applications of SCT

Despite the limited adoption of SCT in industry (Wonham *et al.*, 2018), a few applications of SCT synthesis and implementation have been reported in the literature. These include automata-based SCT application to the control of a rapid thermal multiprocessor (Balemi *et al.*, 1993), a manufacturing workcell (Lauzon *et al.*, 1996), an educational testbed simulating an automated car assembly line (Chandra *et al.*, 2003), waterway locks (Reijnen *et al.*, 2017), a pseudo microgrid setting of a custom power park (Kharrazi *et al.*, 2019) and an AC microgrid (Ghasaei *et al.*, 2021).

Some previous studies have investigated the application of SCT combined with Model-Based System Engineering (MBSE) (Schiffelers *et al.*, 2009; Van der Sanden *et al.*, 2015). Forschelen *et al.* (2012) examined the supervisor implementation issues in MBSE using a theme park vehicle. Systematic approaches for the model-based design and implementation of SCT supervisors are reported in literature for a patient support table for magnetic resonance imaging scanner (Theunissen *et al.*, 2014), an industrial-size baggage handling system of an international airport (Swartjes *et al.*, 2017), an advanced driver assistance system tested in a real vehicle system (Korssen *et al.*, 2018) and robotic operating system-based applications (Torta *et al.*, 2023).

1.3.3 Why Sampled-Data Supervisory Control?

A detailed analysis of previous research on the implementation of untimed and timed DES reveals that these studies propose adhoc implementation approaches that are application-specific. A vast majority of these approaches allow only one event to execute per cycle, thus ignoring concurrency issues and limiting program efficiency. Most of these studies to date are limited to PLC implementation, which further restricts the applicability of the presented methodologies. Moreover, these approaches do not guarantee a controllable and nonblocking implementation, even if designers develop theoretical models that satisfy these properties.

To the best of our knowledge, the sampled-data (SD) supervisory control methodology (Wang, 2009; Wang and Leduc, 2012; Leduc *et al.*, 2014) is the first generic, automata-based formal implementation approach that addresses the implementation, concurrency and timing issues in a well-defined way. The SD approach proposes to implement TDES supervisors as SD controllers, a good example of which is a Moore FSM (Brown and Vranesic, 2013). It allows multiple events to occur in a single clock cycle.

The SD methodology provides sufficient conditions to guarantee that if a theoretical TDES is controllable, nonblocking, and satisfies the desired properties, then the physical system will also exhibit correct behaviour, i.e. the implementation will be controllable, nonblocking and abide by the specified control laws. It ensures this by: 1) identifying a set of existing TDES properties, and especially introducing the new property of *SD controllability* (Definition 3.5.1) that deals with concurrency and timing issues, 2) establishing a formal representation of SD controllers as Moore FSM, and 3) providing a formal translation method to convert TDES supervisors into Moore FSM, which can either be implemented on a PLC, in hardware using a hardware description language such as Verilog (Brown and Vranesic, 2013), or as a computer software program.

Due to these distinctive characteristics of the SD supervisory control theory and its applicability to a variety of TDES applications, we are using this approach as the foundation of our work.

1.3.4 Formal Verification of Existing Systems

A number of existing studies have focused on the extraction of formal models from existing systems. Both automata and petri nets have been used for the formalization of existing PLC programs in order to do formal verification. Below, we outline some important prior research in this area.

Automata-Based Verification

One of the initial studies focusing on the extraction of automata models from LD programs is reported by Moon (1994). The author presented a verification method

that represents LD as a transition system and used a model checker to determine whether the system operates as specified. Rossi and Schnoebelen (2000) described a similar approach that uses a fragment of an LD program for symbolic model checking.

Over time, several approaches and algorithms have been developed to generate Timed Automata (TA) models (Alur and Dill, 1994) from a PLC implementation. Zoubek *et al.* (2003) provided an algorithm to translate seven LD instructions into TA that can be given as an input to the model checking tool UPPAAL (2003) for automatic verification. Bauer *et al.* (2004) presented transformation schemes to convert untimed and timed PLC programs for verification with Cadence-SMV and UPPAAL respectively. Canet *et al.* (2000) and Zhou *et al.* (2009) discussed the conversion of Instruction List (IL) (Adam and Adam, 2022) PLC programs into TA.

Caldwell *et al.* (2016) outlined the use of active learning to extract automata specifications, expressed as a Time Delay Mealy Machine, from existing PLC software. Peixoto *et al.* (2019) presented a model-based testing technique for the automatic verification of LD programs using the Gungnir tool. Zhang *et al.* (2020) proposed a learning algorithm to infer a Moore automaton from system requirements, which can then be used to synthesize an SCT supervisor for the control of embedded systems.

Petri Net-Based Verification

An earlier study to extract petri nets from a subset of IL programs with restrictions was performed by Heiner and Menzel (1998). Fujino *et al.* (2000) defined a set of rules to transform PLC code into petri nets. Bender *et al.* (2008) discussed a model-driven approach to formally verify LD programs using a model checker, Tina (Berthomieu *et al.*, 2004). Wightkin *et al.* (2011); Luo *et al.* (2018) developed algorithms to translate PLC programs into petri net controllers. Quezada *et al.* (2014) presented an element-to-element transformation methodology to obtain a petri net model from an LD control program. Quezada *et al.* (2023) described the conversion of 8 logical blocks of PLC LD into petri net formal models.

1.4 Research Gap

A review of previous studies on the formalization of existing systems reveals that most of the existing work focuses on presenting approaches and algorithms that translate PLC elements into automata or petri net equivalent representation on a case-by-case basis. In most cases, these PLC elements of existing software are expressed in one of the PLC programming languages, as defined by the International Electrotechnical Commission-61131-3 standard (IEC, 2013).

Since PLC, in general, do not have a formal structure of their own, the approaches described in literature to extract formal models from PLC implementation are typically unstructured, implementation-dependent and application-specific. Moreover, they focus on the basic portions of a system, i.e. a subset of PLC elements, rather

than considering all elements of the system together. In addition, to our knowledge, they do not consider and address concurrency and timing issues in a well-defined way.

1.5 Our Proposal: FSM to TDES Translation

In order to address the gaps identified in the previous section, we propose a generic, structured and well-defined approach to automatically translate Moore FSM (Brown and Vranesic, 2013) into TDES supervisors. Specifically, we advocate the notion of expressing new and existing system controllers as Moore FSM. We focus on Moore FSM as they are a standard design structure for digital logic that many control engineers are typically familiar with. Also, Moore FSM provide a complete specification and concrete definition of the controller, yet they are generic enough to be independent of any implementation language and can be expressed as a C program, as hardware using Verilog (Brown and Vranesic, 2013), or as an LD program. In this way, Moore FSM provide a widely accessible and portable means to express a controller, which is more flexible than focusing on part of a platform dependent implementation language, such as PLC LD.

To realize our proposal of FSM-TDES translation, we need to have a formal, structured representation that could make it possible to do the conversion automatically. For this purpose, we make use of the SD supervisory control theory (Wang, 2009; Wang and Leduc, 2012; Leduc *et al.*, 2014) that provides a generic, well-defined, formal framework that we use as the basis of our work. Besides many reasons discussed in Section 1.3.3 for choosing the SD theory, we primarily exploit the structural similarity that the SD methodology creates between TDES supervisors and Moore FSM, while devising our automatic FSM-TDES translation approach.

By proposing the idea of automatic FSM-TDES translation, our aim is to enable the software and hardware designers to design and express their system controllers in a standard way that they are familiar with, i.e. as Moore FSM. Our FSM-TDES translation approach will interpret their Moore FSM as SD controllers, and automatically translate them into a formal TDES supervisor representation. The designers can then use these translated TDES supervisors along with the application's TDES plant models to verify the correctness of their closed-loop systems by checking the desired properties, without requiring them to design TDES supervisors by hand and acquire formal methods expertise.

Preferably, we wish our FSM-TDES translation approach to generate TDES supervisors in such a way that increases the likelihood of the closed-loop behaviour to satisfy the desired properties of the TDES and SD framework, including the key property of SD controllability (Definition 3.5.1). One of the checks enforced by the SD controllability property is that if a prohibitable event is enabled, the *tick* event must be disabled. Currently, in the SD framework, designers are responsible for manually satisfying this condition at every state of the closed-loop system by explicitly

disabling *tick* while designing their TDES supervisors by hand. Since, we propose to automatically generate TDES supervisors from Moore FSM, it is vital to devise a way to automatically satisfy this condition without intervention of the designers. This issue is closely related to the development of our FSM-TDES translation approach and must be addressed first.

Below, we first discuss the existing way of manually disabling the *tick* event in the SD framework, and analyze its intricacies to show how difficult it is currently for the designers to utilize this method, and how complicated it would be for us to express it algorithmically while developing our FSM-TDES translation approach. Then, we present our idea for automatically disabling *tick* in the SD framework. Our proposed solution provides twofold benefits: 1) it keeps our FSM-TDES translation approach simple, and 2) it reduces the complexity of manually designing SD controllable TDES supervisors in the existing SD supervisory control setting ("SD setting," for short). In order to demonstrate the existing method, we use a small independent chunk of our illustrative example described later in this thesis (Chapter 10).

1.5.1 Related Issue

In the TDES framework, the standard method used by software designers to force an event at a given state of the system is to "explicitly disable" the *tick* event at the corresponding state of the TDES supervisor. This has the effect of removing the now impossible behaviour that *tick* could occur before the forcible event, as the forced event is guaranteed to occur before the *tick*.

In the SD supervisory control theory, all prohibitable events are treated as forcible events (Section 3.3). The SD controllability property (Definition 3.5.1) requires that a prohibitable event be forced in the same clock period (before *tick*) in which it is enabled, and must remain disabled otherwise. Specifically, the forward implication ($\Rightarrow$) of Point ii enforces this check to make sure that at a given state, if *tick* and a prohibitable event is possible in the plant model, and the prohibitable event is enabled by the TDES supervisor, then the supervisor must explicitly disable *tick* in order to force the prohibitable event.

Failure to satisfy this property correctly can have serious consequences. For instance, consider the following two scenarios:

1. At a given state, both *tick* and a prohibitable event is possible in the plant and enabled by all modular TDES supervisors. This is highly undesirable as the SD controller would not know whether to let *tick* occur or force a prohibitable event at this state. This would make the translation of TDES supervisors into SD controllers ambiguous.
2. A prohibitable event is enabled by a modular supervisor and the supervisor has also disabled the *tick* event which was possible in the plant model. However, if one of the other modular supervisors has disabled this prohibitable event or the

event is not currently possible in the plant, our system will become uncontrollable (Definition 2.3.2).

In order to avoid these unwanted outcomes, it is important that all plant components and modular supervisors coordinate and agree on the enablement/disablement of *tick* and prohibitable events. For this purpose, currently the designers have to manually keep track of two things while designing their TDES models by hand: 1) when *tick* event is possible in the plant, and 2) when a given prohibitable event is possible in the plant and is not disabled by any of the modular supervisors. In this case, the designers must explicitly disable *tick* event in the supervisor model to force the prohibitable event.

Clearly, keeping track of this information manually, and at the same time guarantee that Point ii ($\Rightarrow$) of SD controllability property is always satisfied at every state of the closed-loop system is not a trivial and trouble-free task. This is further complicated by the fact that multiple modular supervisors are usually in control of the same prohibitable event. As a result, designers typically do not find it convenient to manually satisfy Point ii ($\Rightarrow$) of SD controllability in the existing SD setting, especially while developing modular solutions.

Moreover, the logic of disabling *tick* in the presence of an enabled prohibitable event also needs to be explicitly specified in the design of various TDES supervisors so that modular supervisors have sufficient knowledge of the plant's behaviour as well as each other's behaviour, in order to work together appropriately. In order to make the TDES models aware of each other's behaviour with respect to the *tick* event and common prohibitable events, designers currently rely on two nonautomatic methods: 1) duplicate the logic of one TDES model in the other model(s), or 2) add *expansion events*, i.e. non-physical/virtual events that are added solely to aid in communication between modular supervisors. Five such events, prefixed by "no", are discussed in Section 10.1.

Besides making the TDES modelling process demanding and laborious for designers, these methods also increase the design complexity and size of TDES supervisors, hence the overall SD system, in the existing SD setting. We briefly present an example for the first method of duplicating logic and its associated complexities below. Please see Section 10.3 for a comprehensive discussion on both methods.

Illustrative Example

In Chapter 10, we will discuss the TDES example of a Flexible Manufacturing System (FMS) from Wang (2009); Wang and Leduc (2012) to apply our proposed automatic *tick* disablement/event forcing approach and discuss our results. Here, we briefly present one part of this example to concretely illustrate the aforementioned issues.

The FMS, shown in Figure 1.1, consists of two conveyor belts (Con2 and Con3)[1], four machines (Robot, Lathe, PM and AM), and five buffers (B2, B4, B6, B7 and B8)[1].

[1]This example is taken from a larger example, which is why the part labels are not contiguous.

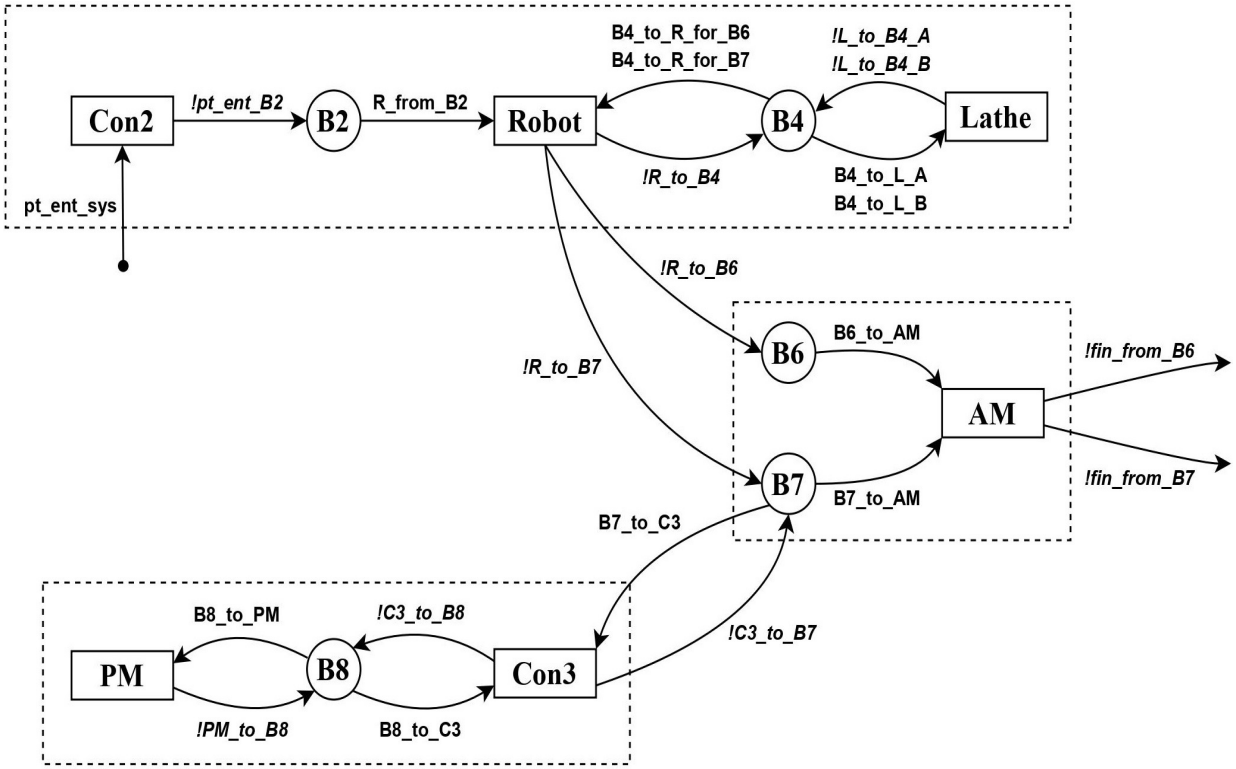


Figure 1.1: An Overview of Flexible Manufacturing System

Each buffer has the capacity to hold a single part and it is desired that buffers never overflow (try to put a part in the buffer when it already has one) or underflow (try to remove a part from the buffer when it is empty). Please note that the behaviour of buffers is treated as specifications, and will be implemented as TDES supervisors. In Figure 1.1, event names preceded by '!' represent uncontrollable events, and those without '!' are prohibitable events.

In the FMS, once a new part enters the system via Con2 (*pt_ent_sys*), it goes to buffer B2 (*pt_ent_B2*). The Robot is responsible for taking parts from buffer B2 (*R_from_B2*) and pass them on to buffer B4 (*R_to_B4*) for further processing by the Lathe. After processing, the part returns to buffer B4, from which Robot moves it either to buffer B6 (*R_to_B6*) or B7 (*R_to_B7*). Please see Section 10.1 for a more in-depth explanation of this system.

In order to manage and force the prohibitable event *R_from_B2* as per the given specifications, TDES plant model **Robot** (Figure 1.2), and modular TDES supervisors, ***B2*** (Figure 1.3) and ***TakeB2*** (Figure 1.4), are designed in the existing SD setting (Wang and Leduc, 2012). Please note that in the complete FMS example (Chapter 10), *R_from_B2* is under the control of four modular supervisors. To keep our discussion simple, here we are considering only two of them that are tightly coupled to one another. To understand the graphical representation and notation of a TDES automaton, please see Section 2.3.

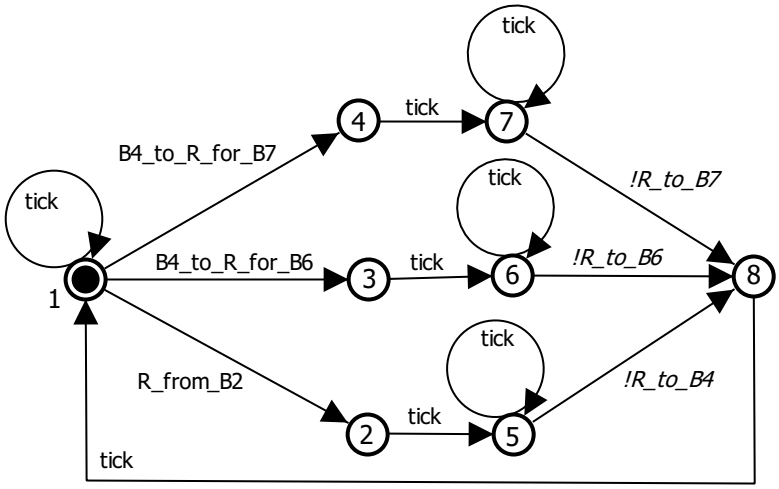


Figure 1.2: TDES Plant **Robot**

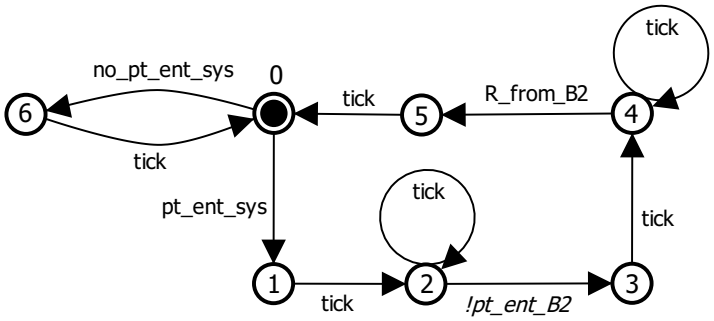


Figure 1.3: TDES Supervisor ***B2***

Supervisor ***TakeB2*** is primarily responsible for forcing event *R_from_B2*. In order to keep the system timed controllable (Definition 2.3.2), it is important to make sure that ***TakeB2*** does not disable *tick* and try to force *R_from_B2* when it is not possible in **Robot** or disabled by supervisor ***B2***. Therefore, ***TakeB2*** must take into account the behaviour of these models before it attempts to force *R_from_B2*.

By looking at supervisor ***B2***, we note that it enables *R_from_B2* after the part has reached buffer B2 (*pt_ent_B2*). This means that ***TakeB2*** needs to keep track of the part's progress. Once the part has reached buffer B2, only then ***TakeB2*** should disable *tick* and force *R_from_B2*, as this is the time when the prohibitable event will be enabled by supervisor ***B2*** and possible in **Robot**.

Now that we have manually figured out the "right time" for forcing *R_from_B2*, this logic needs to be incorporated in the design of supervisor ***TakeB2***. This is done by adding the event *pt_ent_B2* to its event set, and duplicating the related behaviour of ***B2*** in ***TakeB2***, i.e. by replicating the event sequence "$pt\_ent\_B2 - tick$" from supervisor ***B2*** to ***TakeB2***. Only after the occurrence of this sequence of events, ***TakeB2*** forces *R_from_B2* by disabling *tick* at state 2, thus making sure that system does not become uncontrollable while it is trying to force *R_from_B2*.

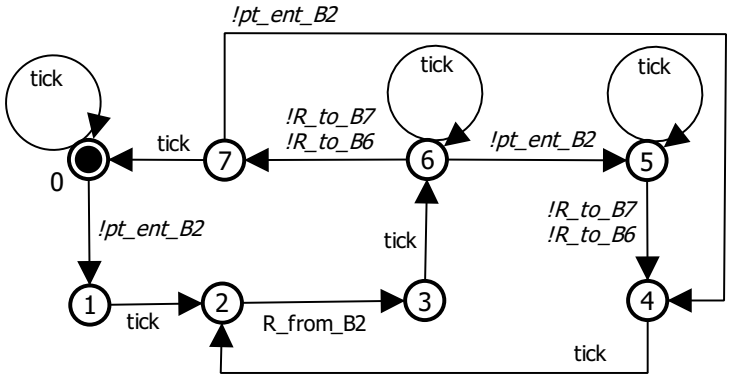


Figure 1.4: TDES Supervisor ***TakeB2***

It is notable that uncontrollable event *pt_ent_B2* gets added to ***TakeB2*** as part of explicitly specifying this forcing logic, and now ***TakeB2*** is also in charge of allowing/disallowing this event to occur in the system. In this case, designers need to effectively handle two more things. First, they must make sure that ***TakeB2*** should always allow *pt_ent_B2* to happen, when needed. In other words, ***TakeB2*** must never block *pt_ent_B2* when it is possible in the plant model, otherwise the system will become uncontrollable. Second, designers must design ***TakeB2*** in such a way that *pt_ent_B2* interleaves properly with the other events of ***TakeB2***, (i.e. *R_to_B6* and *R_to_B7*), in order to prevent the violation of other desired SD properties and overall system specifications.

It is obvious that duplicating the behaviour of supervisor ***B2*** and specifying all this additional logic in relation to *pt_ent_B2* not only make the design of ***TakeB2*** more complicated, but also add more states to ***TakeB2***, thus increasing its state size. It is easy to see that this trend will continue to grow rapidly when the system has several prohibitable events that are under the control of multiple modular supervisors (as evident in Chapter 10). Not to mention the extra effort and time that designers need to invest to manually figure out the logic and the right time for forcing every single prohibitable event of the system, deal with the related complications, and then explicitly specify all this logic in the design of various modular TDES supervisors.

From the above discussion, it is evident that automating this logic of disabling *tick* in order to force an eligible prohibitable event, and express it algorithmically is not straightforward. It will almost certainly make our FSM-TDES translation approach and the translated TDES supervisors overly complicated. Moreover, our translation approach will need to have complete information about the TDES plant models and other supervisors in order to disable *tick* at the right time while generating TDES supervisor from a given Moore FSM, which does not look feasible and practical.

This necessitates the need to formulate a new way to automatically force enabled prohibitable events and disable *tick* in the SD framework. The devised method should not only be helpful in the development of our FSM-TDES translation approach, but also simplify the modelling process and logical design of TDES supervisors that are

manually designed in the existing SD setting.

1.5.2 Proposed Solution

In order to automatically satisfy the condition enforced by Point ii ($\Rightarrow$) of the SD controllability property, we propose an approach to automate the mechanism of forcing eligible prohibitable events in the SD supervisory control framework. Our approach is inspired by the event forcing mechanism of physical controllers. In fact, we adopt the controllers' way of forcing events and apply it to our theoretical TDES setting. In the case of controllers, the forcing of an event is indicated not by disabling the *tick*, but by enabling the event. If a controller wants an event to occur, it simply enables it. As soon as all the controllers that control this event enable it, the event occurs. In this case, none of the controllers is explicitly responsible for forcing the event.

This is exactly what we propose to do in the SD framework. Using our proposed approach, if a prohibitable event needs to be forced, it should simply be enabled in the modular supervisor without explicitly disabling the *tick*. As soon as the prohibitable event is enabled by all concerned supervisors and possible in the plant model, *tick* "automatically" gets disabled to force the prohibitable event, thus automatically satisfying the condition checked by Point ii ($\Rightarrow$) of the SD controllability property. Since none of the supervisors is explicitly disabling the *tick* event, there is no concern of making the system potentially uncontrollable by disabling *tick* at the wrong time.

In order to realize the above-mentioned logic, we change the way of constructing the closed-loop system in the SD framework. Specifically, we devise a new synchronization operator, named the *SD synchronous product* ($||_{SD}$), to form the closed-loop system. While synchronizing the plant and supervisor models, our SD synchronous product operator checks to see that at a given state in the closed-loop system, whether *tick* and a prohibitable event are both possible in the plant and enabled by all modular supervisors. If so, our $||_{SD}$ operator disables the *tick* event at the corresponding state of the closed-loop system without relying on any of the supervisor models to explicitly do this action.

In this way, while manually designing TDES supervisors in the existing SD setting, our approach essentially liberates the designers from manually keeping track of the enablement/disablement of *tick* and prohibitable events in various plant and supervisor models, and instead makes the forcing decision implicit. This implies that in the presence of our $||_{SD}$ operator, designers do not need to explicitly specify the event forcing logic in any of the supervisor models. As none of the modular supervisors is responsible for deciding when to force a prohibitable event, they no longer need to keep track of each others' behaviour. In other words, just like controllers, TDES supervisors are only concerned about their own behaviour in our approach. They simply enable the prohibitable event when they want it to occur.

By automating the *tick* disablement/event forcing mechanism in the theoretical TDES setting, our approach aims at simplifying the design logic and modelling process

of TDES supervisors, hence the overall system, in the existing SD setting. This, in turn, makes the existing SD setting more accessible to software and hardware designers and practitioners. Also, our approach bridges the gap between theoretical TDES supervisors and physical controllers by making the event forcing mechanism of supervisors match with that of controllers. This will make our FSM-TDES translation approach simple and straightforward.

Please refer to Section 10.3.2 to see how the design of supervisor ***TakeB2*** of the SD setting gets simplified in the presence of our proposed approach and taking into consideration the automatic *tick* disablement mechanism of our $||_{SD}$ operator.

1.6 Research Questions

In this thesis, we endeavour to answer the following research questions. Table 1.1 maps each research question to the corresponding chapter(s) and/or section(s) that contribute towards answering this question.

RQ1: How to automate the mechanism of forcing eligible prohibitable events in the SD supervisory control theory, and adapt the existing definitions and properties of the SD theory to match the new setting?

RQ2: How to do formal verification of the proposed SD synchronous product setting with respect to the desired properties of controllability and nonblocking?

RQ3: How to provide tool support for the verification of TDES systems that are designed in the SD synchronous product setting?

RQ4: How does the SD synchronous product setting compare to the existing SD setting with respect to the modelling process, logical design and verification time of TDES systems?

RQ5: How to automatically translate Moore synchronous FSM into TDES supervisors that are more likely to satisfy the desired properties of the SD synchronous product setting?

RQ6: What is the effectiveness of the formulated Moore FSM-TDES translation approach and the correctness of the resultant TDES supervisors?

RQ7: How to build compatibility between the two translation approaches (FSM-TDES and TDES-FSM) of the SD supervisory control theory?

1.7 Thesis Contributions

The novel contributions of this thesis are outlined below:

Table 1.1: Mapping Between Research Questions and Thesis Chapters/Sections

Research Questions	Chapter(s) and/or Section(s)
RQ1	Section 1.5.2, Chapter 4
RQ2	Chapters 5−8
RQ3	Chapter 9 (see DESpot (2023) source code for implementation)
RQ4	Chapter 10
RQ5	Chapters 11−12
RQ6	Chapter 13
RQ7	Section C.2

1. **An approach to automatically force eligible prohibitable events in the SD supervisory control theory**
   In order to address **RQ1** and solve the issue raised in Section 1.5.1, we propose an approach to automatically disable the *tick* event, once *tick* and a prohibitable event is possible in the TDES plant model and enabled by all TDES supervisors. We do this by devising a new synchronization operator, called the *SD synchronous product* ($||_{SD}$), that changes the way of constructing closed-loop systems in the SD framework. Based on the $||_{SD}$ operator, we formulate our novel SD synchronous product setting, and adapt the existing definitions and properties of the SD framework to match with the proposed setting. This also paves our way to address **RQ5** later in this thesis.

   As stated in Section 1.5.2, our approach is inspired by the event forcing mechanism of physical controllers and we apply it to the theoretical TDES setting. In this way, we essentially bridge the gap between theoretical TDES supervisors and physical controllers, and make them behave in a similar way with respect to forcing of events.

2. **Establish logical equivalence between the SD and SD synchronous product settings**
   In most cases, the closed and marked languages generated in our $||_{SD}$ setting will be different than the ones obtained in the existing SD setting. This implies that the two settings are not identical. We bridge this gap between the two settings by establishing logical equivalence. First, we identify the elements of the two settings that need to be proven equivalent. Then, we formally prove that the corresponding elements of the two settings are logically equivalent. In doing so, we partly address **RQ2**.

3. **Extend the controllability and nonblocking verification results of the SD setting to the SD synchronous product setting**
   Due to the distinct nature of the SD and $||_{SD}$ settings, the existing verification results of the SD setting do not remain valid in our $||_{SD}$ setting. In order to resolve this issue and completely address **RQ2**, we formally verify our $||_{SD}$ setting with respect to the desired properties of controllability and nonblocking by making use

of the proven logical equivalence between the SD and $||_{SD}$ settings.

By proving the existing verification results of the SD setting for our $||_{SD}$ setting, we have essentially transferred all benefits offered by the existing SD setting to our proposed $||_{SD}$ setting. This is advantageous for designers, as it guarantees that if they design a theoretical TDES system in our $||_{SD}$ setting that is controllable, nonblocking and satisfy the desired $||_{SD}$ properties, then their physically implemented system will retain these properties and the system abides by the specified control laws.

4. **Tool support for the SD synchronous product setting**
As part of addressing **RQ3**, we adapt predicate-based algorithms of the SD setting (Wang, 2009) in order to check the corresponding properties in our $||_{SD}$ setting. We implement these algorithms in the DES research tool, DESpot (2023). By doing so, we allow the designers to automatically verify the desired $||_{SD}$ properties of the TDES systems they design in our $||_{SD}$ setting. This tool support also enables us to address **RQ4**.

5. **A comparison of the SD and SD synchronous product settings**
For the purpose of addressing **RQ4**, we use the case study of a Flexible Manufacturing System. We compare the SD and $||_{SD}$ settings with respect to the process of manually designing TDES supervisors in the two settings, the logical design complexity and size of the resultant TDES supervisors, and the time taken by DESpot (2023) to run the desired verification checks. This comparison between the two settings is crucial, as it clearly demonstrates the benefits that our $||_{SD}$ operator and $||_{SD}$ setting offer to control system designers as compared to the existing SD setting.

6. **A structured approach for the automatic translation of Moore FSM into TDES supervisors**
In order to address the research gap identified in Section 1.4, we propose a novel, generic and structured approach for the automatic translation of Moore synchronous FSM into TDES supervisors. Specifically, we define the input format for expressing the controllers as Moore FSM, identify the prerequisites of our FSM-TDES translation method, and devise the steps and rules for converting a Moore FSM controller into its equivalent TDES supervisor representation. In doing so, we address major parts of **RQ5**.

   We apply our FSM-TDES translation approach to the example of a 4-bit Combination Lock in order to demonstrate its application, effectiveness and correctness. This addresses **RQ6**.

   Our FSM-TDES translation approach facilitates the designers in the formal representation and verification of their existing and new systems. It enables the designers to obtain a formal representation of their controllers without designing TDES supervisors by hand. It also allows them to formally verify their control systems and make use of the SD supervisory control theory without being proficient

in formal methods.

7. **Algorithms to realize the proposed FSM-TDES translation approach**
   We develop a set of algorithms to realize our FSM-TDES translation approach. Specifically, our algorithms evaluate the preconditions of our FSM-TDES translation method, and implement the steps and rules to perform the actual translation process. This addresses the remainder of **RQ5**.

8. **Build compatibility between the two translation approaches defined in the SD supervisory control framework**
   There are two translation approaches that are developed on the basis of the SD supervisory control theory: i) the existing TDES-FSM translation method (Wang, 2009), and ii) the FSM-TDES translation approach proposed in this thesis. We build compatibility between these two translation approaches by modifying the TDES-FSM translation algorithms designed by Hamid (2014) for DESpot (2023). This addresses **RQ7**.

   The compatibility and consistency that we establish between the two translation approaches allow designers to go back and forth between the Moore FSM and TDES supervisors, and automatically translate one model into the other without experts' intervention. This enhances the benefits the SD supervisory control theory provides to the software and hardware designers and practitioners, and makes it more useful and accessible to them.

1.8 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 2 introduces the area of DES and TDES by describing the basic concepts and terminology used in this thesis. Chapter 3 provides an overview of the sampled-data (SD) supervisory control theory upon which our work is based.

In Chapter 4, we present a novel mechanism for constructing closed-loop systems in the SD supervisory control framework. Specifically, we introduce our *SD synchronous product* ($||_{SD}$) operator and describe our $||_{SD}$ setting.

In Chapters 5−7, we discuss and formally prove logical equivalence between the existing SD supervisory control setting and our $||_{SD}$ setting. Utilizing this equivalence, we present the controllability and nonblocking verification results for our $||_{SD}$ setting in Chapter 8.

In Chapter 9, we provide predicate-based algorithms that we have adapted from Wang (2009) to verify various TDES and SD properties in our $||_{SD}$ setting. In Chapter 10, we discuss the example of a Flexible Manufacturing System (FMS) to demonstrate the application and utilization of our $||_{SD}$ operator and $||_{SD}$ setting, and present our verification results for FMS.

In Chapter 11, we introduce our approach for the automatic translation of Moore

synchronous Finite State Machines (FSM) into TDES supervisors. In Chapter 12, we present a series of algorithms that we have developed to realize our automatic Moore FSM-TDES translation approach. In Chapter 13, we discuss the example of a 4-bit Combination Lock system to demonstrate the application and correctness of our FSM-TDES translation approach.

Finally, Chapter 14 finishes off this thesis by stating our conclusions and discussing future work.

Chapter 2

Preliminaries

This chapter presents a summary of the fundamental Discrete-Event System (DES) and Timed DES (TDES) terminology and concepts that we will use in this thesis. Details can be found in Wonham and Cai (2018).

2.1 Linguistic Preliminaries

This section introduces key language concepts that are required to understand the terminology given in the following sections.

2.1.1 Strings

Let Σ be a finite set of distinct symbols (*events*). We refer to Σ as an *alphabet* e.g. $\Sigma = \{\alpha, \beta, \gamma, \sigma\}$. A *string* s over Σ is a finite sequence of events of the form $s = \sigma_1\sigma_2 \dots \sigma_n$, where $\sigma_i \in \Sigma$ and $0 \leq i \leq n$. A string with no events is called an *empty string*, denoted as ϵ, where $\epsilon \notin \Sigma$.

Let Σ^+ be the set of non-empty, finite sequences of events over Σ. We define Σ^* to be the set of all finite sequences of events over Σ, including the empty string ϵ. Thus, we have $\Sigma^* := \Sigma^+ \cup \{\epsilon\}$. Given a string $s = \sigma_1\sigma_2 \dots \sigma_n, |s| = n$ is the *length* of s. The empty string ϵ has a length of zero, i.e. $|\epsilon| = 0$.

Definition 2.1.1. Let $s, t \in \Sigma^*$, where $s = \alpha_1\alpha_2 \dots \alpha_m$ and $t = \beta_1\beta_2 \dots \beta_n$. The operation of *catenation* of strings s and t, $\text{cat} : \Sigma^* \times \Sigma^* \to \Sigma^*$, is defined as:

$$\begin{aligned} \text{cat}(\epsilon, s) = \text{cat}(s, \epsilon) &= s && s \in \Sigma^* \\ \text{cat}(s, t) = st &= \alpha_1\alpha_2 \dots \alpha_m\beta_1\beta_2 \dots \beta_n && s, t \in \Sigma^+ \end{aligned}$$

As $|s| = m$ and $|t| = n$, the length of catenated string is $|\text{cat}(s, t)| = |s| + |t| = m + n$.

Definition 2.1.2. For some $s, t \in \Sigma^*$, we say that t is a *prefix* of s, written as $t \leq s$, if $(\exists u \in \Sigma^*)\ s = tu$.

By definition, a string $s \in \Sigma^*$ is a prefix of itself, since $s \leq s$. Also, we have that ϵ is a prefix of all strings, since $(\forall s \in \Sigma^*)\, \epsilon \leq s$.

2.1.2 Languages

Languages are used to represent system behaviour. A language is defined as a set of strings. Formally, a *language* L over Σ is any subset of Σ^*, i.e. $L \subseteq \Sigma^*$.

Definition 2.1.3. The *prefix closure* of language $L \subseteq \Sigma^*$ is the language $\overline{L}$, defined as $\overline{L} := \{t \in \Sigma^* \mid t \leq s \text{ for some } s \in L\}$.

This definition says that $\overline{L}$ consists of all prefixes of strings of L. By definition, a language L is a subset of the prefix closure of itself, i.e. $L \subseteq \overline{L}$. A language L is said to be *prefix-closed* if $L = \overline{L}$.

Let $\text{Pwr}(\Sigma)$ denote the set of all possible subsets of Σ. For $\sigma \in \Sigma$, we will use the notation $\Sigma^*.\sigma$ to represent the set of all strings $s\sigma$ for some $s \in \Sigma^*$.

Definition 2.1.4. For language $L \subseteq \Sigma^*$ and string $s \in \Sigma^*$, the *eligibility operator* $Elig_L : \Sigma^* \to \text{Pwr}(\Sigma)$ is defined as $Elig_L(s) := \{\sigma \in \Sigma \mid s\sigma \in L\}$.

In simple words, the eligibility operator returns a set of events $\sigma \in \Sigma$ that can follow string s to create a string $s\sigma \in L$.

2.1.3 Nerode Equivalence Relation

Definition 2.1.5. The *nerode equivalence relation*[1] on Σ^* with respect to L, i.e. Σ^* mod L, is defined as:

$$(\forall s, t \in \Sigma^*)\, s \equiv_L t \text{ or } s \equiv t \,(\text{mod } L) \;\text{ iff }\; (\forall u \in \Sigma^*)\, su \in L \;\text{ iff }\; tu \in L$$

This definition states that two strings s and t are nerode equivalent with respect to L if and only if they can be extended by any string $u \in \Sigma^*$ such that either both strings are in L or neither string is in L.

2.2 Discrete Event Systems

Supervisory control theory (SCT) (Wonham and Ramadge, 1987; Ramadge and Wonham, 1989) provides a formal framework for the analysis and control of discrete-event systems (DES). SCT is automaton-based and models DES as the generator of a formal language. The uncontrolled behaviour of the system of interest, modelled by an automaton, is referred to as the *plant* DES. The desired behaviour of the controlled plant is that its generated language be contained in a specification language. To achieve this desired behaviour as per the given specifications, a *supervisor* DES,

[1] See Definition *A*.1.1 of *equivalence relation* in Appendix A.

modelled by an automaton, is introduced. Supervisor DES alters unrestricted behaviour of the plant DES within prescribed limits by operating synchronously with it and using a feedback control mechanism.

This section presents the formal DES representation and fundamental concepts related to DES.

2.2.1 Generator

Definition 2.2.1. A DES is formally represented as a *generator* which is defined as a 5-tuple:

$$\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$$

where Q is the *state set*, Σ is the *event set*, $\delta : Q \times \Sigma \to Q$ is the partial *transition function*, $q_o \in Q$ is the *initial state*, and $Q_m \subseteq Q$ is the *set of marked states*.

The event set Σ of DES $\mathbf{G}$ can be partitioned into the set of *controllable events* (Σ_c) and *uncontrollable events* (Σ_u), i.e. $\Sigma = \Sigma_c \dot\cup \Sigma_u$, where $\dot\cup$ represents *disjoint union* of the two sets, Σ_c and Σ_u. Controllable events can be enabled or disabled by a supervisor, and can occur only when a supervisor enables them. On the other hand, uncontrollable events are not under the control of the supervisor. These events are assumed to be always enabled. Once the plant DES reaches a state where an uncontrollable event is possible, this event cannot be prevented from occurrence.

Each transition in δ is a 3-tuple (or *triple*) of the form (q, σ, q'), where $\delta(q, \sigma) = q'$ such that $q, q' \in Q$ and $\sigma \in \Sigma$. We refer to q as the *exit (source) state* and q' as the *entrance (destination) state*.

The notation $\delta(q, \sigma)!$ means the transition is defined at state $q \in Q$ for event $\sigma \in \Sigma$. We extend δ to $\delta : Q \times \Sigma^* \to Q$ in the natural way as:

$$\begin{aligned} \delta(q, \epsilon) &= q && \text{for } q \in Q \\ \delta(q, s\sigma) &= \delta(\delta(q, s), \sigma) && \text{for } q \in Q, s \in \Sigma^* \text{ and } \sigma \in \Sigma, \text{ as long as } q' := \delta(q, s)! \\ & && \text{and } \delta(q', \sigma)! \end{aligned}$$

For the following definitions, let DES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$.

Definition 2.2.2. A state $q \in Q$ is *reachable* in $\mathbf{G}$ if $(\exists s \in \Sigma^*)\, \delta(q_o, s)!$ & $\delta(q_o, s) = q$.

This definition states that a state q is reachable if, starting from the initial state q_o, there exists a string $s \in \Sigma^*$ that can take us to state q.

Definition 2.2.3. The *reachable state subset* Q_r of $\mathbf{G}$ is defined as:

$$Q_r := \{q \in Q \mid (\exists s \in \Sigma^*)\, \delta(q_o, s) = q\}$$

Definition 2.2.4. A DES $\mathbf{G}$ is *reachable* if all of its states are reachable, i.e. $Q_r = Q$.

Definition 2.2.5. A DES $\mathbf{G}$ is said to be *deterministic* if it has a single initial state, and for each $q \in Q$, and each $\sigma \in \Sigma$, there is at most one σ transition leaving q.

Note: In this thesis, we always assume that a DES is reachable, deterministic and has a finite state space and a finite event set.

Definition 2.2.6. The *closed behaviour* of $\mathbf{G}$ is defined as $L(\mathbf{G}) := \{s \in \Sigma^* \mid \delta(q_o, s)!\}$.

In simple words, we say that $L(\mathbf{G})$ represents all possible sequences of events that could occur in the system. Clearly, $\epsilon \in L(\mathbf{G})$ as long as $Q \neq \emptyset$.

Definition 2.2.7. The *marked behaviour* of $\mathbf{G}$ is defined as:

$$L_m(\mathbf{G}) := \{s \in \Sigma^* \mid \delta(q_o, s)! \ \ \& \ \ \delta(q_o, s) \in Q_m\}$$

The marked behaviour of $\mathbf{G}$ is interpreted as representing the set of all strings in Σ^* that start at q_o and end at a state in Q_m. Marked behaviour represents "completed" tasks carried out by the system that $\mathbf{G}$ is intended to model. Clearly, $L_m(\mathbf{G}) \subseteq L(\mathbf{G})$.

Definition 2.2.8. A DES $\mathbf{G}$ is said to be *nonblocking* if $\overline{L_m(\mathbf{G})} = L(\mathbf{G})$.

This definition says that any string that can be generated by $\mathbf{G}$ is a prefix of (i.e. can always be extended to) a marked string of $\mathbf{G}$. In other words, every string in $L(\mathbf{G})$ can be extended to a completed task in $L_m(\mathbf{G})$.

Definition 2.2.9. For DES $\mathbf{G}$, let λ be an equivalence relation[2] on Q such that $(\forall q, q' \in Q)\ q \equiv q' \pmod{\lambda}$ if and only if:

1. $(\forall s \in \Sigma^*)\, \delta(q, s)! \Leftrightarrow \delta(q', s)!$
2. $(\forall s \in \Sigma^*)\, \delta(q, s)! \ \ \& \ \ \delta(q, s) \in Q_m \Leftrightarrow \delta(q', s)! \ \ \& \ \ \delta(q', s) \in Q_m$

This definition means that for states q and q' such that $q \equiv q' \pmod{\lambda}$, they have the same future with respect to the closed behaviour $L(\mathbf{G})$ and marked behaviour $L_m(\mathbf{G})$. Based on this, for string $s \in L(\mathbf{G})$, a state $q = \delta(q_o, s)$ represents all strings in Σ^* that are nerode equivalent to s mod $L(\mathbf{G})$ and mod $L_m(\mathbf{G})$.

The λ-equivalence relation allows us to reduce a reachable generator to a minimal state version that represents the same closed and marked behaviour.

Definition 2.2.10. A DES $\mathbf{G}$ is said to be *minimal* if:

$$(\forall q, q' \in Q)\, q \equiv q' \pmod{\lambda} \Leftrightarrow q = q'$$

This definition states that for all states $q, q' \in Q$, q is equivalent to $q' \pmod{\lambda}$ if and only if q and q' are the same state. In other words, $\mathbf{G}$ is minimal if it does not have two distinct states q and q' in Q that are λ-equivalent.

2.2.2 DES Synchronization

From the designer's point of view, it is often easier to model the system as several smaller DES components, rather than designing the whole system as a single, more complex DES all at once. These multiple DES components are synchronized together using a *synchronization operator* to construct the complete system. The commonly used synchronization operators include the synchronous product, the product[3], and

[2] See Definition *A*.1.1 of *equivalence relation* in Appendix A.

[3] See Definition *A*.2.1 of *product* operator in Appendix A.

the meet[4] operator. Before defining the synchronous product operator formally, first we will introduce the *natural projection* operator and its *inverse*.

Natural Projection

Let $L_i \subseteq \Sigma_i^*$, for $i = 1, 2$. Let $\Sigma = \Sigma_1 \cup \Sigma_2$.

Definition 2.2.11. The *natural projection* P_i of Σ^* onto Σ_i^*, i.e. $P_i : \Sigma^* \to \Sigma_i^*$, is defined as:

$$P_i(\epsilon) = \epsilon$$
$$P_i(\sigma) = \begin{cases} \epsilon & \text{if } \sigma \notin \Sigma_i \\ \sigma & \text{if } \sigma \in \Sigma_i \end{cases}$$
$$P_i(s\sigma) = P_i(s)P_i(\sigma) \quad \text{for } s \in \Sigma^*, \sigma \in \Sigma$$

This definition says that the action of P_i on a string s is to erase all occurrences of $\sigma \notin \Sigma_i$, that are in s.

Definition 2.2.12. Let $P_i^{-1} : \text{Pwr}(\Sigma_i^*) \to \text{Pwr}(\Sigma^*)$ be the *inverse image function* of P_i, namely for $L \subseteq \Sigma_i^*$, we have $P_i^{-1}(L) := \{s \in \Sigma^* \mid P_i(s) \in L\}$.

Synchronous Product

First, we will define the synchronous product of two languages L_1 and L_2 in terms of natural projection.

Definition 2.2.13. Let $L_i \subseteq \Sigma_i^*$, for $i = 1, 2$. The *synchronous product* $L_1 \,||\, L_2 \subseteq \Sigma^*$ is defined as $L_1 \,||\, L_2 := P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$.

Thus, $s \in L_1 \,||\, L_2$ if and only if $P_1(s) \in L_1$ and $P_2(s) \in L_2$.

Now, we will define the synchronous product of two DES $\mathbf{G}_1$ and $\mathbf{G}_2$.

Definition 2.2.14. Let $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$, for $i = 1, 2$. The *synchronous product* of the two DES, represented as $\mathbf{G} = \mathbf{G}_1 \,||\, \mathbf{G}_2$, is defined as:

$$\mathbf{G} := (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$$

where $\delta((q_1, q_2), \sigma)$ is only defined and equals:

$$(q_1', q_2') \text{ if } \sigma \in (\Sigma_1 \cap \Sigma_2), \delta_1(q_1, \sigma) = q_1', \delta_2(q_2, \sigma) = q_2' \text{ or}$$
$$(q_1', q_2) \text{ if } \sigma \in \Sigma_1 - \Sigma_2, \delta_1(q_1, \sigma) = q_1' \text{ or}$$
$$(q_1, q_2') \text{ if } \sigma \in \Sigma_2 - \Sigma_1, \delta_2(q_2, \sigma) = q_2'$$

Let $L(\mathbf{G}_1)$ and $L(\mathbf{G}_2)$ be the closed behaviour, and $L_m(\mathbf{G}_1)$ and $L_m(\mathbf{G}_2)$ be the marked behaviour of $\mathbf{G}_1$ and $\mathbf{G}_2$ respectively. Synchronizing $\mathbf{G}_1$ and $\mathbf{G}_2$ using the synchronous product operator will generate the closed and marked behaviour of the resultant DES $\mathbf{G} = \mathbf{G}_1 \,||\, \mathbf{G}_2$ as follows:

$$L(\mathbf{G}) = L(\mathbf{G}_1) \,||\, L(\mathbf{G}_2) \quad \text{and} \quad L_m(\mathbf{G}) = L_m(\mathbf{G}_1) \,||\, L_m(\mathbf{G}_2)$$

[4]See Definition *A*.3.1 of *meet* operator in Appendix A.

It follows from Definition 2.2.13 that:

$$L(\mathbf{G}) = P_1^{-1}(L(\mathbf{G}_1)) \cap P_2^{-1}(L(\mathbf{G}_2)) \text{ and } L_m(\mathbf{G}) = P_1^{-1}(L_m(\mathbf{G}_1)) \cap P_2^{-1}(L_m(\mathbf{G}_2))$$

If both $\mathbf{G}_1$ and $\mathbf{G}_2$ are defined over the same alphabet Σ, i.e. $\Sigma = \Sigma_1 = \Sigma_2$, then:

$$L(\mathbf{G}) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2) \quad \text{and} \quad L_m(\mathbf{G}) = L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2)$$

However, if $\mathbf{G}_1$ and $\mathbf{G}_2$ are not defined over the same alphabet Σ, we can simply add *selfloops*[5] to each DES for the missing events at every state to extend them over the same event set Σ, without any loss of generality.

It is important to note here that if $\Sigma = \Sigma_1 = \Sigma_2$, then synchronizing $\mathbf{G}_1$ and $\mathbf{G}_2$ using the synchronous product, product and meet operator will generate the same closed and marked language of the resultant DES $\mathbf{G}$. In this case, we have:

$$L(\mathbf{G}) = L(\mathbf{G}_1 \,||\, \mathbf{G}_2) = L(\mathbf{G}_1 \times \mathbf{G}_2) = L(\mathbf{meet}(\mathbf{G}_1, \mathbf{G}_2)) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2)$$

$$L_m(\mathbf{G}) = L_m(\mathbf{G}_1 \,||\, \mathbf{G}_2) = L_m(\mathbf{G}_1 \times \mathbf{G}_2) = L_m(\mathbf{meet}(\mathbf{G}_1, \mathbf{G}_2)) = L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2)$$

Note: In this thesis, we assume that we have $m > 1$ plant DES components, $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_m$, and that they are always combined using the synchronous product operator to obtain the composite plant DES $\mathbf{G}$, i.e. $\mathbf{G} = \mathbf{G}_1 \,||\, \mathbf{G}_2 \,||\, \ldots \,||\, \mathbf{G}_m$. Likewise, we have $n > 1$ modular supervisor DES, $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n$, and they are always assumed to be combined using the synchronous product to construct the supervisor DES $\mathbf{S}$, i.e. $\mathbf{S} = \mathbf{S}_1 \,||\, \mathbf{S}_2 \,||\, \ldots \,||\, \mathbf{S}_n$. We also assume that both $\mathbf{G}$ and $\mathbf{S}$ are always defined over the same event set Σ, either by modelling the system in this way or by explicitly adding selfloops later on, unless stated otherwise.

2.2.3 Controllability

Let DES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and DES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. As per Definition 2.2.1, $\Sigma = \Sigma_c \,\dot\cup\, \Sigma_u$.

In order to construct the *closed-loop system*, we synchronize plant $\mathbf{G}$ and supervisor $\mathbf{S}$ using a synchronization operator. The behaviour of $\mathbf{G}$ under the control of $\mathbf{S}$ is referred to as the *closed-loop behaviour* of the system.

Definition 2.2.15. Supervisor $\mathbf{S}$ is *controllable* with respect to plant $\mathbf{G}$ if:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))\,(\forall \sigma \in \Sigma_u)\, s\sigma \in L(\mathbf{G}) \Rightarrow s\sigma \in L(\mathbf{S})$$

This definition can be restated in terms of the eligibility operator as follows:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))\, Elig_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq Elig_{L(\mathbf{S})}(s)$$

This definition states that for all legal strings s that are possible in the closed-loop system, an uncontrollable event must be allowed by $\mathbf{S}$ if it is possible in $\mathbf{G}$ after s.

Note: In this thesis, as we will be focusing on timed DES models (introduced in the next section), we will refer to this definition explicitly as the "untimed controllability" definition.

[5] See Definition *A*.4.1 of *selfloop* operation in Appendix A.

2.3 Timed DES

Timed DES (TDES), introduced by Brandin (1993); Brandin and Wonham (1994), is a discrete-time model that extends untimed DES theory by adding a new event called the *tick* (τ) event. The *tick* event represents the passage of one time unit, and corresponds to the tick of a global clock that the system is assumed to be synchronized with. Thus, the event set of a TDES contains the *tick* event as well as other non-*tick* events called *activity events* (Σ_{act}).

Definition 2.3.1. A TDES automaton $\mathbf{G}$ is formally represented as a 5-tuple:

$$\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$$

where Q is the *state set*, $\Sigma = \Sigma_{act} \dot{\cup} \{\tau\}$ is the *event set*, the partial function $\delta : Q \times \Sigma \to Q$ is the *transition function*, $q_o \in Q$ is the *initial state*, and $Q_m \subseteq Q$ is the *set of marked states*. We extend δ to $\delta : Q \times \Sigma^* \to Q$ in the natural way.

TDES contain *forcible events* (Σ_{for}) and *prohibitable events* (Σ_{hib}). Forcible events represent a class of non-*tick* events which are guaranteed to occur before the next clock *tick*, when required. Hence, they can be relied upon to preempt the *tick* event, when needed. The method used by a TDES supervisor to indicate that an event $\sigma \in \Sigma_{for}$ should be forced at a given state, is to disable *tick* at this state. This has the effect of removing the now impossible behaviour that *tick* could occur before σ. Prohibitable events are non-*tick* events that can be enabled or disabled by a supervisor.

Like a DES generator (Definition 2.2.1), the event set Σ of a TDES automaton can be partitioned into the set of *controllable events* (Σ_c) and *uncontrollable events* (Σ_u), i.e. $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$. The set of controllable events in TDES theory is $\Sigma_c = \Sigma_{hib} \dot{\cup} \{\tau\}$, where $\Sigma_{hib} \subseteq \Sigma_{act}$. The set of uncontrollable events is $\Sigma_u = \Sigma - \Sigma_c = \Sigma_{act} - \Sigma_{hib}$.

Let us consider a TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ with the following tuple information:

State set: $Q = \{\text{q0, q1}\}$

Event set: $\Sigma = \{\text{e1, e2, tick}\}$, where $\Sigma_c = \{\text{e1, tick}\}, \Sigma_u = \{\text{e2}\}, \Sigma_{act} = \{\text{e1, e2}\}$ and $\Sigma_{hib} = \{\text{e1}\}$

Transition function: $\delta = \{(\text{q0, e1, q1}), (\text{q1, e2, q1}), (\text{q1, tick, q0})\}$

Initial state: $q_o = \text{q0}$

Set of marked states: $Q_m = \{\text{q0}\}$

This TDES $\mathbf{G}$ is represented graphically in Figure 2.1. The states of $\mathbf{G}$, q0 and q1, are equated with the nodes (circles) of the graph. Transitions are represented by arrows. Arrows are labelled by events, e1, e2 and *tick*, in Σ. The event name e2 in italics and preceded by "!", indicates that the event is uncontrollable. The initial state q0 is represented by a double circle, whereas a filled circle shows that q0 is also a marked state.

Note: In this thesis, we will use the above-mentioned graphical notation to represent our TDES models.

Since TDES framework is an extension of the DES theory, therefore all DES

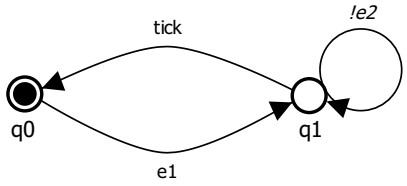


Figure 2.1: An Example TDES Automaton

concepts and properties presented in the previous sections remain valid and applicable to TDES theory. In the following sections, we introduce/restate only those definitions that are specific to TDES framework.

2.3.1 Controllability and Supervision

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor.

Definition 2.3.2. TDES supervisor $\mathbf{S}$ is *timed controllable* with respect to TDES plant $\mathbf{G}$ if $(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))$,

$$Elig_{L(\mathbf{S})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{for} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

This definition states that supervisor $\mathbf{S}$ must accept an uncontrollable event if it is possible in the plant $\mathbf{G}$ after a legal string s. In addition, $\mathbf{S}$ must enable a *tick* event if it is possible in $\mathbf{G}$, unless there exists an eligible forcible event in the system to preempt the *tick*.

Note: In this thesis, as we will only be dealing with TDES models, therefore we will drop the word "timed", and will refer to this property as "$\mathbf{S}$ is controllable with respect to $\mathbf{G}$" for simplicity.

Definition 2.3.3. A *TDES supervisory control* for $\mathbf{G}$ is any map $V : L(\mathbf{G}) \to \mathrm{Pwr}(\Sigma)$ such that $(\forall s \in L(\mathbf{G}))$,

$$V(s) \supseteq \begin{cases} \Sigma_u \cup (\{\tau\} \cap Elig_{L(\mathbf{G})}(s)) & \text{if } V(s) \cap Elig_{L(\mathbf{G})}(s) \cap \Sigma_{for} = \emptyset \\ \Sigma_u & \text{if } V(s) \cap Elig_{L(\mathbf{G})}(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

In the following definitions, we write $V/\mathbf{G}$ to denote the pair $(\mathbf{G}, V)$, i.e. to represent $\mathbf{G}$ under the supervision of V.

Definition 2.3.4. The *closed behaviour* of $V/\mathbf{G}$ is the language $L(V/\mathbf{G}) \subseteq L(\mathbf{G})$ defined inductively as follows:

i. $\epsilon \in L(V/\mathbf{G})$

ii. If $s \in L(V/\mathbf{G}), \sigma \in V(s)$, and $s\sigma \in L(\mathbf{G})$ then $s\sigma \in L(V/\mathbf{G})$

iii. No other strings belong to $L(V/\mathbf{G})$

$L(V/\mathbf{G})$ is prefix-closed, nonempty, and in the range $\{\epsilon\} \subseteq L(V/\mathbf{G}) \subseteq L(\mathbf{G})$.

Definition 2.3.5. The *marked behaviour* of $V/\mathbf{G}$, $L_m(V/\mathbf{G})$, is defined as:

$$L_m(V/\mathbf{G}) := L(V/\mathbf{G}) \cap L_m(\mathbf{G})$$

Definition 2.3.6. V is said to be *nonblocking for* $\mathbf{G}$ if $\overline{L_m(V/\mathbf{G})} = L(V/\mathbf{G})$.

2.3.2 Control Equivalent Supervisors

Let $\mathbf{G} = (Q, \Sigma_{\mathbf{G}}, \delta, q_o, Q_m)$ be a TDES plant. Let $\mathbf{S}_1 = (X_1, \Sigma_1, \xi_1, x_{o,1}, X_{m,1})$ and $\mathbf{S}_2 = (X_2, \Sigma_2, \xi_2, x_{o,2}, X_{m,2})$ be two TDES supervisors.

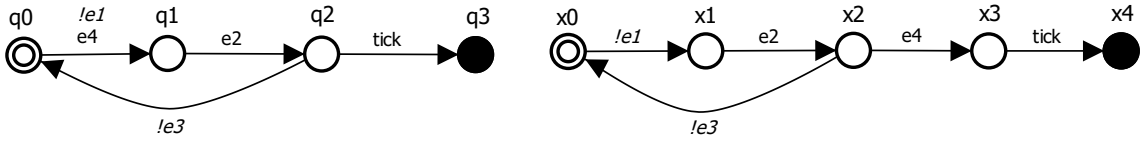
Definition 2.3.7. Supervisors $\mathbf{S}_1$ and $\mathbf{S}_2$ are considered to be *control equivalent* for a given plant $\mathbf{G}$, if they produce the same closed-loop behaviour.

As this definition specifically focuses on the "closed-loop behaviour", two points are notable and worth elaborating.

1. This definition does not make any assumptions about how the two closed-loop systems are constructed, i.e. it is independent of the synchronization operators that are used to form the two closed-loop systems. The two supervisors $\mathbf{S}_1$ and $\mathbf{S}_2$ may be combined with $\mathbf{G}$ using the same synchronization operator, e.g. synchronous product, or two different synchronization operators, e.g. $\mathbf{S}_1$ is combined with $\mathbf{G}$ using synchronous product, and $\mathbf{S}_2$ is combined with $\mathbf{G}$ using the sampled-data synchronous product operator (introduced in Section 4.1). As long as the closed-loop behaviour of the two systems is the same, the definition remains applicable and valid, and the choice of synchronization operator(s) is not important. This will allow us to compare the action of two supervisors that are combined with the same plant, but using different operators to construct the closed-loop systems.
2. The definition is given with respect to the closed-loop behaviour of the two systems, i.e. the closed and marked languages, and not in terms of the actual closed-loop system automata. This is because a TDES representation of the two closed-loop systems having the same closed-loop behaviour might not be exactly the same due to different state labels. They might not even be identical up to state relabelling as one TDES could be in its minimal form and the other one could be a non-minimal version. However, irrespective of their minimal or non-minimal representation, their closed and marked languages will still be same.

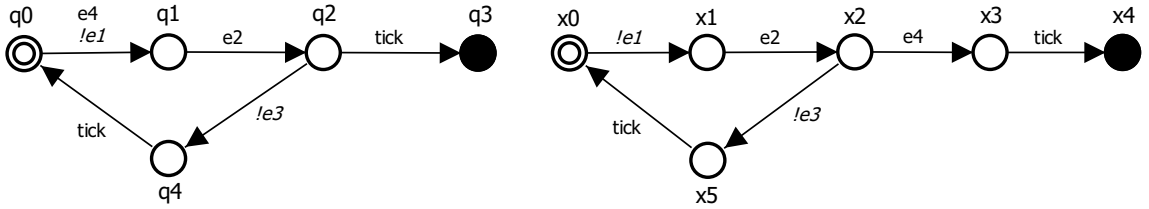
Based on the above discussion, we can restate the definition of two supervisors being control equivalent for a given plant model (Definition 2.3.7) as follows.

Definition 2.3.8. Let $\mathbf{G}_{cl,1}$ be the closed-loop system that is constructed by synchronizing $\mathbf{S}_1$ and $\mathbf{G}$, and let $\mathbf{G}_{cl,2}$ be the closed-loop system that is formed by combining $\mathbf{S}_2$ and $\mathbf{G}$. Then $\mathbf{S}_1$ and $\mathbf{S}_2$ are said to be *control equivalent* for $\mathbf{G}$ if $L(\mathbf{G}_{cl,1}) = L(\mathbf{G}_{cl,2})$ and $L_m(\mathbf{G}_{cl,1}) = L_m(\mathbf{G}_{cl,2})$.



(a) TDES Plant **G** (b) TDES Supervisor **S**

Figure 2.2: An Example to Illustrate Various TDES Properties



(a) ALF TDES Plant **G** (b) ALF TDES Supervisor **S**

Figure 2.3: An Example Satisfying ALF Property

2.3.3 TDES Properties

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor.

We will use an example TDES plant **G** (Figure 2.2a) and TDES supervisor **S** (Figure 2.2b) shown in Figure 2.2 to illustrate various TDES properties.

First, we want to impose a technical condition on our TDES to exclude the physically unrealistic possibility that a *tick* transition might be preempted indefinitely by repeated execution of an activity loop within a fixed unit time interval.

Definition 2.3.9. TDES **G** is said to have an *activity-loop* if:

$$(\exists q \in Q)\,(\exists s \in \Sigma^+_{act})\;\delta(q, s) = q$$

In Figure 2.2a, **G** has activity loops of "e1-e2-e3-e1" and "e4-e2-e3-e4" that could preempt the *tick* event from occurring for an indefinite amount of time. Likewise, *tick* event in **S** can be preempted indefinitely by repeated execution of "e1-e2-e3-e1" activity loop, as shown in Figure 2.2b. To rule this out, we require that a TDES must be activity-loop-free.

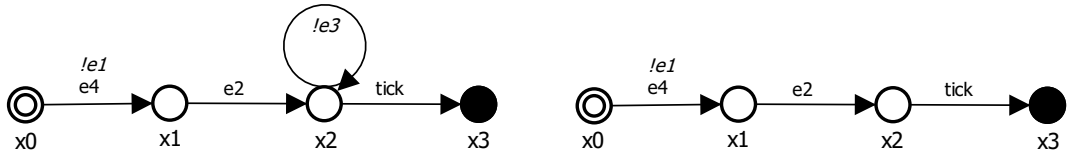
Definition 2.3.10. TDES **G** is *activity-loop-free (ALF)* if:

$$(\forall q \in Q_r)\,(\forall s \in \Sigma^+_{act})\;\delta(q, s) \neq q$$

Please note that this definition is given in terms of only the reachable states, since unreachable states do not contribute to the closed and marked behaviour of a TDES.

One simple way to make our **G** and **S** of Figure 2.2 ALF is by adding a *tick* transition after transition 'e3'. This ALF version of **G** and **S** is shown in Figure 2.3.

Practically, it is not always possible to make supervisors ALF, as they typically have selfloops of activity events. However, these selflooped events are sometimes not possible in the plant model, thus making the closed-loop system ALF. Therefore,



(a) Non-Selfloop ALF TDES **S** (b) ALF TDES **S′**

Figure 2.4: An Example Illustrating Non-Selfloop ALF Property

in the sampled-data supervisory control theory (Wang, 2009; Leduc *et al.*, 2014), the authors desire that their supervisors should preferably satisfy a less restrictive condition of being non-selfloop ALF.

Definition 2.3.11. Let **S** be a TDES, and let **S′** be **S** with all activity event selfloops removed. **S** is *non-selfloop ALF* if **S′** is ALF.

This definition states that if we remove all activity event selfloops from a non-selfloop ALF TDES **S**, then it must become ALF.

A non-selfloop ALF TDES **S** is shown in Figure 2.4a. If we remove the selfloop of activity event e3 at state x2, the TDES becomes ALF, as shown in Figure 2.4b.

The following defintion is taken from Wong and Wonham (1996). Only plant TDES are required to satisfy this property.

Definition 2.3.12. TDES **G** has *proper time behaviour* if:

$$(\forall q \in Q_r)\,(\exists \sigma \in \Sigma_u \cup \{\tau\})\ \delta(q,\sigma)!$$

It says that at each reachable state, either an uncontrollable event or a *tick* event must be possible. This ensures that a TDES can never express that a prohibitable event must occur before the next *tick*, since a supervisor could disable that prohibitable event, thus "stopping the clock". This is neither desirable nor realistic.

TDES plant **G** shown in Figure 2.2a does not have proper time behaviour. The reason is that at state q1, neither an uncontrollable event nor *tick* event is possible. The only event possible at state q1 is the prohibitable event e2.

Usually, controllable events are often part of the supervisor's implementation. This means that supervisors can make these events to occur at any time, even when the plant model says they can't. In order to prevent the violation of the plant model, the property of plant completeness was defined with respect to controllable events by Balemi (1994). It has been adapted to use only prohibitable events for the sampled-data supervisory control theory (Leduc *et al.*, 2014), which is the basis of our work.

Definition 2.3.13. A TDES plant **G** is *complete* for TDES supervisor **S** if:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))\,(\forall \sigma \in \Sigma_{hib})\ s\sigma \in L(\mathbf{S}) \Rightarrow s\sigma \in L(\mathbf{G})$$

This definition states that for every state in **G**, if a prohibitable event σ is enabled by **S**, it must be possible in **G**. This condition can be seen as dual to the definition of **S** being controllable with respect to **G** (Definition 2.3.2).

In Figure 2.2, **S** enables prohibitable event e4 at state x2. However, event e4 is not possible in **G** at state q2, thus violating the property of plant completeness.

Chapter 3

Sampled-Data Supervisory Control

Sampled-Data (SD) supervisory control theory (Wang, 2009; Wang and Leduc, 2012; Leduc *et al.*, 2014) focuses on the implementation of TDES supervisors as SD controllers. It establishes sufficient conditions to ensure that if a theoretical TDES is controllable, nonblocking, and satisfies these properties, then the physical implementation will also have these properties and exhibit correct behaviour as specified by the control laws.

In this chapter, we will only focus on those aspects of the SD methodology that are required to follow our work presented in the following chapters. To gain a thorough understanding of the SD supervisory control theory, please refer to Wang (2009); Wang and Leduc (2012); Leduc *et al.* (2014).

It is worth clarifying here that in the SD supervisory control setting (or "*SD setting,*" for short) described in Wang (2009), the closed-loop system is constructed by combining the TDES plant $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ and the TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ using the **meet** operator, i.e. $\mathbf{meet}(\mathbf{G}, \mathbf{S})$, and all theoretical proofs and results are given in terms of the **meet**. However, in Wang and Leduc (2012); Leduc *et al.* (2014), the product operator is used to form the closed-loop system, expressed as $\mathbf{G} \times \mathbf{S}$, and discuss all verification results.

As noted in Section 2.2.2, if $\mathbf{G}$ and $\mathbf{S}$ are both defined over the same event set, then $\mathbf{meet}(\mathbf{G}, \mathbf{S})$, $\mathbf{G} \times \mathbf{S}$, and $\mathbf{G} \,||\, \mathbf{S}$ will produce the same closed and marked behaviours, and can thus be used interchangeably. To keep things simple and consistent throughout this thesis, we will use the synchronous product operator to discuss the SD supervisory control framework. In this case, we assume that $\mathbf{S}$ and $\mathbf{G}$ are defined over the same event set. We will thus define the closed-loop system to be $\mathbf{S} \,||\, \mathbf{G}$. The system's closed behaviour is thus defined as $L(\mathbf{S} \,||\, \mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$, and its marked behaviour as $L_m(\mathbf{S} \,||\, \mathbf{G}) = L_m(\mathbf{S}) \cap L_m(\mathbf{G})$.

3.1 SD Controllers

A *sampled-data (SD) controller* is driven by a global periodic clock whose clock edge is associated with the *tick* (τ) event of the TDES. It views the system as a series of inputs and outputs that can take the values of *True* and *False* only. On the rising edge of the clock, it samples its inputs, changes its state based on the inputs and current state, and updates its outputs based on the new state it has transitioned to.

To use an SD controller to manage a given system, an input is associated with each non-*tick* event, called an *activity event*, and an output with each non-*tick* controllable event, called a *prohibitable event*. The occurrence of an event is indicated by its input going true during a given clock period. A prohibitable event is considered enabled when its corresponding output has been set true by the controller, disabled otherwise. If a prohibitable event is enabled at a given state, the controller will always make sure it happens before the next clock edge. For example, in a digital logic implementation, the output set to true is usually taken to mean that the event has occurred.

An SD controller samples inputs, changes state and updates outputs only on a clock edge. This has the following implications: 1) An SD controller knows nothing about the occurrence of events in a given sampling period (clock period) until the next clock edge. 2) On the next clock edge, the only information it receives is which events have occurred in a given sampling period. 3) Neither does it know anything about the order the events occurred in, nor the number of times an event has occurred in a given sampling period. 4) An SD controller updates the enablement and forcing information on the clock edge and then keeps it unchanged for the entire clock period.

Figure 3.1 shows an example of event sampling with respect to an SD controller. The left figure shows that $Event1$ and $Event2$ occurred in the 2^{nd} sampling period. However, an SD controller will know nothing about the occurrence of these events until the next clock edge, i.e. 3^{rd} rising edge of the clock. On the next clock edge, the only information it receives is that $Event1$ and $Event2$ occurred in the sampling period that has just ended, without any information about the order or frequency of occurrence of these events (right figure). This means that an SD controller will not know about the exact string that actually happened in the last sampling period, and cannot differentiate between strings such as "Event1-Event2-τ", "Event2-Event1-τ", "Event1-Event2-Event1-τ" or "Event2-Event2-Event1-τ".

3.2 Concurrency and Timing Issues

Timed DES theory assumes that: 1) events occur in an interleaving fashion (we can always determine the event ordering), 2) we know immediately when events occur, and 3) enablement and forcing occur immediately (i.e. no communication delay).

Because these assumptions are not true in general for SD controllers, several concurrency and timing issues arise when representing TDES supervisors as deterministic

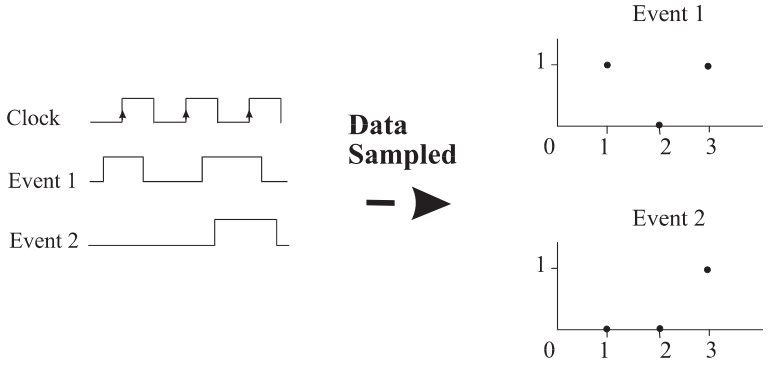


Figure 3.1: An Example for Event Sampling *(Reprinted from Wang (2009))*

SD controllers. For example, if multiple forcible events are enabled in a single clock period (i.e. there is a *choice*), how does the controller decide which events to generate/force in the current clock period, and in which order to force the events? Likewise, if an event is enabled for multiple clock periods (say 3 clock periods), how does the controller decide when to force it and in which clock period (force it in the 1^{st}, 2^{nd} or 3^{rd} clock period)? Also, if an SD controller is forcing multiple events (say e1 and e2) in the same clock period, these events may only actually occur in a specific order (say "e1-e2" only) even though the TDES model says they can occur in multiple orderings (say "e2-e1" as well). This could even vary from one implementation to the other.

These issues have ramifications with respect to controllability, plant model correctness, and the SD controller's ability to determine which state the TDES currently is in. They could make the controller implementation block, uncontrollable, or violate the specified control laws, even though our original TDES is nonblocking and controllable. Also, these issues are important for the unambiguous translation of TDES supervisors into SD controllers and to obtain a deterministic controller.

These issues are primarily addressed in the SD supervisory control framework by introducing the property of SD controllability (Section 3.5).

3.3 SD Assumptions

The SD approach makes the following assumptions that must be met by the system designer while developing the TDES models.

1. The set of prohibitable events is exactly equal to the set of forcible events, i.e. $\Sigma_{for} = \Sigma_{hib}$.
2. A prohibitable event is forced in the same sampling period in which it is enabled. It is only allowed to occur once per clock period.
3. When an event is forced in a given sampling period, no assumptions are made about exactly when the event will occur during that clock period. This is because timing may vary depending upon the controller's implementation.

4. The SD controllers will be implemented centrally with a common clock such that they all are synchronized, i.e. they all sample inputs and update outputs at the same time. Moreover, the controllers generate all prohibitable events, so that there is no issue of communication delay with respect to event enablement/disablement.
5. An event is assumed to have "occurred" when its input goes true. If this happens so close to the clock edge that it shows up in the next sampling period, then it "occurs" immediately after the clock edge. The system designer should reflect this in the plant model.
6. The length of an input pulse should be appropriate to be detected and interpreted correctly by the controller. It should not be so short that it could be missed by the controller (i.e. occurs between two clock edges). It should also not be so long that the controller sees and interprets it as an event occurring multiple times in different clock periods, when the event actually occurred only once in the current clock period.

Assumptions 1, 4, 5 are not very restrictive and essentially represent modelling issues. Assumptions 4, 5 partially deal with timing and communication delay issues.

Note: As we build our work on the SD supervisory control theory, these assumptions apply to our study as well.

3.4 SD Preliminaries

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor.

An SD controller samples inputs and changes state on the clock edge, which is associated with the *tick* event of the TDES. This means an SD controller can only observe strings ending with a *tick*. Additionally, it can also see the empty string (ϵ) that represents the initial state of the system which is always known. Such strings are referred to as sampled strings.

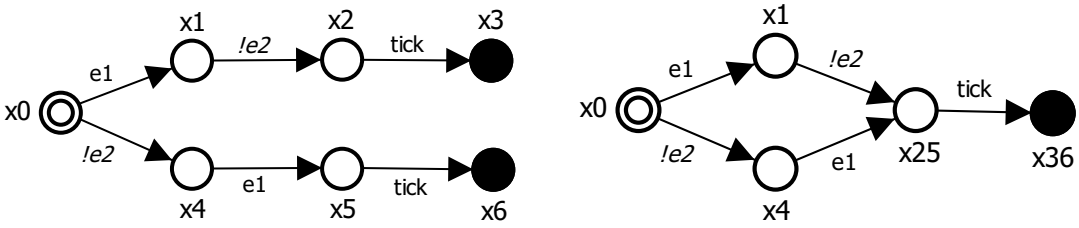
Definition 3.4.1. Given an event set Σ, the set of *sampled strings*, L_{samp}, is defined as $L_{samp} = \Sigma^* \,.\, \tau \cup \{\epsilon\}$.

Sampled strings represent observable points in the system. If the controller is implementing TDES supervisor $\mathbf{S}$, states reached from the initial state by sampled strings represent states in $\mathbf{S}$ that are at least partially observable. These states are referred to as sampled states.

Definition 3.4.2. For supervisor $\mathbf{S}$, the set of *sampled states*, X_{samp}, is defined as:

$$X_{samp} = \{x \in X \mid (\exists s \in L(\mathbf{S}) \cap L_{samp})\, x = \xi(x_o, s)\}$$

An SD controller changes state after each clock edge (*tick*). Its next state is determined by all the strings that can occur containing a single *tick* event at the end, since the last *tick* event. Such strings are referred to as concurrent strings.



(a) Non-CS Deterministic Supervisor **S** (b) CS Deterministic Supervisor **S**′

Figure 3.2: An Example Illustrating CS Deterministic Supervisor Property

Definition 3.4.3. Given a set of sampled strings L_{samp} defined over an event set Σ, the set of *concurrent strings*, L_{conc}, is defined as $L_{conc} = \Sigma^*_{act} \,.\, \tau \subset L_{samp}$.

Two concurrent strings containing the same events but in different order/number are indistinguishable to an SD controller. An occurrence operator is defined to capture this uncertainity. The occurrence operator takes a string and returns the set of events (*occurrence image*) that make up the string.

Definition 3.4.4. For $s \in \Sigma^*$, the *occurrence operator*, $Occu : \Sigma^* \rightarrow \text{Pwr}(\Sigma)$, is defined as $Occu(s) := \{\sigma \in \Sigma \mid s \in \Sigma^*.\sigma.\Sigma^*\}$.

If two concurrent strings with the same occurrence image are possible at a given sampled state and they lead to two different states in **S**, this will make the translation of **S** into an SD controller ambiguous and the translated SD controller non-deterministic. To circumvent this undesirable situation, TDES supervisors are required to be concurrent string deterministic.

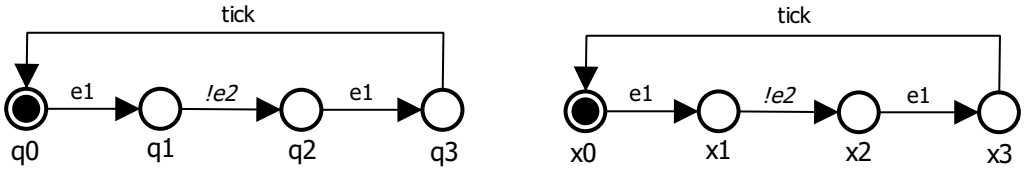
Definition 3.4.5. A TDES supervisor **S** is *concurrent string (CS) deterministic*, if:

$$(\forall s \in L(\mathbf{S}) \cap L_{samp})\,(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\mathbf{S}) \wedge Occu(s') = Occu(s'')] \Rightarrow$$
$$[ss' \equiv_{L(\mathbf{S})} ss'' \wedge ss' \equiv_{L_m(\mathbf{S})} ss'' \wedge \xi(x_o, ss') = \xi(x_o, ss'')]$$

A supervisor **S** failing the CS deterministic property is shown in Figure 3.2a. In **S**, two concurrent strings, "e1-e2-τ" and "e2-e1-τ", leave the initial state x0. Despite having the same occurrence image of {e1, e2, τ}, they go to two different sampled states, x3 and x6. In this case, we note that **S** fails Definition 3.4.5 because it is not minimal (Definition 2.2.10). For example, states x3 and x6 are λ-equivalent (Definition 2.2.9) and can be combined together. This is also true for states x2 and x5. After combining these λ-equivalent states, the resulting minimal TDES **S**′ is shown in Figure 3.2b. Please note that we cannot merge two or more states to obtain a CS deterministic supervisor if they are not λ-equivalent.

One of the assumptions (Point 2 of Section 3.3) says that the controllers allow prohibitable events to occur only once per sampling period. This must be reflected in the TDES plant model and is captured by the following property.

Definition 3.4.6. For TDES plant **G** and TDES supervisor **S**, **G** is said to have



(a) TDES Plant **G** (b) TDES Supervisor **S**

Figure 3.3: An Example Failing **S**-Singular Prohibitable Behaviour Property

S-*singular prohibitable behaviour* if:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp})\,(\forall s' \in \Sigma^*_{act})\, ss' \in L(\mathbf{S}) \cap L(\mathbf{G}) \Rightarrow$$
$$(\forall \sigma \in Occu(s') \cap \Sigma_{hib})\, \sigma \notin Elig_{L(\mathbf{G})}(ss')$$

An example failing the property of **S**-singular prohibitable behaviour is shown in Figure 3.3. Plant **G** (Figure 3.3a) does not have **S**-singular prohibitable behaviour with respect to supervisor **S** (Figure 3.3b). This is because the prohibitable event e1 is possible twice in the given sampling period in **G**, at state q0 and q2, and this event is also allowed by **S**.

3.5 SD Controllability

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor.

Assume a theoretical system with the following properties: 1) TDES **G** and **S** have finite state spaces and finite event sets, 2) **G** has proper time behaviour and is complete for **S**, 3) **S** is controllable with respect to **G**, 4) **S** is CS deterministic, and 5) $\mathbf{S} \,||\, \mathbf{G}$ is ALF and nonblocking. Even if TDES satisfy the above-mentioned properties, the actual system behaviour under the control of the corresponding SD controller could block, violate the control laws, or exhibit behaviour not contained in **G**. To address these issues and handle the problems discussed in Section 3.2, the property of SD controllability is introduced.

Definition 3.5.1. TDES supervisor **S** is *SD controllable* with respect to TDES plant **G** if, $\forall s \in L(\mathbf{S}) \cap L(\mathbf{G})$, the following statements are satisfied:

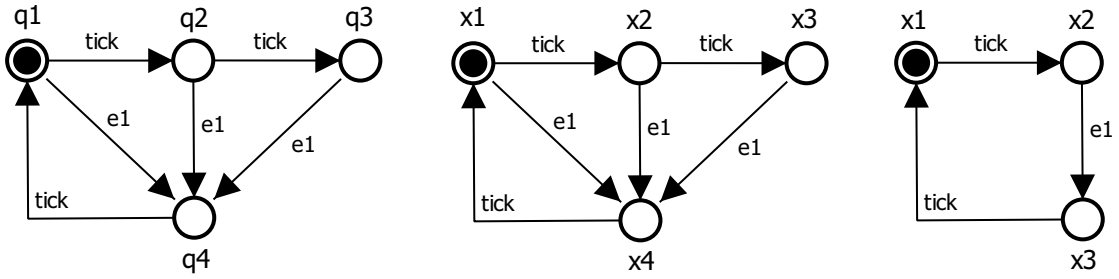
i) $Elig_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq Elig_{L(\mathbf{S})}(s)$

ii) If $\tau \in Elig_{L(\mathbf{G})}(s)$, then $\tau \in Elig_{L(\mathbf{S})}(s) \Leftrightarrow Elig_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

iii) If $s \in L_{samp}$ then

1) $(\forall s' \in \Sigma^*_{act})\,[ss' \in L(\mathbf{S}) \cap L(\mathbf{G})] \Rightarrow$
$$[Elig_{L(\mathbf{S})\cap L(\mathbf{G})}(ss') \cup Occu(s')] \cap \Sigma_{hib} = Elig_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib}$$

2) $(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\mathbf{S}) \cap L(\mathbf{G}) \wedge Occu(s') = Occu(s'')] \Rightarrow$
$$ss' \equiv_{L(\mathbf{S})\cap L(\mathbf{G})} ss'' \wedge ss' \equiv_{L_m(\mathbf{S})\cap L_m(\mathbf{G})} ss''$$



(a) TDES **G** (b) TDES **Sup1** (c) TDES **Sup2**

Figure 3.4: An Example of SD Controllability Point ii (⇒)

iv) $L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \subseteq L_{samp}$

We now give a brief explanation for each of these points.

Point i: This is the standard untimed controllability property (Definition 2.2.15).

Point ii: In the reverse direction (⇐), it says that a *tick* event cannot be disabled unless there exists an eligible prohibitable event to preempt the *tick*. Together with Point i, this implies standard timed controllability (Definition 2.3.2), since $\Sigma_{for} = \Sigma_{hib}$ in the SD setting.

The forward direction (⇒) states that if a prohibitable event is enabled, *tick* must be disabled. This captures the notion that a prohibitable event is enabled only when it needs to be forced, otherwise it must remain disabled. This removes the ambiguity about which sampling period an enabled prohibitable event should be forced in, by making enabling and forcing essentially one and the same. This not only makes the conversion of TDES supervisors into SD controllers simple and straightforward, but also ensures that TDES behaviour is closer to the implementation by removing forcing options that are not actually used in the physical system.

For plant **G** (Figure 3.4a), supervisor **Sup1** (Figure 3.4b) does not satisfy Point ii (⇒) as both *tick* and prohibitable event e1 are enabled at states x1 and x2. This creates uncertainty about when event e1 should be forced (at state x1, x2 or x3) and makes the translation of TDES supervisors into SD controllers ambiguous. However, supervisor **Sup2** (Figure 3.4c) satisfies this property and removes the ambiguous and unused behaviour by allowing *tick* to occur at state x1 and forcing prohibitable event e1 at state x2.

Point iii: For a sampled string s, the following two sub points must be satisfied.

Point iii.1: This point expresses that when a prohibitable event is possible in a clock period, it must be possible immediately after the *tick* and stay possible for the period until it occurs. This captures two ideas: 1) The enablement information of an SD controller is constant for the entire clock period. 2) When a controller forces a prohibitable event, the event must occur before the next *tick*, but we don't know when. So the event must be possible in the plant for the entire clock period till it occurs and must be able to interleave with the other events occurring in the same clock period. Point iii.1 bridges the gap between TDES supervisors and SD controllers by

restricting the way the TDES supervisors change their enablement information and makes it consistent with the SD controllers. This property also emphasizes that if two prohibitable events should occur in a specific order, they must be forced in separate clock periods.

Point iii.2: This point states that if two concurrent strings with the same occurrence image are possible after a given sampled string, they must have the same future with respect to the system's closed behaviour (i.e. same control action must be taken now and in the future for both strings), and with respect to its marked behaviour (i.e. the strings are interchangeable with respect to reaching future marked states).

Point iv: All marked strings in the closed-loop system must be sampled strings.

It is worth noting that Point iii and Point iv apply to both **G** and **S**.

3.6 Formal Model of SD Controller

In the SD supervisory control framework, an SD controller is modelled as a Moore synchronous Finite State Machine (FSM) (Brown and Vranesic, 2013). A *Moore FSM* is a Moore state machine that changes state only on the rising or falling edge of the clock. It chooses its next state based on its current state and inputs. Its outputs are determined by its current state only.

Before giving the formal definition of an SD controller, first we need to introduce some notation.

The inputs and outputs of an SD controller are represented as *boolean vectors*. A boolean vector is a vector whose individual elements can only be assigned the values of *True* (1) or *False* (0). These vectors of information change periodically with respect to some clock.

Let $k \in \{0, 1, 2, \ldots\}$. For any vector $\mathbf{v} = [v_1, v_2, \ldots, v_n] \in V$ or any of its element v_j where $j \in \{1, \ldots, n\}$, "$\mathbf{v}(k)$" and "$v_j(k)$" is used to denote the value of $\mathbf{v}$ and v_j at time k. "At time k" means that k clock ticks have gone by since the starting reference point, $k = 0$. For $k = 0$, $\mathbf{v}(0)$ represents the initial or starting value of $\mathbf{v}$. $k = 0$ represents the time when an SD controller has just been turned on or reset. As index k takes on new values, vector $\mathbf{v}$ defines a sequence with respect to the clock ticks, which are defined to be $\{\mathbf{v}(k) \,|\, k = 0, 1, 2 \ldots\}$, and is denoted as $\{\mathbf{v}(k)\}$. A 'clock tick' corresponds to the occurrence of a *tick* event of a TDES.

Definition 3.6.1. An *SD controller* **C** is defined as a 6-tuple, $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$, where:

- I is the set of possible boolean vectors that the *inputs* of the controller can take on. Each vector $\mathbf{i} = [i_0, i_1, \ldots, i_{v-1}] \in I$ has v input variables. Each element of I corresponds to a unique activity event in the system. When an element is set to 1, this means the corresponding event has occurred at least once in the previous clock period, otherwise it is set to 0. Each input vector $\mathbf{i}(k') \in \{\mathbf{i}(k)\}$ is sampled at

the occurrence of a *tick* event, except for $k = 0$ which occurs when the controller is turned on.

- Z is the set of possible boolean vectors that the *outputs* of the controller can take on. Each vector $\mathbf{z} = [z_0, z_1, \ldots, z_{r-1}] \in Z$ has r output variables. Each element of Z corresponds to a unique prohibitable event in the system. When an element is set to 1, this means the corresponding event is enabled and the controller should make the event occur before the next clock tick, where 0 means it is disabled. Each output vector $\mathbf{z}(k') \in \{\mathbf{z}(k)\}$ is generated at the occurrence of a *tick* event, except for $k = 0$ which occurs when the controller is turned on.
- Q is the set of possible boolean vectors that the *states* of the controller can take on. Each vector $\mathbf{q} = [q_0, q_1, \ldots, q_{l-1}] \in Q$ has l state variables. Starting at $k = 1$, each state $\mathbf{q}(k') \in \{\mathbf{q}(k)\}$ changes to next state $\mathbf{q}(k' + 1) \in \{\mathbf{q}(k)\}$ at the occurrence of a *tick* event.
- $\Omega : Q \times I \to Q$ is the *next-state* function. It takes the current state $\mathbf{q}(k) \in Q$ and an input vector $\mathbf{i}(k + 1) \in I$, and returns the next state $\mathbf{q}(k + 1) \in Q$ such that $\mathbf{q}(k + 1) = \Omega(\mathbf{q}(k), \mathbf{i}(k + 1))$.
- $\Phi : Q \to Z$ is the *state-to-output* map. For state $\mathbf{q} \in Q$, the output $\mathbf{z} \in Z$ at this state is defined as $\mathbf{z} = \Phi(\mathbf{q})$.
- $\mathbf{q}_{res} \in Q$ is the *initial (reset)* state for when the controller starts operating or is reset. Thus we have $\mathbf{q}(0) = \mathbf{q}_{res}$.

Starting at time $k = 0$, a specific run of the controller would give a specific sequence of inputs $\{\mathbf{i}(k)\}$. This sequence, combined with $\mathbf{q}_{res}$ and Ω, will uniquely define the current sequence of states, $\{\mathbf{q}(k)\}$. In turn, $\{\mathbf{q}(k)\}$ and Φ will uniquely define the current sequence of outputs, $\{\mathbf{z}(k)\}$. To distinguish between two vector sequences, different variables will be used, e.g. $\{\mathbf{i}(k)\}$ and $\{\mathbf{i}(k')\}$.

3.7 TDES to FSM Translation

In this section, we introduce the TDES to FSM translation method from Wang and Leduc (2012). We focus on the aspects that are required to comprehend our work presented in the following chapters. Please refer to Wang (2009); Wang and Leduc (2012) for an in-depth discussion of the complete translation method.

The TDES to FSM translation starts with a CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$. By using the information for $\mathbf{S}$, it constructs the corresponding SD controller $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$.

In order to do the translation, each item in the controller's tuple (i.e. I, Z, Ω, etc.) needs to be defined in terms of TDES $\mathbf{S}$. To do this, the authors have defined several translation functions. These functions capture the next state behaviour and enablement information from $\mathbf{S}$, associate events with elements of input and output vectors, and associate sampled states of $\mathbf{S}$ with states of the controller. They also

map event subsets of input or output vectors, as well as define the controller's next state logic (Ω) and state-to-output map (Φ) in terms of supervisor **S**.

3.7.1 Translation Functions

Let TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let $\Sigma_{act} \subset \Sigma$ be the set of all activity events, and $\Sigma_{hib} \subseteq \Sigma_{act}$ be the set of all prohibitable events. Let $X_{samp} \subseteq X$ be the set of sampled states of **S**. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the controller implementation of a CS deterministic supervisor **S**.

A formal TDES to FSM translation method has been developed by defining several translation functions. These translation functions can be used to take the components of **S**, and define the components of **C**. Below, we only list down those functions that we need to prove our equivalence of the SD controllers presented in Chapter 7.

TDES Mapping Functions

The following two functions express the SD behaviour of a TDES.

Definition 3.7.1. Let **S** be a CS deterministic TDES. For $x \in X_{samp}$ and $\Sigma' \subseteq \Sigma_{act}$, the partial function of *next sampling state function*, $\Delta : X_{samp} \times \mathrm{Pwr}(\Sigma_{act}) \to X_{samp}$, is defined as:

$$\Delta(x, \Sigma') := \begin{cases} \xi(x, s) & \text{if } (\exists s \in L_{conc})\, \xi(x, s)! \;\&\; Occu(s) \cap \Sigma_{act} = \Sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

The next sampling state function represents how a TDES will move from one sampled state to the next via concurrent strings.

Definition 3.7.2. Let TDES supervisor **S** be SD controllable with respect to TDES plant **G**. For $x \in X_{samp}$, the *prohibited action function*, $\zeta : X_{samp} \to \mathrm{Pwr}(\Sigma_{hib})$, is defined as $\zeta(x) := \{\sigma \in \Sigma_{hib} \mid \xi(x, \sigma)!\}$.

This function defines the control action that will take place at a given sampled state x, i.e. it captures the prohibitable events that are enabled at x.

Event Mapping Functions

For the following event mapping functions, let $\Sigma_{\mathbf{S}} \subseteq \Sigma$ be the event set of a CS deterministic supervisor **S**.

Definition 3.7.3. Let bijective map[1] $\gamma_g : \Sigma_{act} \to \{0, \ldots, |\Sigma_{act}| - 1\}$ be the *canonical event mapping function* such that $(\forall \sigma_1, \sigma_2 \in \Sigma_{act})\, \sigma_1 = \sigma_2 \Leftrightarrow \gamma_g(\sigma_1) = \gamma_g(\sigma_2)$.

[1] See Definition *A*.5.1 of *bijective function* in Appendix A.

Definition 3.7.4. The *input event mapping function* for $\mathbf{C}$ is a bijective map $\gamma : \Sigma_{\mathbf{S}} \cap \Sigma_{act} \to \{0, 1, \ldots, v-1\}$, where $v = |\Sigma_{\mathbf{S}} \cap \Sigma_{act}|$. It is defined such that:

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{\mathbf{S}} \cap \Sigma_{act})\, \gamma_g(\sigma_1) < \gamma_g(\sigma_2) \Rightarrow \gamma(\sigma_1) < \gamma(\sigma_2)$$

Definition 3.7.5. The *output event mapping function* for $\mathbf{C}$ is a bijective map $\eta : \Sigma_{\mathbf{S}} \cap \Sigma_{hib} \to \{0, 1, \ldots, r-1\}$, where $r = |\Sigma_{\mathbf{S}} \cap \Sigma_{hib}|$. It is defined such that:

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{\mathbf{S}} \cap \Sigma_{hib})\, \gamma_g(\sigma_1) < \gamma_g(\sigma_2) \Rightarrow \eta(\sigma_1) < \eta(\sigma_2)$$

Controller Functions

For the following definitions, let $\mathbf{C}$ be the corresponding controller for CS deterministic supervisor $\mathbf{S}$.

Definition 3.7.6. Let $\Sigma_{act} \subset \Sigma$ be the set of global activity events. Let $\mathbf{i}_g$ be a single input vector that the system sees, i.e. it is globally available. $\mathbf{i}_g = [i_{g,0}, i_{g,1}, \ldots, i_{g,v_g-1}]$ is required to be defined over Σ_{act}, where $v_g = |\Sigma_{act}|$. That is, for any event $\sigma \in \Sigma_{act}$, there is an element in $\mathbf{i}_g$ that corresponds to σ and only σ. We call $\{\mathbf{i}_g(k)\}$ a *canonical input sequence*, and $\mathbf{i}_g \in \{\mathbf{i}_g(k)\}$ a *canonical input vector*[2].

Definition 3.7.7. For CS deterministic supervisor $\mathbf{S}$, let $\Lambda : X_{samp} \to Q$ be an arbitrary injective map, where $X_{samp} \subseteq X$. Λ is a *state mapping function* for $\mathbf{C}$ if, for all $x \in X_{samp}$, $\Lambda(x)$ returns a vector of state variables $\mathbf{q} = [q_0, q_1, \ldots, q_{l-1}]$ such that:

$$(\forall x_1, x_2 \in X_{samp})\, \Lambda(x_1) = \Lambda(x_2) \Leftrightarrow x_1 = x_2$$

The initial state is also a sampled state, and is mapped to be $\Lambda(x_o) = \mathbf{q}_{res} = \mathbf{q}(0)$.

Definition 3.7.8. Let γ be the input event mapping function for $\mathbf{C}$. A bijective map of *input set mapping function* for $\mathbf{C}$, $\Gamma_I : \mathrm{Pwr}(\Sigma_{act}) \to I$, is defined as follows. For arbitrary $\Sigma_I \subseteq \Sigma_{act}$, we have $\Gamma_I(\Sigma_I) = [i_0, i_1, \ldots, i_{v-1}]$ such that for $j = 0, 1, \ldots, v-1$,

$$i_j := \begin{cases} 1 & \text{if } (\exists \sigma \in \Sigma_I)\, \gamma(\sigma) = j \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.7.9. Let η be the output event mapping function for $\mathbf{C}$. A bijective map of *output set mapping function* for $\mathbf{C}$, $\Gamma_Z : \mathrm{Pwr}(\Sigma_{hib}) \to Z$, is defined as follows. For arbitrary $\Sigma_Z \subseteq \Sigma_{hib}$, we have $\Gamma_Z(\Sigma_Z) = [z_0, z_1, \ldots, z_{r-1}]$ such that for $j = 0, 1, \ldots, r-1$,

$$z_j := \begin{cases} 1 & \text{if } (\exists \sigma \in \Sigma_Z)\, \eta(\sigma) = j \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.7.10. Let Δ be the next sampling state function for $\mathbf{S}$, and let $X_{samp} \subseteq X$. For state $\mathbf{q} \in Q$ and arbitrary input $\mathbf{i} \in I$, the *next state function* Ω is defined as:

$$\Omega(\mathbf{q}, \mathbf{i}) := \begin{cases} \Lambda(\Delta(x, \Gamma_I^{-1}(\mathbf{i}))) & \text{if } (\exists x \in X_{samp})\, \mathbf{q} = \Lambda(x)\ \&\ \Delta(x, \Gamma_I^{-1}(\mathbf{i}))! \\ \text{arbitrary} & \text{otherwise} \end{cases}$$

[2]The use of "canonical" here refers to the size and ordering of the inputs, not to the actual values of the input sequence or a given vector.

Definition 3.7.11. Let ζ be the prohibited action function for $\mathbf{S}$. For state $\mathbf{q} \in Q$, the *state-to-output map* Φ is defined as:

$$\Phi(\mathbf{q}) := \begin{cases} \Gamma_Z(\zeta(x)) & \text{if } (\exists x \in X_{samp})\, \mathbf{q} = \Lambda(x) \\ \Gamma_Z(\emptyset) & \text{otherwise} \end{cases}$$

3.7.2 Translation Method

This section defines the TDES to FSM translation method from Wang (2009); Wang and Leduc (2012), and provides a simple example.

To translate a TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ into an SD controller $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$, $\mathbf{S}$ must be CS deterministic to ensure that the resulting SD controller is deterministic. This translation method will not work otherwise. In practice, $\mathbf{S}$ should preferably be non-selfloop ALF as well. However, this is just a design aid, and not a hard requirement.

In order to construct an SD controller $\mathbf{C}$, the values for each member of its tuple need to be defined. To define I, Z and Q, the authors define the size of each vector, as each element will represent a distinct $\sigma \in \Sigma_{act}$, $\sigma \in \Sigma_{hib}$, or state $x \in X_{samp}$ respectively. For each $i \in I$, its size is defined as $v = |\Sigma_{act}|$. For each $z \in Z$, its size is defined as $r = |\Sigma_{hib}|$.

To define the size of Q, the size of each $q \in Q$ needs to be large enough to encode a unique value for each $x \in X_{samp} \subseteq X$. If each state contains l elements, 2 unique values can be expressed. Thus, l is selected such that $2^{l-1} < |X_{samp}| \leq 2^l$.

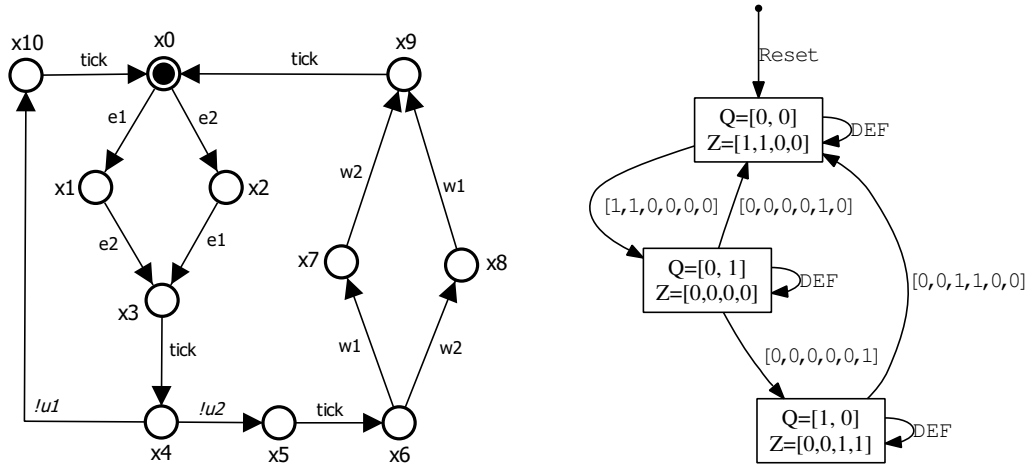
The mapping functions (given in the previous section) are then used to associate event subsets and sampled states to specific values in I (map Γ_I), Z (map Γ_Z) and Q (map Λ). The initial/reset state is immediately set to $\mathbf{q}_{res} = \Lambda(x_o)$.

Next, Definition 3.7.10 is used to define the controller's next state function, Ω. It is notable that if the input vector does not represent a concurrent string accepted by $\mathbf{S}$, the next state (and thus the resulting logic) is defined arbitrary.

Finally, Definition 3.7.11 is used to define the controller's state-to-output map, Φ. Please note that if state q does not represent a sampled state (i.e. $|X_{samp}| < 2^l$, and thus have unused states), then all of its outputs are set to $False$ (0).

Informally, the translation process begins by taking the sampled states of $\mathbf{S}$ as the states of $\mathbf{C}$. The initial state of $\mathbf{S}$ would be the initial (*reset*) state of $\mathbf{C}$. Next step is to determine which concurrent strings are possible from a given sampled state. The occurrence image of these concurrent strings would then define the next-state conditions, and the state will be changed accordingly.

As an example, consider the CS deterministic supervisor $\mathbf{S}$ and its corresponding translated FSM shown in Figure 3.5. The sampled states of $\mathbf{S}$ (Figure 3.5a), x0 (initial state), x4 and x6, are equated to three states in the FSM (Figure 3.5b), $\mathbf{q}_{res}$ = x0 = $[0, 0]$, x4 = $[0, 1]$ and x6 = $[1, 0]$. We assume the ordering I = [e1, e2, w1, w2, u1, u2], and Z = [e1, e2, w1, w2]. As only two prohibitable events, e1 and e2, are possible at state x0 in $\mathbf{S}$, only these outputs are set to 1 at state $[0, 0]$ in the FSM. Similarly, all



(a) TDES Supervisor **S** (b) Translated FSM

Figure 3.5: An Example of TDES to FSM Translation Method *(Reprinted from Wang (2009))*

outputs are set to 0 at state $[0, 1]$, and only w1 and w2 outputs are set to 1 at state $[1, 0]$.

Examining state x0, we see that the only concurrent strings leaving it are "e1-e2-τ" and "e2-e1-τ". They have the same occurrence image and both strings take us to the same next state x4 in **S**. Thus, our next-state condition is that only when e1 and e2 have occurred, we go to state $[0, 1]$ in the FSM. Next-state conditions for other sampled states are determined in the similar fashion.

As ξ of **S** is a partial function and Ω of **C** is a total function, a **DEF** (default) transition usually needs to be added to the translated FSM. **DEF** is a shorthand notation to cover input combinations that are not explicitly specified, i.e. it matches all the remaining unspecified input combinations.

3.8 Supervisory Control

The concept of supervisory control V (Definition 2.3.3) is originally defined in terms of the set of forcible events, Σ_{for}. Since $\Sigma_{for} = \Sigma_{hib}$ in the SD setting, this definition has been expressed with respect to the set of prohibitable events as follows.

Definition 3.8.1. A *TDES supervisory control* for $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ is a map $V : L(\mathbf{G}) \to \mathrm{Pwr}(\Sigma)$, such that $(\forall s \in L(\mathbf{G}))$,

$$V(s) \supseteq \begin{cases} \Sigma_u \cup (\{\tau\} \cap Elig_{L(\mathbf{G})}(s)) & \text{if } V(s) \cap Elig_{L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ \Sigma_u & \text{if } V(s) \cap Elig_{L(\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases}$$

Note: In this thesis, as we will only be dealing with TDES models, therefore we will drop the word "TDES", and will often refer to this property as "V is a supervisory control for **G**".

Definition 3.8.2. For TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, the *concurrent behaviour* of $\mathbf{G}$ is defined to be a map $CB_{\mathbf{G}} : L(\mathbf{G}) \cap L_{samp} \rightarrow \text{Pwr}(L_{conc})$[3], such that for $s \in L(\mathbf{G}) \cap L_{samp}$, $CB_{\mathbf{G}}(s) := \{s' \in L_{conc} \mid ss' \in L(\mathbf{G})\}$.

It says that the possible concurrent behaviour for $\mathbf{G}$ after sampled string s, is the set of concurrent strings that can extend s to a string in the closed behaviour of $\mathbf{G}$.

Since an SD controller only changes state when a *tick* occurs, it is difficult to relate its control action directly to strings. Therefore, a corresponding supervisory control V is constructed to express the enablement information that controller $\mathbf{C}$ would provide to plant $\mathbf{G}$. Precisely, it captures two ideas: 1) Enablement information changes immediately after a *tick* event and then stays constant till the next *tick*. 2) As soon as a prohibitable event is enabled, the controller will force the event to occur before the next *tick*.

Algorithm 3.1 constructs supervisory control V (Proposition 3.3 (page 48) shows that V is indeed a TDES supervisory control) by keeping track of how controller $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ changes state in response to strings generated by plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$. A brief description of the algorithm, and variables used in the algorithm follows.

- $Pend \subseteq L_{samp} \times Q$: Set of pending string-state pairs, $(s, \mathbf{q})$, to be analyzed, where s is a sampled string in $L(\mathbf{G})$, and $\mathbf{q} \in Q$ is the corresponding state in $\mathbf{C}$ reached by input sequences that would match the concurrent strings that make up s. If $s = \epsilon$, then $\mathbf{q} = \mathbf{q}_{res}$.
- Σ_V: Set of prohibitable events enabled by $V(s)$ for current sampled string s that is being processed.
- Σ_{temp}: Copy of Σ_V that is made while processing a concurrent string that extends sampled string s that is currently being processed. It keeps track of the prohibitable events in Σ_V that have not yet occurred in substrings of the concurrent strings that extend s in $L(\mathbf{G})$.

For all strings $s \in L(\mathbf{G})$, the algorithm starts by adding all uncontrollable events (Σ_u) and *tick* ($\boldsymbol{\tau}$) event to $V(s)$ from **lines 1-3**. This is done to satisfy Definition 3.8.1 of supervisory control V.

As controller always starts operating at its reset state, $(\epsilon, \mathbf{q}_{res})$ is the 1st string-state pair that is added to *Pend* at **line 4**. All string-state pairs that get added to *Pend* during the execution of the algorithm are extracted and analyzed one by one in the **while-loop** running from **lines 5-31**.

At **lines 6-7**, the next string-state pair to be analyzed, $(s, \mathbf{q})$, is extracted and removed from *Pend*. At **line 8**, for current state $\mathbf{q}$ of $\mathbf{C}$, the output vector $\mathbf{z}$ is obtained by applying the state-to-output map Φ (Definition 3.7.11). At **line 9**, $\mathbf{z}$ is used to construct Σ_V using the inverse of output set mapping function Γ_Z (Definition 3.7.9).

[3]This map is different from the map of Definition 4.1 given in Leduc *et al.* (2014) due to an error in the original definition.

Algorithm 3.1 Obtaining V from Controller $\mathbf{C}$, Acting on Plant $\mathbf{G}$

1: **for all** $s \in L(\mathbf{G})$ **do**
2: $V(s) \leftarrow \Sigma_u \cup \{\tau\}$
3: **end for**
4: $Pend \leftarrow \{(\epsilon, \mathbf{q}_{res})\}$
5: **while** $Pend \neq \emptyset$ **do**
6: $(s, \mathbf{q}) \leftarrow$ a member from $Pend$
7: $Pend \leftarrow Pend - \{(s, \mathbf{q})\}$
8: $\mathbf{z} \leftarrow \Phi(\mathbf{q})$
9: $\Sigma_V \leftarrow \Gamma_Z^{-1}(\mathbf{z})$
10: **if** $\Sigma_V \neq \emptyset$ **then**
11: $V(s) \leftarrow (V(s) \cup \Sigma_V) - \{\tau\}$
12: **end if**
13: **for all** $s' \leftarrow \sigma_1\sigma_2 \ldots \sigma_j \in CB_{\mathbf{G}}(s)$ **do** // $\sigma_j = \tau$ by definition
14: **if** $(Occu(s') \cap \Sigma_{hib} \subseteq \Sigma_V) \wedge (ss' \in L(\mathbf{S}))$ **then**
15: $\Sigma_{temp} \leftarrow \Sigma_V$
16: $\mathbf{i} \leftarrow \Gamma_I(Occu(s') - \{\tau\})$
17: $\mathbf{q}' \leftarrow \Omega(\mathbf{q}, \mathbf{i})$
18: $Pend \leftarrow Pend \cup \{(ss', \mathbf{q}')\}$
19: **if** $j > 1$ **then**
20: **for** $i \leftarrow 1$ to $j - 1$ **do**
21: $\Sigma_{temp} \leftarrow \Sigma_{temp} - \sigma_i$
22: **if** $\Sigma_{temp} \neq \emptyset$ **then**
23: $V(s\sigma_1\sigma_2 \ldots \sigma_i) \leftarrow (V(s\sigma_1\sigma_2 \ldots \sigma_i) \cup \Sigma_V) - \{\tau\}$
24: **else**
25: $V(s\sigma_1\sigma_2 \ldots \sigma_i) \leftarrow (V(s\sigma_1\sigma_2 \ldots \sigma_i) \cup \Sigma_V)$
26: **end if**
27: **end for**
28: **end if**
29: **end if**
30: **end for**
31: **end while**
32: **return** V

Σ_V now contains the set of all prohibitable events enabled by $\mathbf{C}$ at state $\mathbf{q}$.

Lines 10-12 process $V(s)$. If any prohibitable event is enabled at state $\mathbf{q}$ (**line 10**), the enablement information Σ_V is added to $V(s)$ for current sampled string s (**line 11**). Also, since a prohibitable event is enabled and needs to be forced, *tick* (added at **line 2**) gets removed from $V(s)$ to satisfy Point ii ($\Rightarrow$) of the SD controllability definition (Definition 3.5.1).

Lines 13-30 loops through all possible concurrent strings s' that extend s in

$L(\mathbf{G})$ ($s' = \sigma_1\sigma_2\ldots\sigma_j \in CB_{\mathbf{G}}(s)$). However, at **line 14**, those concurrent strings whose occurrence images contain prohibitable events that have been disabled by $\mathbf{C}$ at state $\mathbf{q}$ (not in Σ_V) are ignored. **Line 14** also disregards concurrent strings that do not represent a valid behaviour by extending s in $L(\mathbf{S})$, thus restricting the valid strings to $L(\mathbf{S}) \cap L(\mathbf{G})$. As these illegal strings represent behaviour that will not actually happen in the closed-loop system, they are left at their default enablement information (**line 2**).

Line 15 copies Σ_V to Σ_{temp}. Using the occurrence image of concurrent string s', **line 16** computes input vector $\mathbf{i}$ by applying input set mapping function Γ_I (Definition 3.7.8). At **line 17**, the next-state function Ω (Definition 3.7.10) is used to compute the next state $\mathbf{q}'$ of $\mathbf{C}$ that is reached from $\mathbf{q}$ by $\mathbf{i}$. This new string-state pair $(ss', \mathbf{q}')$ also needs to be analyzed, so it is added to *Pend* at **line 18**.

Line 19 checks to see if s' contains any activity events (for $j = |s'|$, if $j > 1$). If so, each substring $\sigma_1\sigma_2\ldots\sigma_i$, where $i < j$, is analyzed from **lines 20-27**.

Line 21 potentially removes one prohibitable event from Σ_{temp}. If Σ_{temp} contains more prohibitable events that have not yet occurred (**line 22**), then *tick* is removed from $V(s\sigma_1\sigma_2\ldots\sigma_i)$ to force the remaining enabled prohibitable events in the current sampling period (**line 23**). Otherwise, *tick* event is not removed from $V(s\sigma_1\sigma_2\ldots\sigma_i)$ (**line 25**). Moreover, in both cases, Σ_V is added to $V(s\sigma_1\sigma_2\ldots\sigma_i)$, since the enablement information of $\mathbf{C}$ remains constant until the next *tick*.

It is worth clarifying that this algorithm abstractly describes how map V is related to $\mathbf{C}$. As $L(\mathbf{G})$ may not be finite, there might be infinite number of string-state pairs to analyze, and the algorithm may never terminate. In Wang (2009), the authors have proven that map V constructed from $\mathbf{C}$ using this algorithm is well defined.

Definition 3.8.3. For plant $\mathbf{G}$, and CS deterministic supervisor $\mathbf{S}$ that is SD controllable for $\mathbf{G}$, let $\mathbf{C}$ be the SD controller that is constructed from $\mathbf{S}$ using the translation method described in Section 3.7, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 3.1. The *marked behaviour* of $V/\mathbf{G}$ is defined as $L_m(V/\mathbf{G}) := L(V/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})$.

Definition 3.8.4. V is said to be *nonblocking for* $\mathbf{G}$ if $\overline{L_m(V/\mathbf{G})} = L(V/\mathbf{G})$.

3.9 Verification Results

Comprehensive theoretical proofs and results for verifying the control action of an SD controller and comparing it to that of the TDES supervisor from which it was converted are presented in Wang (2009); Leduc *et al.* (2014). In this section, we only outline some significant conclusions, and restate those theorems/propositions that we will refer to in our work presented in the subsequent chapters.

For TDES plant $\mathbf{G}$, TDES supervisor $\mathbf{S}$ and an SD controller $\mathbf{C}$, the system is required to satisfy the following properties: 1) $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces and finite event sets, 2) $\mathbf{G}$ has proper time behaviour, 3) $\mathbf{G}$ is complete for $\mathbf{S}$, 4) $\mathbf{G}$ has

S-singular prohibitable behaviour, 5) $\mathbf{S} \,||\, \mathbf{G}$ is ALF, 6) **S** is SD controllable with respect to **G**, 7) **S** is CS deterministic, and 8) **C** is an SD controller translated from **S** as described in Section 3.7. Given that these conditions are met, the following results have been proven for the SD supervisory control methodology.

3.9.1 SD Controller as a Supervisory Control

To compare the control action of **S** and **C**, a supervisory control V is constructed using Algorithm 3.1. It is demonstrated in Wang (2009) that V is indeed a map that expresses the enablement and forcing behaviour of **C**.

Proposition 3.1 given below is taken from Leduc *et al.* (2014). Although the control action of **C** could be quite different than that of **S**, this proposition proves that if any string is not accepted by **S**, it will also be rejected by **C**, i.e. if a certain path is not possible in the theoretical model, it can never occur in the implemented system, thus preventing the physical system to behave in an undesirable and unexpected way.

Proposition 3.1. (Leduc *et al.*, 2014) For plant **G** and supervisor **S**, let **S** be CS deterministic and SD controllable for **G**, and let **G** be complete for **S**, and have **S**-singular prohibitable behaviour. Let **C** be the SD controller that is constructed from **S**.
$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp})\,(\forall s' \in CB_{\mathbf{G}}(s))$
If s takes **C** to state $\mathbf{q}$ and $ss' \notin L(\mathbf{S})$ then **C** will reject s'.

3.9.2 Controllability

Using Proposition 3.2, Theorem 3.1 given below proves that the closed-loop behaviour of **G** under the control of **C** (represented as $L(V/\mathbf{G})$) is same as the closed-loop behaviour of **S** and **G**. This is despite the fact that **S** can change its enablement and forcing information at any time, as opposed to **C** that is restricted to do so only on the clock edge and then it must keep it constant during the entire clock period. This shows that SD controllers can be used to implement TDES supervisors and obtain the expected closed-loop behaviour, at least with respect to the required enablement and forcing actions of the controller.

Proposition 3.2. (Leduc *et al.*, 2014) For CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from **S**.
$(\forall s \in L(\mathbf{S}) \cap L_{samp})$
String s will take **C** to state $\mathbf{q} = \Lambda(\xi(x_o, s))$ with outputs $\sigma \in \Sigma_{\mathbf{q}} = Elig_{L(\mathbf{S})}(s) \cap \Sigma_{hib}$ set to *true*.

Theorem 3.1. (Leduc *et al.*, 2014) For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable for **G**, let both TDES have finite state spaces, let **G** be complete for **S**, have proper time and **S**-singular prohibitable behaviour, let $\mathbf{S} \,||\, \mathbf{G}$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD

controller that is constructed from $\mathbf{S}$, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 3.1. Then, $L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$.

By proving the following proposition, it has been demonstrated that map V is indeed a TDES supervisory control for $\mathbf{G}$.

Proposition 3.3. (Leduc *et al.*, 2014) For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, have proper time and $\mathbf{S}$-singular prohibitable behaviour, let $\mathbf{S} \,||\, \mathbf{G}$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 3.1. Then map V is a TDES supervisory control for $\mathbf{G}$.

3.9.3 Event Generation

Theorem 3.2 has been proven in Leduc *et al.* (2014) to show that $\mathbf{C}$ cannot generate a prohibitable event when $\mathbf{G}$ won't accept it. This result guarantees that illegal transitions won't occur, thus preventing the system from violating control laws. It also means that $\mathbf{G}$ will accurately reflect the system's behaviour when controlled by $\mathbf{C}$.

Theorem 3.2. (Leduc *et al.*, 2014) For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, have proper time and $\mathbf{S}$-singular prohibitable behaviour, let $\mathbf{S} \,||\, \mathbf{G}$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 3.1.
$(\forall s \in L(V/\mathbf{G}) \cap L_{samp})\,(\forall s' \in \Sigma^*_{act})\,(\forall \sigma \in \Sigma_{hib})$
If $ss' \in L(V/\mathbf{G})$ and σ then physically occurs after ss' and before any other events can occur, then $ss'\sigma \in L(\mathbf{G})$.

3.9.4 Nonblocking

Before discussing the nonblocking verification results, the following concept has been introduced in the SD setting.

Definition 3.9.1. Let $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a TDES plant, and let V and V' be supervisory controls for $\mathbf{G}$. V' is said to be *concurrent supervisory control equivalent (CSCE)* to V if:

1. $(\forall s \in L(\mathbf{G}))\, V'(s) \subseteq V(s)$
2. $(\forall s \in L(V'/\mathbf{G}) \cap L_{samp})\,(\forall s' \in L_{conc})\, ss' \in L(V/\mathbf{G}) \Rightarrow$
$$(\exists s'' \in L_{conc})\, ss'' \in L(V'/\mathbf{G}) \wedge Occu(s') = Occu(s'')$$

Point 1 requires that each event allowed by $V'(s)$ is also allowed by $V(s)$. This is to ensure that $L(V'/\mathbf{G})$ does not include any unwanted behaviour. Point 2 requires

that if $V'/\mathbf{G}$ accepts a sampled string s, and $V/\mathbf{G}$ accepts a concurrent string s' after s, then $V'/\mathbf{G}$ must accept a concurrent string s'' that has the same occurrence image as s'.

In the SD setting, the following theorem has been proven to show that $\mathbf{G}$ under the control of $\mathbf{C}$ is nonblocking if and only if $\mathbf{S}||\mathbf{G}$ is nonblocking. This is true even if only a single concurrent string, out of multiple possible concurrent strings with the same occurrence image possible in the TDES model at a given sampled state, is actually possible in the physical system. The SD approach has been proven to be robust with respect to such variations and nonblocking.

Theorem 3.3. (Leduc *et al.*, 2014) For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, have proper time and $\mathbf{S}$-singular prohibitable behaviour, let $\mathbf{S} \,||\, \mathbf{G}$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 3.1. Let V' be a supervisory control for $\mathbf{G}$. If V is nonblocking for $\mathbf{G}$ and V' is CSCE to V, then V' is also nonblocking for $\mathbf{G}$.

Chapter 4

Sampled-Data Synchronous Product

In this chapter, we present a novel mechanism for constructing closed-loop system in the SD supervisory control framework. Specifically, we devise a new synchronization operator, called the *sampled-data (SD) synchronous product*, to combine TDES plant **G** and TDES supervisor **S** to form the closed-loop system. After defining our SD synchronous product operator, we discuss and prove the relevant fundamental properties of this synchronization operator. This is followed by a description of our SD synchronous product setting.

As we are proposing a new way of constructing the closed-loop system, existing properties of the SD supervisory control theory need to be adapted to work with our new synchronization operator. The rest of this chapter focuses on adapting these properties to make them compatible with our SD synchronous product setting. Finally, this chapter finishes off with some useful results about the activity-loop-free property (Definition 2.3.10) with respect to our SD synchronous product setting.

4.1 SD Synchronous Product Operator

In this section, we define our new synchronization operator, called the *sampled-data (SD) synchronous product*, represented as $||_{SD}$, to combine two TDES models. This operator is specifically designed to synchronize TDES plant **G** and TDES supervisor **S** in order to construct a closed-loop system in the SD supervisory control framework, and to address the issues discussed in Section 1.5.1.

The SD synchronous product operator is basically an intelligent and powerful version of the standard synchronous product operator. It is smart enough to automatically disable a *tick* event in the closed-loop system, if both *tick* and prohibitable events are possible in **G** and enabled by **S**.

This implies that in the presence of the SD synchronous product operator, while

designing the system, designers no longer need to keep track of the enablement/disablement of *tick* event and prohibitable events, and incorporate this logic of explicit *tick* disablement manually in various modular TDES supervisors. This also means that while verifying the system model, the property of SD controllability Point ii ($\Rightarrow$) no longer needs to be explicitly checked, as the SD synchronous product operator guarantees that this property will always be satisfied at every state of the closed-loop system (we elaborate this point later in Section 4.5).

Definition 4.1.1. Let TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$, for $i = 1, 2$. The *sampled-data (SD) synchronous product* of two TDES, represented as $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$, is defined as:

$$\mathbf{G} := (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$$

where $\delta((q_1, q_2), \sigma)$, for $(q_1, q_2) \in Q_1 \times Q_2$ and $\sigma \in \Sigma_1 \cup \Sigma_2$, is only defined and equals:

i) (q_1', q_2') if $\sigma \in (\Sigma_1 \cap \Sigma_2) \ \wedge \ \delta_1(q_1, \sigma) = q_1' \ \wedge \ \delta_2(q_2, \sigma) = q_2' \ \wedge$ $[(\sigma \neq \tau) \ \vee \ ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib}) \neg \delta((q_1, q_2), \sigma')!)]$

ii) (q_1', q_2) if $\sigma \in (\Sigma_1 - \Sigma_2) \ \wedge \ \delta_1(q_1, \sigma) = q_1'$

iii) (q_1, q_2') if $\sigma \in (\Sigma_2 - \Sigma_1) \ \wedge \ \delta_2(q_2, \sigma) = q_2'$

Note: From now on, we will refer to this synchronization operator by interchangeably using its name "SD synchronous product" and its symbol "$||_{SD}$" (to be concise).

We will now explain the logic used by the SD synchronous product operator to construct the transition function δ, as this is the only element where the logic of the SD synchronous product differs from the standard synchronous product operator.

The $||_{SD}$ operator constructs the transition function δ of $\mathbf{G}$ based on the component transition functions, δ_1 of $\mathbf{G}_1$ and δ_2 of $\mathbf{G}_2$. As δ_1 and δ_2 are partial functions, the transition function δ constructed by $||_{SD}$ is a partial function as well.

The $||_{SD}$ operator states three rules to define δ. For every state (q_1, q_2) of $\mathbf{G}$ and each $\sigma \in \Sigma_1 \cup \Sigma_2$, these rules are used to determine: (I) If σ transition would be defined at state (q_1, q_2) in $\mathbf{G}$? (II) If so, what would be the destination state that σ would take $\mathbf{G}$ to? These three rules defined by $||_{SD}$ to construct δ are elaborated next.

i) Point i applies to events that $\mathbf{G}_1$ and $\mathbf{G}_2$ have in common. This point makes a distinction between the *tick* and non-*tick* (activity) events, and specifies two different rules for defining the *tick* and activity event transitions in $\mathbf{G}$.

a) $\sigma \neq \tau$

For an activity event σ, a transition will be defined at a state in $\mathbf{G}$ if it is defined at the corresponding states in both $\mathbf{G}_1$ and $\mathbf{G}_2$. This means that $\mathbf{G}_1$ and $\mathbf{G}_2$ act together to cooperatively determine and agree on the definition of σ transition, and its corresponding destination state in $\mathbf{G}$. This is essentially the same logic that synchronous product uses to determine its transitions.

It is important to clarify here that the $||_{SD}$ operator is not capable of adding any non-*tick* transition to δ if it does not exist in either δ_1 or δ_2 or both. Likewise,

$||_{SD}$ cannot remove a non-*tick* transition from δ if it is defined in both δ_1 and δ_2.

b) $\sigma = \tau$
This is the case where the logic of $||_{SD}$ operator differs from the standard synchronous product, i.e. the case of figuring out the definition of *tick* transitions in $\mathbf{G}$. This point says that a *tick* transition will be defined at a state in $\mathbf{G}$ if the following conditions are satisfied:

I) *tick* transition is defined at the corresponding states in both $\mathbf{G}_1$ and $\mathbf{G}_2$.

II) No prohibitable event is possible at the current state in $\mathbf{G}$.

Point I signifies that $||_{SD}$ will not define a *tick* transition in $\mathbf{G}$ if it is blocked by either $\mathbf{G}_1$ or $\mathbf{G}_2$ or both. This means that our synchronization operator is not capable of adding any *tick* transition to δ on its own. It will 'potentially' add a *tick* transition to δ only if it exists in both δ_1 and δ_2.

However, **Point II** imposes an important condition, which if not satisfied, then $||_{SD}$ operator is capable of deciding "not" to add a *tick* transition to δ, even if is defined in both δ_1 and δ_2. In this case, $||_{SD}$ is smart enough to automatically "disable" a *tick* event if a prohibitable event is currently possible in $\mathbf{G}$.

This means that if *tick* is defined in $\mathbf{G}_1$ and $\mathbf{G}_2$, $||_{SD}$ will not immediately add this *tick* transition to $\mathbf{G}$. First, it will figure out whether or not any prohibitable event σ' is currently possible in $\mathbf{G}$. To determine this, $||_{SD}$ evaluates the transitions for all prohibitable events one by one at the current state in $\mathbf{G}$. Depending upon whether σ' is in $(\Sigma_1 \cap \Sigma_2)$, $(\Sigma_1 - \Sigma_2)$, or $(\Sigma_2 - \Sigma_1)$, the $||_{SD}$ operator will recursively make use of Points i ($\sigma \neq \tau$), ii or iii respectively to figure out if σ' is defined at the corresponding states in both $\mathbf{G}_1$ and $\mathbf{G}_2$.

If any prohibitable event transition is possible at the current state in $\mathbf{G}$, **Point II** fails, and $||_{SD}$ will "not" add *tick* transition to $\mathbf{G}$. In this way, the $||_{SD}$ operator disables the *tick* event to automatically satisfy Point ii ($\Rightarrow$) of the SD controllability definition (Definition 3.5.1) at every state of $\mathbf{G}$. On the other hand, if none of the prohibitable events is currently possible in $\mathbf{G}$, **Point II** is satisfied, and $||_{SD}$ will define a *tick* transition in $\mathbf{G}$, given that *tick* is currently possible in both $\mathbf{G}_1$ and $\mathbf{G}_2$.

Please note that since the $||_{SD}$ operator only deals with TDES models, *tick* event will certainly be present in both $\mathbf{G}_1$ and $\mathbf{G}_2$. Thus, $||_{SD}$ will always use this point (and never Point i ($\sigma \neq \tau$), Point ii or Point iii) to determine the definition of *tick* transition and its corresponding next state in $\mathbf{G}$.

Points ii and **iii** are applicable to events that are present in the event set of only one TDES, $\mathbf{G}_1$ or $\mathbf{G}_2$, respectively. These points of the SD synchronous product's definition are identical to the synchronous product's definition.

ii) If an event σ is only in the event set of $\mathbf{G}_1$, then $||_{SD}$ will use **Point ii** to determine the definition and next state of σ transition in $\mathbf{G}$. At a given state, σ will be allowed to occur in $\mathbf{G}$ if it is possible at the corresponding state in $\mathbf{G}_1$. As $\mathbf{G}_2$

does not care about σ, it can neither prevent σ from occurring in $\mathbf{G}$, nor it will change its state as a result of this σ transition.

iii) **Point iii** applies to an event σ that is present only in the event set of $\mathbf{G}_2$. This point says that σ transition will be defined at a state in $\mathbf{G}$ if it is possible at the corresponding state in $\mathbf{G}_2$. $\mathbf{G}_1$ is not related to σ in any way, therefore it cannot block σ transition in $\mathbf{G}$. Also, the occurrence of σ transition will not have any affect on $\mathbf{G}_1$'s current state.

Example

Figure 4.1 shows an example of the $||_{SD}$ operator, and compares its synchronization mechanism to that of the synchronous product operator. In the example, we have two TDES, $\mathbf{G}_1$ (Figure 4.1a) and $\mathbf{G}_2$ (Figure 4.1b), that are defined over the same event set Σ, such that $\Sigma = \{e1, e2, \tau\}$, $\Sigma_{hib} = \{e1\}$ and $\Sigma_u = \{e2\}$. At the initial state, q0 of $\mathbf{G}_1$ and x0 of $\mathbf{G}_2$, both *tick* and prohibitable event e1 are defined.

If we construct $\mathbf{G}' = \mathbf{G}_1 \,||\, \mathbf{G}_2$, the synchronous product operator enables both *tick* and prohibitable event e1 at the initial state of $\mathbf{G}'$, as shown in Figure 4.1c.

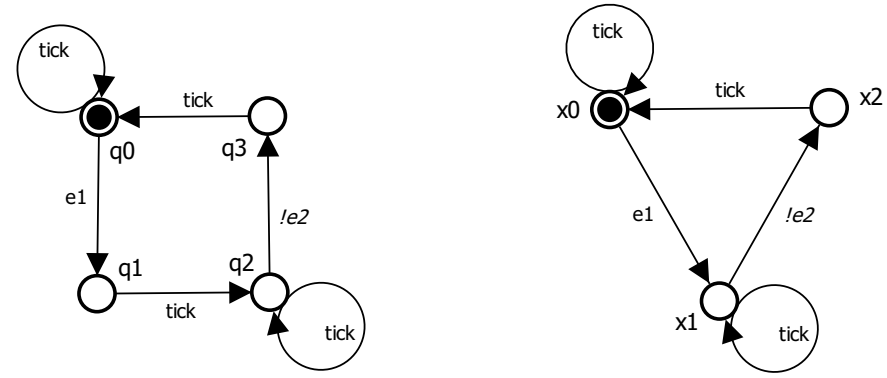
Figure 4.1d illustrates the result of synchronizing $\mathbf{G}_1$ and $\mathbf{G}_2$ using the $||_{SD}$ operator. For $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$, we note that *tick* transition is not defined at the initial state s0, although it is defined at the initial states of $\mathbf{G}_1$ and $\mathbf{G}_2$. The reason is that both $\mathbf{G}_1$ and $\mathbf{G}_2$ have enabled prohibitable event e1 at their initial states. Therefore, the $||_{SD}$ operator enables prohibitable event e1 at the initial state of $\mathbf{G}$ and disables the *tick* event, as desired to satisfy Point ii ($\Rightarrow$) of the SD controllability property.

We also note that this is the only difference between $\mathbf{G}'$ and $\mathbf{G}$, indicating that the rest of the synchronization mechanism of the $||_{SD}$ operator is essentially the same as the synchronous product.

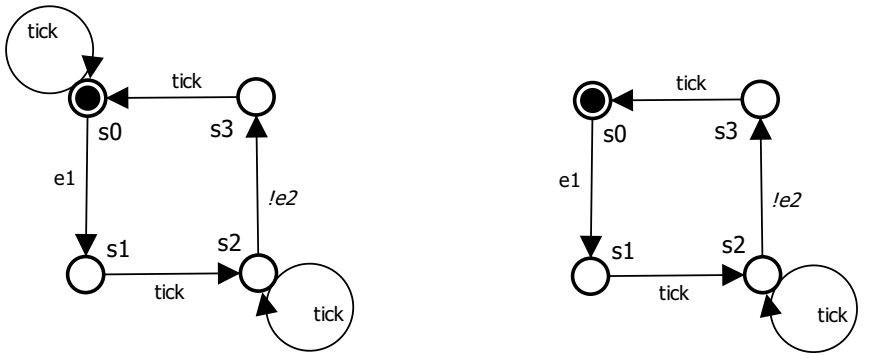
4.2 Properties of SD Synchronous Product Operator

In this section, we discuss and prove some fundamental properties of our SD synchronous product operator. We will start by showing that when we synchronize two TDES automata using the $||_{SD}$ operator, this will result in the generation of a model that is also a TDES automaton.

For any synchronization operator, the two key properties of interest are *commutativity* and *associativity*. We will also examine our $||_{SD}$ operator with respect to these properties. Precisely, we demonstrate that the $||_{SD}$ operator is commutative, but not associative. These results will later help us in defining our strategy of constructing the closed-loop system using the $||_{SD}$ operator in our setting, described in Section 4.3.



(a) TDES $\mathbf{G}_1$ (b) TDES $\mathbf{G}_2$



(c) $\mathbf{G}' = \mathbf{G}_1 \,||\, \mathbf{G}_2$ (d) $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$

Figure 4.1: An Example of SD Synchronous Product Operator

4.2.1 SD Synchronous Product Defines a TDES

As we have defined a new synchronization operator to combine two TDES automata, it is important to show that the resultant model is also a TDES automaton with all of its elements being well defined. We formally prove this in the following proposition. As the SD synchronous product operator is an adapted version of the standard synchronous product, we will base our argument on the fact that the model generated by the synchronous product operator has these properties.

Proposition 4.1. Let $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2, \delta_2, q_{o,2}, Q_{m,2})$ be two TDES. The SD synchronous product of $\mathbf{G}_1$ and $\mathbf{G}_2$, represented as $\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$, defines a TDES automaton.

Proof. The SD synchronous product defines $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$ as a quintuple:

$$\mathbf{G} := (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$$

By Definition 2.3.1, a TDES automaton is formally represented as a quintuple $(Q, \Sigma, \delta, q_o, Q_m)$.

In order to prove that $\mathbf{G}$ is a TDES automaton, it is sufficient to show that $\mathbf{G}$'s tuple is comprised of the five standard elements of a TDES automaton's tuple.

By looking at Definition 4.1.1 of the $||_{SD}$ operator, it is obvious that the tuple elements of Q, Σ, q_o and Q_m are defined by the $||_{SD}$ operator in exactly the same way as the synchronous product. Clearly these elements of **G** are well defined, as we know that the synchronous product operator is well defined.

Below, we analyze the transition function δ to show that δ defined by $||_{SD}$ is well defined. We will base our argument on the fact that the transition function defined by the synchronous product is well defined.

In order to show that δ is well defined, we need to show that δ unambiguously determines: **(I)** if $\sigma \in (\Sigma_1 \cup \Sigma_2)$ transition would be defined at state $(q_1, q_2) \in Q_1 \times Q_2$ in **G**? **(II)** what would be the destination state for each σ transition that would be defined in **G**?

I) By looking at the definition of δ in the SD synchronous product's definition, we note that **Point ii** and **Point iii** are identical to the synchronous product's transition function. As the synchronous product's transition function is well defined, we deduce that Point ii and Point iii construct δ in a well defined way.

The only rule that makes δ different from the synchronous product's transition function is **Point i**. Therefore, it is sufficient to show that Point i constructs δ in a well defined way.

Depending upon whether an event σ is a *tick* or a non-*tick* (activity) event, Point i specifies two different rules for determining whether or not σ transition would be defined at state (q_1, q_2) in **G**. Thus, we have two cases: **(a)** $\sigma \neq \tau$, and **(b)** $\sigma = \tau$.

In order to show that Point i constructs δ in a well defined, we need to show that δ is constructed in a well defined way in both cases.

Case a) $\sigma \neq \tau$
For an activity event σ, it is decided whether or not $\delta((q_1, q_2), \sigma)!$ in **G**, by evaluating whether or not $\delta_1(q_1, \sigma)!$ and $\delta_2(q_2, \sigma)!$.

This is the same logic that is used by the synchronous product's transition function to figure out its transitions for shared events while synchronizing two TDES models.

As the transition function of synchronous product is well defined, we conclude that for each $\sigma \neq \tau$, δ is well defined in the way it decides whether or not $\delta((q_1, q_2), \sigma)!$ in **G**.

Case (a) complete.

Case b) $\sigma = \tau$
In order to decide whether or not $\delta((q_1, q_2), \tau)!$ in **G**, it is evaluated whether or not: **(1)** $\delta_1(q_1, \tau)!$, **(2)** $\delta_2(q_2, \tau)!$, and **(3)** $(\forall \sigma' \in \Sigma_{hib})\, \delta((q_1, q_2), \sigma')!$.

The process of determining if $\delta_1(q_1, \tau)!$ in $\mathbf{G}_1$ and $\delta_2(q_2, \tau)!$ in $\mathbf{G}_2$ is straightforward and will always give a unique result without any ambiguity, since δ_1 and δ_2 are individually well defined.

In order to figure out whether or not, for all $\sigma' \in \Sigma_{hib}, \delta((q_1, q_2), \sigma')!$, it is examined whether or not for each individual σ', $\delta((q_1, q_2), \sigma')!$.

For individual σ', depending upon whether $\sigma' \in (\Sigma_1 \cap \Sigma_2)$, $\sigma' \in (\Sigma_1 - \Sigma_2)$ or $\sigma' \in (\Sigma_2 - \Sigma_1)$, Point i ($\sigma \neq \tau$), Point ii or Point iii will respectively be used to determine whether or not $\delta_1(q_1, \sigma')!$ and/or $\delta_2(q_2, \sigma')!$. Since we have already shown that Point i ($\sigma \neq \tau$), Point ii and Point iii construct δ in a well defined way, we infer that for each individual $\sigma' \in \Sigma_{hib}$, the process of determining whether or not $\delta((q_1, q_2), \sigma')!$ is well defined.

This implies that the overall process of determining whether or not, for all $\sigma' \in \Sigma_{hib}, \delta((q_1, q_2), \sigma')!$ will always give a unique result without any ambiguity.

Hence, we conclude that the overall decision process of δ to determine whether or not $\delta((q_1, q_2), \tau)!$ in $\mathbf{G}$ is well defined.

Case (b) complete.

By Cases (a) and (b), we conclude that **Point i** constructs δ in a well defined way.

As Points (i-iii) of the $||_{SD}$ operator construct δ in a well defined way, hence we conclude that δ is well defined in the way it determines if σ transition would be defined at state (q_1, q_2) in $\mathbf{G}$.

Part (I) complete.

II) By looking at the definition of δ in $||_{SD}$, we note that δ uses the same strategy as the synchronous product's transition function to determine the destination state of each σ transition that would be defined in $\mathbf{G}$. Since the transition function of synchronous product is well defined, we deduce that δ is also well defined in this perspective.

Part (II) complete.

By Parts (I) and (II), we conclude that the transition function δ, defined by $||_{SD}$, is well defined.

We have thus shown that $\mathbf{G}$'s quintuple defined by $||_{SD}$ comprises of five standard elements of a TDES automaton's tuple and all these elements are well defined.

Hence, we conclude that $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$ defines a TDES automaton. □

4.2.2 Commutative Property

The SD synchronous product operator is commutative up to isomorphism, i.e. $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$ and $\mathbf{G}_2 ||_{SD} \mathbf{G}_1$ will give us the same resultant TDES automaton up to relabelling of state components in the composed states. More formally, TDES $\mathbf{G}$ and $\mathbf{G}'$ are isomorphic up to state relabelling if we can define a bijective function[1] that maps $\mathbf{G}$ to $\mathbf{G}'$. Our next proposition formally proves this concept and property.

Proposition 4.2. Let $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2, \delta_2, q_{o,2}, Q_{m,2})$

[1] See Definition *A*.5.1 of *bijective function* in Appendix A.

be two TDES. The SD synchronous product of $\mathbf{G}_1$ and $\mathbf{G}_2$ is commutative up to isomorphism.

Proof. Let $\mathbf{G}$ be a TDES constructed as $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$, and let $\mathbf{G}'$ be a TDES constructed as $\mathbf{G}' = \mathbf{G}_2 \,||_{SD}\, \mathbf{G}_1$.

The SD synchronous product operator defines $\mathbf{G}$ and $\mathbf{G}'$ as follows:

$\mathbf{G} := (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$

$\mathbf{G}' := (Q_2 \times Q_1, \Sigma_2 \cup \Sigma_1, \delta', (q_{o,2}, q_{o,1}), Q_{m,2} \times Q_{m,1})$

First, we note that the $||_{SD}$ operator defines the event sets of $\mathbf{G}$ and $\mathbf{G}'$ as $\Sigma_1 \cup \Sigma_2$ and $\Sigma_2 \cup \Sigma_1$ respectively.

The *commutative property for set union* says the order of sets in which we do the union operation does not change the result. This means taking the union of sets Σ_1 and Σ_2 in either order will give the same resulting set, i.e. $\Sigma = \Sigma_1 \cup \Sigma_2 = \Sigma_2 \cup \Sigma_1$.

This implies that both $\mathbf{G}$ and $\mathbf{G}'$ are defined over the same event set Σ. Therefore, the quintuples of TDES automata $\mathbf{G}$ and $\mathbf{G}'$ can be restated as follows:

$\mathbf{G} := (Q_1 \times Q_2, \Sigma, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$

$\mathbf{G}' := (Q_2 \times Q_1, \Sigma, \delta', (q_{o,2}, q_{o,1}), Q_{m,2} \times Q_{m,1})$

In order to show that the SD synchronous product of $\mathbf{G}_1$ and $\mathbf{G}_2$ is commutative up to isomorphism, it is sufficient to show that $\mathbf{G}$ and $\mathbf{G}'$ are isomorphic up to state relabelling.

By definition, $\mathbf{G}$ and $\mathbf{G}'$ are said to be isomorphic by states if there exists an isomorphic function, $\boldsymbol{iso}$, that maps $\mathbf{G}$ to $\mathbf{G}'$ while preserving all automata-theoretic structure of $\mathbf{G}$ and $\mathbf{G}'$, as defined by $||_{SD}$, up to relabelling of states.

We will show this first by defining a function $\boldsymbol{iso}$, and proving that $\boldsymbol{iso}$ is indeed an isomorphic map. Then we will show that $\boldsymbol{iso}$ maps $\mathbf{G}$ to $\mathbf{G}'$ while preserving all automata-theoretic structure of $\mathbf{G}$ and $\mathbf{G}'$ up to state relabelling.

First, we will define and construct our function $\boldsymbol{iso}$.

We define $\boldsymbol{iso}$ as: $\boldsymbol{iso} : \mathbf{G} \to \mathbf{G}'$

Our goal is to define $\boldsymbol{iso}$ so we achieve the following result:

$\boldsymbol{iso}((Q_1 \times Q_2, \Sigma, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})) = (Q_2 \times Q_1, \Sigma, \delta', (q_{o,2}, q_{o,1}), Q_{m,2} \times Q_{m,1})$

To construct our function $\boldsymbol{iso}$, we define two functions: **(i)** iso_Q, and **(ii)** id_Σ.

i) $iso_Q : Q_1 \times Q_2 \to Q_2 \times Q_1 : (q_1, q_2) \mapsto (q_2, q_1)$

The function iso_Q is defined to map the state set of $\mathbf{G}$ to the state set of $\mathbf{G}'$. Specifically, it takes a given state of $\mathbf{G}$ and maps it to its corresponding state in $\mathbf{G}'$ by swapping the elements of $\mathbf{G}$'s state tuple.

$$(\forall (q_1, q_2) \in Q_1 \times Q_2)\ iso_Q\,((q_1, q_2)) = (q_2, q_1)$$

Clearly, iso_Q is bijective as $Q_1 \times Q_2$ and $Q_2 \times Q_1$ are the same size, and:

$$(\forall (q_2, q_1) \in Q_2 \times Q_1)\, iso_Q^{-1}((q_2, q_1)) = (q_1, q_2)$$

ii) $id_\Sigma : \Sigma \to \Sigma : \sigma \mapsto \sigma$

id_Σ is defined as an identity function on Σ. Since both $\mathbf{G}$ and $\mathbf{G}'$ are defined over the same event set Σ, this function maps the event set of $\mathbf{G}$ to the event set of $\mathbf{G}'$ by mapping event σ to itself.

$$(\forall \sigma \in \Sigma)\ id_\Sigma(\sigma) = \sigma$$

Clearly, id_Σ is bijective as it is an identity function.

Using these two functions, we can map each element of $\mathbf{G}$'s quintuple to its corresponding element in $\mathbf{G}'$'s quintuple, as elaborated next.

We first note that for function $f : X \to Y$, we can define for $A \subseteq X$,

$$f(A) = \{f(x) \mid x \in A\}$$

1) State Set

We will use $iso_Q(Q_1 \times Q_2) = \{iso_Q((q_1, q_2)) \mid (q_1, q_2) \in Q_1 \times Q_2\}$.

As iso_Q is bijective, $iso_Q(Q_1 \times Q_2) = Q_2 \times Q_1$.

2) Event Set

We will use $id_\Sigma(\Sigma) = \{id_\Sigma(\sigma) \mid \sigma \in \Sigma\}$. Clearly, $id_\Sigma(\Sigma) = \Sigma$.

3) Transition Function

In order to clearly argue about the preservation of transitions of $\mathbf{G}$ and $\mathbf{G}'$ later in the proof, we will express our transitions as a 3-tuple. The transitions are represented as a triple of the form $(q, \sigma, q') \subseteq Q \times \Sigma \times Q$, where $\delta(q, \sigma) = q'$. As such, $\delta \subseteq (Q_1 \times Q_2) \times \Sigma \times (Q_1 \times Q_2)$ and $\delta' \subseteq (Q_2 \times Q_1) \times \Sigma \times (Q_2 \times Q_1)$.

To convert δ, we will use:

$$iso_Q \times id_\Sigma \times iso_Q(\delta) = \{iso_Q \times id_\Sigma \times iso_Q(((q_1, q_2), \sigma, (q_1', q_2'))) \mid ((q_1, q_2), \sigma, (q_1', q_2')) \in \delta\}$$

We will still need to show that this produces δ'. As iso_Q and id_Σ are bijective functions, their cross product will also be bijective. As $\mathbf{G}_1$ and $\mathbf{G}_2$ are arbitrary TDES, showing that the above produces δ', is thus sufficient to prove the inverse function applied to δ' will produce δ.

4) Initial State

We will use $iso_Q((q_{o,1}, q_{o,2})) = (q_{o,2}, q_{o,1})$.

Clearly, this is a bijective process as $iso_Q^{-1}((q_{o,2}, q_{o,1})) = (q_{o,1}, q_{o,2})$.

5) Set of Marked States

We will use $iso_Q(Q_{m,1} \times Q_{m,2}) = \{iso_Q((q_1, q_2)) \mid (q_1, q_2) \in Q_{m,1} \times Q_{m,2}\}$.

As iso_Q is bijective, $iso_Q(Q_{m,1} \times Q_{m,2}) = Q_{m,2} \times Q_{m,1}$.

To map $\mathbf{G}$ to $\mathbf{G}'$, we combine the above mappings, and we can express our function

iso as follows:

$$iso\left((Q_1 \times Q_2, \Sigma, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})\right) =$$
$$(iso_Q(Q_1 \times Q_2), id_\Sigma(\Sigma), iso_Q \times id_\Sigma \times iso_Q(\delta), iso_Q(q_{o,1}, q_{o,2}), iso_Q(Q_{m,1} \times Q_{m,2})) =$$
$$(Q_2 \times Q_1, \Sigma, iso_Q \times id_\Sigma \times iso_Q(\delta), (q_{o,2}, q_{o,1}), Q_{m,2} \times Q_{m,1})$$

From the above discussion, it is clear that except for δ, every part of the conversion correctly maps each remaining component of $\mathbf{G}$ onto the corresponding component of $\mathbf{G}'$, and in a bijective manner, i.e. applying the mapping in reverse will map these components of $\mathbf{G}'$ to the corresponding components of $\mathbf{G}$.

Now all that remains is to show that $iso_Q \times id_\Sigma \times iso_Q(\delta) = \delta'$.

The $||_{SD}$ operator defines the transition function δ of $\mathbf{G}$ and δ' of $\mathbf{G}'$ as follows:

$\delta((q_1, q_2), \sigma)$ is only defined and equals:

i) (q_1', q_2') if $\sigma \in (\Sigma_1 \cap \Sigma_2) \ \wedge \ \delta_1(q_1, \sigma) = q_1' \ \wedge \ \delta_2(q_2, \sigma) = q_2' \ \wedge$
$[(\sigma \neq \tau) \ \vee \ ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib}) \neg \delta((q_1, q_2), \sigma')!)]$

ii) (q_1', q_2) if $\sigma \in (\Sigma_1 - \Sigma_2) \ \wedge \ \delta_1(q_1, \sigma) = q_1'$

iii) (q_1, q_2') if $\sigma \in (\Sigma_2 - \Sigma_1) \ \wedge \ \delta_2(q_2, \sigma) = q_2'$

$\delta'((q_2, q_1), \sigma)$ is only defined and equals:

i) (q_2', q_1') if $\sigma \in (\Sigma_2 \cap \Sigma_1) \ \wedge \ \delta_1(q_1, \sigma) = q_1' \ \wedge \ \delta_2(q_2, \sigma) = q_2' \ \wedge$
$[(\sigma \neq \tau) \ \vee \ ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib}) \neg \delta'((q_2, q_1), \sigma')!)]$

ii) (q_2, q_1') if $\sigma \in (\Sigma_1 - \Sigma_2) \ \wedge \ \delta_1(q_1, \sigma) = q_1'$

iii) (q_2', q_1) if $\sigma \in (\Sigma_2 - \Sigma_1) \ \wedge \ \delta_2(q_2, \sigma) = q_2'$

By examining and comparing the definitions of δ and δ', we note that the rules specified by δ and δ' are logically identical, i.e. they specify the same logic, in terms of δ_1 of $\mathbf{G}_1$ and δ_2 of $\mathbf{G}_2$, to make decisions about defining transitions and determining next states.

The only difference is the way δ and δ' label the exit and entrance states of their transitions while specifying their rules. Precisely, if we swap the elements in exit and entrance states' tuples in the rules defined by δ, we essentially get the corresponding rules defined by δ'. This means the definitions of δ and δ' are identical up to reordering of elements in the tuples of their exit and entrance states respectively.

This implies that $\mathbf{G}$ and $\mathbf{G}'$, constructed by $||_{SD}$, essentially have the same set of defined transitions, up to relabelling of their exit and entrance states respectively.

Our isomorphic function ***iso*** uses iso_Q and id_Σ to map the transition triples defined in $\mathbf{G}$ to their corresponding transition triples in $\mathbf{G}'$ as follows.

$$(iso_Q\left((q_1, q_2)\right), id_\Sigma\left(\sigma\right), iso_Q\left((q_1', q_2')\right)) = ((q_2, q_1), \sigma, (q_2', q_1')),$$
$$\text{where } \ (q_1, q_2), (q_1', q_2') \in Q_1 \times Q_2 \ \text{ and } \ \sigma \in \Sigma$$

It is noticeable that iso_Q maps the exit and entrance states of a given transition in $\mathbf{G}$ to the respective exit and entrance states of its corresponding transition in $\mathbf{G}'$ by swapping the elements individually in exit and entrance states' tuples.

Since $\mathbf{G}$ and $\mathbf{G}'$ are defined over the same event set Σ, id_Σ preserves the identity of an event $\sigma \in \Sigma$ by mapping σ of $\mathbf{G}$ to σ of $\mathbf{G}'$.

This makes it evident that ***iso*** maps the transition triple of $\mathbf{G}$ to its corresponding transition triple in $\mathbf{G}'$ by relabelling the exit and entrance states, and preserving the identity of the event, i.e. if σ transition takes $\mathbf{G}$ from state (q_1, q_2) to (q'_1, q'_2), ***iso*** maps it to its corresponding σ transition that takes $\mathbf{G}'$ from state (q_2, q_1) to (q'_2, q'_1).

As $\mathbf{G}$ starts at $(q_{o,1}, q_{o,2})$ and $\mathbf{G}'$ at $iso_Q(q_{o,1}, q_{o,2}) = (q_{o,2}, q_{o,1})$, then for any $\sigma \in \Sigma$ such that $\delta((q_{o,1}, q_{o,2}), \sigma) = (q'_1, q'_2)$, it will also be true that $\delta'((q_{o,2}, q_{o,1}), \sigma) = (q'_2, q'_1) = iso_Q((q'_1, q'_2))$.

This means that all transitions leaving the initial state of $\mathbf{G}$ will have a matching isomorphic transition leaving the initial state of $\mathbf{G}'$. It is easy to see that all states reached from the initial state of $\mathbf{G}$ will have an isomorphic state reached from the initial state of $\mathbf{G}'$.

Following this to the logical conclusion, any state reachable in $\mathbf{G}$ will have an isomorphic state reachable in $\mathbf{G}'$. Also, at each reachable state (q_1, q_2) in $\mathbf{G}$, the set of transitions leaving (q_1, q_2) will be isomorphic to the set of transitions leaving state (q_2, q_1) in $\mathbf{G}'$.

Hence, we conclude that ***iso*** preserves the structure of the transition function of $\mathbf{G}$ and $\mathbf{G}'$ up to relabelling of exit and entrance states in the defined transitions. In other words, $iso_Q \times id_\Sigma \times iso_Q(\delta) = \delta'$.

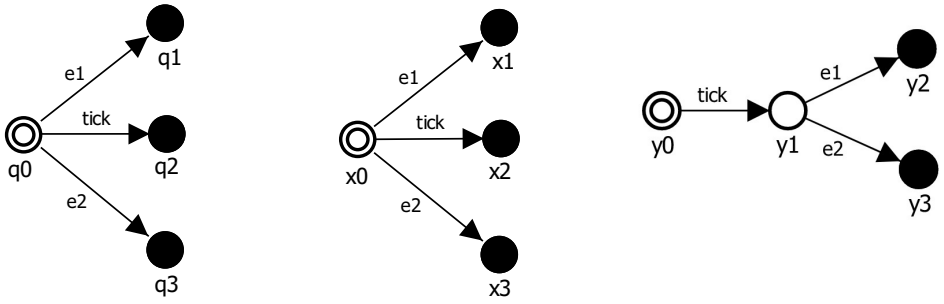
We have thus shown that by using two bijective functions, iso_Q and id_Σ, our isomorphic function ***iso*** maps each individual element of $\mathbf{G}$'s quintuple to its corresponding element in $\mathbf{G}'$'s quintuple while preserving its original structure, as defined by $||_{SD}$, up to relabelling of states. Hence, we conclude that ***iso*** preserves all automata-theoretic structure of $\mathbf{G}$ and $\mathbf{G}'$ up to state relabelling.

In this way, by constructing our desired isomorphic function, ***iso***, we have shown that $\mathbf{G}$ and $\mathbf{G}'$ are isomorphic up to state relabelling.

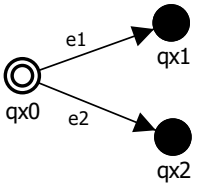
Hence, we conclude that the SD synchronous product of $\mathbf{G}_1$ and $\mathbf{G}_2$ is commutative up to isomorphism. □

4.2.3 Non-Associative Property

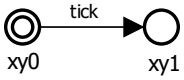
The SD synchronous product operator is inherently non-associative, i.e. the order of synchronizing three or more TDES automata using $||_{SD}$ is important and might make a difference in the resultant TDES. In other words, if we have three TDES automata, $\mathbf{G}_1, \mathbf{G}_2$ and $\mathbf{G}_3$, then in general $(\mathbf{G}_1 ||_{SD} \mathbf{G}_2) ||_{SD} \mathbf{G}_3 \neq \mathbf{G}_1 ||_{SD} (\mathbf{G}_2 ||_{SD} \mathbf{G}_3)$. Below, we demonstrate it with the help of an example.

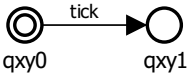


(a) TDES $\mathbf{G}_1$ (b) TDES $\mathbf{G}_2$ (c) TDES $\mathbf{G}_3$



(d) $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$ (e) $\mathbf{G} = (\mathbf{G}_1 ||_{SD} \mathbf{G}_2) ||_{SD} \mathbf{G}_3$





(f) $\mathbf{G}_2 ||_{SD} \mathbf{G}_3$ (g) $\mathbf{G}' = \mathbf{G}_1 ||_{SD} (\mathbf{G}_2 ||_{SD} \mathbf{G}_3)$

Figure 4.2: SD Synchronous Product Operator is Non-Associative

Figure 4.2 illustrates the non-associative nature of the $||_{SD}$ operator using three TDES automata, $\mathbf{G}_1$ (Figure 4.2a), $\mathbf{G}_2$ (Figure 4.2b) and $\mathbf{G}_3$ (Figure 4.2c). All TDES are defined over the same event set Σ, such that $\Sigma = \{\text{e1, e2, } \tau\}$ and $\Sigma_{hib} = \{\text{e1, e2}\}$.

First, let us discuss the synchronization mechanism of $||_{SD}$ for constructing TDES $\mathbf{G}$ as $\mathbf{G} = (\mathbf{G}_1 ||_{SD} \mathbf{G}_2) ||_{SD} \mathbf{G}_3$. Figure 4.2d shows the result of synchronizing $\mathbf{G}_1$ and $\mathbf{G}_2$ using $||_{SD}$. As two prohibitable events, e1 and e2, are enabled at the initial states of $\mathbf{G}_1$ and $\mathbf{G}_2$, $||_{SD}$ disables *tick* event at the initial state of $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$. Thus, the only events possible at the initial state of $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$ are e1 and e2.

In order to construct $\mathbf{G}$, we synchronize $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$ with $\mathbf{G}_3$ using $||_{SD}$. We see in Figure 4.2e that no events are possible at the initial state of $\mathbf{G}$. This is because prohibitable events e1 and e2 that are possible at the initial state of $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$ have been blocked by $\mathbf{G}_3$ at its initial state. Likewise, *tick* event is possible in $\mathbf{G}_3$ but not in $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$. This is because the *tick* event, that was originally possible in both $\mathbf{G}_1$ and $\mathbf{G}_2$, has already been disabled by $||_{SD}$ while constructing $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$.

Now we will change the order of synchronizing our three TDES, and construct TDES $\mathbf{G}'$ as $\mathbf{G}' = \mathbf{G}_1 ||_{SD} (\mathbf{G}_2 ||_{SD} \mathbf{G}_3)$. At the initial state, prohibitable events e1 and e2 are possible in $\mathbf{G}_2$ but not in $\mathbf{G}_3$. Thus, $||_{SD}$ does not enable these events at the initial state of $\mathbf{G}_2 ||_{SD} \mathbf{G}_3$. As *tick* is possible in both $\mathbf{G}_2$ and $\mathbf{G}_3$, and no prohibitable event is possible in $\mathbf{G}_2 ||_{SD} \mathbf{G}_3$, $||_{SD}$ defines a *tick* transition at the initial state of $\mathbf{G}_2 ||_{SD} \mathbf{G}_3$, as shown in Figure 4.2f.

To construct $\mathbf{G}'$, we now synchronize $\mathbf{G}_1$ with $\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3$ using $||_{SD}$. $\mathbf{G}_1$ enables prohibitable events e1 and e2 at its initial state, but since these events are not possible in $\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3$, $||_{SD}$ does not add their transitions at the initial state of $\mathbf{G}'$. Given that *tick* is possible in both $\mathbf{G}_1$ and $\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3$, and no prohibitable event is currently possible in $\mathbf{G}'$, $||_{SD}$ enables *tick* at the initial state of $\mathbf{G}'$, as shown in Figure 4.2g.

By comparing our $\mathbf{G}$ and $\mathbf{G}'$, we note that $\mathbf{G} \neq \mathbf{G}'$. This example clearly demonstrates that the order of synchronizing three TDES using $||_{SD}$ does matter, and we might get different resultant TDES. Hence, we deduce that the $||_{SD}$ operator is inherently non-associative.

4.3 SD Synchronous Product Setting

In our SD synchronous product setting (or "$||_{SD}$ *setting,*" for short), we will use the SD synchronous product operator to combine our TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$. Hence, our closed-loop system is $\mathbf{S} \,||_{SD}\, \mathbf{G}$. Due to the commutative property of the $||_{SD}$ operator, we can synchronize $\mathbf{G}$ and $\mathbf{S}$ in either order, i.e. $\mathbf{G} \,||_{SD}\, \mathbf{S} = \mathbf{S} \,||_{SD}\, \mathbf{G}$.

Note: For consistency, we will always write our closed-loop system as $\mathbf{S} \,||_{SD}\, \mathbf{G}$.

In our $||_{SD}$ setting, we also assume that both $\mathbf{G}$ and $\mathbf{S}$ are defined over the same event set. In case where $\mathbf{G}$ and $\mathbf{S}$ are not defined over the same alphabet, we can simply add selfloops to each TDES for the missing events at every state to extend them over the same event set, without any loss of generality.

In the real world, software designers typically design $\mathbf{G}$ and $\mathbf{S}$ in a modular fashion, rather than as monolithic models. In this case, we assume that these modular plant and supervisor models will be independently synchronized using the standard synchronous product operator to obtain $\mathbf{G}$ and $\mathbf{S}$ respectively. For $m > 1$ plant components, $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_m$, our $\mathbf{G}$ will be obtained as $\mathbf{G} = \mathbf{G}_1 \,||\, \mathbf{G}_2 \,||\, \ldots \,||\, \mathbf{G}_m$. Similarly, for $n > 1$ modular supervisors, $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n$, our $\mathbf{S}$ will be constructed as $\mathbf{S} = \mathbf{S}_1 \,||\, \mathbf{S}_2 \,||\, \ldots \,||\, \mathbf{S}_n$.

There are two reasons for *not* using the $||_{SD}$ operator to combine individual plant and supervisor components to construct $\mathbf{G}$ and $\mathbf{S}$ respectively. The primary reason is the non-associative nature of the $||_{SD}$ operator due to which the order of combining various plant (or supervisor) components becomes important, and different synchronization order will potentially give us a different $\mathbf{G}$ (or $\mathbf{S}$). Moreover, it might also cause our closed-loop system to block, as no events remain possible in TDES $\mathbf{G}$ (Figure 4.2e) of the example discussed in Section 4.2.3.

Secondly, applying the $||_{SD}$ operator either to plant or supervisor models individually does not look practical and reasonable. The key characteristic of $||_{SD}$ is to automatically disable a *tick* event in the resultant model in cases where source models *agree* on the enablement of one or more prohibitable events. Strictly speaking, there is no concept of enablement/disablement of *tick* event and forcing of prohibitable events

solely with respect to either plant or supervisor. Plant model just represents the behaviour of the physical system without any restrictions and constraints. A supervisor is designed to impose control action on the plant model by operating synchronously with it. Hence, it is not justifiable to combine either plant or supervisor components independently using $||_{SD}$, and let only one model decide about the disablement of *tick* event without having any knowledge of the other model's behaviour.

Considering the non-associative property of the $||_{SD}$ operator, it is also evident that once we have formed the closed-loop system using $||_{SD}$, we cannot add any new plant or supervisor component directly to $\mathbf{S}\,||_{SD}\,\mathbf{G}$. This might give us unexpected and problematic results. After constructing $\mathbf{S}\,||_{SD}\,\mathbf{G}$, if we want to add more plant or supervisor models, we need to form our closed-loop system again. We should first reconstruct our $\mathbf{G}$ and $\mathbf{S}$ separately using the synchronous product, and then combine $\mathbf{G}$ and $\mathbf{S}$ to obtain our closed-loop system $\mathbf{S}\,||_{SD}\,\mathbf{G}$.

However, one possible way to use the $||_{SD}$ operator to combine plant and supervisor components is to synchronize all system models in parallel. In this case, instead of constructing $\mathbf{G}$, $\mathbf{S}$, and $\mathbf{S}\,||_{SD}\,\mathbf{G}$ sequentially, we will synchronize m plant components and n modular supervisors using $||_{SD}$ all at once to construct our closed-loop system. Hence, our closed-loop system will be $\mathbf{G}_1\,||_{SD}\,\mathbf{G}_2\,||_{SD}\,\ldots||_{SD}\,\mathbf{G}_m\,||_{SD}\,\mathbf{S}_1\,||_{SD}\,\mathbf{S}_2\,||_{SD}\,\ldots||_{SD}\,\mathbf{S}_n$. In this way, the non-associative nature of the $||_{SD}$ operator can be circumvented.

4.4 SD Properties with SD Synchronous Product

TDES and SD properties discussed in the SD setting (Chapter 3) assume that the closed-loop system is formed by combining TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ using the synchronous product. In our $||_{SD}$ setting, as we have devised a new way of constructing the closed-loop system, these properties need to be adapted with respect to our SD synchronous product operator. In this section, we redefine the TDES and SD properties to match with our $||_{SD}$ setting.

We would like to clarify here that the definitions presented in the following sections are conceptually similar (but not identical) to the ones given in the SD setting. For this reason, we will use the same name followed by "*with SD synchronous product*" to define the adapted version of these properties for our $||_{SD}$ setting. As a shorthand, we will simply write "$\langle$*property name*$\rangle$ *with* $||_{SD}$".

For the following definitions, let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Please note that both $\mathbf{G}$ and $\mathbf{S}$ are defined over the same event set Σ.

4.4.1 Plant Completeness with $||_{SD}$

Definition 4.4.1. A TDES plant $\mathbf{G}$ is *complete with* $||_{SD}$ for TDES supervisor $\mathbf{S}$ if:

$$(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}))\ (\forall \sigma \in \Sigma_{hib})\ s\sigma \in L(\mathbf{S}) \Rightarrow s\sigma \in L(\mathbf{G})$$

In the $||_{SD}$ setting, it says that for all strings s that are possible in $\mathbf{S}\,||_{SD}\,\mathbf{G}$, if a prohibitable event σ is enabled by $\mathbf{S}$ after s, then it must be possible in $\mathbf{G}$ as well.

4.4.2 S-Singular Prohibitable Behaviour with $||_{SD}$

Definition 4.4.2. For TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$, we say that $\mathbf{G}$ has $\mathbf{S}$-*singular prohibitable behaviour with* $||_{SD}$ if:

$$(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp})\ (\forall s' \in \Sigma^*_{act})\ ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \Rightarrow$$
$$(\forall \sigma \in Occu(s') \cap \Sigma_{hib})\ \sigma \notin Elig_{L(\mathbf{G})}(ss')$$

In the $||_{SD}$ setting, this definition states that for a given sampling period, if a prohibitable event σ has already occurred in $\mathbf{S}\,||_{SD}\,\mathbf{G}$, then σ must not be possible in $\mathbf{G}$ again in the same sampling period.

4.4.3 Timed Controllability with $||_{SD}$

As we are building our work on the SD supervisory control, where $\Sigma_{for} = \Sigma_{hib}$, we will adapt and discuss the timed controllability property (Definition 2.3.2) in terms of prohibitable events only.

In the SD setting, the closed-loop system is formed by combining plant and supervisor TDES using the synchronous product. In this case, a supervisor is solely in charge of enabling/disabling a *tick* event and forcing prohibitable events in the closed-loop system. The correct behaviour of the supervisor with respect to these decisions is ensured by checking the timed controllability property.

In our $||_{SD}$ setting, we are constructing the closed-loop system using $||_{SD}$. Our $||_{SD}$ operator is also capable of disabling a *tick* event, once a prohibitable event is possible in the plant and enabled by the supervisor. Thus in our setting, in addition to checking that supervisor is enabling/disabling *tick* at the right time, we also need to make sure that the $||_{SD}$ operator does not disable a *tick* event when it is not supposed to, i.e. the $||_{SD}$ operator must not disable a *tick* event when it is possible in the plant and enabled by the supervisor, and no prohibitable events are currently possible in $\mathbf{S}\,||_{SD}\,\mathbf{G}$. Otherwise, our system will become uncontrollable. We capture this notion in the following timed controllability property adapted for our $||_{SD}$ setting.

Definition 4.4.3. TDES supervisor $\mathbf{S}$ is *timed controllable with* $||_{SD}$ with respect to TDES plant $\mathbf{G}$ if for all $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$,

$$Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases}$$

It states that all uncontrollable events that are currently possible in $\mathbf{G}$ must be allowed to occur in the closed-loop system, $\mathbf{S}\,||_{SD}\,\mathbf{G}$. In addition, *tick* event must be enabled in $\mathbf{S}\,||_{SD}\,\mathbf{G}$ if it is possible in $\mathbf{G}$, unless there exists an eligible prohibitable event in $\mathbf{S}\,||_{SD}\,\mathbf{G}$ to preempt it. This property makes sure that neither $\mathbf{S}$ nor the $||_{SD}$

operator can disable a *tick* event, if it is possible in $\mathbf{G}$ and no prohibitable events are currently eligible to be forced in the closed-loop system to preempt *tick*.

Note: As our $||_{SD}$ setting is specific to TDES, we will drop the word "timed", and will simply refer to this property as "$\mathbf{S}$ is controllable with $||_{SD}$ with respect to $\mathbf{G}$".

It is notable that the untimed controllability property (Definition 2.2.15) is part of the standard timed controllability definition (Definition 2.3.2). Since we have adapted the timed controllability definition for our $||_{SD}$ setting, the untimed controllability property automatically gets redefined as part of it. Below, we explicitly state the untimed controllability with $||_{SD}$ property.

Definition 4.4.4. TDES supervisor $\mathbf{S}$ is *untimed controllable with* $||_{SD}$ with respect to TDES plant $\mathbf{G}$ if $(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}))\; Elig_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s)$.

4.5 SD Controllability with SD Synchronous Product

This section provides a detailed explanation of how we have adapted the property of SD controllability (Definition 3.5.1) defined in the SD setting into the property of *SD controllability with SD synchronous product* (*SD controllability with* $||_{SD}$, as a shorthand) for our $||_{SD}$ setting.

In the SD setting, the closed-loop system is constructed by synchronizing $\mathbf{G}$ and $\mathbf{S}$ using the synchronous product, along with the assumption that both $\mathbf{G}$ and $\mathbf{S}$ are defined over the same event set. Therefore, the authors have defined the SD controllability property with respect to the closed language $L(\mathbf{S}) \cap L(\mathbf{G})$, and marked language $L_m(\mathbf{S}) \cap L_m(\mathbf{G})$.

As the synchronization mechanism of the $||_{SD}$ operator is different than the synchronous product, the closed and marked languages generated in the $||_{SD}$ setting will potentially be different than the ones assumed in the SD setting. Keeping this in view, we need to modify the SD controllability definition with respect to the closed and marked languages to make it suitable for our $||_{SD}$ setting. Specifically, we have replaced its $L(\mathbf{S}) \cap L(\mathbf{G})$ with our closed language $L(\mathbf{S}\,||_{SD}\,\mathbf{G})$, and its $L_m(\mathbf{S}) \cap L_m(\mathbf{G})$ with our marked language $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$ to make it work for our $||_{SD}$ setting.

Below, we give a formal definition of the SD controllability with $||_{SD}$ property. This is followed by a description of how we logically adapted the individual points of the SD controllability definition to define our SD controllability with $||_{SD}$ property.

Definition 4.5.1. TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ is *SD controllable with* $||_{SD}$ with respect to TDES plant $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ if, $\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$, the following statements are satisfied:

i) $Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases}$

ii) If $s \in L_{samp}$ then

1) $(\forall s' \in \Sigma^*_{act})\,[ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})] \Rightarrow$
$[Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(ss') \cup Occu(s')] \cap \Sigma_{hib} = Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib}$

2) $(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \wedge Occu(s') = Occu(s'')] \Rightarrow$
$ss' \equiv_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}\ ss'' \wedge ss' \equiv_{L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})}\ ss''$

iii) $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \subseteq L_{samp}$

Point i: It says that **S** is controllable with $||_{SD}$ with respect to **G** (Definition 4.4.3).

Now we will discuss how this point logically corresponds to Point i and Point ii of the SD controllability definition. Point i of the SD controllability definition is the standard untimed controllability property. Together with Point ii reverse direction ($\Leftarrow$), it becomes the timed controllability property (Definition 2.3.2) of the SD setting. As we have adapted the timed controllability definition for our $||_{SD}$ setting, we will use our timed controllability with $||_{SD}$ property instead. In this way, Point i of our SD controllability with $||_{SD}$ definition is logically equivalent to Point i and Point ii ($\Leftarrow$) of the SD controllability definition.

In the forward direction ($\Rightarrow$), Point ii of the SD controllability definition states that if a prohibitable event is enabled in the closed-loop system, then *tick* must be disabled. It is noteworthy that this condition is essentially in agreement with the synchronization mechanism of our $||_{SD}$ operator. In simple words, this is exactly what our $||_{SD}$ operator does while synchronizing **G** and **S**, i.e. if a prohibitable event is enabled in the closed-loop system, our $||_{SD}$ operator automatically disables *tick* event in the closed-loop system, even if it is possible in both **G** and **S**.

This implies that our $||_{SD}$ operator guarantees that any closed-loop system constructed as $\mathbf{S}\,||_{SD}\,\mathbf{G}$ will always satisfy the condition imposed by Point ii ($\Rightarrow$) of the SD controllability definition. In our $||_{SD}$ setting, as we construct our closed-loop system as $\mathbf{S}\,||_{SD}\,\mathbf{G}$, this means that we do not need to explicitly check this condition, as it will always be satisfied by the $||_{SD}$ operator while synchronizing **G** and **S**. As a result, we eliminate this explicit condition from our SD controllability with $||_{SD}$ definition. In fact, ensuring the automatic satisfaction of this condition and removing this explicit check is the primary purpose of introducing the $||_{SD}$ operator and our $||_{SD}$ setting.

In this way, Points i and ii of the SD controllability definition get simplified, and are represented only by Point i in our SD controllability with $||_{SD}$ definition.

Point ii: As a result of the simplification discussed above, Point iii of the SD controllability definition becomes Point ii of the SD controllability with $||_{SD}$ definition. These two points are logically identical except for the way they assume their closed-loop systems to be constructed, which are different for the two settings, SD and $||_{SD}$.

Point iii: Point iii of the SD controllability with $||_{SD}$ definition corresponds to Point iv of the SD controllability definition. These two points essentially represent the same

logic. The only difference is their way of representing the marked behaviours, as per the SD and $||_{SD}$ setting.

Since Points ii and iii of the SD controllability with $||_{SD}$ definition are logically identical to Points iii and iv of the SD controllability definition respectively, we have not reexamined these points here. Please refer to Definition 3.5.1 of SD controllability to see a logical explanation of these points.

4.6 ALF Modularity and SD Synchronous Product

In this section, we present and discuss some important results for the ALF property with respect to our SD synchronous product operator in the $||_{SD}$ setting.

In our $||_{SD}$ setting, we wish our closed-loop system $\mathbf{S}\,||_{SD}\,\mathbf{G}$ to be ALF to rule out the possibility of having the physically unrealistic behaviour that activity events can preempt *tick* for an indefinite amount of time. Instead of first constructing the closed-loop system and then checking its ALF property, it would be much easier and economical if we could find a way to determine whether our closed-loop system is ALF or not before actually constructing it.

One possible way to do this is to apply the ALF check individually on $\mathbf{S}$ and $\mathbf{G}$ before synchronizing them to construct the closed-loop system. The following proposition formally proves that if $\mathbf{S}$ or $\mathbf{G}$ is ALF, then our closed-loop system constructed as $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is guaranteed to be ALF.

Proposition 4.3. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. If either $\mathbf{S}$ or $\mathbf{G}$ is ALF, then the closed-loop system $\boldsymbol{\mathcal{S}} = (Y, \Sigma, \eta, y_o, Y_m)$ constructed as $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF.

Proof. Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and that either $\mathbf{S}$ or $\mathbf{G}$ is ALF. By Definition 2.3.10 of the ALF property, this implies:

$$[(\forall x \in X_r)\,(\forall s \in \Sigma^+_{act})\,\xi(x,s) \neq x] \;\vee\; [(\forall q \in Q_r)\,(\forall s \in \Sigma^+_{act})\,\delta(q,s) \neq q] \qquad (1)$$

Must show: $\boldsymbol{\mathcal{S}}$ is ALF

By the ALF definition, it is sufficient to show: $(\forall y \in Y_r)\,(\forall s \in \Sigma^+_{act})\;\eta(y,s) \neq y$, where $Y_r \subseteq Y$ is the set of reachable states in $\boldsymbol{\mathcal{S}}$.

We will use proof by contradiction to show that $\boldsymbol{\mathcal{S}}$ is ALF.

Assume $\boldsymbol{\mathcal{S}}$ is not ALF, i.e. there exists an activity loop in $\boldsymbol{\mathcal{S}}$. By Definition 2.3.9 of activity loop, this implies: $(\exists y \in Y_r)\,(\exists s' \in \Sigma^+_{act})\;\eta(y,s') = y$ (2)

Let $y \in Y_r$, and let $s' \in \Sigma^+_{act}$ such that $\eta(y,s') = y$. (3)

As $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ by (1), by the definition of state set Y in the $||_{SD}$ operator (Definition 4.1.1), we have: $y = (x,q)$, such that $x \in X$ and $q \in Q$. (4)

Also, by the definition of Y in $||_{SD}$, we know that y is a reachable state in $\boldsymbol{\mathcal{S}}$, only if x is a reachable state in $\mathbf{S}$ and q is a reachable state in $\mathbf{G}$, i.e. $x \in X_r \wedge q \in Q_r$.

By (3), we have: $\eta(y, s') = y$

$\Rightarrow \eta((x, q), s') = (x, q)$ *by* (4)

As $\mathbf{S}$ and $\mathbf{G}$ are defined over the same event set Σ, by *Point i* of the $||_{SD}$ definition, we have that a transition will be defined at a state in $\boldsymbol{\mathcal{S}}$, only if it is defined at the corresponding states in both $\mathbf{S}$ and $\mathbf{G}$.

Since we have that transition for string s' is defined at state $y = (x, q)$ in $\boldsymbol{\mathcal{S}}$, this implies that s' transition is defined at state x in $\mathbf{S}$ and state q in $\mathbf{G}$.

$\Rightarrow \xi(x, s') = x \wedge \delta(q, s') = q$ *by Point i of* $||_{SD}$ *definition*

These transitions indicate that both $\mathbf{S}$ and $\mathbf{G}$ are not ALF. This contradicts our assumption of (1) that either $\mathbf{S}$ or $\mathbf{G}$ is ALF.

Thus, we deduce that our assumption of (2) is false, and $\boldsymbol{\mathcal{S}}$ is ALF.

$\Rightarrow (\forall y \in Y_r)\,(\forall s \in \Sigma^+_{act})\ \eta(y, s) \neq y$

Hence, we conclude that if either $\mathbf{S}$ or $\mathbf{G}$ is ALF, then $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF. □

As $\mathbf{S}$ and $\mathbf{G}$ are typically designed modularly by designers, our $\mathbf{S}$ and $\mathbf{G}$ will most likely be constructed as $\mathbf{S} = \mathbf{S}_1 \,||\, \mathbf{S}_2 \,||\, \ldots \,||\, \mathbf{S}_m$ and $\mathbf{G} = \mathbf{G}_1 \,||\, \mathbf{G}_2 \,||\, \ldots \,||\, \mathbf{G}_n$, where $m, n > 1$. Here, it is worthwhile to mention a proposition from Wang (2009) that presents an easy and modular way of obtaining an ALF TDES. The following proposition states that if each individual TDES is ALF, then their synchronous product is ALF. This proposition is useful in our $||_{SD}$ setting as we can make our $\mathbf{S}$ or $\mathbf{G}$ ALF just by making sure that each individual plant or supervisor component is ALF, even if these components are defined over different event sets.

Proposition 4.4. (Wang, 2009) For TDES $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2, \delta_2, q_{o,2}, Q_{m,2})$, if $\mathbf{G}_1$ and $\mathbf{G}_2$ are each ALF, then their synchronous product $\mathbf{G} = \mathbf{G}_1 \,||\, \mathbf{G}_2$ is ALF.

In the presence of Proposition 4.3 and Proposition 4.4, it is evident that if we want to construct an ALF closed-loop system in our $||_{SD}$ setting, we simply need to design ALF plant or supervisor components. This is because individual ALF plant or supervisor components ensure that when we synchronize them using synchronous product, our $\mathbf{S}$ or $\mathbf{G}$ will be ALF (Proposition 4.4). This in turn guarantees that our closed-loop system constructed as $\mathbf{S}\,||_{SD}\,\mathbf{G}$ will be ALF (Proposition 4.3). In this way, we can verify the ALF property of our closed-loop system before actually constructing $\mathbf{S}\,||_{SD}\,\mathbf{G}$, or even before constructing composite $\mathbf{S}$ and $\mathbf{G}$.

For our closed-loop system $\mathbf{S}\,||_{SD}\,\mathbf{G}$, we are also interested in making sure that our system does not try to "stop the clock", i.e. it should never reach a state where *tick* events are not possible anymore, as this behaviour is undesirable and physically unrealistic. Therefore, we want to guarantee that after a finite number of activity events, our system should always reach a state where the *tick* event is possible. In

the following proposition, we present sufficient conditions to ensure this behaviour.

Proposition 4.5. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, and let $\boldsymbol{\mathcal{S}} = (Y, \Sigma, \eta, y_o, Y_m)$ be the closed-loop system constructed as $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$. If both $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces, $\mathbf{G}$ has proper time behaviour, $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}}$ is ALF, then:

$$(\forall y \in Y_r)\,(\exists s \in \Sigma^*_{act})\ \eta(y, s\tau)!$$

Proof. Assume initial conditions, and let $y \in Y_r$.

Must show: $(\exists s \in \Sigma^*_{act})\ \eta(y, s\tau)!$

As both $\mathbf{G}$ and $\mathbf{S}$ have finite, non-empty state spaces (as they both contain an initial state), it follows from the definition of state set Y in the $||_{SD}$ operator (Definition 4.1.1) that the closed-loop system $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ has a finite, non-empty state space.

Let $n = |Y_r|$.

By our initial assumption, we have that $\boldsymbol{\mathcal{S}}$ is ALF. This implies that starting at state y in $\boldsymbol{\mathcal{S}}$, the system can do at most $n-1$ activity event transitions before it has visited all n reachable states. At this point, there must be no more activity event transitions possible in $\boldsymbol{\mathcal{S}}$. Otherwise, the system would have to visit a state twice, thus creating an activity loop and failing the ALF definition.

This idea can be formally expressed as follows:

$$(\exists s \in \Sigma^*_{act})\ |s| \leq n-1\ \wedge\ (\exists y' \in Y_r)\ \eta(y, s) = y'\ \wedge\ (\forall \sigma \in \Sigma_{act})\ \neg\, \eta(y', \sigma)! \qquad (1)$$

Now we will present our argument to show that *tick* transition is defined at state y' in $\boldsymbol{\mathcal{S}}$.

By (1), we have that no activity events are possible at state y' in $\boldsymbol{\mathcal{S}}$. As $\Sigma_u \subseteq \Sigma_{act}$, this means that no uncontrollable events are possible at y' in $\boldsymbol{\mathcal{S}}$.

By our initial assumption, we have that $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$. This implies that no uncontrollable events are defined at the corresponding state in $\mathbf{G}$, as $\mathbf{S}$ would not have restricted them, and there are none possible in $\boldsymbol{\mathcal{S}}$.

Lets refer to this state of $\mathbf{G}$ as $q' \in Q$.

As $\mathbf{G}$ has proper time behaviour, this implies that *tick* is defined at state q' in $\mathbf{G}$.

By (1), we have that no activity events are possible at state y' in $\boldsymbol{\mathcal{S}}$. As $\Sigma_{hib} \subseteq \Sigma_{act}$, this means no prohibitable events are enabled at y' in $\boldsymbol{\mathcal{S}}$.

We have that *tick* event is defined at state q' in $\mathbf{G}$ and no prohibitable event is eligible at state y' to preempt the *tick* in $\boldsymbol{\mathcal{S}}$. As $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$, this implies that neither $\mathbf{S}$ nor the $||_{SD}$ operator can disable the *tick* event at this point in time. Thus, the *tick* event must be enabled at state y' in $\boldsymbol{\mathcal{S}}$.

Thus, we have: $\eta(y', \tau)!$

$\Rightarrow \eta(\eta(y, s), \tau)!$ *by* (1)

$\Rightarrow \eta(y, s\tau)!$ *by definition of transition function*

Hence, we conclude $(\forall y \in Y_r)\,(\exists s \in \Sigma^*_{act})\ \eta(y, s\tau)!$. □

Chapter 5

Equivalence of SD and SD Synchronous Product Setting

Now that we have described our SD synchronous product setting in detail, our next target is to establish equivalence between the SD setting (Chapter 3) and our $||_{SD}$ setting (Chapter 4). This chapter serves as the first stepping stone to achieve this goal.

We begin this chapter by presenting a discussion on why this equivalence between the two settings is needed, how it will be established, and how it will pave the way for proving controllability, nonblocking and all SD verification results in our $||_{SD}$ setting (Chapter 8). After this discussion, we state some assumptions that apply to our complete study. This is followed by our language equivalence results, where we establish and formally prove equivalence between the closed and marked languages of the SD and $||_{SD}$ setting. Utilizing these results, we then demonstrate the equivalence between various SD properties in the two settings.

5.1 Establishing Equivalence

In this section, we present a detailed discussion on establishing equivalence between the SD and $||_{SD}$ settings. First, we explain why we opted for establishing equivalence between the two settings. After that, we provide a complete road map to establish our desired equivalence by presenting a comprehensive description of how did we plan to prove this equivalence and make use of it while performing our controllability and nonblocking verification in the $||_{SD}$ setting.

5.1.1 Why Equivalence is Needed?

In our $||_{SD}$ setting, we have presented a novel way of constructing the closed loop system by synchronizing TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ using the $||_{SD}$ operator.

By doing this, we have essentially changed the way of obtaining closed and marked languages for the system. Since our $||_{SD}$ operator generates a new system language that, in most cases, will not be the same as the language generated by synchronous product in the SD setting, the controllability and nonblocking verification results of the SD setting do not remain valid in our $||_{SD}$ setting. This means that we need to reprove all verification results of the SD setting for our $||_{SD}$ setting.

There are two possible ways to perform controllability and nonblocking verification in our $||_{SD}$ setting: 1) prove all SD results from scratch, or 2) establish some kind of logical equivalence between the SD and $||_{SD}$ setting, so that the results that have already been proven in the SD setting remain applicable to our $||_{SD}$ setting as well. This will allow us to reuse and base our results on some of the existing results from the SD setting while performing the controllability and nonblocking verification in our $||_{SD}$ setting.

Hypothetically, we could follow the first approach and prove all SD results in our $||_{SD}$ setting from scratch. But the issue with this approach is that there is no closed and obvious form of the closed and marked language that is generated by synchronizing $\mathbf{G}$ and $\mathbf{S}$ using the $||_{SD}$ operator, i.e. $L(\mathbf{S}||_{SD}\mathbf{G})$ and $L_m(\mathbf{S}||_{SD}\mathbf{G})$. By this we mean that, apparently, we cannot easily express these languages in terms of the natural projection or its equivalent, as has been done for the synchronous product. For plant $\mathbf{G}$ and an arbitrary supervisor $\mathbf{S}$ that are defined over the same event set Σ, we know that $L(\mathbf{S}\,||\,\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$ and $L_m(\mathbf{S}\,||\,\mathbf{G}) = L_m(\mathbf{S}) \cap L_m(\mathbf{G})$. However, in most cases, we do not expect to have this kind of equality for our $||_{SD}$ operator, i.e. $L(\mathbf{S}||_{SD}\mathbf{G}) \neq L(\mathbf{S}) \cap L(\mathbf{G})$ and $L_m(\mathbf{S}||_{SD}\mathbf{G}) \neq L_m(\mathbf{S}) \cap L_m(\mathbf{G})$. This is due to the automatic *tick* disablement mechanism of the $||_{SD}$ operator that is not present in the synchronous product operator.

In the absence of such a closed and clear cut form for $L(\mathbf{S}||_{SD}\mathbf{G})$ and $L_m(\mathbf{S}||_{SD}\mathbf{G})$, working with these languages and proving all SD verification results from scratch in the $||_{SD}$ setting does not look like a straightforward and trouble-free task. Due to these reasons, we opt for the second approach of establishing logical equivalence between the SD and $||_{SD}$ setting, and then utilize this equivalence to perform our controllability and nonblocking verification in the $||_{SD}$ setting.

5.1.2 How to Establish Equivalence?

Note: In this discussion, in fact in the rest of this thesis, we need to talk about two supervisors, one from the SD setting and the other from our $||_{SD}$ setting. Since these two supervisors will most likely be different (as they may satisfy different properties of the two settings), therefore, in order to avoid any ambiguity, we will use two different symbols to refer to them. The supervisor of the SD setting will be stated as $\boldsymbol{\mathcal{S}}$ ($\boldsymbol{\mathcal{S}}$ maps and refers to $\mathbf{S}$ of Chapter 3), whereas the supervisor of our $||_{SD}$ setting will be referred to as $\mathbf{S}$.

In the SD setting, since TDES plant $\mathbf{G}$ and TDES supervisor $\boldsymbol{\mathcal{S}}$ are assumed to be

combined with the synchronous product, therefore all verification results have been proven using the closed language $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and marked language $L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$. The authors have assumed that $\boldsymbol{\mathcal{S}}$ is an arbitrary supervisor that satisfies certain SD properties, independently and when combined with $\mathbf{G}$ to form the closed-loop system, $\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}$. This supervisor $\boldsymbol{\mathcal{S}}$ is then used to generate its corresponding controller implementation in the SD setting using the translation method described in Section 3.7. Therefore, in order to make the SD results valid in our $||_{SD}$ setting and derive our results based on these existing results, we need to satisfy all these conditions and prove equivalence at all levels, i.e. 1) prove language equivalence, 2) satisfy all properties that have been considered as preconditions for concluding the controllability and nonblocking verification results, and 3) prove controller's equivalence.

To establish the required logical equivalence, first and foremost, we need to establish language equivalence between the two settings. Since the closed and marked languages generated in the SD and $||_{SD}$ settings are $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and $L(\mathbf{S} \,||_{SD}\, \mathbf{G})$, and $L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$ and $L_m(\mathbf{S} \,||_{SD}\, \mathbf{G})$ respectively, we can potentially establish language equivalence between the two settings if we could somehow prove that $L(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and $L_m(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$. To do this, we need to find an appropriate and concrete definition for supervisor $\boldsymbol{\mathcal{S}}$ of the SD setting, that is not only guaranteed to exist, but should be based on or somehow related to our $\mathbf{S} \,||_{SD}\, \mathbf{G}$ to achieve the above-mentioned equivalence.

An intriguing idea is what if we define $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$? Can we establish our desired language equivalence between the two settings with this definition of $\boldsymbol{\mathcal{S}}$? Can we demonstrate that $\boldsymbol{\mathcal{S}}$ satisfies all the properties as required by existing verification results of the SD setting? Can we prove that the controller implementation of $\boldsymbol{\mathcal{S}}$ will be according to the requirements of the SD setting? If we can prove these things, then we can certainly define $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ to prove the desired equivalence, and make use of the existing SD results while verifying our $||_{SD}$ setting.

This is exactly the approach that we adopt for proving equivalence between the two settings. We start by establishing language equivalence between the two settings and proving that if $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, then both settings have the same closed and marked behaviours. Specifically, we prove that $L(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and $L_m(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$ (Section 5.3).

Then, we focus on satisfying the preconditions (various SD properties) of the SD verification results. Specifically, we demonstrate that if certain SD properties are satisfied in the $||_{SD}$ setting with respect to our supervisor $\mathbf{S}$, this implies that their corresponding SD properties are guaranteed to be satisfied with respect to supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ in the SD setting (Section 5.4). We also show how to process $\boldsymbol{\mathcal{S}}$ to satisfy some other properties that are required in the SD setting but may not be directly satisfied as $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ (Chapter 6).

Finally, we prove that the SD controller that is obtained by translating $\mathbf{S}$ in our $||_{SD}$ setting is output equivalent to the controller that is generated by supervisor $\boldsymbol{\mathcal{S}}$ of the SD setting with respect to valid input strings, i.e. strings that are possible in the

two closed-loop behaviours (Chapter 7). In other words, controller implementation of the two supervisors, $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}$, will exhibit exactly the same control behaviour with respect to TDES plant $\mathbf{G}$.

Once we have this formal equivalence between the two settings in place, we will successfully satisfy all the assumptions and preconditions that have been identified for proving all verification results in the SD setting. Since we have fulfilled all the prerequisites, we can rightly conclude the SD verification results. In this way, the existing SD results become valid in the $||_{SD}$ setting and we can easily reuse them to build our controllability and nonblocking verification results of the $||_{SD}$ setting (Chapter 8).

Before closing this section, it is also important to clearly state the relationship that we have established between the two settings to do our formal theoretical verification. The basic idea is that in the $||_{SD}$ setting, supervisor $\mathbf{S}$ is expected to be manually designed by the designers for plant $\mathbf{G}$, and is required to satisfy certain SD properties with $||_{SD}$, defined in Chapter 4. It is worth-mentioning here that while designing $\mathbf{S}$, designers do not need to manually take care of the tricky condition imposed by Point ii ($\Rightarrow$) of the SD controllability definition, as required in the SD setting. This $\mathbf{S}$ should then be synchronized with $\mathbf{G}$ using our $||_{SD}$ operator to construct $\mathbf{S}\,||_{SD}\,\mathbf{G}$.

Instead of using this $\mathbf{S}\,||_{SD}\,\mathbf{G}$ as our closed-loop system for theoretical verification of the $||_{SD}$ setting, we treat this $\mathbf{S}\,||_{SD}\,\mathbf{G}$ as the "supervisor" of the SD setting, i.e. $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$. We will be able to do this because of our equivalence results, since these results ensure that $\boldsymbol{\mathcal{S}}$ is guaranteed to satisfy all properties that a supervisor of the SD setting is required to satisfy. It is noteworthy that $\boldsymbol{\mathcal{S}}$ is also guaranteed to automatically satisfy Point ii ($\Rightarrow$) of the SD controllability definition with respect to $\mathbf{G}$ because of the synchronization mechanism of our $||_{SD}$ operator that is used to construct $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$.

This $\boldsymbol{\mathcal{S}}$ is then assumed to be synchronized with $\mathbf{G}$ using the synchronous product to construct the closed-loop system $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$, with closed language $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and marked language $L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$, as done in the existing SD setting. All the existing SD verification results then follow immediately, as they have been proven using the same closed and marked languages in the SD setting.

We would like to clarify that software and hardware practitioners do not actually need to construct supervisor $\boldsymbol{\mathcal{S}}$ or closed-loop system $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ of the SD setting in practice. Also, they are not required to physically implement $\boldsymbol{\mathcal{S}}$ as their controller. This additional step is only considered and discussed here with respect to theoretical verification of our $||_{SD}$ setting. Practically, designers and practitioners only need to design supervisor $\mathbf{S}$ with the desired SD properties of the $||_{SD}$ setting. This supervisor can then be translated to generate its corresponding controller implementation using the translation method described in Section 3.7.

In this way, our $||_{SD}$ setting inherently liberates the designers from manually designing the potentially intricate supervisor of the SD setting that must satisfy all SD conditions, especially the stringent SD controllability Point ii ($\Rightarrow$). Using our

approach, they should now be able to design a much simpler and less complicated supervisor $\mathbf{S}$ of the $||_{SD}$ setting that, when combined with $\mathbf{G}$ using the $||_{SD}$ operator, is equivalent in its closed-loop behaviour, control action and controller implementation to the one required by the SD setting. In other words, by introducing the concrete definition of $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, we essentially provide a way to automatically generate a TDES $\boldsymbol{\mathcal{S}}$ that qualifies as the supervisor of the SD setting by satisfying all the required properties and conditions.

5.2 Implicit Assumptions

In this section, we list down our implicit assumptions that hold true for our $||_{SD}$ setting. Since these assumptions apply to our complete study, we are stating them together at one place, and will not repeat them in any of the upcoming sections/chapters.

1. For TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ of the $||_{SD}$ setting, we assume that both $\mathbf{G}$ and $\mathbf{S}$ are always defined over the same event set. However, in the case where $\mathbf{G}$ and $\mathbf{S}$ are not defined over the same alphabet, we can simply add selfloops to each TDES for the missing events at every state to extend them over the same event set, without any loss of generality. If we assume otherwise in any particular section of this thesis, we will explicitly state that.
2. Let TDES $\boldsymbol{\mathcal{S}}$ be constructed by synchronizing plant $\mathbf{G}$ and supervisor $\mathbf{S}$ using the SD synchronous product operator, i.e. $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$. Since both $\mathbf{G}$ and $\mathbf{S}$ are defined over the same event set, by definition of the $||_{SD}$ operator, the resultant TDES $\boldsymbol{\mathcal{S}}$ will also have the same event set as $\mathbf{G}$ and $\mathbf{S}$.
3. As $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ are defined over the same event set, by Definition 2.2.14 of the synchronous product, we have that $L(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$. In the rest of this thesis, we might interchangeably use these two representations of synchronous product without explicit explanation.
4. In the SD supervisory control theory, it has been assumed that the set of prohibitable events (Σ_{hib}) is exactly equal to the set of forcible events (Σ_{for}), i.e. $\Sigma_{for} = \Sigma_{hib}$. Since we are using this methodology as the basis of our work, this assumption holds true for our study as well.
5. All TDES discussed in this thesis are assumed to be reachable and deterministic with a finite state space and a finite event set.

5.3 Equivalence of Languages

In this section, we present our desired language equivalence results for the SD and $||_{SD}$ setting. Specifically, we formally prove that the closed and marked languages generated in the two settings are equivalent.

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let $\boldsymbol{\mathcal{S}} = (Y, \Sigma, \eta, y_o, Y_m)$ be a TDES constructed as $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$.

We start by proving two propositions that will help us in showing our main language equivalence result. The basic idea of these two propositions has been taken from Definition 4.1.1 of our SD synchronous product operator.

By looking at the synchronization mechanism of the $||_{SD}$ operator, we note that $||_{SD}$ 'potentially' adds a transition to $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, only if that transition is defined in both $\mathbf{S}$ and $\mathbf{G}$. It does not add any transition to $\boldsymbol{\mathcal{S}}$ that is not defined in either $\mathbf{S}$ or $\mathbf{G}$. This implies that the strings defined in $L(\boldsymbol{\mathcal{S}})$ are going to be a subset of the strings that are defined in both $L(\mathbf{S})$ and $L(\mathbf{G})$. The proposition given below uses proof by induction to formally prove this notion.

Proposition 5.1. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$, then: **(i)** $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S})$, and **(ii)** $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{G})$.

Proof. We will prove these two points together.

Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ (1)

Must show: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S})$ and $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{G})$

Sufficient to show: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S}) \cap L(\mathbf{G})$

Let $s \in L(\boldsymbol{\mathcal{S}})$. Must show this implies: $s \in L(\mathbf{S}) \cap L(\mathbf{G})$

We will use induction on the length of s to show: $s \in L(\mathbf{S}) \cap L(\mathbf{G})$

Base Case: $s = \epsilon$

As $\mathbf{S}$ contains an initial state x_o, and $\mathbf{G}$ contains an initial state q_o, it follows that $\epsilon \in L(\mathbf{S})$ and $\epsilon \in L(\mathbf{G})$.

$\Rightarrow \epsilon \in L(\mathbf{S}) \cap L(\mathbf{G})$

Base case complete.

Inductive Step: For some $k \geq 0$, we assume:

- $s = \sigma_1 \ldots \sigma_k \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{S}) \cap L(\mathbf{G})$ (2)
- $s\sigma_{k+1} \in L(\boldsymbol{\mathcal{S}})$ (3)

We will now show this implies: $s\sigma_{k+1} \in L(\mathbf{S}) \cap L(\mathbf{G})$

By (2), we have: $s \in L(\boldsymbol{\mathcal{S}})$

$\Rightarrow \eta(y_o, s)!$ *by definition of* $L(\boldsymbol{\mathcal{S}})$ (4)

We have $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ by (1). The $||_{SD}$ operator defines the initial state of $\boldsymbol{\mathcal{S}}$ as an ordered pair of the initial states of $\mathbf{S}$ and $\mathbf{G}$.

$\Rightarrow \eta((x_o, q_o), s)!$ *by definition of* y_o *in* $||_{SD}$ *definition* (5)

As $\mathbf{S}$ and $\mathbf{G}$ are defined over the same event set Σ, by *Point i* of the $||_{SD}$ operator's definition, we have that a transition will be defined at a state in $\boldsymbol{\mathcal{S}}$, only if it is defined at corresponding states in both $\mathbf{S}$ and $\mathbf{G}$. Also, we know that the $||_{SD}$ operator is defined in such a way that it may remove a *tick* transition from $\boldsymbol{\mathcal{S}}$ under certain

conditions, even though that *tick* transition is possible in both **S** and **G**, but it cannot add any *tick* or non-*tick* transition to $\boldsymbol{\mathcal{S}}$ that is not defined in either **S** or **G**.

Since, by (4), we have that string s is defined at state y_o in $\boldsymbol{\mathcal{S}}$, by (5) this implies that s is defined at state x_o in **S** and state q_o in **G**.

$\Rightarrow \xi(x_o, s)! \wedge \delta(q_o, s)!$ *by Point i of* $||_{SD}$ *definition* (6)

By (3), we have: $s\sigma_{k+1} \in L(\boldsymbol{\mathcal{S}})$

$\Rightarrow \eta(y_o, s\sigma_{k+1})!$ *by definition of* $L(\boldsymbol{\mathcal{S}})$

$\Rightarrow \eta(\eta(y_o, s), \sigma_{k+1})!$ *by* (4) *and definition of transition function*

$\Rightarrow \eta(\eta((x_o, q_o), s), \sigma_{k+1})!$ *by* (5)

As σ_{k+1} transition is defined in $\boldsymbol{\mathcal{S}}$, by *Point i* of the $||_{SD}$ definition, this implies that σ_{k+1} transition is defined at corresponding states in both **S** and **G**.

$\Rightarrow \xi(\xi(x_o, s), \sigma_{k+1})! \wedge \delta(\delta(q_o, s), \sigma_{k+1})!$ *by* (6) *and Point i of* $||_{SD}$ *definition*

$\Rightarrow \xi(x_o, s\sigma_{k+1})! \wedge \delta(q_o, s\sigma_{k+1})!$ *by definition of transition function*

$\Rightarrow s\sigma_{k+1} \in L(\mathbf{S}) \wedge s\sigma_{k+1} \in L(\mathbf{G})$ *by definition of* $L(\mathbf{S})$ *and* $L(\mathbf{G})$

$\Rightarrow s\sigma_{k+1} \in L(\mathbf{S}) \cap L(\mathbf{G})$

Inductive step complete.

By our base case and inductive step, we have shown that for some arbitrary string s, $s \in L(\boldsymbol{\mathcal{S}})$ implies $s \in L(\mathbf{S}) \cap L(\mathbf{G})$. Thus, we have shown that $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S}) \cap L(\mathbf{G})$.

Hence, we conclude: **(i)** $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S})$, and **(ii)** $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{G})$. □

In the next proposition, we prove same idea with respect to marked languages of **S**, **G** and $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$. Specifically, we show that the marked strings that make up $L_m(\boldsymbol{\mathcal{S}})$ is a subset of the marked strings that are defined in both $L_m(\mathbf{S})$ and $L_m(\mathbf{G})$. This proof is partially based on the result of our previous proposition.

Proposition 5.2. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$, then: **(i)** $L_m(\boldsymbol{\mathcal{S}}) \subseteq L_m(\mathbf{S})$, and **(ii)** $L_m(\boldsymbol{\mathcal{S}}) \subseteq L_m(\mathbf{G})$.

Proof. We will prove these two points together.

Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$. Sufficient to show: $L_m(\boldsymbol{\mathcal{S}}) \subseteq L_m(\mathbf{S}) \cap L_m(\mathbf{G})$

Let $s \in L_m(\boldsymbol{\mathcal{S}})$. Must show this implies: $s \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})$

We have: $s \in L_m(\boldsymbol{\mathcal{S}})$

$\Rightarrow \eta(y_o, s)! \wedge \eta(y_o, s) \in Y_m$ *by definition of* $L_m(\boldsymbol{\mathcal{S}})$

$\Rightarrow s \in L(\boldsymbol{\mathcal{S}}) \wedge \eta(y_o, s) \in Y_m$ *by definition of* $L(\boldsymbol{\mathcal{S}})$ (1)

As $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, thus by Proposition 5.1, we have: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S})$ and $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{G})$

$\Rightarrow s \in L(\mathbf{S}) \wedge s \in L(\mathbf{G})$ *by Proposition* 5.1 (2)

The $||_{SD}$ operator defines the initial state of $\boldsymbol{\mathcal{S}}$ as an ordered pair of the initial states of **S** and **G**, and the set of marked states of $\boldsymbol{\mathcal{S}}$ as cross product of the set of marked states of **S** and **G**.

By (1), we have: $\eta(y_o, s) \in Y_m$
$\Rightarrow \eta((x_o, q_o), s) \in X_m \times Q_m$ *by (2) and definition of y_o and Y_m in $||_{SD}$ definition*
$\Rightarrow \xi(x_o, s) \in X_m \wedge \delta(q_o, s) \in Q_m$ *by Point i and definition of Y_m in $||_{SD}$ definition*
$\Rightarrow s \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})$
Hence, we conclude: **(i)** $L_m(\boldsymbol{\mathcal{S}}) \subseteq L_m(\mathbf{S})$, and **(ii)** $L_m(\boldsymbol{\mathcal{S}}) \subseteq L_m(\mathbf{G})$. □

Based on the above two propositions, we now present our main result of proving language equivalence between the SD and our $||_{SD}$ setting. In the following proposition, we prove that the closed and marked languages generated by synchronizing TDES supervisor $\mathbf{S}$ and TDES plant $\mathbf{G}$ using $||_{SD}$ operator in the $||_{SD}$ setting are the same as the closed and marked languages obtained by combining TDES supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and TDES plant $\mathbf{G}$ using synchronous product operator in the SD setting.

Proposition 5.3. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$, then: **(i)** $L(\boldsymbol{\mathcal{S}}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$, and **(ii)** $L_m(\boldsymbol{\mathcal{S}}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$.

Proof. Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$

i) Show: $L(\boldsymbol{\mathcal{S}}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$
Sufficient to show: **(1)** $L(\boldsymbol{\mathcal{S}}) \subseteq L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$, and **(2)** $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \subseteq L(\boldsymbol{\mathcal{S}})$.

1) Show: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$
Let $s \in L(\boldsymbol{\mathcal{S}})$. Must show this implies: $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ (1)
As $s \in L(\boldsymbol{\mathcal{S}})$ by (1), sufficient to show: $s \in L(\mathbf{G})$
As $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, thus by Proposition 5.1, we have: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S}) \cap L(\mathbf{G})$
$\Rightarrow s \in L(\mathbf{G})$ *by (1) and Proposition 5.1*
We thus conclude that $L(\boldsymbol{\mathcal{S}}) \subseteq L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$.

2) Show: $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \subseteq L(\boldsymbol{\mathcal{S}})$
This follows automatically from the definition of set intersection.

By Parts (1) and (2), we conclude that $L(\boldsymbol{\mathcal{S}}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$.

ii) Show: $L_m(\boldsymbol{\mathcal{S}}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$
Proof is identical to Part (i) up to relabelling closed languages $L(\mathbf{S})$, $L(\mathbf{G})$ and $L(\boldsymbol{\mathcal{S}})$ to marked languages $L_m(\mathbf{S})$, $L_m(\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}})$ respectively, and replacing Proposition 5.1 with Proposition 5.2 in Part (1). □

By our assumptions, we know that $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, $L(\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$, and $L_m(\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$. This means that Proposition 5.3 can be stated in multiple ways. Below we derive a corollary based on our main language equivalence result. We will then refer to the various points of this corollary to directly cite the result in the required form in the upcoming proofs.

Corollary 5.1. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$, then:

i) $L(\boldsymbol{\mathcal{S}}) = L(\boldsymbol{\mathcal{S}} || \mathbf{G})$ **ii)** $L_m(\boldsymbol{\mathcal{S}}) = L_m(\boldsymbol{\mathcal{S}} || \mathbf{G})$

iii) $L(\mathbf{S} ||_{SD} \mathbf{G}) = L(\boldsymbol{\mathcal{S}} || \mathbf{G})$ **iv)** $L_m(\mathbf{S} ||_{SD} \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}} || \mathbf{G})$

v) $L(\mathbf{S} ||_{SD} \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ **vi)** $L_m(\mathbf{S} ||_{SD} \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$

Proof. Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$

i) Show: $L(\boldsymbol{\mathcal{S}}) = L(\boldsymbol{\mathcal{S}} || \mathbf{G})$
As both $\boldsymbol{\mathcal{S}}$ and $\mathbf{G}$ are defined over Σ, we thus have: $L(\boldsymbol{\mathcal{S}} || \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$
The result follows automatically from Proposition 5.3.

ii) Show: $L_m(\boldsymbol{\mathcal{S}}) = L_m(\boldsymbol{\mathcal{S}} || \mathbf{G})$
Proof is identical to Part (i) up to relabelling closed languages $L(\mathbf{S})$, $L(\mathbf{G})$ and $L(\boldsymbol{\mathcal{S}})$ to marked languages $L_m(\mathbf{S})$, $L_m(\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}})$ respectively.

iii) Show: $L(\mathbf{S} ||_{SD} \mathbf{G}) = L(\boldsymbol{\mathcal{S}} || \mathbf{G})$
As $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$, the result follows immediately from Part (i).

iv) Show: $L_m(\mathbf{S} ||_{SD} \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}} || \mathbf{G})$
As $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$, the result follows immediately from Part (ii).

v) Show: $L(\mathbf{S} ||_{SD} \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$
As $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$, the result follows immediately from Proposition 5.3(i).

vi) Show: $L_m(\mathbf{S} ||_{SD} \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$
As $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$, the result follows immediately from Proposition 5.3(ii). □

5.4 Equivalence of SD Properties

In this section, we prove equivalence between the two versions of various properties that are defined in the SD and $||_{SD}$ setting.

In our $||_{SD}$ setting, we expect TDES supervisor $\mathbf{S}$ to be manually designed by software designers, and is required to satisfy certain properties. By introducing the $||_{SD}$ setting, we are devising a way to automatically construct the supervisor of the SD setting as $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$. This means we must also provide a way to automatically satisfy various properties that the supervisor of the SD setting is required to satisfy. This is discussed in the following subsections. Specifically, in these subsections, we formally prove that if certain $||_{SD}$ properties are satisfied with respect to $\mathbf{S}$ and TDES plant $\mathbf{G}$ in our $||_{SD}$ setting, this implies that the corresponding SD properties are guaranteed to be satisfied with respect to TDES supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$ and $\mathbf{G}$ in the SD setting.

5.4.1 Plant Completeness

In the SD setting, it is required that plant TDES should be complete for the supervisor TDES. In the following proposition, we prove that if plant $\mathbf{G}$ is complete with $||_{SD}$ for supervisor $\mathbf{S}$ in our $||_{SD}$ setting, then this is sufficient to ensure that $\mathbf{G}$ is complete for supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ in the SD setting.

Proposition 5.4. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, and TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor. If $\mathbf{G}$ is complete with $||_{SD}$ for $\mathbf{S}$, then $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$.

Proof. Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, and $\mathbf{G}$ is complete with $||_{SD}$ for $\mathbf{S}$ (1)

To show that $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, it is sufficient to show:

$$(\forall s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}))\ (\forall \sigma \in \Sigma_{hib})\ s\sigma \in L(\boldsymbol{\mathcal{S}}) \Rightarrow s\sigma \in L(\mathbf{G})$$

Let $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and let $\sigma \in \Sigma_{hib}$. Assume: $s\sigma \in L(\boldsymbol{\mathcal{S}})$ (2)

Must show this implies: $s\sigma \in L(\mathbf{G})$

By (2), we have: $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

$\Rightarrow s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ *by* (1) *and Corollary* 5.1(v) (3)

As $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ by (1), thus by Proposition 5.1, we have: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S})$

$\Rightarrow s\sigma \in L(\mathbf{S})$ *by* (2) *and Proposition* 5.1 (4)

$\Rightarrow s\sigma \in L(\mathbf{G})$ *by* (1-4)

Hence, we conclude that $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$. □

5.4.2 S-Singular Prohibitable Behaviour

One of the assumptions made in the SD setting is that controllers allow prohibitable events to occur only once per sampling period. This should be reflected in the plant model as well. Hence, plant $\mathbf{G}$ is required to satisfy $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour with respect to supervisor $\boldsymbol{\mathcal{S}}$ in the SD setting. The following proposition proves that if $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$ with respect to supervisor $\mathbf{S}$ in the $||_{SD}$ setting, then $\mathbf{G}$ is guaranteed to have $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour with respect to supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ in the SD setting.

Proposition 5.5. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, and TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor. If $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, then $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour.

Proof. Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, and $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$ (1)

To show that $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, it is sufficient to show:

$$(\forall s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \cap L_{samp})\ (\forall s' \in \Sigma^*_{act})\ ss' \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \Rightarrow$$
$$(\forall \sigma \in Occu(s') \cap \Sigma_{hib})\ \sigma \notin Elig_{L(\mathbf{G})}(ss')$$

Let $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \cap L_{samp}$, and let $s' \in \Sigma^*_{act}$. Assume: $ss' \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ (2)

Let $\sigma \in Occu(s') \cap \Sigma_{hib}$. Must show: $\sigma \notin Elig_{L(\mathbf{G})}(ss')$ (3)

By (2), we have: $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \cap L_{samp}$

$\Rightarrow s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ *by* (1) *and Corollary* 5.1(v) (4)

By (2), we have: $ss' \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

$\Rightarrow ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ *by* (1) *and Corollary* 5.1(v) (5)

$\Rightarrow \sigma \notin Elig_{L(\mathbf{G})}(ss')$ *by* (1-5)

Hence, we conclude that $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour. □

5.4.3 Timed Controllability

In the SD setting, supervisor TDES is assumed to be timed controllable with respect to plant TDES. The following proposition proves that while designing supervisor $\mathbf{S}$ of the $||_{SD}$ setting, if designers make sure that $\mathbf{S}$ is timed controllable with $||_{SD}$ with respect to plant $\mathbf{G}$, this guarantees that supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is timed controllable with respect to $\mathbf{G}$ in the SD setting.

Proposition 5.6. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor, and let $\Sigma_{for} = \Sigma_{hib}$. If $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$, then $\boldsymbol{\mathcal{S}}$ is timed controllable for $\mathbf{G}$.

Proof. Let $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and $\Sigma_{for} = \Sigma_{hib}$. (1)

Assume: $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$ (Definition 4.4.3) (2)

Must show: $\boldsymbol{\mathcal{S}}$ is timed controllable for $\mathbf{G}$

Substituting (1) in Definition 2.3.2 of timed controllability, it is sufficient to show:

$(\forall s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}))$

$$Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases} \quad (3)$$

Let $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$.

$\Rightarrow s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ *by* (1) *and Corollary* 5.1(v) (4)

As $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ by (1), applying Corollary 5.1(v) on the R.H.S of (3), we get:

$$Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases} \quad (5)$$

By (4) and (5), we thus have $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and:

$$Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases}$$

This is true by our assumption of (2), as s is an arbitrary string.

Hence, we conclude that $\boldsymbol{\mathcal{S}}$ is timed controllable for $\mathbf{G}$. □

5.4.4 SD Controllability

One of the most important assumptions made by the authors while proving controllability and nonblocking verification results in the SD setting is that the supervisor TDES is SD controllable with respect to the plant TDES. The proposition given below provides sufficient conditions to automatically satisfy this property in the SD setting. It proves that in the $||_{SD}$ setting, if designers create a supervisor $\mathbf{S}$ that is SD controllable with $||_{SD}$ with respect to plant $\mathbf{G}$, then supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is guaranteed to be SD controllable with respect to $\mathbf{G}$ in the SD setting.

Proposition 5.7. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor, and let $\Sigma_{for} = \Sigma_{hib}$. If $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$, then $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$.

Proof. Let $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and $\Sigma_{for} = \Sigma_{hib}$. (1)

Assume: $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$ (Definition 4.5.1) (2)

Must show $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$. By Definition 3.5.1 of SD controllability, it is sufficient to show the following:

$(\forall s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}))$

i) $Elig_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq Elig_{L(\boldsymbol{\mathcal{S}})}(s)$

ii) If $\tau \in Elig_{L(\mathbf{G})}(s)$, then $\tau \in Elig_{L(\boldsymbol{\mathcal{S}})}(s) \Leftrightarrow Elig_{L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

iii) If $s \in L_{samp}$ then

1) $(\forall s' \in \Sigma^*_{act})\,[ss' \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})] \Rightarrow$
$[Elig_{L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})}(ss') \cup Occu(s')] \cap \Sigma_{hib} = Elig_{L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib}$

2) $(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \wedge Occu(s') = Occu(s'')] \Rightarrow$
$ss' \equiv_{L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})} ss'' \wedge ss' \equiv_{L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})} ss''$

iv) $L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G}) \subseteq L_{samp}$

Let $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$.
$\Rightarrow s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ *by* (1) *and Corollary* 5.1(v) (3)

Now we will analyze the four points of the SD controllability definition individually.

i) To show Point i, we need to show: $Elig_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq Elig_{L(\boldsymbol{\mathcal{S}})}(s)$

This concept can be restated as: $Elig_{L(\boldsymbol{\mathcal{S}})}(s) \supseteq Elig_{L(\mathbf{G})}(s) \cap \Sigma_u$ (4)

In the next step, we will combine this with Part(a) of Point ii, and show this matches Point i of Definition 4.5.1, and is thus satisfied by (2).

ii) Point ii of the SD controllability definition represents an "if and only if" statement. We will analyze it in two parts.

If $\tau \in Elig_{L(\mathbf{G})}(s)$ then:

a) Reverse implication ($\Leftarrow$): $\tau \in Elig_{L(\boldsymbol{\mathcal{S}})}(s) \Leftarrow Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

b) Forward implication ($\Rightarrow$): $\tau \in Elig_{L(\boldsymbol{\mathcal{S}})}(s) \Rightarrow Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

Part a) The reverse implication can be restated as:

$$Elig_{L(\boldsymbol{\mathcal{S}})}(s) \supseteq Elig_{L(\mathbf{G})}(s) \cap \{\tau\} \text{ if } Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \quad (5)$$

Combining (4) and (5), we get:

$$Elig_{L(\boldsymbol{\mathcal{S}})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases}$$

Applying (1) on the L.H.S., and (1) and Corollary 5.1(v) on the R.H.S, we get:

$$Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases} \quad (6)$$

As this now matches Point i of Definition 4.5.1, it is satisfied by (2).

Part b) The forward implication says that if *tick* is possible in $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$, then no prohibitable events are possible after string s in $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$.

From (3), we have: $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$

We now need to show: $\tau \in Elig_{L(\boldsymbol{\mathcal{S}})}(s) \Rightarrow Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

Assume: $\tau \in Elig_{L(\boldsymbol{\mathcal{S}})}(s)$

$\Rightarrow \tau \in Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s)$ *as* $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ *by* (1) (7)

We now need to show this implies: $Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

By (1) and Corollary 5.1(v), it is sufficient to show: $Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

As we have $\tau \in Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s)$ by (7), this follows automatically from Point i of Definition 4.1.1 of the $||_{SD}$ operator. (8)

Combining with Point i and Part(a) of Point ii, we have now satisfied both Points i and ii of the SD controllability definition.

iii) From Corollary 5.1(v,vi), we have:

$$L(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \quad \text{and} \quad L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$$

We can thus rewrite Point iii of Definition 3.5.1 using these identities as follows:

If $s \in L_{samp}$ then (9)

1) $(\forall s' \in \Sigma^*_{act})\,[ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})] \Rightarrow$

$$[Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(ss') \cup Occu(s')] \cap \Sigma_{hib} = Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \quad (10)$$

2) $(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \wedge Occu(s') = Occu(s'')] \Rightarrow$
$ss' \equiv_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}\ ss'' \wedge ss' \equiv_{L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})}\ ss''$ (11)

As this now exactly matches Point ii of Definition 4.5.1, it is satisfied by (2).

iv) From Corollary 5.1(vi), we have: $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$

We can thus rewrite Point iv of Definition 3.5.1 as: $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \subseteq L_{samp}$ (12)

As this now exactly matches Point iii of Definition 4.5.1, it is satisfied by (2).

Combining (3), (6) and (8-12), we have shown that Points (i-iv) of the SD controllability definition are satisfied for $\boldsymbol{\mathcal{S}}$ and $\mathbf{G}$, as required.

Hence, we conclude that $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$. □

5.4.5 ALF

In order to show our equivalence result with respect to the ALF property, we will make use of one of the propositions from Wang (2009). Proposition 5.8 stated below says that the synchronous product of two TDES will be ALF, if one TDES is ALF, and the ALF TDES contains all events in the event set of the other TDES.

Proposition 5.8. (Wang, 2009) Let $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2, \delta_2, q_{o,2}, Q_{m,2})$ be two TDES. If $\mathbf{G}_1$ is ALF and $\Sigma_1 \supseteq \Sigma_2$, then $\mathbf{G}_1 \,||\, \mathbf{G}_2$ is also ALF.

In the SD setting, one of the preconditions of the controllability and nonblocking verification results is that the closed-loop system constructed by synchronizing the plant and supervisor models using the synchronous product is ALF. In order to automatically satisfy this condition of the SD setting, we require that the closed-loop system constructed as $\mathbf{S}\,||_{SD}\,\mathbf{G}$ in the $||_{SD}$ setting must be ALF. This is because if $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF, then the closed loop system constructed as $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ in the SD setting is guaranteed to be ALF by Proposition 5.8 (as both $\boldsymbol{\mathcal{S}}$ and $\mathbf{G}$ are defined over the same event set (Section 5.2)). Please note that in Section 4.6, we have already presented an easy and modular way of making $\mathbf{S}\,||_{SD}\,\mathbf{G}$ ALF.

Chapter 6

Equivalence using Minimal Automaton

In this chapter, we present some more results with respect to establishing equivalence between the SD and our $||_{SD}$ setting. The primary focus of this chapter is on describing the approach that we have formulated to process TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}||_{SD}\,\mathbf{G}$ (if required) and ensure that $\boldsymbol{\mathcal{S}}$ satisfies the property of concurrent string (CS) deterministic supervisors, as required by the supervisor of the SD setting.

This chapter begins with a discussion on why supervisor $\boldsymbol{\mathcal{S}}$ needs to be CS deterministic, and the significance of minimizing $\boldsymbol{\mathcal{S}}$. Then, we present our algorithms to obtain the minimal version of $\boldsymbol{\mathcal{S}}$ from its non-minimal TDES automaton. After that, we identify sufficient conditions and formally prove that minimized $\boldsymbol{\mathcal{S}}$ is guaranteed to be CS deterministic. Finally, we finish this chapter off by revisiting and re-evaluating our equivalence results presented in the previous chapter to make sure that they remain valid with the minimal version of $\boldsymbol{\mathcal{S}}$ as well.

6.1 Why Minimal Automaton is Needed?

The SD supervisory control methodology (Chapter 3) presents a formal translation method to translate a TDES supervisor into an SD controller. This translation process requires that the TDES supervisor must be CS deterministic (Definition 3.4.5). Otherwise, this conversion technique is not guaranteed to work. Since we are defining a concrete way to automatically construct TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}||_{SD}\,\mathbf{G}$ that we intend to use as the supervisor of the SD setting, we need $\boldsymbol{\mathcal{S}}$ to be CS deterministic.

We also want to make $\boldsymbol{\mathcal{S}}$ CS deterministic because in the SD setting, the developed translation method is used to convert the CS deterministic TDES supervisor into an SD controller (in fact, the controller would otherwise be non-deterministic). This CS deterministic supervisor and its corresponding SD controller have then been used in the SD setting as the basis to conclude various SD controllability and nonblocking

verification results. As we want to make these existing SD verification results valid in our $||_{SD}$ setting, and use them to derive and conclude our controllability and nonblocking verification results of the $||_{SD}$ setting, we must make sure that all preconditions of the SD verification results are satisfied.

Moreover, one of the goals of defining our $||_{SD}$ setting is to enable the software and hardware practitioners to design and implement our TDES supervisor $\mathbf{S}$ instead of the potentially more complex supervisor of the SD setting. In order to be able to do that, we need to show that the SD controller generated by translating $\mathbf{S}$ in our $||_{SD}$ setting is output equivalent (Definition 7.1.2) to the SD controller that is obtained by converting $\boldsymbol{\mathcal{S}}$ in the SD setting (this is demonstrated in Chapter 7). To theoretically prove this equivalence, we assume and require that the two supervisors $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}$ have been translated into their corresponding SD controllers. For this reason, both supervisors must be CS deterministic, as their translation into SD controllers is not possible otherwise.

In our $||_{SD}$ setting, since we want practitioners to design and implement our TDES supervisor $\mathbf{S}$, therefore we require them to design $\mathbf{S}$ in such a way that it must satisfy the property of CS deterministic supervisor. However, making $\mathbf{S}$ CS deterministic does not guarantee that $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$ will be CS deterministic. This is owing to the fact that in order to construct $\boldsymbol{\mathcal{S}}$, $\mathbf{S}$ needs to be synchronized with TDES plant $\mathbf{G}$ using $||_{SD}$ operator, and neither $\mathbf{G}$ nor the $||_{SD}$ operator guarantees to preserve the property of CS deterministic supervisor in any way. This means if $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$ is not CS deterministic, then we need to somehow process $\boldsymbol{\mathcal{S}}$ to make it CS deterministic.

Our approach of making $\boldsymbol{\mathcal{S}}$ CS deterministic relies on generating its minimal version. As we are proposing a strategy of obtaining a CS deterministic version of $\boldsymbol{\mathcal{S}}$, it is also important to show that $\boldsymbol{\mathcal{S}}$ will indeed become CS deterministic after applying our state space minimization algorithms (presented in the next section), and satisfying some other conditions. We formally prove this in Section 6.3.1.

In summary, if $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$ is CS deterministic in its original form, we can directly use it as the supervisor of the SD setting and generate its corresponding SD controller. However, if $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$ is not CS deterministic in its current form, then $\boldsymbol{\mathcal{S}}$ must be minimized using our state space minimization algorithms to make it CS deterministic, and essentially make it work within our $||_{SD}$ setting for use in our proofs. In practice, we would never need to actually minimize $\boldsymbol{\mathcal{S}}$, as once we have proven the required equivalence, we would just implement our $\mathbf{S}$ as an SD controller.

6.2 Obtaining a Minimal Automaton

In this section, we present our approach to *minimize* a given TDES automaton, i.e. obtain an equivalent TDES automaton that has as few states as possible as any automaton accepting the same closed and marked languages. This minimal TDES automaton is unique for the given language up to relabelling of states.

Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a TDES automaton. Without any loss of generality, we assume that $\mathbf{G}$ is a reachable automaton. We will describe our approach of obtaining the minimal version of TDES automaton $\mathbf{G}$ in two steps: 1) identify distinct λ-equivalent states of $\mathbf{G}$, and 2) construct minimal TDES automaton $\mathbf{G}'$. We elaborate these two steps and present their corresponding algorithms in the following two subsections. Please note that these algorithms are generic (not specific to our $\boldsymbol{\mathcal{S}}$) and can be used to generate the minimal version of any given TDES automaton.

6.2.1 Identify Distinct $\lambda-$Equivalent States

Algorithm 6.1 identifies distinct λ-equivalent states of a generator $\mathbf{G}$ (where δ is a partial function). The algorithm begins by unflagging all state pairs at **Step 0**. We have added this step just to ensure the accuracy of our results. Our approach to find all possible sets of λ-equivalent states of $\mathbf{G}$, is by flagging state pairs that are not λ-equivalent at **Steps 1-4**. It is notable that as the relation λ is symmetric, for states $q_1, q_2 \in Q$, if we flag state pair (q_1, q_2), we must also flag pair (q_2, q_1).

At **Step 1**, we flag every state pair such that one state of the pair is marked and the other state is unmarked, as marked and unmarked states are not λ-equivalent. **Step 2** is performed for every remaining unflagged state pair $(q_1, q_2) \in Q \times Q$. At **Step 2.1**, we look for some event $\sigma \in \Sigma$, such that σ is defined at exactly one state of the state pair, i.e. either at q_1 or q_2. If such σ exists, we flag state pairs (q_1, q_2) and (q_2, q_1) (**Step 2.1.1**). This is because q_1 and q_2 are not λ-equivalent, as they have different sets of σ transitions leaving them.

At **Step 3**, we initialize our boolean variable $flagging$ to $True$. **Step 4** is repeated as long as $flagging$ is $True$, i.e. there is a possibility to flag more state pairs. At **Step 4.1**, we set $flagging$ to $False$ by assuming that no more state pairs could be flagged in the current iteration. However, if we are able to flag more state pairs, then we set $flagging$ to $True$ again (**Step 4.2.1.2**) to repeat **Step 4** one more time. This is because there is a possibility that flagging might propagate from the recently flagged state pairs to some unflagged state pair(s) in the next iteration. However, if we do not flag any state pairs in the current iteration of **Step 4**, $flagging$ remains $False$, and while loop of **Step 4** terminates.

Step 4.2 is performed for every unflagged state pair $(q_1, q_2) \in Q \times Q$. At **Step 4.2.1**, we check to see if there is some event σ, such that σ is defined at both q_1 and q_2, and σ leads them to a state pair that is flagged. If so, we flag (q_1, q_2) and (q_2, q_1) (**Step 4.2.1.1**). The reason is that σ takes q_1 and q_2 to some destination states that are not λ-equivalent. Once **Step 4** finishes, the flagging process is complete and the state pairs that are not flagged correspond to states that are λ-equivalent.

At **Step 5**, we create a list $\mathbf{L}$, and add all non-singular (a state pair with distinct states) unflagged state pairs to $\mathbf{L}$. This means that if a state is only λ-equivalent to itself, then $(q, q) \in Q \times Q$ will not be added to $\mathbf{L}$. Only unflagged state pairs $(q_1, q_2) \in Q \times Q$, with $q_1 \neq q_2$, will be added to $\mathbf{L}$. At **Step 6**, we initialize our

Algorithm 6.1 Identify Distinct λ-Equivalent States of Generator **G**

Step 0: For every pair $(q_1, q_2) \in Q \times Q$, unflag (q_1, q_2).
Step 1: For every pair $(q_1, q_2) \in Q \times Q$, if $(q_1 \in Q_m \wedge q_2 \notin Q_m) \vee (q_1 \notin Q_m \wedge q_2 \in Q_m)$, then:
  Step 1.1: Flag $(q_1, q_2), (q_2, q_1)$.
Step 2: For every pair $(q_1, q_2) \in Q \times Q$ not flagged at Step 1:
  Step 2.1: For some $\sigma \in \Sigma$, if $(\delta(q_1, \sigma)! \wedge \neg\delta(q_2, \sigma)!) \vee (\neg\delta(q_1, \sigma)! \wedge \delta(q_2, \sigma)!)$, then:
    Step 2.1.1: Flag $(q_1, q_2), (q_2, q_1)$.
Step 3: Set $flagging := True$.
Step 4: While $(flagging)$:
  Step 4.1: Set $flagging := False$.
  Step 4.2: For every pair $(q_1, q_2) \in Q \times Q$ not flagged at Steps 1 and 2:
    Step 4.2.1: For some $\sigma \in \Sigma$ such that $\delta(q_1, \sigma)! \wedge \delta(q_2, \sigma)!$, if $(\delta(q_1, \sigma), \delta(q_2, \sigma))$ is flagged, then:
      Step 4.2.1.1: Flag $(q_1, q_2), (q_2, q_1)$.
      Step 4.2.1.2: Set $flagging := True$.
Step 5: Add all unflagged, non-singular pairs (no pairs $(q, q) \in Q \times Q$) to list **L**.
Step 6: Set $k := 0$.
Step 7: While $\mathbf{L} \neq \emptyset$:
  Step 7.1: Set $k := k + 1$.
  Step 7.2: Take a pair (q_1, q_2) from **L**. Create a new set $\mathbf{E}_k$ and add both states q_1 and q_2 of the pair to $\mathbf{E}_k$. Remove all occurrences of the pair (q_1, q_2) and (q_2, q_1) from **L**.
  Step 7.3: For every pair (q_1', q_2') in **L**, if the pair has exactly one state in common with $\mathbf{E}_k$, then add the uncommon state of the pair to $\mathbf{E}_k$. Remove all occurrences of the pair (q_1', q_2') and (q_2', q_1') from **L**. Then, repeat this step until no pair in **L** has exactly one state in common with $\mathbf{E}_k$.

counter variable k to 0, and increment it by 1 (**Step 7.1**) every time we construct a new set of λ-equivalent states, E_k.

At **Step 7**, we use the list **L** to form disjoint sets of λ-equivalent states in such a way that each state is exactly in one set, all states in the same set are λ-equivalent, and no two states from different sets are λ-equivalent. These sets will thus contain at least two (and possibly more) distinct λ-equivalent states that need to be combined. We use the *transitive property* (i.e. if $x \equiv y$ and $y \equiv z$, then $x \equiv z$) of the λ-equivalence relation to form these sets. **Step 7** is repeated until **L** becomes empty.

At **Step 7.2**, we create a new set $\mathbf{E}_k$ by removing one state pair (q_1, q_2) from **L**, and adding both states of the pair to $\mathbf{E}_k$. As these two states of the pair are λ-equivalent, they must be in the same set. We then remove all occurrences of (q_1, q_2) and (q_2, q_1) from **L**. This ensures that each state pair is added to only one set exactly

once, and guarantees that all sets of λ-equivalent states are disjoint.

At **Step 7.3**, we check to see if there exists a state pair (q'_1, q'_2) in $\mathbf{L}$ that has exactly one state in common with $\mathbf{E}_k$. If yes, this means the common state is λ-equivalent to all other states of $\mathbf{E}_k$. As two states of the pair are λ-equivalent, this step adds the uncommon state of the pair to $\mathbf{E}_k$ as well. This ensures that all states in the same set are λ-equivalent. As both states of this pair have now been added to the appropriate set, we remove all occurrences of (q'_1, q'_2) and (q'_2, q'_1) from $\mathbf{L}$. This step is repeated until there does not exist any state pair in $\mathbf{L}$ that has exactly one state in common with $\mathbf{E}_k$. It is notable that if no state of the pair is in common with $\mathbf{E}_k$, then the states of the pair are not λ-equivalent to the states of $\mathbf{E}_k$. In this case, they must not be added to $\mathbf{E}_k$, as only λ-equivalent states must be in the same set.

After **Step 7.3** is complete for set $\mathbf{E}_k$, there is no state pair in $\mathbf{L}$ that has one or more states in common with $\mathbf{E}_k$. For other state pairs that are in $\mathbf{L}$ but not λ-equivalent to the states of $\mathbf{E}_k$, we repeat **Step 7** and create new sets, as needed.

Upon completion, Algorithm 6.1 creates one or more disjoint sets of λ-equivalent states of the input TDES automaton $\mathbf{G}$, if $\mathbf{G}$ was not minimal. However, if $\mathbf{G}$ was already in its minimal form, our algorithm will flag all state pairs at **Steps 1-4**, as no two distinct states of $\mathbf{G}$ are λ-equivalent. In this case, there will be no non-singular (i.e. no pairs $(q, q) \in Q \times Q$) unflagged pairs to be added to list $\mathbf{L}$ at **Step 5**. As $\mathbf{L}$ is empty, **Step 7** is not executed and no sets of λ-equivalent states will be formed by the algorithm.

6.2.2 Construct a Minimal Automaton

A TDES automaton is said to be *minimal* (Definition 2.2.10) if it does not have two distinct states that are λ-equivalent. This means in order to obtain a minimal version of a non-minimal TDES, all distinct λ-equivalent states of the non-minimal automaton should be merged and replaced by a single "aggregate" state. This process is called *state aggregation* (Cassandras and Lafortune, 2008). For example, if the non-minimal TDES automaton $\mathbf{G}$ has $n > 1$ distinct λ-equivalent states $q_1, \ldots, q_n \in Q$, these n states should be replaced by a single aggregate state, say q, in the minimal TDES automaton, such that q behaves like $q_1, \ldots, q_n$. There can be one or more groups of distinct λ-equivalent states in the non-minimal automaton. The minimal automaton will have an aggregate state corresponding to each one of these groups.

Let TDES automaton $\mathbf{G}' = (Q', \Sigma, \delta', q'_o, Q'_m)$ be the minimum-state version of the non-minimal TDES automaton $\mathbf{G}$. Here, the state space Q' represents the smallest set of states after combining all states within each group of distinct λ-equivalent states of $\mathbf{G}$. Σ is the event set of $\mathbf{G}'$ and is same as the event set of $\mathbf{G}$. δ' is the resulting transition function, $q'_o \in Q'$ is the initial state, and $Q'_m \subseteq Q'$ is the set of marked states of the minimal automaton $\mathbf{G}'$. To clearly argue about the transitions of $\mathbf{G}$ and $\mathbf{G}'$, we will express transitions as a 3-tuple (as described in Section 2.2.1).

By utilizing the disjoint sets of distinct λ-equivalent states of $\mathbf{G}$ identified by

Algorithm 6.2 Construct Minimal TDES Automaton $\mathbf{G}'$ from $\mathbf{G}$

Step 1: $\mathbf{G}' := \mathbf{G}$, such that $Q' := Q, \Sigma := \Sigma, \delta' := \delta, q'_o := q_o, Q'_m := Q_m$.
Step 2: For every set $\mathbf{E}_k$ of distinct λ-equivalent states of $\mathbf{G}$:
  Step 2.1: For all $q \in \mathbf{E}_k$, remove q from Q'.
  Step 2.2: Add q' to Q', such that $q' \notin Q$ and $q' \notin Q'$.
  Step 2.3: If $q_o \in \mathbf{E}_k$, then $q'_o := q'$.
  Step 2.4: If $(\mathbf{E}_k \cap Q_m \neq \emptyset)$, then:
    Step 2.4.1: For all $q'' \in \mathbf{E}_k$, remove q'' from Q'_m.
    Step 2.4.2: Add q' to Q'_m.
  Step 2.5: For every transition $(q_1, \sigma, q_2) \in \delta'$:
    Step 2.5.1: If $q_1 \in \mathbf{E}_k$, then replace q_1 with q' in the transition triple in δ'.
    Step 2.5.2: If $q_2 \in \mathbf{E}_k$, then replace q_2 with q' in the transition triple in δ'.

Algorithm 6.1, Algorithm 6.2 presents steps for the iterative construction of minimal automaton $\mathbf{G}'$. The algorithm begins by copying the non-minimal automaton $\mathbf{G}$ to $\mathbf{G}'$. **Step 1** copies the state set Q to Q', event set Σ to Σ, transition function δ to δ', initial state q_o to q'_o, and the set of final states Q_m to Q'_m. At **Step 2**, we iteratively update the automaton structure of $\mathbf{G}'$ to make it minimal. This step is repeated for each set of λ-equivalent states $\mathbf{E}_k$, where $1 \leq k \leq t$ and $t \geq 1$ is the total number of sets of distinct λ-equivalent states formed by Algorithm 6.1.

Steps 2.1 and **2.2** merge all λ-equivalent states of set $\mathbf{E}_k$ and replace them with a single aggregate state in $\mathbf{G}'$. In other words, we remove all distinct λ-equivalent states of $\mathbf{E}_k$ from Q' and add one state, q', corresponding to $\mathbf{E}_k$ in Q'. It is important to make sure that the state label q' does not already exist in Q or Q'. At **Step 2.3**, we check to see if $\mathbf{E}_k$ contains the initial state of $\mathbf{G}$. If so, we make q' the initial state of $\mathbf{G}'$. The set of marked states of $\mathbf{G}'$ should include all aggregate states that correspond to sets that contain the marked states of $\mathbf{G}$. At **Step 2.4**, we determine if E_k contains any marked state. If so, all the λ-equivalent states of $\mathbf{E}_k$ are removed from Q'_m and replaced by the corresponding aggregate state q'.

At **Step 2.5**, we perform relabelling of λ-equivalent states of $\mathbf{E}_k$ in the transitions of δ'. This is required because all the distinct λ-equivalent states of set $\mathbf{E}_k$ have been replaced by a single aggregate state in $\mathbf{G}'$. Therefore, all the transitions, copied from δ to δ' at **Step 1**, that have these λ-equivalent states as their exit and/or entrance states should now have the corresponding aggregate state q' as their exit and/or entrance states respectively in δ'.

It is important to clarify here that **Step 2.5** does not add or remove any transitions from δ'. It just relabels the exit and/or entrance states of transitions in δ' by replacing the state labels of λ-equivalent states with their corresponding aggregate state labels. In other words, we can say that δ' is essentially δ, with the distinct λ-equivalent states of $\mathbf{G}$ being replaced by their corresponding aggregate states in $\mathbf{G}'$. In this way, every iteration of **Step 2** updates the automaton structure of $\mathbf{G}'$ to make it minimal.

It is noteworthy that Algorithm 6.2 does not make any changes to states that are identified by Algorithm 6.1 as not being λ-equivalent. At **Step 1**, Algorithm 6.2 copies the entire automaton structure of $\mathbf{G}$ to $\mathbf{G}'$. Thus, these non-λ-equivalent states and their transitions are a part of $\mathbf{G}'$ and remain unchanged throughout the execution of **Step 2**, as they do not belong to any set $\mathbf{E}_k$. Therefore, the automaton structure of $\mathbf{G}'$ with respect to these non-λ-equivalent states is the same as $\mathbf{G}$.

Once Algorithm 6.2 completes its execution, $\mathbf{G}'$ will have as few states as any automaton accepting the same closed and marked language as $\mathbf{G}$. In other words, $\mathbf{G}'$ now represents the minimal version of $\mathbf{G}$.

6.3 SD Properties with Minimal Automata

In this section, we discuss our equivalence results for all SD properties with respect to replacing $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ with a minimal version of $\boldsymbol{\mathcal{S}}$, referred to as $min(\boldsymbol{\mathcal{S}})$ (i.e. the result of applying Algorithms 6.1 and 6.2 to TDES $\boldsymbol{\mathcal{S}}$). We would need to do this if $\boldsymbol{\mathcal{S}}$ is not CS deterministic in its current form, and would use $min(\boldsymbol{\mathcal{S}})$ to address this (Section 6.3.1). If we make this change, we will need to re-evaluate our previous equivalence results from Chapter 5 with respect to $min(\boldsymbol{\mathcal{S}})$, and present new results for the property of CS deterministic supervisors with $min(\boldsymbol{\mathcal{S}})$.

In order to assess our previous results with respect to replacing $\boldsymbol{\mathcal{S}}$ by $min(\boldsymbol{\mathcal{S}})$, we first note that our equivalence results of the SD and $||_{SD}$ setting for language equivalence (Section 5.3), plant completeness (Section 5.4.1), **S**-singular prohibitable behaviour (Section 5.4.2), timed controllability (Section 5.4.3) and SD controllability (Section 5.4.4) are all proved in terms of the closed and/or marked languages of the involved TDES, and not the actual automaton structure. As the state space minimization Algorithms 6.1 and 6.2 produce minimal automaton with the same closed and marked languages as the original, i.e. $L(min(\boldsymbol{\mathcal{S}})) = L(\boldsymbol{\mathcal{S}})$ and $L_m(min(\boldsymbol{\mathcal{S}})) = L_m(\boldsymbol{\mathcal{S}})$, it thus follows that the results from Sections 5.3 and 5.4.1-5.4.4 remain valid if we replace $\boldsymbol{\mathcal{S}}$ by $min(\boldsymbol{\mathcal{S}})$. As a result, we do not need to adapt or reprove these results.

The only definition that is given in terms of the states of TDES automaton is the definition of ALF (Definition 2.3.10). Since we intend to minimize $\boldsymbol{\mathcal{S}}$ by merging various groups of distinct λ-equivalent states, the state space of $min(\boldsymbol{\mathcal{S}})$ will be different than the non-minimal $\boldsymbol{\mathcal{S}}$. This implies that while talking about $min(\boldsymbol{\mathcal{S}})$, we can no longer argue in terms of the states and state tuples of $\boldsymbol{\mathcal{S}}$. Hence, we will revisit our ALF equivalence results (discussed in Section 5.4.5) later in this section to make them work with $min(\boldsymbol{\mathcal{S}})$. However, we will first describe our new CS deterministic result with respect to $\boldsymbol{\mathcal{S}}$ and $min(\boldsymbol{\mathcal{S}})$.

6.3.1 CS Deterministic Supervisors

Our ultimate goal of generating the minimal version of $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ is to make it CS deterministic, if it is not already. However, minimizing $\boldsymbol{\mathcal{S}}$ alone does not guarantee

that $min(\boldsymbol{\mathcal{S}})$ will always be CS deterministic. We also need to make sure that our TDES supervisor $\mathbf{S}$ is SD controllable with $||_{SD}$ for TDES plant $\mathbf{G}$ to guarantee that $min(\boldsymbol{\mathcal{S}})$ is CS deterministic. This is proved in our next proposition (Proposition 6.2). In order to prove our desired result, we will use Proposition 6.1 from Wonham and Cai (2018). This proposition says that for a given TDES $\mathbf{G}$, two strings s and s' are nerode equivalent with respect to $L(\mathbf{G})$ and $L_m(\mathbf{G})$ if and only if both of these strings start from the initial state and take us to states that are λ-equivalent.

Proposition 6.1. (Wonham and Cai, 2018) For a generator $\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$, we have $(\forall s, s' \in \Sigma^*)\, \eta(y_o, s) \equiv \eta(y_o, s') \,(\text{mod } \lambda) \Leftrightarrow s \equiv_{L(\mathbf{G})} s' \wedge s \equiv_{L_m(\mathbf{G})} s'$.

Proposition 6.2. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G}) = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor, where $min(\mathbf{S} \,||_{SD}\, \mathbf{G})$ is constructed using Algorithms 6.1 and 6.2. If $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$, then $\boldsymbol{\mathcal{S}}$ is CS deterministic.

Proof. Assume initial conditions.
Must show: $\boldsymbol{\mathcal{S}}$ is CS deterministic. By Definition 3.4.5, it is sufficient to show:

$$(\forall s \in L(\boldsymbol{\mathcal{S}}) \cap L_{samp})\,(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\boldsymbol{\mathcal{S}}) \wedge Occu(s') = Occu(s'')] \Rightarrow$$
$$[ss' \equiv_{L(\boldsymbol{\mathcal{S}})} ss'' \;\wedge\; ss' \equiv_{L_m(\boldsymbol{\mathcal{S}})} ss'' \;\wedge\; \eta(y_o, ss') = \eta(y_o, ss'')]$$

We have $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G})$, and thus $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S} \,||_{SD}\, \mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}}) = L_m(\mathbf{S} \,||_{SD}\, \mathbf{G})$. (1)

Let $s \in L(\boldsymbol{\mathcal{S}}) \cap L_{samp}$, and let $s', s'' \in L_{conc}$. (2)

By (1), this implies: $s \in L(\mathbf{S} \,||_{SD}\, \mathbf{G}) \cap L_{samp}$ (3)

Assume: $ss', ss'' \in L(\boldsymbol{\mathcal{S}})$ and $Occu(s') = Occu(s'')$ (4)

By (1), this implies: $ss', ss'' \in L(\mathbf{S} \,||_{SD}\, \mathbf{G})$ (5)

Must show this implies: $ss' \equiv_{L(\boldsymbol{\mathcal{S}})} ss'' \;\wedge\; ss' \equiv_{L_m(\boldsymbol{\mathcal{S}})} ss'' \;\wedge\; \eta(y_o, ss') = \eta(y_o, ss'')$

We have that $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$. By (2-5), we note that all assumptions of *Point ii.2* of the SD controllability with $||_{SD}$ definition are satisfied.

$\Rightarrow ss' \equiv_{L(\mathbf{S} \,||_{SD}\, \mathbf{G})} ss'' \;\wedge\; ss' \equiv_{L_m(\mathbf{S} \,||_{SD}\, \mathbf{G})} ss''$

$\Rightarrow ss' \equiv_{L(\boldsymbol{\mathcal{S}})} ss'' \;\wedge\; ss' \equiv_{L_m(\boldsymbol{\mathcal{S}})} ss''$ *by* (1) (6)

$\Rightarrow \eta(y_o, ss') \equiv \eta(y_o, ss'') \,(\text{mod } \lambda)$ *by Proposition* 6.1

$\Rightarrow \eta(y_o, ss') = \eta(y_o, ss'')$ *by Definition* 2.2.10 *of minimal* $\boldsymbol{\mathcal{S}}$ (7)

By (6) and (7), we have thus shown that $\boldsymbol{\mathcal{S}}$ is CS deterministic. □

The above proposition tells us that as long as $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$, $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G})$ will be CS deterministic. We note that if $\mathbf{S} \,||_{SD}\, \mathbf{G}$ is already minimal, then Algorithms 6.1 and 6.2 will not make any changes, and $\mathbf{S} \,||_{SD}\, \mathbf{G} = min(\mathbf{S} \,||_{SD}\, \mathbf{G})$. This implies that $\mathbf{S} \,||_{SD}\, \mathbf{G}$ will be CS deterministic in this case. However, if $\mathbf{S} \,||_{SD}\, \mathbf{G}$ is not minimal, then we just take $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G})$, and we have a CS deterministic supervisor in both cases.

As discussed in Section 5.1, we intend to base our controllability and nonblocking verification results of the $||_{SD}$ setting on some of the existing results of the SD setting. To do this, we will need to construct an SD controller from $\boldsymbol{\mathcal{S}}$, a prerequisite of which is that $\boldsymbol{\mathcal{S}}$ must be CS deterministic. We now know that this will require considering $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is minimal, or $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ is $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is not minimal.

It is worth noting that in either case, both $\mathbf{S}\,||_{SD}\,\mathbf{G}$ and $min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ will have the same closed and marked languages, i.e. $L(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L(min(\mathbf{S}\,||_{SD}\,\mathbf{G}))$ and $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L_m(min(\mathbf{S}\,||_{SD}\,\mathbf{G}))$. This in turn means that all equivalence results that are solely related to the closed and marked languages remain applicable to both $\mathbf{S}\,||_{SD}\,\mathbf{G}$ and $min(\mathbf{S}\,||_{SD}\,\mathbf{G})$.

However, whether we use $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ or $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ will affect our argument about defining the states of $\boldsymbol{\mathcal{S}}$ in terms of the states of $\mathbf{S}$ and $\mathbf{G}$. Precisely, if $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, then state $y \in Y$ of $\boldsymbol{\mathcal{S}}$ will be a cross product of the states $x \in X$ of $\mathbf{S}$ and $q \in Q$ of $\mathbf{G}$, i.e. $y = (x, q)$. But this might not be true if we minimize the automaton $\mathbf{S}\,||_{SD}\,\mathbf{G}$ and use it as our $\boldsymbol{\mathcal{S}}$, i.e. $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$. Therefore, in our future proofs, whenever we want to argue in terms of the states of $\boldsymbol{\mathcal{S}}$, we will consider two ways of constructing $\boldsymbol{\mathcal{S}}$ separately.

6.3.2 ALF

In order to keep our ALF result of Section 5.4.5 valid for $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$, we need to show that the ALF property is preserved by the TDES minimization process. Before we prove this, we first give three utility propositions. Their goal is to allow us to convert key information (information about λ-equivalent states and their successors, converting transition in $\mathbf{G}'$ to transition in $\mathbf{G}$ and vice versa, and translating state reachability) from $\mathbf{G}' = min(\mathbf{G})$ to equivalent results about $\mathbf{G}$. This will be key in removing redundancies from later proofs to make them more compact.

Please note that for a non-minimal TDES $\mathbf{G}$, Algorithm 6.1 will create $t \geq 1$ sets of distinct λ-equivalent states (Definition 2.2.9) of $\mathbf{G}$. For each such set E_k $(1 \leq k \leq t)$, Algorithm 6.2 will replace all instances of state $q \in E_k$ from $\mathbf{G}'$. Each instance would be replaced by a unique aggregate state q', such that $q' \notin Q$ and $q' \notin Q'$ (before the replacement). As each E_k gets associated with a unique state q' by this replacement, we will refer to E_k as $E_{q'}$ in the following propositions to make it clear that the λ-equivalent states in $E_{q'}$ were replaced by q' when Algorithm 6.2 was executed and $\mathbf{G}'$ was constructed.

Proposition 6.3. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a non-minimal TDES and $\mathbf{G}' = min(\mathbf{G}) = (Q', \Sigma, \delta', q'_o, Q'_m)$ be the TDES constructed using Algorithms 6.1 and 6.2. Then:

i) $(\forall q_a, q_b \in Q)\, q_a \equiv q_b \,(\text{mod } \lambda) \Rightarrow (\forall s \in \Sigma^*)\, \delta(q_a, s)! \Rightarrow \delta(q_a, s) \equiv \delta(q_b, s) (\text{mod } \lambda)$

ii) $(\forall q', q'' \in Q')\, (\forall s \in \Sigma^*)\, \delta'(q', s) = q'' \Rightarrow (\exists\, q_a, q_b \in Q)\, \delta(q_a, s) = q_b \wedge (q_a = q' \vee q_a \in E_{q'}) \wedge (q_b = q'' \vee q_b \in E_{q''})$

iii) $(\forall q_a, q_b \in Q)\,(\forall s \in \Sigma^*)\,\delta(q_a, s) = q_b \Rightarrow (\exists q', q'' \in Q')\,\delta'(q', s) = q'' \wedge (q' = q_a \vee q_a \in E_{q'}) \wedge (q'' = q_b \vee q_b \in E_{q''})$

Proof. Assume initial conditions.

i) Show: $(\forall q_a, q_b \in Q)\,q_a \equiv q_b \,(\text{mod } \lambda) \Rightarrow (\forall s \in \Sigma^*)\,\delta(q_a, s)! \Rightarrow \delta(q_a, s) \equiv \delta(q_b, s) \,(\text{mod } \lambda)$

Let $q_a, q_b \in Q$. Assume: $q_a \equiv q_b \,(\text{mod } \lambda)$ (1)

Let $s \in \Sigma^*$. Assume: $\delta(q_a, s)!$

$\Rightarrow \delta(q_b, s)!$ *by* (1)

By Definition 2.2.9, it is sufficient to show Parts (1) and (2) below.

Part 1) Show: $(\forall s' \in \Sigma^*)\,\delta(\delta(q_a, s), s')! \Leftrightarrow \delta(\delta(q_b, s), s')!$

By definition of δ, it is sufficient to show: $(\forall s' \in \Sigma^*)\,\delta(q_a, ss')! \Leftrightarrow \delta(q_b, ss')!$

This follows automatically from (1), Point 1 of the λ-equivalence definition, and the fact that $s, s' \in \Sigma^*$ implies $ss' \in \Sigma^*$.

Part 2) Show: $(\forall s' \in \Sigma^*)\,\delta(\delta(q_a, s), s')! \wedge \delta(\delta(q_a, s), s') \in Q_m \Leftrightarrow \delta(\delta(q_b, s), s')! \wedge \delta(\delta(q_b, s), s') \in Q_m$

By definition of δ, it is sufficient to show:

$$(\forall s' \in \Sigma^*)\,\delta(q_a, ss')! \wedge \delta(q_a, ss') \in Q_m \Leftrightarrow \delta(q_b, ss')! \wedge \delta(q_b, ss') \in Q_m$$

This follows automatically from (1), Point 2 of the λ-equivalence definition, and the fact that $s, s' \in \Sigma^*$ implies $ss' \in \Sigma^*$.

By Parts (1) and (2), we have proven **Part (i)**.

ii) Show: $(\forall q', q'' \in Q')\,(\forall s \in \Sigma^*)\,\delta'(q', s) = q'' \Rightarrow (\exists\, q_a, q_b \in Q)\,\delta(q_a, s) = q_b \wedge (q_a = q' \vee q_a \in E_{q'}) \wedge (q_b = q'' \vee q_b \in E_{q''})$

Let $q', q'' \in Q'$ and $s \in \Sigma^*$. Assume: $\delta'(q', s) = q''$ (2)

To be consistent with Algorithm 6.2, we will treat $\delta \subseteq Q \times \Sigma \times Q$ as a relation, where $(q_1, \sigma, q_2) \in \delta$ if and only if $\delta(q_1, \sigma) = q_2$. Similarly, we will treat $\delta' \subseteq Q' \times \Sigma \times Q'$, where $(q_1', \sigma, q_2') \in \delta'$ if and only if $\delta'(q_1', \sigma) = q_2'$.

As $s \in \Sigma^*$, we have two cases: **(1)** $s = \epsilon$, or **(2)** $s \in \Sigma^+$.

Case 1) $s = \epsilon$

As $\delta'(q', s) = q''$ by (2), this implies $q' = q''$. (3)

We now have two cases: **(a)** $q' \in Q$, or **(b)** $q' \notin Q$.

Case 1.a) $q' \in Q$

We can then take $q_a = q_b = q' = q''$, and we immediately have $\delta(q_a, \epsilon) = \delta(q_a, s) = q_a = q_b$.

Case 1.b) $q' \notin Q$

$\Rightarrow \exists q_a \in E_{q'}$, as $E_{q'}$ is not empty by Algorithm 6.1.

We thus have $q_a \in Q$ (by Algorithms 6.1 and 6.2), and can set $q_b = q_a$, and we

have $\delta(q_a, \epsilon) = \delta(q_a, s) = q_a = q_b$.
As $q' = q''$ by (3), we have $E_{q'} = E_{q''}$, thus $q_b \in E_{q''}$ as $q_a = q_b$ and $q_a \in E_{q'}$.

By Cases (1.a) and (1.b), we have proven the desired condition for **Case (1)** ($s = \epsilon$).

Case 2) $s \in \Sigma^+$
Let $n = |s| \geq 1$.
$\Rightarrow (\exists \sigma_1, \sigma_2, \ldots, \sigma_n \in \Sigma)\, s = \sigma_1 \sigma_2 \ldots \sigma_n$
As $\delta'(q', s) = q''$ by (2), and $\delta' \subseteq Q' \times \Sigma \times Q'$, we can conclude there exists states $q'_1, q'_2, \ldots, q'_{n+1} \in Q'$ such that they form the following sequence of transitions in δ':

$$(q'_1, \sigma_1, q'_2), (q'_2, \sigma_2, q'_3), \ldots, (q'_n, \sigma_n, q'_{n+1}), \text{where } q'_1 = q' \text{ and } q'_{n+1} = q'' \tag{4}$$

By Algorithm 6.2, there exists states $q_1, q_2, \ldots, q_n, q_{n+1} \in Q$, and that $\delta \subseteq Q \times \Sigma \times Q$ contains the corresponding sequence of transitions:

$$\begin{gathered}(q_1, \sigma_1, q_2), (q_2, \sigma_2, q_3), \ldots, (q_n, \sigma_n, q_{n+1}), \\ \text{where for } 1 \leq i \leq n+1, q_i = q'_i \text{ or } q_i \in E_{q'_i}\end{gathered} \tag{5}$$

We thus have: $\delta(q_1, s) = q_{n+1}$
We can thus take $q_a = q_1$ and $q_b = q_{n+1}$, and we have $\delta(q_a, s) = q_b$, $q_a = q'$ or $q_a \in E_{q'}$, and $q_b = q''$ or $q_b \in E_{q''}$ by (4) and (5).
Case (2) complete.

By Cases (1) and (2), we have constructed suitable $q_a, q_b \in Q$ with properties:

$$\delta(q_a, s) = q_b \wedge (q_a = q' \vee q_a \in E_{q'}) \wedge (q_b = q'' \vee q_b \in E_{q''})$$

Part (ii) complete.

iii) Show: $(\forall q_a, q_b \in Q)\, (\forall s \in \Sigma^*)\, \delta(q_a, s) = q_b \Rightarrow (\exists q', q'' \in Q')\, \delta'(q', s) = q'' \wedge (q' = q_a \vee q_a \in E_{q'}) \wedge (q'' = q_b \vee q_b \in E_{q''})$
Let $q_a, q_b \in Q$ and $s \in \Sigma^*$. Assume: $\delta(q_a, s) = q_b$ (6)
As $s \in \Sigma^*$, we have two cases: **(1)** $s = \epsilon$, or **(2)** $s \in \Sigma^+$.

Case 1) $s = \epsilon$
As $\delta(q_a, s) = q_b$ by (6), this implies: $q_a = q_b$ (7)
We now have two cases: **(a)** $q_a \in Q'$, or **(b)** $q_a \notin Q'$.

Case 1.a) $q_a \in Q'$
We can thus take $q' = q'' = q_a = q_b$, and we immediately have $\delta'(q', \epsilon) = q' = q''$.

Case 1.b) $q_a \notin Q'$
By Algorithms 6.1 and 6.2, this implies: $(\exists q' \in Q')\, q_a \in E_{q'}$ (8)
As $q_a = q_b$ by (7), we also set $q'' = q'$, and we have $q'' \in Q'$ with $q_b \in E_{q''}$.
As $q' \in Q'$ by (8), we have $\delta'(q', \epsilon) = q' = q''$.

By Cases (1.a) and (1.b), we have proven the desired condition for **Case (1)**

$(s = \epsilon)$.

Case 2) $s \in \Sigma^+$

Let $n = |s| \geq 1$.

$\Rightarrow (\exists \sigma_1, \sigma_2, \ldots, \sigma_n \in \Sigma)\, s = \sigma_1 \sigma_2 \ldots \sigma_n$

As $\delta(q_a, s) = q_b$ by (6), and $\delta \subseteq Q \times \Sigma \times Q$, we can conclude there exists states $q_1, q_2, \ldots, q_{n+1} \in Q$ such that they form the following sequence of transitions in δ:

$$(q_1, \sigma_1, q_2), (q_2, \sigma_2, q_3), \ldots, (q_n, \sigma_n, q_{n+1}), \text{where } q_1 = q_a \text{ and } q_{n+1} = q_b \tag{9}$$

By Algorithm 6.2, there exists states $q'_1, q'_2, \ldots, q'_n, q'_{n+1} \in Q'$, and that $\delta' \subseteq Q' \times \Sigma \times Q'$ contains the corresponding sequence of transitions:

$$\begin{gathered}(q'_1, \sigma_1, q'_2), (q'_2, \sigma_2, q'_3), \ldots, (q'_n, \sigma_n, q'_{n+1}), \\ \text{where for } 1 \leq i \leq n+1, q'_i = q_i \text{ or } q_i \in E_{q'_i}\end{gathered} \tag{10}$$

We thus have: $\delta'(q'_1, s) = q'_{n+1}$

We can thus take $q' = q'_1$, and $q'' = q'_{n+1}$, and by (9) and (10) we have:

$$(q', q'' \in Q') \wedge (\delta'(q', s) = q'') \wedge (q' = q_a \vee q_a \in E_{q'}) \wedge (q'' = q_b \vee q_b \in E_{q''})$$

Case (2) complete.

By Cases (1) and (2), we have constructed suitable $q', q'' \in Q'$ with the desired properties.

Part (iii) complete.

By Parts (i)-(iii), we conclude that Points (i-iii) of the proposition are satisfied. □

Proposition 6.4. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a non-minimal TDES and $\mathbf{G}' = min(\mathbf{G}) = (Q', \Sigma, \delta', q'_o, Q'_m)$ be the TDES constructed using Algorithms 6.1 and 6.2. Then for $q', q'' \in Q'$ and $s \in \Sigma^*$ such that $\delta'(q', s) = q''$, the following properties hold:
i) $q', q'' \in Q \Rightarrow \delta(q', s) = q''$ **ii)** $q', q'' \notin Q \Rightarrow (\forall q_1 \in E_{q'})\, (\exists q_2 \in E_{q''})\, \delta(q_1, s) = q_2$
iii) $[q' \notin Q \wedge q'' \in Q] \Rightarrow (\forall q \in E_{q'})\, \delta(q, s) = q''$
iv) $[q' \in Q \wedge q'' \notin Q] \Rightarrow (\exists q \in E_{q''})\, \delta(q', s) = q$

Proof. Assume initial conditions.

Let $q', q'' \in Q'$, and $s \in \Sigma^*$. Assume: $\delta'(q', s) = q''$

By Proposition 6.3(ii), we can conclude:

$$(\exists q_a, q_b \in Q)\, \delta(q_a, s) = q_b \wedge (q_a = q' \vee q_a \in E_{q'}) \wedge (q_b = q'' \vee q_b \in E_{q''}) \tag{1}$$

We will now show that Points (i-iv) are satisfied.

i) Show: $q', q'' \in Q \Rightarrow \delta(q', s) = q''$

Assume: $q', q'' \in Q$

It thus follows by Algorithm 6.2 that both q' and q'' are λ-equivalent only to themselves, and we can thus conclude by (1) that $q_a = q'$ and $q_b = q''$.

$\Rightarrow \delta(q', s) = q''$ *by* (1)

Part (i) complete.

ii) Show: $q', q'' \notin Q \Rightarrow (\forall q_1 \in E_{q'})\,(\exists q_2 \in E_{q''})\,\delta(q_1, s) = q_2$
Assume: $q', q'' \notin Q$ (2)
By (1), we have: $(q_a = q' \vee q_a \in E_{q'}) \wedge (q_b = q'' \vee q_b \in E_{q''})$
$\Rightarrow q_a \in E_{q'}$ and $q_b \in E_{q''}$ *by* (2) *and Algorithm* 6.2
Let $q_1 \in E_{q'}$.
As $\delta(q_a, s) = q_b$ by (1), and $q_a \in E_{q'}$, it follows that $\delta(q_1, s)!$.
By Proposition 6.3(i), we have: $\delta(q_a, s) \equiv \delta(q_1, s) (\text{mod } \lambda)$
$\Rightarrow \delta(q_1, s) \in E_{q''}$ *as* $q_b \in E_{q''}$
We can thus take $q_2 = \delta(q_1, s)$, and we have $q_1 \in E_{q'}, q_2 \in E_{q''}$, and $\delta(q_1, s) = q_2$, as required.
Part (ii) complete.

iii) Show: $[q' \notin Q \wedge q'' \in Q] \Rightarrow (\forall q \in E_{q'})\,\delta(q, s) = q''$
Assume: $q' \notin Q \wedge q'' \in Q$ (3)
By (1) and Algorithm 6.2, we can conclude: $q_a \in E_{q'}$ and $q_b = q''$
$\Rightarrow \delta(q_a, s) = q''$ *by* (1) (4)
Let $q \in E_{q'}$.
As $\delta(q_a, s) = q''$ and $q_a \in E_{q'}$, it follows that $\delta(q, s)!$.
By Proposition 6.3(i), we have: $\delta(q_a, s) \equiv \delta(q, s) (\text{mod } \lambda)$
As $q'' \in Q$ by (3), it follows by Algorithm 6.2 that q'' is only λ-equivalent to itself.
$\Rightarrow \delta(q, s) = q''$ *by* (4)
We thus have $q \in E_{q'}$ and $\delta(q, s) = q''$, as required.
Part (iii) complete.

iv) Show: $[q' \in Q \wedge q'' \notin Q] \Rightarrow (\exists q \in E_{q''})\,\delta(q', s) = q$
Assume: $q' \in Q \wedge q'' \notin Q$
By (1) and Algorithm 6.2, we can conclude: $q_a = q', q_b \in E_{q''}$ and $\delta(q', s) = q_b$
We can thus take $q = q_b$, and we have $q \in E_{q''}$ with $\delta(q', s) = q$, as required.
Part (iv) complete.

By Parts (i)-(iv), we conclude that Points (i-iv) of the proposition are satisfied. □

Proposition 6.5. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a non-minimal TDES and $\mathbf{G}' = min(\mathbf{G}) = (Q', \Sigma, \delta', q'_o, Q'_m)$ be the TDES constructed using Algorithms 6.1 and 6.2. Then for $q' \in Q'_r$, the following properties hold: **(i)** $q' \in Q \Rightarrow q' \in Q_r$, and **(ii)** $q' \notin Q \Rightarrow (\exists q \in E_{q'})\, q \in Q_r$.

Proof. Let $q' \in Q'_r$ and assume initial conditions.
As $q' \in Q'_r$, we have: $(\exists s \in \Sigma^*)\,\delta'(q'_o, s) = q'$

i) Show: $q' \in Q \Rightarrow q' \in Q_r$

Assume: $q' \in Q$ (1)

We have two cases: **(1)** $q'_o \in Q$, or **(2)** $q'_o \notin Q$.

Case 1) $q'_o \in Q$
$\Rightarrow q'_o, q' \in Q$ *by* (1)
By Proposition 6.4(i), we have: $\delta(q'_o, s) = q'$ (2)
As $q'_o \in Q$, we have: $q'_o = q_o$ *by Steps* 1 *and* 2.3 *of Algorithm* 6.2
$\Rightarrow \delta(q_o, s) = q'$ *by* (2)
$\Rightarrow q' \in Q_r$

Case 2) $q'_o \notin Q$
$\Rightarrow q'_o \notin Q$ and $q' \in Q$ *by* (1)
By Proposition 6.4(iii), we have: $(\forall q \in E_{q'_o})\, \delta(q, s) = q'$
As $q_o \in E_{q'_o}$ by Algorithm 6.2, we thus have: $\delta(q_o, s) = q'$
$\Rightarrow q' \in Q_r$

By Cases (1) and (2), we have $q' \in Q_r$, as required.
Part (i) complete.

ii) Show: $q' \notin Q \Rightarrow (\exists q \in E_{q'})\, q \in Q_r$

Assume: $q' \notin Q$ (3)

We have two cases: **(1)** $q'_o \in Q$, or **(2)** $q'_o \notin Q$.

Case 1) $q'_o \in Q$
$\Rightarrow q'_o \in Q$ and $q' \notin Q$ *by* (3)
By Proposition 6.4(iv), we have: $(\exists q \in E_{q'})\, \delta(q'_o, s) = q$ (4)
As $q'_o \in Q$, we have: $q'_o = q_o$ *by Steps* 1 *and* 2.3 *of Algorithm* 6.2
$\Rightarrow \delta(q_o, s) = q$ *by* (4)
$\Rightarrow q \in Q_r$

Case 2) $q'_o \notin Q$
$\Rightarrow q'_o, q' \notin Q$ *by* (3)
By Proposition 6.4(ii), we have: $(\forall q_1 \in E_{q'_o})\, (\exists q \in E_{q'})\, \delta(q_1, s) = q$
As $q_o \in E_{q'_o}$ by Algorithm 6.2, we thus have: $\delta(q_o, s) = q$
$\Rightarrow q \in Q_r$

By Cases (1) and (2), we have constructed $q \in E_{q'}$ with $q \in Q_r$.
Part (ii) complete.

By Parts (i) and (ii), we conclude that Points (i-ii) of the proposition are satisfied. □

Now we will present our main ALF result. The theorem given below proves that if

a non-minimal TDES having a finite state space is ALF, then the minimal version of this TDES will also be ALF. This will allow us to use our ALF result of Section 5.4.5 whether $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ or $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$.

Theorem 6.1. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a TDES with finite state space. Let TDES $\mathbf{G}' = min(\mathbf{G}) = (Q', \Sigma, \delta', q'_o, Q'_m)$ be the minimal version of $\mathbf{G}$ that is constructed using Algorithms 6.1 and 6.2. If $\mathbf{G}$ is ALF, then $\mathbf{G}'$ is ALF.

Proof. Assume initial conditions.

Assume $\mathbf{G}$ is ALF and has a finite state space. (1)

If $\mathbf{G}$ is minimal, then $\mathbf{G} = \mathbf{G}'$, and it follows immediately that $\mathbf{G}'$ is ALF.

We now consider the case that $\mathbf{G}$ is non-minimal.

To show that $\mathbf{G}'$ is ALF, it is sufficient to show: $(\forall q' \in Q'_r)(\forall s \in \Sigma^+_{act})\,\delta'(q', s) \neq q'$

We will use proof by contradiction to show our desired result.

Assume $\mathbf{G}'$ is not ALF.

$\Rightarrow (\exists q' \in Q'_r)\,(\exists s \in \Sigma^+_{act})\,\delta'(q', s) = q'$ (2)

We will now show this implies $\mathbf{G}$ is not ALF, contradicting (1).

To do this, we will need to construct $q \in Q_r$ and $s' \in \Sigma^+_{act}$ such that $\delta(q, s') = q$.

We have two cases: **(i)** $q' \in Q$, or **(ii)** $q' \notin Q$.

Case i) $q' \in Q$

As $\delta'(q', s) = q'$ by (2), we apply Proposition 6.4(i) and conclude: $\delta(q', s) = q'$

As $q' \in Q'_r$ by (2), and $q' \in Q$, we apply Proposition 6.5(i) and conclude: $q' \in Q_r$

We thus take $q = q'$, $s' = s$, and we have $q \in Q_r$, $s' \in \Sigma^+_{act}$ by (2), and $\delta(q, s') = q$, thus contradicting $\mathbf{G}$ being ALF.

Case ii) $q' \notin Q$

As $\delta'(q', s) = q'$ by (2), we apply Proposition 6.4(ii) and conclude:

$$(\forall q_1 \in E_{q'})\,(\exists q_2 \in E_{q'})\,\delta(q_1, s) = q_2 \tag{3}$$

As $q' \in Q'_r$ by (2), we apply Proposition 6.5(ii) and conclude: $(\exists q_1 \in E_{q'})\,q_1 \in Q_r$ (4)

Applying this to (3), we have: $(\exists q_2 \in E_{q'})\,\delta(q_1, s) = q_2$

As $q_2 \in E_{q'}$, we could apply (3) to q_2 and so on to create a chain of transitions.

Let $n = |E_{q'}|$. As $E_{q'} \subseteq Q$ by Algorithm 6.1 and the fact that Q is finite by (1), we have $n < \infty$. (5)

Starting with q_1, we could apply (3) repeatedly n times and construct a chain of transitions in δ as: $q_1 \xrightarrow{s} q_2 \xrightarrow{s} \cdots \xrightarrow{s} q_n \xrightarrow{s} q_{n+1}$, where for $1 \leq i \leq n+1, q_i \in E_{q'}$. (6)

As $q_1 \in Q_r$ by (4), it follows that each $q_i \in Q_r$. (7)

We note that as $n < \infty$ by (5), after the n^{th} transition $(q_1 \to q_n)$, it is possible that each q_i was a distinct state in $E_{q'}$, but the transition $\delta(q_n, s) = q_{n+1}$ must then involve

a duplicate state. (8)

This means states $q_1, \ldots, q_{n+1}$ must contain two duplicate states.

Let $1 \leq i < n+1$ be the index for the first duplicate state, and let $1 < j \leq n+1$ be the index for the second occurrence of this state (i.e. $q_i = q_j$). (9)

Let $k = j - i$. This is the number of transitions separating the two states (i.e. for q_2 and q_1, 2-1=1).

We then take $q = q_i$ and $s' = s \ldots s$. We thus have $q \in Q_r$ by (7), $s' \in \Sigma^+_{act}$ as $s \in \Sigma^+_{act}$ by (2), and $\delta(q, s') = q$ by (6), (8) and (9), which contradicts $\mathbf{G}$ being ALF.

By **Cases (i)** and **(ii)**, we have proven that $\mathbf{G}$ is not ALF, which contradicts (1).

As our assumption that $\mathbf{G}'$ is not ALF caused a contradiction, we thus conclude that $\mathbf{G}'$ is ALF. □

In the SD setting, one of the preconditions of the SD controllability and nonblocking verification results is that the closed-loop system formed by synchronizing TDES plant and supervisor models using the synchronous product will not "stop the clock". In Leduc *et al.* (2014), this has been proven using the following proposition.

Proposition 6.6. (Leduc *et al.*, 2014) If TDES plant $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ and TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ both have finite state spaces, $\mathbf{G}$ has proper time behaviour, $\mathbf{S} \,||\, \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ is ALF, and $\mathbf{S}$ is timed controllable for $\mathbf{G}$, then $(\forall y \in Y_r)\,(\exists s \in \Sigma^*_{act})\ \eta(y, s\tau)!$.

Since we wish to make our $\boldsymbol{\mathcal{S}}$ eligible to be used as the supervisor of the SD setting, we need to show that this result is satisfied by our $\boldsymbol{\mathcal{S}}$ as well. As there are two possible ways to construct $\boldsymbol{\mathcal{S}}$, i.e. $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ or $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G})$, below we show this result with respect to both cases.

Proposition 6.7. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, and TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}}' = \min(\boldsymbol{\mathcal{S}}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 6.1 and 6.2. Let the closed-loop system be $\boldsymbol{\mathcal{S}}' \,||\, \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$. If both $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces, $\mathbf{G}$ has proper time behaviour, $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}}$ is ALF, then:

$$(\forall y \in Y_r)\,(\exists s \in \Sigma^*_{act})\ \eta(y, s\tau)!$$

Proof. Assume initial conditions.

To obtain our desired result, we will show that the assumptions of Proposition 6.6 are satisfied. It is notable that if $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ is already minimal, then $\boldsymbol{\mathcal{S}} = \boldsymbol{\mathcal{S}}'$. Thus, we need to prove conditions for both $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}'$.

First, we have that $\mathbf{G}$ has finite state space and proper time behaviour. (1)

Next, we have that both $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces.

$\Rightarrow \boldsymbol{\mathcal{S}}$ has a finite state space (2)

$\Rightarrow \boldsymbol{\mathcal{S}}' = min(\boldsymbol{\mathcal{S}})$ has a finite state space *by Algorithms* 6.1 *and* 6.2 (3)

By our initial assumptions, $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$.
$\Rightarrow \boldsymbol{\mathcal{S}}$ is timed controllable for $\mathbf{G}$ *by Proposition* 5.6 (4)
As state space minimization does not change the automaton's closed behaviour, we have $L(\boldsymbol{\mathcal{S}}') = L(\boldsymbol{\mathcal{S}})$. As timed controllability is a language based property, this implies that $\boldsymbol{\mathcal{S}}'$ is timed controllable for $\mathbf{G}$. (5)
We have that $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF.
$\Rightarrow \boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ is ALF *by Proposition* 5.8 (6)
We now have two cases: **(i)** $\boldsymbol{\mathcal{S}}$ is minimal, or **(ii)** $\boldsymbol{\mathcal{S}}$ is non-minimal.

Case i) $\boldsymbol{\mathcal{S}}$ is minimal
This means $\boldsymbol{\mathcal{S}} = \boldsymbol{\mathcal{S}}' = min(\boldsymbol{\mathcal{S}})$ as Algorithms 6.1 and 6.2 will make no changes. We can thus use properties for $\boldsymbol{\mathcal{S}}$.
By (1), (2), (4) and (6), all assumptions of Proposition 6.6 are satisfied.

Case ii) $\boldsymbol{\mathcal{S}}$ is non-minimal
This means $\boldsymbol{\mathcal{S}} \neq \boldsymbol{\mathcal{S}}' = min(\boldsymbol{\mathcal{S}})$, so we must use the results for $\boldsymbol{\mathcal{S}}'$.
As $\boldsymbol{\mathcal{S}}$ is ALF, we can conclude by Theorem 6.1 that $\boldsymbol{\mathcal{S}}'$ is ALF.
$\Rightarrow \boldsymbol{\mathcal{S}}'\,||\,\mathbf{G}$ is ALF *by Proposition* 5.8 (7)
By (1), (3), (5) and (7), all assumptions of Proposition 6.6 are satisfied.

By **Cases (i)** and **(ii)**, we can apply Proposition 6.6 and conclude:

$$(\forall y \in Y_r)\,(\exists s \in \Sigma^*_{act})\;\eta(y, s\tau)! \qquad \square$$

Chapter 7

Equivalence of SD Controllers

In this chapter, we present our final set of results with respect to establishing equivalence between the SD and our $||_{SD}$ setting. Specifically, this chapter proves the output equivalence between the two SD controllers that are generated by translating CS deterministic TDES supervisors $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}$ of the $||_{SD}$ and SD settings respectively. In other words, we will show that the two SD controllers generate the same sequence of outputs in response to a given valid (possible according to system model) sequence of inputs.

We begin this chapter by stating some preliminary definitions that we have defined for our $||_{SD}$ setting. Then, we present our supporting propositions that will help us in proving our final result that the two SD controllers translated from $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}$ produce the same output information for the same valid input sequence with respect to the closed-loop behaviour.

Please note that the functions and notation used in this chapter have already been introduced in Chapter 3. We will provide a brief introduction, but recommend the reader to refresh the details (specifically Sections 3.6 and 3.7) before reading this section.

7.1 Preliminary Definitions

In this section, we present some definitions that we have adapted from Wang (2009) to define the concepts related to SD controllers in our $||_{SD}$ setting.

One of the goals of devising the $||_{SD}$ setting is to liberate the software and hardware practitioners from designing and implementing a potentially intricate supervisor in the SD setting. Rather, we want them to design and implement a much simpler TDES supervisor $\mathbf{S}$ in the $||_{SD}$ setting. In order to do that, it is important to show that the SD controller generated by translating CS deterministic supervisor $\mathbf{S}$ in our $||_{SD}$ setting is output equivalent to the SD controller that is obtained by translating CS deterministic supervisor $\boldsymbol{\mathcal{S}}$ (possibly $min(\boldsymbol{\mathcal{S}})$) of the SD setting. In other words,

we wish to prove that whether practitioners physically implement $\mathbf{S}$ or $\boldsymbol{\mathcal{S}}$, they are going to achieve the same physical control action with respect to a given TDES plant $\mathbf{G}$. This result will also be essential for the proofs of Chapter 8 so we can use the SD controller for $\mathbf{S}$ and apply it for proofs using $\boldsymbol{\mathcal{S}}$.

It is important to clarify that we do not require the two SD controllers to be identical. We only wish to demonstrate that they produce the same enablement and forcing information for a given plant $\mathbf{G}$ for valid input sequences.

First, we provide a definition for valid input sequences with respect to the closed-loop behaviour. It is worth-mentioning that the definition given below is generic with respect to forming the closed-loop system, $\mathbf{G}_{cl}$. By this we mean that our definition is independent of the operator that is used to form $\mathbf{G}_{cl}$. $\mathbf{G}_{cl}$ could be constructed by synchronizing TDES plant and supervisor models using the $||_{SD}$ operator, the synchronous product, the meet or the product operator. For this definition, we are only interested in the language obtained as a result of combining the plant and supervisor models, i.e. $L(\mathbf{G}_{cl})$. Our goal is to ensure that whichever operator we use to obtain $L(\mathbf{G}_{cl})$, our definition will remain applicable and valid.

Definition 7.1.1. For TDES plant $\mathbf{G} = (Q, \Sigma_{\mathbf{G}}, \delta, q_o, Q_m)$ and CS deterministic TDES supervisor $\mathbf{S} = (X, \Sigma_{\mathbf{S}}, \xi, x_o, X_m)$, let $\mathbf{G}_{cl} = (Y, \Sigma, \eta, y_o, Y_m)$ be the closed-loop system constructed by synchronizing $\mathbf{S}$ and $\mathbf{G}$. For system event set Σ, with canonical event mapping function γ_g, global input vector $\mathbf{i}_g$, and activity event set Σ_{act}, a canonical input sequence $\{\mathbf{i}_g(k)\}$ is said to be *input valid for* $L(\mathbf{G}_{cl})$, if:

$$(\forall k \in \{1, 2, \ldots\})\,(\exists\, s_1, s_2, \ldots, s_k \in L_{conc})\,[s_1 s_2 \ldots s_k \in L(\mathbf{G}_{cl})] \wedge$$
$$[(\forall n \in \{1, 2, \ldots, k\})\,(\forall \sigma \in \Sigma_{act})\, i_{g,\gamma_g(\sigma)}\,(n) = 1 \text{ iff } \sigma \in Occu(s_n)]$$

In the above definition, γ_g is a bijective map that associates each $\sigma \in \Sigma_{act}$ with a unique element of input vector $\mathbf{i}_g = [i_{g,0}, i_{g,1}, \ldots, i_{g,v-1}]$ $(v = |\Sigma_{act}|)$, $\{\mathbf{i}_g(k)\} = \{i_g(1), i_g(2), \ldots\}$ is a sequence of input vectors taken at different sampling instances, $i_{g,\gamma_g(\sigma)}\,(n)$ is element $i_{g,\gamma_g(\sigma)}$ (the element γ_g associate with σ) of the n^{th} vector in sequence $\{\mathbf{i}_g(k)\}$, and $\sigma \in Occu(s_n)$ means the string s_n contains event σ. For more information, see Sections 3.4, 3.6 and 3.7.

Essentially, in this definition, we require the input sequence $\{\mathbf{i}_g(k)\}$ to correspond to a sequence of concurrent strings that our closed-loop system $\mathbf{G}_{cl}$ will accept. This is necessary as the TDES supervisor to SD controller translation method (Section 3.7) only dictates outputs for these inputs and leaves the outputs for other inputs unspecified. This means controllers could differ for input sequences that are not possible in the system.

Before we proceed to our next definition, please note that in our $||_{SD}$ setting, we construct our closed-loop system as $\mathbf{S}\,||_{SD}\,\mathbf{G}$, whereas the SD setting constructs the closed-loop system as $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$. Because of our language equivalence results (Section 5.3), we know that both closed-loop systems have the same closed and marked languages. This implies that input sequences that represent valid input strings in the behaviour of the two closed-loop systems will be the same.

In order to prove our controller equivalence results, we wish to prove that two SD controllers, $\mathbf{C}_1$ and $\mathbf{C}_2$, are output equivalent with respect to the closed-loop behaviour for plant $\mathbf{G}$ and supervisors $\mathbf{S}_1$ and $\mathbf{S}_2$, where $\mathbf{C}_1$ is constructed from $\mathbf{S}_1$ and $\mathbf{C}_2$ is constructed from $\mathbf{S}_2$. For this definition, we are assuming that the two closed-loop systems, represented by TDES $\mathbf{G}_{cl,1}$ and $\mathbf{G}_{cl,2}$, have the same closed languages, i.e. $L(\mathbf{G}_{cl,1}) = L(\mathbf{G}_{cl,2})$. Before we present our formal definition, three points are notable and worth elaborating.

1. This definition is independent of the synchronization operators that are used to form the closed-loop systems. As long as the closed-loop behaviour of the two systems is the same, the definition remains applicable and valid, and the choice of synchronization operator(s) is trivial.

2. This definition is stated with respect to the closed-loop behaviour of the two systems, and not in terms of the actual system automata. This is because the TDES representation of the two closed-loop systems might not be exactly the same due to different state labels or if one TDES is non-minimal, despite them having the same closed behaviour.

3. As the closed behaviour of the two closed-loop systems is the same, i.e. $L(\mathbf{G}_{cl,1}) = L(\mathbf{G}_{cl,2})$, using either $L(\mathbf{G}_{cl,1})$ or $L(\mathbf{G}_{cl,2})$ will not make any difference because we are eventually referring to the same language. However, to be clear and avoid any ambiguity, instead of using either one of these two labels, we will refer to this language using a more generic label $L(\mathbf{G}_{cl})$, without any loss of generality.

Definition 7.1.2. For TDES plant $\mathbf{G} = (Y, \Sigma_{\mathbf{G}}, \delta, y_o, Y_m)$, let $\mathbf{S}_j = (X_j, \Sigma_j, \xi_j, x_{o,j}, X_{m,j})$ $(j = 1, 2)$ be two CS deterministic TDES supervisors. Let $\mathbf{G}_{cl,j}$ be the closed-loop system formed by synchronizing $\mathbf{G}$ and $\mathbf{S}_j$. Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be control equivalent for $\mathbf{G}$, i.e. $L(\mathbf{G}_{cl}) = L(\mathbf{G}_{cl,1}) = L(\mathbf{G}_{cl,2})$. For system event set Σ, with canonical event mapping function γ_g, and activity event set Σ_{act}, let $\mathbf{C}_j = (I_j, Z_j, Q_j, \Omega_j, \Phi_j, \mathbf{q}_{res,j})$ be the SD controller constructed from $\mathbf{S}_j$. Let r_j be the number of output variables for a vector in Z_j, and η_j be the output event mapping function for $\mathbf{C}_j$. $\mathbf{C}_1$ and $\mathbf{C}_2$ are said to be *output equivalent with respect to the closed-loop behaviour* $L(\mathbf{G}_{cl})$ if, for any canonical input sequence $\{\mathbf{i}_g(k)\}$ that is input valid for $L(\mathbf{G}_{cl})$ and induced output $\mathbf{z}_j(k') = [z_{j,0}(k'), z_{j,1}(k'), \ldots, z_{j,r_j-1}(k')] \in Z_j$ at time $k' = \{0, 1, 2, \ldots\}$, the following conditions are satisfied:

1. $r_1 = r_2$

2. $(\forall\, 0 \leq i < r_1)\, \eta_1^{-1}(i) = \eta_2^{-1}(i)$

3. $(\forall k' \in \{0, 1, 2, \ldots\})\, \mathbf{z}_1(k') = \mathbf{z}_2(k')$

In the above definition, $\mathbf{z}_j(k')$ is the current output vector for controller $\mathbf{C}_j$, at time k'. Also, η_j is a bijective map that associates each $\sigma \in \Sigma_{hib} \cap \Sigma_j$ with a unique element in $\mathbf{z}_j(k')$ in a way that respects the event ordering of γ_g. See Section 3.7.1 for details.

In this definition, Point 1 requires that output vectors of the two controllers must

be the same size, i.e. they must have same number of output variables. Point 2 enforces the condition that the two output vectors must have the same prohibitable events stored in exactly the same order/sequence. Finally, Point 3 imposes the constraint that for any value of k', the two output vectors must have the same enablement information, i.e. one controller should enable a prohibitable event if and only if the other does. This means that the two controllers must agree with respect to the enablement of prohibitable events at the reset state, and must continue to agree in the future as well.

The above definition gives us a way to compare the output information of the SD controller translated from supervisor $\mathbf{S}$ in the $||_{SD}$ setting to the SD controller translated from $\boldsymbol{\mathcal{S}}$ (or $min(\boldsymbol{\mathcal{S}})$, if $\boldsymbol{\mathcal{S}}$ is not minimal) in the SD setting. If the two controllers are output equivalent with respect to the shared closed-loop behaviour, then they will assert the same enablement and forcing action on plant $\mathbf{G}$. This will allow us to implement the controller for $\mathbf{S}$, but apply the controllability and nonblocking results of the SD setting to $\boldsymbol{\mathcal{S}}$ and this controller.

7.2 Supporting Propositions

In this section, we introduce two supporting propositions that will be used in the next section to prove our main result that the corresponding SD controllers generated in the SD and $||_{SD}$ settings are output equivalent. Please recall that as per our assumptions (Section 5.2), all TDES are deterministic automata.

To convert a TDES supervisor to an SD controller, it must be CS deterministic. As discussed in Section 6.3.1, we might need to minimize the TDES supervisor $\boldsymbol{\mathcal{S}}$ in order to make it CS deterministic. As λ-equivalent states (Definition 2.2.9) are combined during the state minimization process, this will make it complicated to compare $\boldsymbol{\mathcal{S}}$ to our supervisor $\mathbf{S}$ of the $||_{SD}$ setting. As a result, in the proofs presented in this section, we will refer to the sets of distinct λ-equivalent states, labelled as $E_k(|E_k| \geq 2)$, that are created by Algorithm 6.1 during the minimization process. We will refer to E_k as $E_{q'}$, where q' is the aggregate state label associated with E_k by Algorithm 6.2. Please refer to Section 6.3.2 for details.

In Proposition 7.1 given below, X_{samp} (Definition 3.4.2) is the set of sampled states for TDES supervisor $\mathbf{S}$. These are the states of $\mathbf{S}$ that are reached from the initial state by a sampled string (Definition 3.4.1). The prohibited action function ζ (Definition 3.7.2) is associated with a specific supervisor, and maps sampled states of the supervisor to the set of prohibitable events enabled at these states.

Proposition 7.1. Let $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a non-minimal TDES supervisor and $\mathbf{S}' = min(\mathbf{S}) = (X', \Sigma, \xi', x'_o, X'_m)$ be the minimal TDES constructed using Algorithms 6.1 and 6.2. Let ζ be the prohibited action function for $\mathbf{S}$ and ζ' be the prohibited action function for $\mathbf{S}'$. Then, for $x_1, x_2 \in X$ and $x' \in X'$, the following properties hold:

i) $x_1 \equiv x_2 \,(\text{mod } \lambda) \Rightarrow (\forall \sigma \in \Sigma)\, \xi(x_1, \sigma)! \Rightarrow \xi(x_2, \sigma)!$

ii) $[x_1 \equiv x_2 \,(\text{mod } \lambda) \wedge x_1, x_2 \in X_{samp}] \Rightarrow \zeta(x_1) = \zeta(x_2)$

iii) $[x' \notin X \wedge x' \in X'_{samp}] \Rightarrow (\forall x \in E_{x'} \cap X_{samp})\, \zeta(x) = \zeta'(x')$

iv) $[x' \in X \wedge x' \in X'_{samp}] \Rightarrow x' \in X_{samp} \wedge \zeta(x') = \zeta'(x')$

Proof. Let $x_1, x_2 \in X$ and $x' \in X'$. Assume initial conditions. (1)

i) Show: $x_1 \equiv x_2 \,(\text{mod } \lambda) \Rightarrow (\forall \sigma \in \Sigma)\, \xi(x_1, \sigma)! \Rightarrow \xi(x_2, \sigma)!$
Assume: $x_1 \equiv x_2 \,(\text{mod } \lambda)$
Let $\sigma \in \Sigma$. Let $s = \sigma$.
As $s \in \Sigma^*$, $\xi(x_1, s)! \Leftrightarrow \xi(x_2, s)!$ follows automatically from Definition 2.2.9 of λ-equivalence.
Part (i) complete.

ii) Show: $[x_1 \equiv x_2 \,(\text{mod } \lambda) \wedge x_1, x_2 \in X_{samp}] \Rightarrow \zeta(x_1) = \zeta(x_2)$
Assume: $x_1 \equiv x_2 \,(\text{mod } \lambda)$ and $x_1, x_2 \in X_{samp}$ (2)
To show $\zeta(x_1) = \zeta(x_2)$, by Definition 3.7.2 of ζ it is sufficient to show:

$$\{\sigma \in \Sigma_{hib} \,|\, \xi(x_1, \sigma)!\} = \{\sigma \in \Sigma_{hib} \,|\, \xi(x_2, \sigma)!\}$$

As $\Sigma_{hib} \subseteq \Sigma$ and $x_1 \equiv x_2 \,(\text{mod } \lambda)$ by (2), this follows automatically from Part (i).
Part (ii) complete.

iii) Show: $[x' \notin X \wedge x' \in X'_{samp}] \Rightarrow (\forall x \in E_{x'} \cap X_{samp})\, \zeta(x) = \zeta'(x')$
Assume: $x' \notin X$ and $x' \in X'_{samp}$ (3)
As $x' \notin X$, this means that x' was added to $\mathbf{S}'$ by Algorithm 6.2.
Let $x \in E_{x'} \cap X_{samp}$. (4)
We first note that by Definition 3.7.2, we have:

$$\zeta(x) = \{\sigma \in \Sigma_{hib} \,|\, \xi(x, \sigma)!\} \text{ and } \zeta'(x') = \{\sigma \in \Sigma_{hib} \,|\, \xi'(x', \sigma)!\}$$

To show that $\zeta(x) = \zeta'(x')$, it is sufficient to show: $(\forall \sigma \in \Sigma_{hib})\, \xi(x, \sigma)! \Leftrightarrow \xi'(x', \sigma)!$
Let $\sigma \in \Sigma_{hib}$.

Part 1) Show: $\xi(x, \sigma)! \Rightarrow \xi'(x', \sigma)!$
Assume: $\xi(x, \sigma)!$
Let $x_b = \xi(x, \sigma)$, and thus $x_b \in X$.
By Proposition 6.3(iii), we can conclude:

$$(\exists x'_1, x'_2 \in X')\, \xi'(x'_1, \sigma) = x'_2 \wedge (x'_1 = x \vee x \in E_{x'_1}) \wedge (x'_2 = x_b \vee x_b \in E_{x'_2}) \quad (5)$$

As $x \in E_{x'}$ by (4), it follows that $x \notin X'$.
As $(x'_1 = x \vee x \in E_{x'_1})$ by (5), it follows that $x \neq x'_1$, thus following that $x \in E_{x'_1}$.
$\Rightarrow x' = x'_1$ *as Algorithm* 6.1 *will put a state in* X *into at most one distinct set of* λ*-equivalent states*

$\Rightarrow \xi'(x', \sigma) = x'_2$ *by* (5)
$\Rightarrow \xi'(x', \sigma)!$

Part 2) Show: $\xi'(x', \sigma)! \Rightarrow \xi(x, \sigma)!$
Assume: $\xi'(x', \sigma)!$
Let $x'' = \xi'(x', \sigma)$. (6)
We have two cases: **(a)** $x'' \in X$, or **(b)** $x'' \notin X$.

Case 2.a) $x'' \in X$
$\Rightarrow x' \notin X$ and $x'' \in X$ *by* (3)
By Proposition 6.4(iii), we can conclude: $(\forall x_a \in E_{x'})\, \xi(x_a, \sigma) = x''$
As $x \in E_{x'}$ by (4), we have: $\xi(x, \sigma) = x''$
$\Rightarrow \xi(x, \sigma)!$

Case 2.b) $x'' \notin X$
$\Rightarrow x', x'' \notin X$ and $\xi'(x', \sigma) = x''$ *by* (3) *and* (6)
By Proposition 6.4(ii), we can conclude: $(\forall x_a \in E_{x'})\, (\exists x_b \in E_{x''})\, \xi(x_a, \sigma) = x_b$
As $x \in E_{x'}$ by (4), we have: $\xi(x, \sigma) = x_b$
$\Rightarrow \xi(x, \sigma)!$

By Cases (2.a) and (2.b), we have $\xi(x, \sigma)!$. We thus conclude $\xi'(x', \sigma)! \Rightarrow \xi(x, \sigma)!$.

By Parts (1) and (2), we conclude that for $\sigma \in \Sigma_{hib}$, $\xi(x, \sigma)! \Leftrightarrow \xi'(x', \sigma)!$.
We thus conclude $\zeta(x) = \zeta(x')$.
Part (iii) complete.

iv) Show: $[x' \in X \wedge x' \in X'_{samp}] \Rightarrow x' \in X_{samp} \wedge \zeta(x') = \zeta'(x')$
Assume: $x' \in X$ and $x' \in X'_{samp}$ (7)
We will now show this implies: $x' \in X_{samp}$ and $\zeta(x') = \zeta'(x')$

Part 1) Show: $x' \in X_{samp}$
By Definition 3.4.2 of sampled states, it is sufficient to show:

$$x' \in \{x_a \in X \mid (\exists s \in L(\mathbf{S}) \cap L_{samp})\, x_a = \xi(x_o, s)\}$$

As $x' \in X$ by (7), all that remains is to show: $(\exists s \in L(\mathbf{S}) \cap L_{samp})\, x' = \xi(x_o, s)$
As $x' \in X'_{samp}$ by (7), it follows that: $(\exists s \in L(\mathbf{S}') \cap L_{samp})\, x' = \xi'(x'_o, s)$ (8)
We have two cases: **(a)** $x'_o \in X$, or **(b)** $x'_o \notin X$.

Case 1.a) $x'_o \in X$
As $x' \in X$ by (7), and $x'_o \in X$, we apply Proposition 6.4(i) and conclude: $\xi(x'_o, s) = x'$
As $x'_o \in X$, it follows by Algorithms 6.1 and 6.2 that $x'_o = x_o$.
$\Rightarrow \xi(x_o, s) = x'$

Case 1.b) $x'_o \notin X$

As $x' \in X$ by (7), $x'_o \notin X$, and $\xi'(x'_o, s) = x'$ by (8), we can apply Proposition 6.4(iii) and conclude: $(\forall x_a \in E_{x'_o})\, \xi(x_a, s) = x'$

As $x'_o \notin X$, by Algorithms 6.1 and 6.2 we can conclude $x_o \in E_{x'_o}$.

$\Rightarrow \xi(x_o, s) = x'$

By Cases (1.a) and (1.b), we have: $\xi(x_o, s) = x'$

$\Rightarrow s \in L(\mathbf{S}) \cap L_{samp}$ *by* (8)

$\Rightarrow x' \in X_{samp}$

Part (1) complete.

Part 2) Show: $\zeta(x') = \zeta'(x')$

To show $\zeta(x') = \zeta'(x')$, by Definition 3.7.2 it is sufficient to show:

$$(\forall \sigma \in \Sigma_{hib})\, \xi(x', \sigma)! \Leftrightarrow \xi'(x', \sigma)!$$

Let $\sigma \in \Sigma_{hib}$.

Part 2.a) Show: $\xi(x', \sigma)! \Rightarrow \xi'(x', \sigma)!$

Assume: $\xi(x', \sigma)!$

Let $x_b = \xi(x', \sigma)$.

By Proposition 6.3(iii), we can conclude:

$$(\exists x'_a, x'_b \in X')\, \xi'(x'_a, \sigma) = x'_b \wedge (x'_a = x' \vee x' \in E_{x'_a}) \wedge (x'_b = x_b \vee x_b \in E_{x'_b})$$

As $x' \in X \cap X'$ by (1) and (7), by Algorithms 6.1 and 6.2 we have $x'_a = x'$, as x' is λ-equivalent only to itself.

$\Rightarrow \xi'(x', \sigma) = x'_b$

$\Rightarrow \xi'(x', \sigma)!$

Part (2.a) complete.

Part 2.b) Show: $\xi'(x', \sigma)! \Rightarrow \xi(x', \sigma)!$

Assume: $\xi'(x', \sigma)!$

Let $x'_b = \xi'(x', \sigma)$.

$\Rightarrow x'_b \in X', x' \in X \cap X'$, and $\xi'(x', \sigma) = x'_b$ *by* (1) *and* (7) (9)

By Proposition 6.3(ii), we can conclude:

$$(\exists x_a, x_b \in X)\, \xi(x_a, \sigma) = x_b \wedge (x_a = x' \vee x_a \in E_{x'}) \wedge (x_b = x'_b \vee x_b \in E_{x'_b})$$

As $x' \in X$ by (9), by Algorithms 6.1 and 6.2 this implies that x' is λ-equivalent only to itself.

$\Rightarrow x_a = x'$

$\Rightarrow \xi(x', \sigma) = x_b$

$\Rightarrow \xi(x', \sigma)!$

Part (2.b) complete.

By Parts (2.a) and (2.b), we conclude $(\forall \sigma \in \Sigma_{hib})\, \xi(x', \sigma)! \Leftrightarrow \xi'(x', \sigma)!$.

$\Rightarrow \zeta(x') = \zeta'(x')$

Part (2) complete.

By Parts (1) and (2), we thus conclude $x' \in X_{samp}$ and $\zeta(x') = \zeta'(x')$.

Part (iv) complete.

By Parts (i)-(iv), we conclude that Points (i-iv) of the proposition are satisfied. □

We now introduce another supporting proposition that will be key in proving our main result of output equivalent controllers. The following proposition shows that a sampled string accepted by the closed-loop system of our $||_{SD}$ setting, $\mathbf{S}\,||_{SD}\,\mathbf{G}$, will take each supervisor $\mathbf{S}$, $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}'$ to a state with the same prohibitable events enabled. This means whether we use $\boldsymbol{\mathcal{S}}$ or $\boldsymbol{\mathcal{S}}'$, we get the same result that matches our supervisor $\mathbf{S}$.

Proposition 7.2. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$. Let TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor and $\boldsymbol{\mathcal{S}}' = min(\boldsymbol{\mathcal{S}}) = (Y', \Sigma, \eta', y'_o, Y'_m)$ be the minimal TDES constructed using Algorithms 6.1 and 6.2. Let $\zeta_{\mathbf{S}}$, $\zeta_{\boldsymbol{\mathcal{S}}}$ and $\zeta_{\boldsymbol{\mathcal{S}}'}$ be the prohibited action functions for supervisors $\mathbf{S}$, $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}'$ respectively. Then:

$$(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp})\, \zeta_{\mathbf{S}}(\xi(x_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}}(\eta(y_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}'}(\eta'(y'_o, s))$$

Proof. Assume initial conditions. Let $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and $\boldsymbol{\mathcal{S}}' = min(\boldsymbol{\mathcal{S}})$. (1)

Let $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$. (2)

Let X_{samp}, Y_{samp} and Y'_{samp} be the sets of sampled states for $\mathbf{S}$, $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}'$ respectively.

First, we note that $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ means that $s \in L(\boldsymbol{\mathcal{S}})$ by definition. As state minimization process does not affect the closed behaviour of an automaton, this implies $s \in L(\boldsymbol{\mathcal{S}}')$. (3)

$\Rightarrow \eta(y_o, s)!$ and $\eta'(y'_o, s)!$

Let $y = \eta(y_o, s)$ and $y' = \eta'(y'_o, s)$. (4)

By Definition 4.1.1 of $||_{SD}$ operator, we have: $(\exists x \in X)\,(\exists q \in Q)\, y = (x, q)$ (5)

As both $\mathbf{G}$ and $\mathbf{S}$ are defined over Σ, it follows by the definition of $||_{SD}$ that:

$\xi(x_o, s) = x$ and $\delta(q_o, s) = q$ (6)

$\Rightarrow s \in L(\mathbf{S})$ and $s \in L(\mathbf{G})$ (7)

$\Rightarrow s \in L(\mathbf{S}) \cap L_{samp}$ *by* (2)

$\Rightarrow x \in X_{samp}$ *by Definition* 3.4.2 *of sampled states*

As $s \in L(\boldsymbol{\mathcal{S}})$ by (3), by (2) we have: $s \in L(\boldsymbol{\mathcal{S}}) \cap L_{samp}$

As $y = \eta(y_o, s)$ by (4), by Definition 3.4.2 we have: $y \in Y_{samp}$ (8)

Similarly, we have: $y' \in Y'_{samp}$ (9)

This means that $\zeta_{\mathbf{S}}(x), \zeta_{\boldsymbol{\mathcal{S}}}(y)$ and $\zeta_{\boldsymbol{\mathcal{S}}'}(y')$ are defined.

We will now show: $\zeta_{\mathbf{S}}(\xi(x_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}}(\eta(y_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}'}(\eta'(y'_o, s))$

By (4) and (6), it is sufficient to show: $\zeta_{\mathbf{S}}(x) = \zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$
We will show this in two steps.

Part 1) Show: $\zeta_{\mathbf{S}}(x) = \zeta_{\boldsymbol{\mathcal{S}}}(y)$
By Definition 3.7.2 of ζ, it is sufficient to show:

$$\{\sigma \in \Sigma_{hib} \,|\, \xi(x,\sigma)!\} = \{\sigma \in \Sigma_{hib} \,|\, \eta(y,\sigma)!\}$$

This is equivalent to showing: $(\forall \sigma \in \Sigma_{hib})\, \xi(x,\sigma)! \Leftrightarrow \eta(y,\sigma)!$
Let $\sigma \in \Sigma_{hib}$. (10)

Part 1.a) Show: $\xi(x,\sigma)! \Rightarrow \eta(y,\sigma)!$
Assume: $\xi(x,\sigma)!$ (11)
$\Rightarrow s\sigma \in L(\mathbf{S})$ *by* (6) *and* (7)
As $s \in L(\mathbf{S}) \cap L(\mathbf{G})$ by (7), $\sigma \in \Sigma_{hib}$ by (10), and $\mathbf{G}$ is complete with $||_{SD}$ for $\mathbf{S}$ by (1), we can conclude: $s\sigma \in L(\mathbf{G})$
$\Rightarrow \delta(q,\sigma)!$ *by* (6)
As $\xi(x,\sigma)!$ by (11), $\delta(q,\sigma)!$, $\sigma \in \Sigma_{hib}$ by (10), and $y = (x,q)$ by (5), by the definition of $||_{SD}$, we conclude: $\eta((x,q),\sigma)!$
$\Rightarrow \eta(y,\sigma)!$

Part 1.b) Show: $\eta(y,\sigma)! \Rightarrow \xi(x,\sigma)!$
Assume: $\eta(y,\sigma)!$
$\Rightarrow \eta((x,q),\sigma)!$ *by* (5)
$\Rightarrow \xi(x,\sigma)!$ *by definition of* $||_{SD}$ *and the fact that* $\mathbf{G}$ *and* $\mathbf{S}$ *are defined over* Σ
By Parts (1.a) and (1.b), we can conclude: $(\forall \sigma \in \Sigma_{hib})\, \xi(x,\sigma)! \Leftrightarrow \eta(y,\sigma)!$
$\Rightarrow \zeta_{\mathbf{S}}(x) = \zeta_{\boldsymbol{\mathcal{S}}}(y)$
Part (1) complete.

Part 2) Show: $\zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$
By (4), we have: $y = \eta(y_o, s)$
Using $\eta(y_o, s) = y$, we can apply Proposition 6.3(iii) and conclude:

$$(\exists y'_a, y'_b \in Y')\, \eta'(y'_a, s) = y'_b \wedge (y'_a = y_o \vee y_o \in E_{y'_a}) \wedge (y'_b = y \vee y \in E_{y'_b}) \quad (12)$$

From Algorithms 6.1 and 6.2, we know that y_o belongs to at most one set of λ-equivalent states (E_k), and that either $y_o = y'_o$ or $y_o \in E_{y'_o}$.
$\Rightarrow \eta'(y'_o, s) = y'_b$
$\Rightarrow y'_b = y'$ *as* $y' = \eta'(y'_o, s)$ *by* (4)
$\Rightarrow y' = y \vee y \in E_{y'}$ *by* (12)
We thus have two cases: **(a)** $y' = y$, or **(b)** $y \in E_{y'}$.

Case 2.a) $y' = y$ (13)
$\Rightarrow y' \in Y$
As we have $y' \in Y \cap Y'$ by (4), and $y' \in Y'_{samp}$ by (9), we can apply Proposition 7.1(iv)

and we have: $\zeta_{\boldsymbol{\mathcal{S}}}(y') = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$
$\Rightarrow \zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$ *by* (13)

Case 2.b) $y \in E_{y'}$ (14)

By Algorithms 6.1 and 6.2, this implies: $y' \notin Y$

As $y' \in Y'$ by (4), $y' \notin Y$, and $y' \in Y'_{samp}$ by (9), we can apply Proposition 7.1(iii) and conclude: $(\forall y_a \in E_{y'} \cap Y_{samp})\, \zeta_{\boldsymbol{\mathcal{S}}}(y_a) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$

As $y \in E_{y'}$ by (14) and $y \in Y_{samp}$ by (8), we can conclude: $\zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$

By Cases (2.a) and (2.b), we have: $\zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$

Part (2) complete.

Combining Parts (1) and (2), we can conclude: $\zeta_{\mathbf{S}}(x) = \zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$

By (4) and (6), we can conclude $\zeta_{\mathbf{S}}(\xi(x_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}}(\eta(y_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}'}(\eta'(y'_o, s))$, as required. □

7.3 Output Equivalent Controllers

In this section, we present our main result for output equivalence between two SD controllers that are translated using the method presented in Section 3.7.

Theorem 7.1 given below proves that an SD controller translated from supervisor $\mathbf{S}$ will be output equivalent to a controller translated from supervisor $\boldsymbol{\mathcal{S}}' = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$. In this theorem, we only consider the controller for $\boldsymbol{\mathcal{S}}' = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$, and not $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$. This is because if $\boldsymbol{\mathcal{S}}$ is already minimal, then $min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = \boldsymbol{\mathcal{S}}$. Thus, examining $\boldsymbol{\mathcal{S}}'$ without assuming that $\boldsymbol{\mathcal{S}}$ is minimal will cover both cases. Also, in Proposition 7.2, we have already proven that for a valid sampled string, both $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}'$ produce the same enablement information for prohibitable events.

Theorem 7.1. Let TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor that is SD controllable with $||_{SD}$ for $\mathbf{G}$, and let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$. Let TDES supervisor $\boldsymbol{\mathcal{S}}' = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ constructed using Algorithms 6.1 and 6.2 be CS deterministic. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller translated from $\mathbf{S}$, and $\boldsymbol{\mathcal{C}}' = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller translated from $\boldsymbol{\mathcal{S}}'$. Then, $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ are output equivalent with respect to the closed-loop behaviour $L(\mathbf{G}_{cl})$, with $\mathbf{G}_{cl} = \mathbf{S}\,||_{SD}\,\mathbf{G}$.

Proof. Assume initial conditions.

First, we will describe our setting and notation for the proof.

Let $\Sigma_{act} \subset \Sigma$ be the set of activity events and $\Sigma_{hib} \subseteq \Sigma_{act}$ be the set of prohibitable events.

Let $\mathbf{G}_{cl} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and $\boldsymbol{\mathcal{S}}' = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$. (1)

As state minimization process does not change the closed behaviour of an automaton, thus we have: $L(\mathbf{G}_{cl}) = L(\boldsymbol{\mathcal{S}}')$

By Corollary 5.1(iii), we have $L(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}' \,||\, \mathbf{G})$. We thus have: $L(\mathbf{G}_{cl}) = L(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}' \,||\, \mathbf{G})$

As $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ are control equivalent (Definition 2.3.8), this means we can apply Definition 7.1.2 of output equivalence to $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$.

Let $X_{samp} \subseteq X$ and $X'_{samp} \subseteq X'$ be the sets of sampled states for $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ respectively.

Let $\Lambda : X_{samp} \to Q$ and $\Lambda' : X'_{samp} \to Q'$ be the injective state mapping functions (Definition 3.7.7) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ respectively.

Let $\Gamma_Z : \text{Pwr}(\Sigma_{hib}) \to Z$ and $\Gamma_{Z'} : \text{Pwr}(\Sigma_{hib}) \to Z'$ be the bijective output set mapping functions (Definition 3.7.9) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ respectively.

Let $\Phi : Q \to Z$ and $\Phi' : Q' \to Z'$ be the state-to-output maps (Definition 3.7.11) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ respectively.

Let $\zeta : X_{samp} \to \text{Pwr}(\Sigma_{hib})$ and $\zeta' : X'_{samp} \to \text{Pwr}(\Sigma_{hib})$ be the prohibited action functions (Definition 3.7.2) for $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ respectively.

Let γ_g be the canonical event mapping function (Definition 3.7.3) for the system. This is the default way to order event variables in vectors.

Let $v = |\Sigma_{act}|$.

As both $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ are defined over Σ, it follows that each input vector $i \in I$ and $i' \in I'$ is the same size, i.e. each contains v variables.

Let $\gamma : \Sigma_{act} \to \{0, 1, \ldots, v-1\}$ and $\gamma' : \Sigma_{act} \to \{0, 1, \ldots, v-1\}$ be the input event mapping functions (Definition 3.7.4) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ respectively. By definition of γ and γ', we have:

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{act})\, \gamma_g(\sigma_1) < \gamma_g(\sigma_2) \Rightarrow \gamma(\sigma_1) < \gamma(\sigma_2) \wedge \gamma'(\sigma_1) < \gamma'(\sigma_2)$$

This implies that there is only one way to define γ and γ', and they both must equal γ_g, i.e. $\gamma = \gamma' = \gamma_g$. (2)

Let $\{\mathbf{i}_g(k'')\}$ be a canonical input sequence with respect to γ_g (i.e. its event variables ordering matches γ_g), and let the sequence be input valid for $L(\mathbf{G}_{cl})$. (3)

As $\gamma = \gamma' = \gamma_g$ by (2), it follows that $\{\mathbf{i}_g(k'')\}$ can be used as input vectors for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ directly, without any conversion.

Let $r = |\Sigma_{hib}|$.

As both $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ are defined over Σ, this implies that each output vector $\mathbf{z} \in Z$ and $\mathbf{z}' \in Z'$ is the same size, i.e. each contains r variables. (4)

Let $\eta : \Sigma_{hib} \to \{0, 1, \ldots, r-1\}$ and $\eta' : \Sigma_{hib} \to \{0, 1, \ldots, r-1\}$ be the bijective output event mapping functions (Definition 3.7.5) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ respectively. By definition of η and η', we have:

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{hib})\, \gamma_g(\sigma_1) < \gamma_g(\sigma_2) \Rightarrow \eta(\sigma_1) < \eta(\sigma_2) \wedge \eta'(\sigma_1) < \eta'(\sigma_2)$$

This implies that there is only one way to define η and η'. Thus we have $\eta = \eta'$. (5)

We note that the definition of Γ_Z and $\Gamma_{Z'}$ is defined in terms of η and η' respectively. Since $\eta = \eta'$ by (5), this implies that $\Gamma_Z = \Gamma_{Z'}$. (6)

This implies that output vectors $\mathbf{z}$ and $\mathbf{z}'$ are the same size (r) and represent prohibitable events in exactly the same order.

For input sequence $\{\mathbf{i}_g(k'')\}$, let $\mathbf{z}(k) \in Z$ and $\mathbf{z}'(k) \in Z'$ be the induced output vector at time k for controllers $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ respectively.

Let $\mathbf{q}(k) \in Q$ and $\mathbf{q}'(k) \in Q'$ be the induced state vectors at time k for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ respectively.

Now, we will prove our main result.

To show that $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ are output equivalent with respect to $L(\mathbf{G}_{cl})$, by Definition 7.1.2 we need to show:

1. Both output vectors $\mathbf{z}$ and $\mathbf{z}'$ are of size r.

2. $(\forall\, 0 \leq i < r)\, \eta^{-1}(i) = \eta'^{-1}(i)$

3. $(\forall k \in \{0, 1, 2, \ldots\})\, \mathbf{z}(k) = \mathbf{z}'(k)$

We note that Points 1 and 2 follow immediately from (4) and (5) respectively.

Now all that remains is to show: $(\forall k \in \{0, 1, 2, \ldots\})\, \mathbf{z}(k) = \mathbf{z}'(k)$

Let $k \in \{0, 1, \ldots\}$.

We first note that by the TDES to FSM translation method (Section 3.7), we have:

$$\mathbf{z}(k) = \Phi(\mathbf{q}(k)) \text{ and } \mathbf{z}'(k) = \Phi'(\mathbf{q}'(k))$$

By definition of Φ and Φ', we have:

$$\begin{gathered}\mathbf{z}(k) = \Phi(\mathbf{q}(k)) = \Gamma_Z(\zeta(x)) \text{ and } \mathbf{z}'(k) = \Phi'(\mathbf{q}'(k)) = \Gamma_{Z'}(\zeta'(x')) \\ \text{where } \mathbf{q}(k) = \Lambda(x) \text{ and } \mathbf{q}'(k) = \Lambda'(x') \text{ for some } x \in X_{samp} \text{ and } x' \in X'_{samp}\end{gathered} \tag{7}$$

As $\Gamma_Z = \Gamma_{Z'}$ by (6), all we need to complete the proof is to construct a suitable $x \in X_{samp}$ and $x' \in X'_{samp}$ and show that $\zeta(x) = \zeta'(x')$.

We have two cases: **(1)** $k = 0$, and **(2)** $k \in \{1, 2, \ldots\}$.

Case 1) $k = 0$

By definition of the TDES to FSM translation method (Section 3.7), we have:

$$\mathbf{q}(0) = \mathbf{q}_{res} = \Lambda(x_o) \text{ and } \mathbf{q}'(0) = \mathbf{q}'_{res} = \Lambda'(x'_o) \tag{8}$$

Let $s = \epsilon$.

$\Rightarrow \xi(x_o, s) = x_o$ and $\xi'(x'_o, s) = x'_o$ (9)

As $s = \epsilon \in L_{samp} = \{\epsilon\} \cup \Sigma^*.\tau$, and $\mathbf{S}$ and $\mathbf{G}$ have initial states implies that $\mathbf{S}\,||_{SD}\,\mathbf{G}$ has an initial state, it follows that $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$.

By applying Proposition 7.2, we conclude: $\zeta(\xi(x_o, s)) = \zeta'(\xi'(x'_o, s))$

$\Rightarrow \zeta(x_o) = \zeta'(x'_o)$ *by* (9)

We note that by Definition 3.4.2 of sampled states, initial states are always sampled states. We then take $x = x_o$ and $x' = x'_o$. We thus have $x \in X_{samp}$ and $x' \in X'_{samp}$, $\mathbf{q}(k) = \Lambda(x)$ and $\mathbf{q}'(k) = \Lambda'(x')$ by (8), $k = 0$, and $\zeta(x) = \zeta'(x')$.

Case 2) $k \in \{1, 2, \ldots\}$

As $\{\mathbf{i}_g(k'')\}$ is input valid for $L(\mathbf{G}_{cl})$ by (3), we have:

$$(\exists s_1, s_2, \ldots, s_k \in L_{conc})\,[s_1 s_2 \ldots s_k \in L(\mathbf{G}_{cl})] \wedge$$
$$[(\forall n \in \{1, 2, \ldots, k\})\,(\forall \sigma \in \Sigma_{act})\, i_{g,\gamma_g(\sigma)}\,(n) = 1 \text{ iff } \sigma \in Occu(s_n)]$$

Let $s = s_1 s_2 \ldots s_k$, and we have $s \in L_{samp}$ as $L_{conc} = \Sigma^*_{act}.\boldsymbol{\tau}$ (Definition 3.4.1). (10)

As $\mathbf{G}_{cl} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and $\boldsymbol{\mathcal{S}}' = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ by (1), we have: $s \in L(\boldsymbol{\mathcal{S}}') \cap L_{samp}$ (11)

As $L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \subseteq L(\mathbf{S})$ by Proposition 5.1, we have: $s \in L(\mathbf{S}) \cap L_{samp}$

By applying Proposition 3.2, we conclude:

$$\mathbf{q}(k) = \Lambda(\xi(x_o, s)) \text{ and } \mathbf{q}'(k) = \Lambda'(\xi'(x'_o, s)) \quad (12)$$

Let $x = \xi(x_o, s)$ and $x' = \xi'(x'_o, s)$. (13)

$\Rightarrow x \in X_{samp}$ and $x' \in X'_{samp}$ *as* $s \in L_{samp}$ *by* (10) (14)

As $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ by (11), by applying Proposition 7.2 we conclude:

$$\zeta(\xi(x_o, s)) = \zeta'(\xi'(x'_o, s))$$

$\Rightarrow \zeta(x) = \zeta'(x')$ *by* (13)

We thus have $x \in X_{samp}$ and $x' \in X'_{samp}$ by (14), $\mathbf{q}(k) = \Lambda(x)$ and $\mathbf{q}'(k) = \Lambda'(x')$ by (12) and (13), and $\zeta(x) = \zeta'(x')$.

By Cases (1) and (2), we have constructed a suitable x and x' with $\zeta(x) = \zeta'(x')$.

We thus conclude by (7) that $\mathbf{z}(k) = \mathbf{z}'(k)$, as required.

Hence, we conclude that $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ are output equivalent with respect to the closed-loop behaviour, $L(\mathbf{G}_{cl})$. □

We close this section with a proposition that we will find useful in Chapter 8. It essentially shows that SD controllers $\boldsymbol{\mathcal{C}}$ and $\boldsymbol{\mathcal{C}'}$ will produce the same output for every sampled string accepted by the closed-loop system, $\mathbf{G}_{cl} = \mathbf{S}\,||_{SD}\,\mathbf{G}$.

Proposition 7.3. Let TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor that is SD controllable with $||_{SD}$ for $\mathbf{G}$, and let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$. Let TDES supervisor $\boldsymbol{\mathcal{S}}' = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ constructed using Algorithms 6.1 and 6.2 be CS deterministic. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller translated from $\mathbf{S}$, and $\boldsymbol{\mathcal{C}'} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller translated from $\boldsymbol{\mathcal{S}}'$. Let Λ and Λ' be the state mapping functions for $\boldsymbol{\mathcal{C}}$ and $\boldsymbol{\mathcal{C}'}$ respectively. Then:

$$(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp})\ \Phi(\Lambda(\xi(x_o, s))) = \Phi'(\Lambda'(\xi'(x'_o, s)))$$

Proof. Assume initial conditions.

Let ζ and ζ' be the prohibited action functions (Definition 3.7.2) for $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ respectively.

Let η and η' be the bijective output event mapping functions (Definition 3.7.5) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ respectively.

Let Γ_Z and $\Gamma_{Z'}$ be the bijective output set mapping functions (Definition 3.7.9) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ respectively.

Let $s \in L(\mathbf{S} ||_{SD} \mathbf{G}) \cap L_{samp}$. (1)

Applying Proposition 7.2, we conclude: $\zeta(\xi(x_o, s)) = \zeta'(\xi'(x'_o, s))$ (2)

Let $\mathbf{q} = \Lambda(\xi(x_o, s))$ and $\mathbf{q}' = \Lambda'(\xi'(x'_o, s))$. (3)

Applying Theorem 7.1, we conclude that $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ are output equivalent with respect to $L(\mathbf{G}_{cl})$.

This implies $\eta = \eta'$, and thus $\Gamma_Z = \Gamma_{Z'}$, as they are defined in terms of η and η' respectively. (4)

By Definition 3.7.11 of Φ, we have: $\Phi(\mathbf{q}) = \Gamma_Z(\zeta(x))$ if $(\exists x \in X_{samp})\, \mathbf{q} = \Lambda(x)$

As $s \in L_{samp}$ by (1), we can take $x = \xi(x_o, s)$ and we have $x \in X_{samp}$, and $\Lambda(x) = \mathbf{q}$ by (3).

We thus have: $\Phi(\mathbf{q}) = \Gamma_Z(\zeta(\xi(x_o, s)))$

Similarly, we can take $x' = \xi'(x'_o, s)$ and we have $x' \in X'_{samp}$, and $\Lambda'(x') = \mathbf{q}'$ by (3).

$\Rightarrow \Phi'(\mathbf{q}') = \Gamma_{Z'}(\zeta'(\xi'(x'_o, s)))$

As $\Gamma_Z = \Gamma_{Z'}$ by (4), and $\zeta(\xi(x_o, s)) = \zeta'(\xi'(x'_o, s))$ by (2), we have: $\Phi(\mathbf{q}) = \Phi'(\mathbf{q}')$

$\Rightarrow \Phi(\Lambda(\xi(x_o, s))) = \Phi'(\Lambda'(\xi'(x'_o, s)))$ *by* (3) □

Chapter 8

Controllability and Nonblocking Results for SD Synchronous Product Setting

In this chapter, we present the controllability and nonblocking verification results for our $||_{SD}$ setting. This chapter begins with the construction of a TDES supervisory control V, stating the relevant definitions, and proving its various properties. After that, we thoroughly describe and formally prove our controllability and nonblocking results. Essentially, we show that if our theoretical $||_{SD}$ system is controllable, nonblocking and abide by the specified control laws, then the physically implemented system will also have these properties, given that the $||_{SD}$ system satisfies our adapted properties that were originally identified by the SD supervisory control methodology.

Please note that in this chapter, we will use the notation of TDES supervisor $\mathbf{S}$, SD controller $\mathbf{C}$ and TDES supervisory control V while discussing about our $||_{SD}$ setting. The notation of TDES supervisor $\boldsymbol{\mathcal{S}}$, SD controller $\boldsymbol{\mathcal{C}}$ and TDES supervisory control $\mathcal{V}$ will be used while referring to the SD setting, and they map to $\mathbf{S}$, $\mathbf{C}$ and V of Chapter 3 respectively. In this chapter, whenever we refer to an SD controller constructed from a supervisor, we will always assume that it is translated using the method described in Section 3.7.

From this chapter onwards, we will take our SD supervisor $\boldsymbol{\mathcal{S}}$ to be $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$, i.e. the minimal version of $\mathbf{S}\,||_{SD}\,\mathbf{G}$ that is constructed using Algorithms 6.1 and 6.2. The reason is that if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is already minimal, there is no change. However, if it is not already minimal, we must minimize it to ensure that $\boldsymbol{\mathcal{S}}$ is CS deterministic. Our assumption that we are always using the minimal version of $\mathbf{S}\,||_{SD}\,\mathbf{G}$ will keep things simple.

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be an SD controller, and the closed-loop system of our $||_{SD}$ setting be $\mathbf{S}\,||_{SD}\,\mathbf{G}$. For the rest of this chapter, we require our system to satisfy the following properties: 1) $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces and finite

event sets, 2) $\mathbf{G}$ has proper time behaviour, 3) $\mathbf{G}$ is complete with $||_{SD}$ for $\mathbf{S}$, 4) $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, 5) $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF, 6) $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$, 7) $\mathbf{S}$ is CS deterministic, and 8) $\mathbf{C}$ is an SD controller translated from $\mathbf{S}$ using the translation method described in Section 3.7.

By looking at Proposition 4.5, it is evident that these conditions are sufficient to guarantee that our system will not "stop the clock", i.e. for any string $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$, our $||_{SD}$ system will always be able to do a *tick* event after at most a finite number of activity events. This ensures that after a sampled string, all new behaviour of the system can be represented as a sequence of concurrent strings.

8.1 Supervisory Control V

In the SD supervisory control theory (Wang, 2009; Leduc *et al.*, 2014), the authors have pointed out that an SD controller is more constrained than a TDES supervisor. This is due to the fact that an SD controller only changes state on the occurrence of the *tick* event, whereas a supervisor can do so every time an event occurs. This in turn implies that the enablement and forcing information of an SD controller does not always exactly match with that of a TDES supervisor.

To address this issue in the SD setting, a TDES supervisory control is used to express the enablement and forcing behaviour of an SD controller in terms of strings. In Wang (2009), the authors presented Algorithm 3.1 to construct this supervisory control. This algorithm's definition is then used to argue about the behaviour of the SD controller in various controllability and nonblocking verification proofs of the SD setting. Since we are building our work on the SD supervisory control methodology, we will adopt the same approach to capture the control action of SD controller in our $||_{SD}$ setting and prove our desired results.

In this section, we first discuss the construction of a TDES supervisory control V in our $||_{SD}$ setting (we will formally prove that V is indeed a TDES supervisory control later in Proposition 8.4). Specifically, we explain how we have adapted Algorithm 3.1 to make it compatible with our $||_{SD}$ setting. Then, we present some definitions in relation to our V. Finally, we prove some properties with respect to V that will help us in proving our $||_{SD}$ controllability and nonblocking verification results afterwards.

8.1.1 Construction of V

Note: To be clear in our discussion and avoid any ambiguity, we will refer to $\mathbf{S}$, $\mathbf{C}$, V and Σ_V of Algorithm 3.1 as $\boldsymbol{\mathcal{S}}$, $\boldsymbol{\mathcal{C}}$, $\mathcal{V}$ and $\Sigma_{\mathcal{V}}$ respectively.

In order to construct TDES supervisory control V from our SD controller $\mathbf{C}$ in the $||_{SD}$ setting, we adapt Algorithm 3.1 from Wang (2009). Our algorithm for the $||_{SD}$ setting is presented as Algorithm 8.1. It is worth-mentioning that the two algorithms

Algorithm 8.1 Obtaining V from Controller $\mathbf{C}$, Acting on Plant $\mathbf{G}$

1: **for all** $s \in L(\mathbf{G})$ **do**
2: $V(s) \leftarrow \Sigma_u \cup \{\tau\}$
3: **end for**
4: $Pend \leftarrow \{(\epsilon, \mathbf{q}_{res})\}$
5: **while** $Pend \neq \emptyset$ **do**
6: $(s, \mathbf{q}) \leftarrow$ a member from $Pend$
7: $Pend \leftarrow Pend - \{(s, \mathbf{q})\}$
8: $\mathbf{z} \leftarrow \Phi(\mathbf{q})$
9: $\Sigma_V \leftarrow \Gamma_Z^{-1}(\mathbf{z})$
10: **if** $\Sigma_V \neq \emptyset$ **then**
11: $V(s) \leftarrow (V(s) \cup \Sigma_V) - \{\tau\}$
12: **end if**
13: **for all** $s' \leftarrow \sigma_1\sigma_2 \ldots \sigma_j \in CB_{\mathbf{G}}(s)$ **do** // $\sigma_j = \tau$, *by definition of* L_{conc}
14: **if** $(Occu(s') \cap \Sigma_{hib} \subseteq \Sigma_V) \wedge (ss' \in L(\mathbf{S}||_{SD}\, \mathbf{G}))$ **then**
15: $\Sigma_{temp} \leftarrow \Sigma_V$
16: $\mathbf{i} \leftarrow \Gamma_I(Occu(s') - \{\tau\})$
17: $\mathbf{q}' \leftarrow \Omega(\mathbf{q}, \mathbf{i})$
18: $Pend \leftarrow Pend \cup \{(ss', \mathbf{q}')\}$
19: **if** $j > 1$ **then**
20: **for** $i \leftarrow 1$ to $j - 1$ **do**
21: $\Sigma_{temp} \leftarrow \Sigma_{temp} - \sigma_i$
22: **if** $\Sigma_{temp} \neq \emptyset$ **then**
23: $V(s\sigma_1\sigma_2 \ldots \sigma_i) \leftarrow (V(s\sigma_1\sigma_2 \ldots \sigma_i) \cup \Sigma_V) - \{\tau\}$
24: **else**
25: $V(s\sigma_1\sigma_2 \ldots \sigma_i) \leftarrow (V(s\sigma_1\sigma_2 \ldots \sigma_i) \cup \Sigma_V)$
26: **end if**
27: **end for**
28: **end if**
29: **end if**
30: **end for**
31: **end while**
32: **return** V

are logically identical, although they differ at **line 14**, where $L(\boldsymbol{\mathcal{S}})$ of Algorithm 3.1 has been replaced by $L(\mathbf{S}||_{SD}\, \mathbf{G})$ in Algorithm 8.1.

Please note that the complete description of Algorithm 3.1 to construct TDES supervisory control from an SD controller is given in Section 3.8. Most of the items given in Algorithm 8.1 are defined in Section 3.7. The map of *Occu* is defined in Section 3.4, and TDES supervisory control and $CB_{\mathbf{G}}$ are defined in Section 3.8. In this section, we only focus on explaining and comparing those aspects of the two

algorithms that differ and need clarification.

For all strings $s \in L(\mathbf{G})$, Algorithm 3.1 sets the default enablement information at **lines 1-3** by adding all uncontrollable events and *tick* event to $\mathcal{V}(s)$. This is done to satisfy Definition 3.8.1 of TDES supervisory control. As this definition is given only in terms of $L(\mathbf{G})$, it remains valid in our $||_{SD}$ setting as well. Therefore, this part of Algorithm 3.1 remains unchanged in Algorithm 8.1.

By looking at Algorithm 3.1, we note that it updates the default enablement information only for those strings that represent valid behaviour in the closed-loop system. These strings are identified at **lines 13-14**. **Line 13** of Algorithm 3.1 considers all possible concurrent strings s' that extend a sampled string s in $L(\mathbf{G})$. The **if** statement at **line 14** then uses the following two conditions to filter out those strings that do not meet the required criteria.

1) $Occu(s') \cap \Sigma_{hib} \subseteq \Sigma_{\mathcal{V}}$
This condition excludes concurrent strings that are possible in the closed behaviour of $\mathbf{G}$, but their occurrence images contain prohibitable events that are not in $\Sigma_{\mathcal{V}}$. As these prohibitable events are disabled by controller $\boldsymbol{\mathcal{C}}$, therefore these strings will not occur in the physical system.

Since we are using the translation method of the SD setting to generate our SD controller $\mathbf{C}$ from TDES supervisor $\mathbf{S}$, therefore this condition does not need to be changed for our $||_{SD}$ setting, and shows up as it is in Algorithm 8.1.

2) $ss' \in L(\boldsymbol{\mathcal{S}})$
In Algorithm 3.1, this condition disregards concurrent strings that do not represent valid behaviour in $L(\boldsymbol{\mathcal{S}})$ after sampled string s. This ensures to restrict the set of valid strings to concurrent strings that are accepted by the supervisor. Ultimately, it results in restricting the valid strings overall to $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ in the SD setting.

Using the same logic for our $||_{SD}$ setting, we want to restrict the set of valid strings to our closed-loop behaviour, $L(\mathbf{S}\,||_{SD}\,\mathbf{G})$. In order to do that, we cannot simply replace $L(\boldsymbol{\mathcal{S}})$ at **line 14** of Algorithm 3.1 with $L(\mathbf{S})$ in our Algorithm 8.1. This is due to the fact that in the SD setting, supervisor $\boldsymbol{\mathcal{S}}$ is solely responsible for the enablement/disablement of *tick* event in the closed-loop system. However, in our $||_{SD}$ setting, the task of enabling/disabling the *tick* event is *cooperatively* performed by supervisor $\mathbf{S}$ and $||_{SD}$ operator. In our case, a *tick* event that is possible in $\mathbf{G}$ might be enabled by $\mathbf{S}$ too. Still, this *tick* event might not be possible in the closed-loop system as our $||_{SD}$ operator is authorized to remove *tick* from the closed-loop system in the presence of enabled prohibitable events. To further clarify, in most cases, our closed-loop behaviour $L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \neq L(\mathbf{S}) \cap L(\mathbf{G})$ due to the synchronization mechanism of our $||_{SD}$ operator.

For this reason, in order to restrict the set of valid strings to our closed-loop behaviour, we have replaced their $L(\boldsymbol{\mathcal{S}})$ with our $L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ at **line 14**. By making this change, we guarantee that if a string does not represent valid behaviour in our closed-loop system, then its enablement information, once assigned at **line 2**,

will remain unmodified throughout the execution of Algorithm 8.1.

It is worth clarifying that this replacement does not change the original logic of Algorithm 3.1 for constructing supervisory control from the SD controller. In fact, this change at **line 14** actually ensures that the original logic of Algorithm 3.1 remains untouched in Algorithm 8.1. We will formally prove this in Proposition 8.2 by showing that the two supervisory controls V and $\mathcal{V}$ constructed using Algorithms 8.1 and 3.1 respectively are equal with respect to a given plant $\mathbf{G}$.

Another way, probably an easier and straightforward one, to look at this modification at **line 14** is that in our $||_{SD}$ setting, we have concretely defined $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, or $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G})$ for that matter. We have already proven in our previous chapters that $\boldsymbol{\mathcal{S}}$ possesses all the required properties and does qualify to be used as the supervisor of the SD setting. Since $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S} \,||_{SD}\, \mathbf{G})$, replacing $L(\boldsymbol{\mathcal{S}})$ with $L(\mathbf{S} \,||_{SD}\, \mathbf{G})$ does not bring any logical change at **line 14**, as the two closed languages are the same. Therefore, both algorithms will restrict the update of enablement information to the same set of valid strings in the closed-loop behaviour due to the way we have constructed $\boldsymbol{\mathcal{S}}$ in the $||_{SD}$ setting.

Another important point that we want to highlight is about **line 11** of Algorithm 3.1. If any prohibitable event is enabled at state $\mathbf{q}'$ in $\boldsymbol{\mathcal{C}}$ (**line 10**), this prohibitable event needs to be forced in the current sampling period. Therefore, **line 11** removes *tick* event from $\mathcal{V}(s)$ to satisfy Point ii ($\Rightarrow$) of the SD controllability definition. This *tick* was added at **line 2** while initializing $\mathcal{V}(s)$ with its default enablement information.

It is worth recalling here that Point ii ($\Rightarrow$) of the SD controllability definition does not exist in our definition of SD controllability with $||_{SD}$ property, and we are not checking this condition explicitly in our $||_{SD}$ setting. We are able to get rid of this explicit check because of the synchronization mechanism of our $||_{SD}$ operator that guarantees to automatically satisfy this condition while forming the closed-loop system. Therefore, although *tick* event does get removed at **line 11** in Algorithm 8.1, it is for a different reason. In our case, this removal of *tick* is not to satisfy any point of the SD controllability with $||_{SD}$ definition. Rather, it is to keep things consistent with the synchronization mechanism used by our $||_{SD}$ operator to construct the closed-loop system; hence, **line 11** remains unmodified in Algorithm 8.1.

8.1.2 Preliminary Definitions

In order to define the closed behaviour of $V/\mathbf{G}$, represented as $L(V/\mathbf{G})$, Definition 2.3.4 uses TDES plant $\mathbf{G}$ and supervisory control V. This definition neither takes into account the supervisor model nor the synchronization operator while defining $L(V/\mathbf{G})$. Thus, this definition remains valid for our $||_{SD}$ setting and does not need to be redefined. Below, we present some definitions in relation to V that are specific to our $||_{SD}$ setting.

Definition 8.1.1. For TDES plant $\mathbf{G}$ and CS deterministic TDES supervisor $\mathbf{S}$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let $\mathbf{C}$ be the SD controller constructed from $\mathbf{S}$ using the translation method described in Section 3.7, and let V be the map constructed from $\mathbf{C}$ using Algorithm 8.1. In the $||_{SD}$ setting, the *marked behaviour* of $V/\mathbf{G}$, represented as $L_m(V/\mathbf{G})_{||_{SD}}$, is defined as:

$$L_m(V/\mathbf{G})_{||_{SD}} := L(V/\mathbf{G}) \cap L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$$

Definition 8.1.2. In the $||_{SD}$ setting, V is said to be *nonblocking for* $\mathbf{G}$ if:

$$\overline{L_m(V/\mathbf{G})_{||_{SD}}} = L(V/\mathbf{G})$$

8.1.3 Map V is Well Defined

In Wang (2009), map $\mathcal{V}$ generated from SD controller $\boldsymbol{\mathcal{C}}$ using Algorithm 3.1 is shown to be well defined. Since we have modified Algorithm 3.1 to suit our needs, it is important to show the same result in our $||_{SD}$ setting so that we can consider V as a potential TDES supervisory control.

The proposition given below proves that map V constructed from SD controller $\mathbf{C}$ using Algorithm 8.1 is well defined. As Algorithms 3.1 and 8.1 are logically identical, we have taken the basic idea of this proof from Wang (2009) to prove our desired result.

Proposition 8.1. For TDES plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, and CS deterministic TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let $\mathbf{C}$ be the SD controller constructed from $\mathbf{S}$ using the translation method described in Section 3.7, and let V be the map constructed from $\mathbf{C}$ using Algorithm 8.1. Then, map V is well defined.

Proof. Assume initial conditions.

In order to show that map V is well defined, we need to show that for all $s \in L(\mathbf{G})$, Algorithm 8.1 defines $V(s)$ in only one way. We will show this by analyzing the logic used by Algorithm 8.1 to construct $V(s)$ from $\mathbf{C}$.

By examining Algorithm 8.1, first we note that for all $s \in L(\mathbf{G})$, the algorithm initializes $V(s)$ at **line 2**, and then potentially updates it at **lines 11**, **23** and **25**.

Further examination reveals that for all $s \notin \overline{L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}}$, the algorithm adds Σ_u and $\{\tau\}$ to $V(s)$ at **line 2**, and these strings are not evaluated again in the algorithm. This means for all such s, $V(s)$ is defined only once at **line 2**. Therefore, it is evident that for all $s \notin \overline{L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}}$, $V(s)$ is well defined.

Now we will analyze all the remaining strings s, such that $s \in \overline{L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}}$.

Let $s \in \overline{L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}}$

$\Rightarrow (\exists u \in \Sigma^*)\, su \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ *by definition of prefix closure of* L

$\Rightarrow su \in L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp}$ *by Proposition* 5.1

$\Rightarrow su \in L(\mathbf{S}) \cap L_{samp}$

$\Rightarrow s \in L(\mathbf{S})$ *as* $L(\mathbf{S})$ *is a prefix-closed language* (1)

After **line 2**, the enablement information of $V(s)$ can be modified at **lines 11**, **23** and **25** of the algorithm. By analyzing these lines, we observe that **line 11** updates $V(s)$ if $s \in L_{samp}$. Otherwise, if $s \notin L_{samp}$, then $V(s)$ could possibly be updated once or more at **line 23** or **25**. Thus, we have two cases: **(1)** $s \in L_{samp}$, and **(2)** $s \notin L_{samp}$.

Case 1) $s \in L_{samp}$

Algorithm 8.1 evaluates string-state pairs $(s, \mathbf{q})$, by retrieving them one by one from the set *Pend* at **line 6**. This means the algorithm re-evaluates $V(s)$ of only those strings that were added to *Pend*.

If a sampled string s is never added to *Pend*, its $V(s)$ cannot be modified at **line 11**. Such sampled strings will retain their default enablement information that was assigned to them at **line 2**. Hence, for such s, $V(s)$ will always be well defined.

Thus, without any loss of generality, we assume that s was added to *Pend* at some point during the execution of Algorithm 8.1.

Line 11 updates $V(s)$ by adding the set of enabled prohibitable events, Σ_V, and removing the *tick* event. As we want to show that the algorithm defines $V(s)$ in only one way, it is sufficient to show that whenever **line 11** is executed for s, we always have the same Σ_V to append to $V(s)$. Clearly, as long as Σ_V is the same, executing **line 11** once or more will not make any difference, as $V(s)$ will be updated in the same way every time.

By reviewing the algorithm, we note that Σ_V is formed from output vector $\mathbf{z}$ of controller $\mathbf{C}$ at **line 9**. This output vector $\mathbf{z}$ is in turn obtained from state $\mathbf{q}$ of $\mathbf{C}$ at **line 8**. This means that Σ_V is uniquely defined by state $\mathbf{q}$. Thus, it is sufficient to show that sampled string s will always be paired with state $\mathbf{q}$ of controller $\mathbf{C}$.

As $s \in L_{samp}$, by Definition 3.4.1 of L_{samp}, we have two possible cases: **(a)** $s = \epsilon$, and **(b)** $s \in \Sigma^*.\tau$.

Case 1.a) $s = \epsilon$

The controller $\mathbf{C}$ always starts at its initial or reset state, $\mathbf{q}_{res}$. By the definition of SD controller, $\mathbf{q}_{res}$ corresponds to the empty string, ϵ.

From **line 4** of the algorithm, it is clear that ϵ is always paired with state $\mathbf{q}_{res}$ of $\mathbf{C}$.

Hence, we conclude that if $s = \epsilon$, then s is always paired with the same state $\mathbf{q}_{res}$ of $\mathbf{C}$.

Case (1.a) complete.

Case 1.b) $s \in \Sigma^*.\tau$

By examining the algorithm, we note that for every string-state pair $(s, \mathbf{q})$ added to *Pend*, the non-empty sampled string s of the pair is constructed by concatenating one or more concurrent strings together. Thus, for every such s, we have:

$$(\exists n \in \{1, 2, \ldots\})\,(\exists s_1, s_2, \ldots, s_n \in L_{conc})\; s_1 s_2 \ldots s_n = s$$

By Definition 3.4.3 of concurrent string, we have: $L_{conc} = \Sigma^*_{act}.\tau$

This implies that for a given sampled string s, there is only one way to define the sequence of concurrent strings $s_1 s_2 \dots s_n$. In other words, the sequence of concurrent strings $s_1 s_2 \dots s_n$ in one sampled string s will always be the same.

Except for the first pair $(\epsilon, \mathbf{q}_{res})$, all string-state pairs are added to *Pend* at **line 18**. These pairs are determined at **lines 16** and **17** of the algorithm. These two lines show that starting from the initial state $\mathbf{q}_{res}$, each subsequent state of $\mathbf{C}$ is determined by the current state and the occurrence image of the next concurrent string which is possible in the closed-loop system.

As $\mathbf{S}$ is a CS deterministic supervisor and $s \in L(\mathbf{S}) \cap L_{samp}$ by (1), by the definition of translation functions Γ_I (Definition 3.7.8), Ω (Definition 3.7.10), Λ (Definition 3.7.7) and Δ (Definition 3.7.1), it is evident that the sequence of states reached by the sequence of concurrent strings $s_1 s_2 \dots s_n$ will be unique. This implies the state $\mathbf{q}$ of controller $\mathbf{C}$ that is reached by sampled string $s = s_1 s_2 \dots s_n$ will also be unique.

Hence, we conclude that if $s = \Sigma^*.\tau$, then s is always paired with the same state $\mathbf{q}$ of $\mathbf{C}$.

Case (1.b) complete.

By Cases (1.a) and (1.b), we have shown that sampled string s will always be paired with the same state $\mathbf{q}$ of controller $\mathbf{C}$. In other words, whenever **line 11** is executed for $s \in L_{samp}$, we always have same Σ_V to append to $V(s)$.

Hence, we conclude that for $s \in L_{samp}$, Algorithm 8.1 defines $V(s)$ in only one way.

Case (1) complete.

Case 2) $s \notin L_{samp}$

If $s \notin L_{samp}$, this implies: $(\exists t \in L_{samp})\,(\exists t' \in L_{conc})\, t < s < tt'$

This in turn implies: $(\exists j > 1)\,(\exists \sigma_1, \dots, \sigma_j \in \Sigma)\, t' = \sigma_1 \dots \sigma_j$

As $t' \in L_{conc}$, by the definition of L_{conc}, $\sigma_j = \tau$.

We thus have: $(\exists i \in \{1, \dots, j-1\})\, t\sigma_1, \dots, \sigma_i = s$

In the above setting, we have $j > 1$. This is because if we consider $j = 0$ or $j = 1$, then $t < s < tt'$ would cause a contradiction.

If $j = 0$, then $t' = \epsilon$. As $t' \in L_{conc}$, by the definition of L_{conc}, $t' \neq \epsilon$. Moreover, $t' = \epsilon$ implies $tt' = t$. In this case, we would have $t < s < t$, that could not be true in any case.

If $j = 1$, then $t' = \tau$. Since we require $t < s$, s must contain at least one event more than t, and since $s \notin L_{samp}$, $s \neq \epsilon$ and must not end with a τ. As t' contains only one event, τ, this would not allow $s < tt'$ and $s \notin L_{samp}$. Thus, we must have $j > 1$.

We note that in Algorithm 8.1, if: i) t was never added to *Pend*, or ii) t was added to *Pend* but for all such t' discussed above, if t' fail the condition at **line 14**, then $V(s)$ will never be updated in the algorithm after its initialization. This implies that

$V(s)$ will keep the value assigned to it on **line 2**, even after the complete execution of the algorithm. In such case, we know that $V(s)$ will be well-defined.

Thus, without any loss of generality, we assume that t was added to $Pend$, and our t' passes the condition at **line 14**.

This implies: $t, tt' \in L(\mathbf{S} \,||_{SD}\, \mathbf{G})$

We have: $t' = \sigma_1 \ldots \sigma_i \, \sigma_{i+1} \ldots \sigma_j \in L_{conc}$

As $s = t\sigma_1 \ldots \sigma_i$, it is obvious, from the definition of L_{conc}, that there is only one way to define activity events $\sigma_1 \ldots \sigma_i$, and thus the sampled string t. This implies that there is only one way to define $s = t\sigma_1 \ldots \sigma_i$. Of course, it is possible that there might be multiple ways to define $\sigma_{i+1} \ldots \sigma_j$.

From the result of Case (1), we know that whenever **line 11** is executed for a given $t \in L_{samp}$, we always have the same Σ_V to append to $V(t)$.

By examining Algorithm 8.1, we note that for $s \notin L_{samp}$, the portion of the algorithm that we are interested in, with respect to the modification of $V(s)$, is defined from **lines 19-28**.

In this section, we see that **lines 23** and **25** update $V(s)$ by appending Σ_V, which we know will always be same for $t \in L_{samp}$. In addition, $V(s)$ is determined by $t\sigma_1, \ldots, \sigma_i$ which is also unique for our s, as discussed above. Thus, it is evident that whenever **line 23** or **25** is executed, we always get the same updated $V(s)$.

Hence, we conclude that for $s \notin L_{samp}$, Algorithm 8.1 defines $V(s)$ in only one way.

Case (2) complete.

By Cases (1) and (2), we have shown that for all $s \in \overline{L(\mathbf{S} \,||_{SD}\, \mathbf{G}) \cap L_{samp}}$, Algorithm 8.1 defines $V(s)$ in only one way.

Thus, we have shown that for all $s \in L(\mathbf{G})$, $V(s)$ is well defined.

Hence, we conclude that map V, constructed from SD controller $\mathbf{C}$ using Algorithm 8.1, is well defined. □

8.1.4 Equivalence of V and $\mathcal{V}$

As discussed in Section 8.1.1, Algorithm 8.1 is logically equivalent to Algorithm 3.1 in its way of constructing TDES supervisory control from the SD controller. By Theorem 7.1, we know that the two SD controllers $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$, of the $||_{SD}$ and SD setting respectively, are output equivalent with respect to the closed-loop behaviour, $\mathbf{S} \,||_{SD}\, \mathbf{G}$. This means that for a given plant $\mathbf{G}$, two maps V and $\mathcal{V}$ constructed from SD controllers $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$ using Algorithms 8.1 and 3.1 should also be equivalent. This is formally proven in our next proposition.

This equivalence result essentially bridges the gap between the two settings in terms of their supervisory controls. This further paves our way for reusing some of the existing SD results in deriving and concluding our controllability and nonblocking verification results presented in the next section.

Proposition 8.2. Let TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant to be controlled. Let TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor that is SD controllable with $||_{SD}$ for $\mathbf{G}$, and let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$. Let TDES supervisor $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ constructed using Algorithms 6.1 and 6.2 be CS deterministic. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and V be the map constructed from $\mathbf{C}$ using Algorithm 8.1. Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 3.1. Then, $V = \mathcal{V}$.

Proof. Assume initial conditions.

We first note that $L(\boldsymbol{\mathcal{S}}) = L(min(\mathbf{S}\,||_{SD}\,\mathbf{G})) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$, as state space minimization does not change the closed-loop behaviour of an automaton. (1)

We next note that Algorithm 3.1 will be applied to $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{C}}$, while Algorithm 8.1 will be applied to $\mathbf{S}$ and $\mathbf{C}$.

We note that Algorithms 3.1 and 8.1 are identical except for **line 14**, where Algorithm 3.1 has $ss' \in L(\boldsymbol{\mathcal{S}})$ and Algorithm 8.1 has $ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$.

However, as $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ by (1), **line 14** is now identical for both. Hence, the two algorithms now only differ by the fact that Algorithm 8.1 is applied to $\mathbf{C}$ while Algorithm 3.1 is applied to $\boldsymbol{\mathcal{C}}$. (2)

We will now show that we can replace controller $\boldsymbol{\mathcal{C}}$ by $\mathbf{C}$ in Algorithm 3.1, and $V = \mathcal{V}$ will immediately follow.

First, we need to prove the following claim.

Claim: In the tuples added to *Pend* in either algorithm, the string t of the tuple will always satisfy: $t \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$

As for our purpose, the two algorithms are equal by (2), we will examine Algorithm 8.1 but the result will equally apply to Algorithm 3.1.

We will prove this by induction.

Base Case:

We first note that at **line 4**, *Pend* is initialized to $(\epsilon, \mathbf{q}_{res})$. We thus have $\epsilon \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$, as $\epsilon \in L_{samp}$ by Definition 3.4.2, and as $\mathbf{S}$ and $\mathbf{G}$ have initial states, and by Definition 4.1.1 of the $||_{SD}$ operator.

Inductive Step:

Show: $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ at **line 6** $\Rightarrow ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ at **line 18**

Assume: $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ at **line 6**

For ss' to reach **line 18**, we have $s' \in CB_{\mathbf{G}}(s)$ and $ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ from **lines 13** and **14**.

$\Rightarrow ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $s' \in L_{conc}$ *by Definition* 3.8.2 *of* $CB_{\mathbf{G}}$

$\Rightarrow ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ *by Definition* 3.4.3 *of* L_{conc}

By base case and inductive step, we conclude that each string t of the tuple at **line 6** satisfies $t \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$.

Claim proven.

This implies that for both algorithms, we only care about the outputs of controllers $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$ for strings $t \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$.

Applying Proposition 7.3, it follows that $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$ provide exactly the same output for strings $t \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$.

This means we can replace controller $\boldsymbol{\mathcal{C}}$ by controller $\mathbf{C}$ in Algorithm 3.1 without affecting the algorithm.

The application of Algorithms 3.1 and 8.1 are now identical, so we immediately have $V = \mathcal{V}$, as required. □

8.2 Controllability and Nonblocking Verification

This section presents our controllability and nonblocking verification results for the $||_{SD}$ setting. Essentially, we show that the behaviour of TDES plant $\mathbf{G}$ under the action of SD controller $\mathbf{C}$ is the same as the behaviour of $\mathbf{G}$ under the supervision of TDES supervisor $\mathbf{S}$, given that $||_{SD}$ system satisfies the properties that are stated in the beginning of this chapter. Our results clearly indicate that if the theoretical $||_{SD}$ system is controllable, nonblocking and satisfies the specified properties, then the physically implemented system will also have these properties, and the SD controller will behave as expected with respect to control action, event forcing and nonblocking.

As discussed before, instead of proving all results from scratch, we will use some of the existing SD results to derive and conclude our formal $||_{SD}$ verification results. We will do this by utilizing the equivalence that we have established between the SD and our $||_{SD}$ setting in the previous chapters. As we will be needing these equivalence results in various proofs, we summarize them together in the following corollary. We will then simply cite this corollary in our upcoming proofs, instead of repeating the same argument in multiple proofs.

Corollary 8.1. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$ and has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S}\,||_{SD}\,\mathbf{G}$ be ALF. Let TDES $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 6.1 and 6.2. Then, the following properties are satisfied: **(1)** $\mathbf{S}\,||_{SD}\,\mathbf{G}$ has a finite state space, **(2)** $\boldsymbol{\mathcal{S}}$ has a finite state space, **(3)** $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}}) = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$, **(4)** $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, **(5)** $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, **(6)** $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$, **(7)** $\boldsymbol{\mathcal{S}}$ is CS deterministic, **(8)** $\boldsymbol{\mathcal{S}}$ is ALF, and **(9)** $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ is ALF.

Proof. Assume initial conditions. (1)

1) Show: $\mathbf{S}\,||_{SD}\,\mathbf{G}$ has a finite state space
By (1), we have that $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces. It follows from Definition 4.1.1 of the $||_{SD}$ operator that $\mathbf{S}\,||_{SD}\,\mathbf{G}$ has a finite state space.

2) Show: $\boldsymbol{\mathcal{S}}$ has a finite state space
By (1), we have that $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ is constructed using Algorithms 6.1 and 6.2. It thus follows automatically from Point (1) that $\boldsymbol{\mathcal{S}}$ has a finite state space.

3) Show: $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}}) = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$
By (1), we have that $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ is constructed using Algorithms 6.1 and 6.2. As state space minimization process does not affect the closed and marked languages of an automaton, we conclude that $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}}) = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$.

4) Show: $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$
By (1), we have that $\mathbf{G}$ is complete with $||_{SD}$ for $\mathbf{S}$. As plant completeness is a language based property, by Point (3) and Proposition 5.4, we conclude that $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$.

5) Show: $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour
By (1), we have that $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$. By Point (3) and Proposition 5.5, we conclude that $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour.

6) Show: $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$
By (1), we have that $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$. By Point (3) and Proposition 5.7, we conclude that $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$.

7) Show: $\boldsymbol{\mathcal{S}}$ is CS deterministic
By (1), we have that $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$ and $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ is constructed using Algorithms 6.1 and 6.2. Applying Proposition 6.2, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic.

8) Show: $\boldsymbol{\mathcal{S}}$ is ALF
By Point (1) we have that $\mathbf{S}\,||_{SD}\,\mathbf{G}$ has a finite state space, and by (1) we have that $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF and $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ is constructed using Algorithms 6.1 and 6.2. Applying Theorem 6.1, we conclude that $\boldsymbol{\mathcal{S}}$ is ALF.

9) Show: $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ is ALF
By Point (8), we have that $\boldsymbol{\mathcal{S}}$ is ALF. As $\boldsymbol{\mathcal{S}}$ and $\mathbf{G}$ are defined over the same Σ, by Proposition 5.8 we conclude that $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ is ALF. □

8.2.1 SD Controller as a Supervisory Control

In the $||_{SD}$ setting, the controlled behaviour of the closed-loop system, $\mathbf{S}\,||_{SD}\,\mathbf{G}$, is a combination of the control action of TDES supervisor $\mathbf{S}$ and *tick* disablement mechanism of the $||_{SD}$ operator. This means a *tick* event that is possible in TDES plant $\mathbf{G}$ might be enabled by $\mathbf{S}$ too. However, it still might not be possible in $\mathbf{S}\,||_{SD}\,\mathbf{G}$, as our $||_{SD}$ operator is capable of removing *tick* from the closed-loop system in the presence of enabled prohibitable events. As this path is not possible in the theoretical system model, we want to make sure that our SD controller forbids such strings from

occurring in the implemented system as well, thus preventing the physical system to behave in an undesirable and unexpected way.

We show this in our next proposition by providing sufficient conditions and proving that if a concurrent string is not possible in our theoretical closed-loop system $\mathbf{S}\,||_{SD}\,\mathbf{G}$, then the SD controller $\mathbf{C}$ will not allow it to occur in the physical implementation. By proving this result, we essentially guarantee that the physical system under the control action of SD controller $\mathbf{C}$ does not violate the behaviour, constraints and control laws specified by our theoretical $||_{SD}$ system.

It is important to point out that in the following proposition, we are not comparing the control action of $\mathbf{C}$ with $\mathbf{S}$ only. This is because in the $||_{SD}$ setting, designers are not required to manually incorporate all of the logic of explicit *tick* disablement in the supervisor model, as they have the option of leaving it up to the $||_{SD}$ operator to automatically perform this task for them while constructing the closed-loop system. This implies that the individual control action of $\mathbf{S}$ might not always match with $\mathbf{C}$, which is neither required nor expected in the presence of the $||_{SD}$ operator. Therefore, our goal is to ensure that the control action of $\mathbf{C}$ always remains exactly in line with the controlled behaviour of $\mathbf{S}\,||_{SD}\,\mathbf{G}$, and not only $\mathbf{S}$, which is what we are proving in the proposition given below.

Proposition 8.3. For TDES plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, let TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor that is SD controllable with $||_{SD}$ for $\mathbf{G}$. Let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$ and have $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$.
$(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp})\,(\forall s' \in CB_{\mathbf{G}}(s))$
If s takes $\mathbf{C}$ to state $\mathbf{q}$ and $ss' \notin L(\mathbf{S}\,||_{SD}\,\mathbf{G})$, then $\mathbf{C}$ will reject s'.

Proof. Assume initial conditions. (1)

Let $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ and $s' \in CB_{\mathbf{G}}(s)$. (2)

Assume: s takes $\mathbf{C}$ to state $\mathbf{q}$ and $ss' \notin L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ (3)

We will now show this implies $\mathbf{C}$ will reject s'.

We will use Proposition 3.1 of the SD setting to show our desired result. To do this, we first need to setup things for the SD setting, and show that the preconditions are satisfied.

Let $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 6.1 and 6.2. (4)

$\Rightarrow L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ *by Corollary* 8.1 (5)

We note that except for the CS deterministic property, the remaining conditions needed to apply Proposition 3.1 are all language based. Thus, if they apply to $\mathbf{S}\,||_{SD}\,\mathbf{G}$, they also apply to $\boldsymbol{\mathcal{S}}$.

By Corollary 5.1(v), we have: $L(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$
$\Rightarrow s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \cap L_{samp}$ *by* (2) (6)

By (1), we have that $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$. By (4) and Corollary 8.1,

we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic. (7)

Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$.

Let Λ and Λ' be the state mapping functions (Definition 3.7.7) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$ respectively.

As $\boldsymbol{\mathcal{S}}$ is CS deterministic by (7) and $s \in L(\boldsymbol{\mathcal{S}}) \cap L_{samp}$ by (6), by Proposition 3.2 we conclude that s will take $\boldsymbol{\mathcal{C}}$ to state $\mathbf{q}' = \Lambda'(\xi'(x'_o, s))$. (8)

As $s \in L(\mathbf{S} \,||_{SD}\, \mathbf{G}) \cap L_{samp}$ by (2), by Proposition 5.1 we conclude that $s \in L(\mathbf{S}) \cap L_{samp}$.

We can thus apply Proposition 3.2 and conclude $\mathbf{q} = \Lambda(\xi(x_o, s))$. (9)

As $ss' \notin L(\mathbf{S} \,||_{SD}\, \mathbf{G})$ by (3), this implies $ss' \notin L(\boldsymbol{\mathcal{S}})$ by (5). (10)

We now have $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \cap L_{samp}$ by (6), $s' \in CB_{\mathbf{G}}(s)$ by (2), $\boldsymbol{\mathcal{S}}$ is CS deterministic by (7), s takes $\boldsymbol{\mathcal{C}}$ to state $\mathbf{q}'$ by (8), and $ss' \notin L(\boldsymbol{\mathcal{S}})$ by (10). Also, by (1), (5) and Corollary 8.1 we have that $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$ and has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, and $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$.

We can now apply Proposition 3.1 and conclude that $\boldsymbol{\mathcal{C}}$ will reject s' at state $\mathbf{q}'$. (11)

As all assumptions of Proposition 7.3 are satisfied, we thus conclude:

$\Phi(\Lambda(\xi(x_o, s))) = \Phi'(\Lambda'(\xi'(x'_o, s)))$

$\Rightarrow \Phi(\mathbf{q}) = \Phi'(\mathbf{q}')$ *by* (8) *and* (9)

As $\mathbf{q}$ and $\mathbf{q}'$ have the same output, it follows that if $\boldsymbol{\mathcal{C}}$ rejects s' at state $\mathbf{q}'$ (by (11)), then $\mathbf{C}$ will also reject s' at state $\mathbf{q}$, as required. □

8.2.2 SD Controller and Controllability

In general, a TDES supervisor is more expressive than an SD controller in terms of updating its enablement and forcing information. This is because a supervisor can change this information every time an event occurs. On the other hand, an SD controller is restricted to update its enablement and forcing actions only after a *tick* event, and then it must keep this information constant until the occurrence of the next *tick*.

For our $||_{SD}$ setting, we are interested in showing that despite these differences between the supervisor and the SD controller, the closed-loop behaviour of TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ is exactly the same as the closed-loop behaviour of $\mathbf{G}$ and SD controller $\mathbf{C}$. Please note that the *closed-loop behaviour* of $\mathbf{G}$ and $\mathbf{C}$ is represented as $L(V/\mathbf{G})$.

This notion is proved in our next theorem. We base our result on Theorem 3.1 of the SD setting. Our result is useful as it demonstrates that when we implement our supervisor $\mathbf{S}$ as an SD controller $\mathbf{C}$, we are guaranteed to get the same expected closed-loop behaviour in the physical implementation as our theoretical $||_{SD}$ system, at least with respect to the required enablement and forcing actions of the controller.

Theorem 8.1. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$, have proper time behaviour and $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S} \,||_{SD}\, \mathbf{G}$ be ALF. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and let V be the map constructed from $\mathbf{C}$ using Algorithm 8.1. Then:

$$L(V/\mathbf{G}) = L(\mathbf{S} \,||_{SD}\, \mathbf{G})$$

Proof. Assume initial conditions. (1)

Must show: $L(V/\mathbf{G}) = L(\mathbf{S} \,||_{SD}\, \mathbf{G})$

In order to use Theorem 3.1 of the SD setting to conclude our desired result, we first need to setup things and show that its preconditions are satisfied.

Let $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 6.1 and 6.2. (2)

By (1), (2) and Corollary 8.1, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic.

Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and let $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 3.1.

Let $L(V/\mathbf{G})$ be the closed behaviour of $V/\mathbf{G}$, and let $L(\mathcal{V}/\mathbf{G})$ be the closed behaviour of $\mathcal{V}/\mathbf{G}$.

Now we will show that $L(V/\mathbf{G}) = L(\mathbf{S} \,||_{SD}\, \mathbf{G})$.

We first apply Proposition 8.2 and conclude: $V = \mathcal{V}$

By Definition 2.3.4 of $L(V/\mathbf{G})$ and $L(\mathcal{V}/\mathbf{G})$, this implies: $L(V/\mathbf{G}) = L(\mathcal{V}/\mathbf{G})$

By Corollary 5.1(v), we have: $L(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

Thus, to show that $L(V/\mathbf{G}) = L(\mathbf{S} \,||_{SD}\, \mathbf{G})$, it is sufficient to show:

$$L(\mathcal{V}/\mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$$

By (1), (2), and Corollary 8.1, we have that $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ have finite state spaces, $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, $\mathbf{G}$ has proper time and $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, $\boldsymbol{\mathcal{S}}$ is CS deterministic and SD controllable for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}$ is ALF.

We can now apply Theorem 3.1 and conclude $L(\mathcal{V}/\mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$, as required. □

Our next proposition shows that map V constructed from SD controller $\mathbf{C}$ using Algorithm 8.1 is indeed a TDES supervisory control for TDES plant $\mathbf{G}$. We will base our result on Proposition 3.3 of the SD setting which shows similar result for map $\mathcal{V}$ that is constructed from SD controller $\boldsymbol{\mathcal{C}}$ using Algorithm 3.1.

Proposition 8.4. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$, have proper time behaviour and $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S} \,||_{SD}\, \mathbf{G}$ be ALF. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and let V be the map

constructed from $\mathbf{C}$ using Algorithm 8.1. Then, map V is a TDES supervisory control for $\mathbf{G}$.

Proof. Assume initial conditions. (1)

In order to apply Proposition 3.3 of the SD setting, we first need to setup things and show that its preconditions are satisfied.

Let $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 6.1 and 6.2. (2)

By (1), (2) and Corollary 8.1, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic.

Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and let $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 3.1.

By (1), (2) and Corollary 8.1, we have that $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ have finite state spaces, $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, $\mathbf{G}$ has proper time and $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, $\boldsymbol{\mathcal{S}}$ is CS deterministic and SD controllable for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}$ is ALF.

We can now apply Proposition 3.3 and conclude that map $\mathcal{V}$ is a TDES supervisory control for $\mathbf{G}$. (3)

We next apply Proposition 8.2 and conclude: $V = \mathcal{V}$

As $\mathcal{V}$ is a TDES supervisory control for $\mathbf{G}$ by (3) and $V = \mathcal{V}$, it follows immediately that V is also a TDES supervisory control for $\mathbf{G}$. □

8.2.3 SD Controller and Event Generation

In a typical system, prohibitable events are often part of a supervisor's implementation and they completely depend on the supervisor's discretion for their occurrence. In our $||_{SD}$ setting, this means that the resulting SD controller could potentially make these prohibitable events to occur whenever it wants, possibly even when the plant model does not want them to happen. The occurrence of a prohibitable event might correspond to setting an output of the controller to true, executing a software routine, or sending a message.

In the following theorem, we provide sufficient conditions to make sure that the aforementioned undesirable situation does not occur in our $||_{SD}$ setting. Specifically, we formally prove that if the stated conditions are met, then the SD controller $\mathbf{C}$, translated from TDES supervisor $\mathbf{S}$, cannot generate a prohibitable event when TDES plant $\mathbf{G}$ won't accept it.

This result is beneficial as it forbids the occurrence of *illegal transitions* and prevents the implemented system from violating control laws. It also means that plant model will accurately reflect the $||_{SD}$ system's behaviour when controlled by the SD controller $\mathbf{C}$.

Theorem 8.2. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$, have proper time behaviour

and $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S} \,||_{SD}\, \mathbf{G}$ be ALF. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and let V be the map constructed from $\mathbf{C}$ using Algorithm 8.1.
$(\forall s \in L(V/\mathbf{G}) \cap L_{samp})\,(\forall s' \in \Sigma^*_{act})\,(\forall \sigma \in \Sigma_{hib})$
If $ss' \in L(V/\mathbf{G})$ and σ then physically occurs after ss' and before any other events can occur, then $ss'\sigma \in L(\mathbf{G})$.

Proof. Assume initial conditions. (1)

Let $s \in L(V/\mathbf{G}) \cap L_{samp}$, $s' \in \Sigma^*_{act}$, and $\sigma \in \Sigma_{hib}$. (2)

Assume: $ss' \in L(V/\mathbf{G})$ and that σ physically occurs after ss' and before any other events can occur (3)

Must show: $ss'\sigma \in L(\mathbf{G})$

We will use Theorem 3.2 of the SD setting to show our desired result. To do this, we first need to establish the preconditions of Theorem 3.2 and then the result will follow.

Let $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 6.1 and 6.2. (4)

By (1), (4) and Corollary 8.1, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic.

Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and let $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 3.1.

Let $L(\mathcal{V}/\mathbf{G})$ be the closed behaviour of $\mathcal{V}/\mathbf{G}$.

We can first apply Proposition 8.2 and conclude: $V = \mathcal{V}$
$\Rightarrow L(V/\mathbf{G}) = L(\mathcal{V}/\mathbf{G})$
$\Rightarrow s \in L(\mathcal{V}/\mathbf{G}) \cap L_{samp}, s' \in \Sigma^*_{act}$, and $\sigma \in \Sigma_{hib}$ *by* (2)

We also have that $ss' \in L(\mathcal{V}/\mathbf{G})$ and that σ physically occurs after ss' and before any other events can occur by (3).

By (1), (4) and Corollary 8.1, we have that $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ have finite state spaces, $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, $\mathbf{G}$ has proper time and $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, $\boldsymbol{\mathcal{S}}$ is CS deterministic and SD controllable for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}$ is ALF.

We can now apply Theorem 3.2 and conclude $ss' \in L(\mathbf{G})$. □

8.2.4 SD Controller and Nonblocking

One of the fundamental properties that a TDES is required to satisfy is nonblocking. In the $||_{SD}$ setting, we wish to guarantee that if our theoretical $||_{SD}$ system is nonblocking, then the physical system implemented under the control action of SD controller will retain this property. This is the main focus of our next proof.

The following proposition proves that if specified conditions are satisfied in the $||_{SD}$ setting, then the closed-loop behaviour of TDES plant $\mathbf{G}$ and SD controller $\mathbf{C}$ is

nonblocking if and only if the closed-loop behaviour of $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ is nonblocking.

Proposition 8.5. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$, have proper time behaviour and $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S}\,||_{SD}\,\mathbf{G}$ be ALF. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and let V be the map constructed from $\mathbf{C}$ using Algorithm 8.1. Then V is nonblocking for $\mathbf{G}$ if and only if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is nonblocking.

Proof. Assume initial conditions. (1)

Must show: V is nonblocking for $\mathbf{G}$ if and only if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is nonblocking

To show this, it is sufficient to show:

$$L(V/\mathbf{G}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \quad \text{and} \quad L_m(V/\mathbf{G})_{||SD} = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$$

Applying Theorem 8.1 (by (1)), we conclude: $L(V/\mathbf{G}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ (2)

Now all that remains is to show: $L_m(V/\mathbf{G})_{||SD} = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$

By Definition 8.1.1 of $L_m(V/\mathbf{G})_{||SD}$, we have:

$$\begin{aligned} L_m(V/\mathbf{G})_{||SD} &= L(V/\mathbf{G}) \cap L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \\ &= L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \quad by\ (2) \\ &= L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \quad as\ L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \subseteq L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \end{aligned}$$

We thus conclude that V is nonblocking for $\mathbf{G}$ if and only if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is nonblocking. □

In the SD supervisory control theory, the SD setting is proven to be robust with respect to multiple variations of concurrent strings and nonblocking. Specifically, if theoretical TDES system is nonblocking, then TDES plant $\mathbf{G}$ under the control of SD controller $\boldsymbol{\mathcal{C}}$ is shown to be nonblocking, even if multiple concurrent strings with the same occurrence image are possible at a given sampled state in the theoretical SD system and only one of these concurrent strings is actually possible in the physical implementation.

We also wish to demonstrate such robustness with respect to nonblocking for our $||_{SD}$ setting. In order to be able to do that, first we present a supporting proposition that will help us in proving our main result. In the following proposition, we show that for any V' that is a TDES supervisory control for TDES plant $\mathbf{G}$, V' is concurrent supervisory control equivalent (CSCE: Definition 3.9.1) to TDES supervisory control V of the $||_{SD}$ setting if and only if V' is CSCE to TDES supervisory control $\mathcal{V}$ of the SD setting.

Proposition 8.6. Let TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant to be controlled. Let TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor that is SD controllable with $||_{SD}$ for $\mathbf{G}$, and let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$. Let TDES supervisor $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ constructed using Algorithms 6.1 and

6.2 be CS deterministic. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and V be the map constructed from $\mathbf{C}$ using Algorithm 8.1. Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 3.1. Then, for all V' that are TDES supervisory controls for $\mathbf{G}$, V' is concurrent supervisory control equivalent to V if and only if V' is concurrent supervisory control equivalent to $\mathcal{V}$.

Proof. Assume initial conditions. (1)

Let V' be a TDES supervisory control for $\mathbf{G}$.

Must show: V' is concurrent supervisory control equivalent (CSCE) to V if and only if V' is CSCE to $\mathcal{V}$

Let $L(V/\mathbf{G})$, $L(\mathcal{V}/\mathbf{G})$ and $L(V'/\mathbf{G})$ be the closed behaviours of $V/\mathbf{G}$, $\mathcal{V}/\mathbf{G}$ and $V'/\mathbf{G}$ respectively.

We can now apply Proposition 8.2 (by (1)) and conclude: $V = \mathcal{V}$ (2)

$\Rightarrow L(V/\mathbf{G}) = L(\mathcal{V}/\mathbf{G})$ (3)

We next note that by Definition 3.9.1, if V' is CSCE to V, this implies:

1) $(\forall s \in L(\mathbf{G}))\, V'(s) \subseteq V(s)$

2) $(\forall s \in L(V'/\mathbf{G}) \cap L_{samp})\, (\forall s' \in L_{conc})\, ss' \in L(V/\mathbf{G}) \Rightarrow$
$(\exists s'' \in L_{conc})\, ss'' \in L(V'/\mathbf{G}) \wedge Occu(s') = Occu(s'')$

However, as $V = \mathcal{V}$ by (2), and $L(V/\mathbf{G}) = L(\mathcal{V}/\mathbf{G})$ by (3), we can substitute into the above and get:

1) $(\forall s \in L(\mathbf{G}))\, V'(s) \subseteq \mathcal{V}(s)$

2) $(\forall s \in L(V'/\mathbf{G}) \cap L_{samp})\, (\forall s' \in L_{conc})\, ss' \in L(\mathcal{V}/\mathbf{G}) \Rightarrow$
$(\exists s'' \in L_{conc})\, ss'' \in L(V'/\mathbf{G}) \wedge Occu(s') = Occu(s'')$

This implies that V' is CSCE to $\mathcal{V}$. □

We will now present our final result that proves the robustness of the $||_{SD}$ setting with respect to different variations of concurrent strings and nonblocking. Our next theorem shows that if a theoretical $||_{SD}$ system satisfies the stated conditions and the closed-loop behaviour of TDES plant $\mathbf{G}$ and SD controller $\mathbf{C}$ is nonblocking, then any of its CSCE variations will also be nonblocking. This theorem makes use of Theorem 3.3 of the SD setting to conclude the desired result.

This result is beneficial as it provides liberty to practitioners to choose any specific implementation of $\mathbf{S}\,||_{SD}\,\mathbf{G}$ without having to worry about potential blocking of the physical system. If they fulfill the specified conditions, then they are guaranteed that the physical system under the action of the SD controller $\mathbf{C}$ will be nonblocking, even if their chosen implementation only allows a subset of variations of a concurrent string out of all variations possible in $\mathbf{S}\,||_{SD}\,\mathbf{G}$.

Theorem 8.3. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} =$

$(X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$, have proper time behaviour and $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S} \,||_{SD}\, \mathbf{G}$ be ALF. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and let V be the map constructed from $\mathbf{C}$ using Algorithm 8.1. Then, for all V' that are TDES supervisory controls for $\mathbf{G}$, if V is nonblocking for $\mathbf{G}$ and V' is concurrent supervisory control equivalent to V, then V' is also nonblocking for $\mathbf{G}$.

Proof. Assume initial conditions. (1)

Let V' be a TDES supervisory control for $\mathbf{G}$.

Assume: V' is concurrent supervisory control equivalent (CSCE) to V and that V is nonblocking for $\mathbf{G}$. By Definition 8.1.2, this implies: $\overline{L_m(V/\mathbf{G})_{||_{SD}}} = L(V/\mathbf{G})$ (2)

Must show: V' is nonblocking for $\mathbf{G}$

Let $L(V'/\mathbf{G})$ and $L_m(V'/\mathbf{G})$ be the closed and marked behaviour of $V'/\mathbf{G}$.

Sufficient to show: $\overline{L_m(V'/\mathbf{G})_{||_{SD}}} = L(V'/\mathbf{G})$

We first define our setting and notation for the proof.

Let $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 6.1 and 6.2. (3)

By (1), (3) and Corollary 8.1, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic.

Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and let $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 3.1.

Let $L(V/\mathbf{G})$ and $L(\mathcal{V}/\mathbf{G})$ be the closed behaviours of $V/\mathbf{G}$ and $\mathcal{V}/\mathbf{G}$ respectively.

We now apply Proposition 8.2 and conclude: $V = \mathcal{V}$

$\Rightarrow L(V/\mathbf{G}) = L(\mathcal{V}/\mathbf{G})$ (4)

Now we will show that $\overline{L_m(V'/\mathbf{G})_{||_{SD}}} = L(V'/\mathbf{G})$.

By Definition 8.1.1 of $L_m(V'/\mathbf{G})_{||_{SD}}$, it is sufficient to show:

$\overline{L(V'/\mathbf{G}) \cap L_m(\mathbf{S} \,||_{SD}\, \mathbf{G})} = L(V'/\mathbf{G})$

$\Rightarrow \overline{L(V'/\mathbf{G}) \cap L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})} = L(V'/\mathbf{G})$ *by Corollary* 5.1(vi)

$\Rightarrow \overline{L_m(V'/\mathbf{G})} = L(V'/\mathbf{G})$ *by Definition* 3.8.3 *of* $L_m(V'/\mathbf{G})$

This means, in order to show that V' is nonblocking for $\mathbf{G}$ in our $||_{SD}$ setting, it is sufficient to show that V' is nonblocking for $\mathbf{G}$ in the SD setting (Definition 3.8.4).

We will show this by using Theorem 3.3 of the SD setting. We will first establish the preconditions of Theorem 3.3 and then the result will follow.

By (2), we have: $\overline{L_m(V/\mathbf{G})_{||_{SD}}} = L(V/\mathbf{G})$

$\Rightarrow \overline{L(V/\mathbf{G}) \cap L_m(\mathbf{S} \,||_{SD}\, \mathbf{G})} = L(V/\mathbf{G})$ *by Definition* 8.1.1 *of* $L_m(V/\mathbf{G})_{||_{SD}}$

$\Rightarrow \overline{L(\mathcal{V}/\mathbf{G}) \cap L_m(\mathbf{S} \,||_{SD}\, \mathbf{G})} = L(\mathcal{V}/\mathbf{G})$ *by* (4)

$\Rightarrow \overline{L(\mathcal{V}/\mathbf{G}) \cap L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})} = L(\mathcal{V}/\mathbf{G})$ *by Corollary* 5.1(vi)

$\Rightarrow \overline{L_m(\mathcal{V}/\mathbf{G})} = L(\mathcal{V}/\mathbf{G})$ *by Definition* 3.8.3 *of* $L_m(\mathcal{V}/\mathbf{G})$

By Definition 3.8.4, this indicates that $\mathcal{V}$ is nonblocking for $\mathbf{G}$.

Applying Proposition 8.6, we note that as V' is CSCE to V by (2), this implies that V' is CSCE to $\mathcal{V}$.

By (1), (3) and Corollary 8.1, we have that $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ have finite state spaces, $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, $\mathbf{G}$ has proper time and $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, $\boldsymbol{\mathcal{S}}$ is CS deterministic and SD controllable for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}$ is ALF.

We now apply Theorem 3.3 and conclude that V' is nonblocking for $\mathbf{G}$ in the SD setting.

By showing that V' is nonblocking for $\mathbf{G}$ in the SD setting, we have thus shown that V' is nonblocking for $\mathbf{G}$ in our $||_{SD}$ setting, as required. □

Chapter 9

Symbolic Verification in SD Synchronous Product Setting

In this chapter, we discuss theoretical concepts and predicate-based algorithms to symbolically verify various properties in our $||_{SD}$ setting. This chapter is based on symbolic verification of the SD supervisory control methodology presented in Wang (2009), who in turn built upon the symbolic computation and verification work done by Song (2006) and Ma (2004).

We begin this chapter by introducing the fundamental concepts of predicates and predicate transformers. This is followed by a discussion on how to use logic formulas to represent state subsets and transitions in our $||_{SD}$ setting. After that, we describe the symbolic computation of transitions, inverse transitions and predicate transformers. Finally, we present algorithms that can be used to verify various properties in our $||_{SD}$ setting. All data representations, computations and verifications discussed in this chapter are based on ordered binary decision diagrams (BDD) (Bryant, 1986, 1992).

Please note that the algorithms discussed in this chapter were originally developed as part of the SD supervisory control methodology by Wang (2009). We have tweaked them to match our adapted properties of the $||_{SD}$ setting introduced in Chapter 4. Since some properties of our $||_{SD}$ setting are logically similar to the SD setting, their corresponding algorithm steps remain unchanged. These unmodified algorithms are included in Appendix B for the sake of completeness.

Note: In this chapter, we will represent *logical equivalence* between state predicates by '$\equiv$', *logical true* by 'T' and *logical false* by 'F' respectively. Also, we will use $\boldsymbol{\mathcal{S}}$ to refer to the supervisor of the SD setting (Chapter 3).

9.1 Predicates and Predicate Transformers

This section introduces the concepts of state predicates and predicate transformers from Song (2006).

9.1.1 State Predicates

For the following definitions, let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$.

Definition 9.1.1. A *predicate* P defined on state set Q is a function $P : Q \to \{T, F\}$ *identified* by the corresponding state subset $Q_P := \{q \in Q \mid P(q) = T\} \subseteq Q$.

If $q \in Q_P$, then $q \models P$ means "q *satisfies* P" or "P *includes* q". Thus, we have $q \models P \iff P(q) = T$.

Definition 9.1.2. A predicate defined on the state set of a TDES is referred to as a *state predicate*. The state predicate *true* is identified by Q, state predicate *false* by $\emptyset$, and state predicate P_m by Q_m.

We write $Pred(Q)$ to represent the set of all predicates defined on Q. Thus, $Pred(Q)$ is identified by $\text{Pwr}(Q)$. For $P \in Pred(Q)$, $st(P)$ denotes the corresponding state subset $Q_P \subseteq Q$ which identifies P. We use $pr(Q)$ to represent the predicate that is identified by Q.

For $q \in Q$ and $P, P_1, P_2 \in Pred(Q)$, the following predicate operations can be used to build various boolean expressions:

- $(\neg P)(q) = T \iff P(q) = F$
- $(P_1 \wedge P_2)(q) = T \iff P_1(q) = T$ and $P_2(q) = T$
- $(P_1 \vee P_2)(q) = T \iff P_1(q) = T$ or $P_2(q) = T$
- $(P_1 - P_2)(q) = T \iff P_1(q) = T$ and $P_2(q) = F$

Definition 9.1.3. The partial order relation $\preceq$ over $Pred(Q)$ is defined as:

$$(\forall P_1, P_2 \in Pred(Q))\, P_1 \preceq P_2 \iff (P_1 \wedge P_2) \equiv P_1$$

It is obvious that $Q_{P_1} \subseteq Q_{P_2} \iff P_1 \preceq P_2$. Thus, we have $(\forall q \in Q)\, q \models P_1 \implies q \models P_2$.

Definition 9.1.4. For some state set Q, let $P_1, P_2 \in Pred(Q)$. P_1 is a *subpredicate* of P_2 if $P_1 \preceq P_2$. We say P_1 is *stronger* than P_2, and P_2 is *weaker* than P_1.

$Sub(P)$ represents the set of all subpredicates of $P \in Pred(Q)$ such that $Sub(P)$ is identified by $\text{Pwr}(Q_P)$.

9.1.2 Predicate Transformers

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ and $P \in Pred(Q)$. A *predicate transformer* is defined as a function $f : Pred(Q) \to Pred(Q)$. In our subsequent sections, we will use the following predicate transformers from Song (2006).

i) $R(\mathbf{G}, P)$

The *reachability predicate* $R(\mathbf{G}, P)$ holds true for those states in $\mathbf{G}$ that can be reached from q_o by states satisfying P. It is inductively defined as follows:

1. $q_o \models P \implies q_o \models R(\mathbf{G}, P)$.

2. $q \models R(\mathbf{G}, P)$ & $\sigma \in \Sigma$ & $\delta(q,\sigma)!$ & $\delta(q,\sigma) \models P \implies \delta(q,\sigma) \models R(\mathbf{G}, P)$.

3. No other states satisfy $R(\mathbf{G}, P)$.

In simple words, a state $q \models R(\mathbf{G}, P)$ if and only if there exists a path from q_o to q in $\mathbf{G}$ and each state in that path satisfies P. $R(\mathbf{G}, true)$ represents the set of all reachable states in Q.

ii) $CR(\mathbf{G}, P)$

The *coreachability predicate* $CR(\mathbf{G}, P)$ holds true for those states in $\mathbf{G}$ that can reach a marked state by states satisfying P. It is inductively defined as follows:

1. $P_m \wedge P \equiv false \implies CR(\mathbf{G}, P) \equiv false$.

2. $q \models P_m \wedge P \implies q \models CR(\mathbf{G}, P)$.

3. $q \models CR(\mathbf{G}, P)$ & $q' \models P$ & $\sigma \in \Sigma$ & $\delta(q',\sigma)!$ & $\delta(q',\sigma) = q \implies q' \models CR(\mathbf{G}, P)$.

4. No other states satisfy $CR(\mathbf{G}, P)$.

In other words, a state $q \models CR(\mathbf{G}, P)$ if and only if there exists a path from q to some marked state in $\mathbf{G}$ and each state in that path satisfies P. $CR(\mathbf{G}, true)$ represents the set of all coreachable states in Q.

iii) $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$

Let $P' \in Pred(Q)$ and $\Sigma' \subseteq \Sigma$. Once $\mathbf{G}$, P' and Σ' are fixed, $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ becomes a predicate transformer. The predicate $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ holds true for those states in $\mathbf{G}$ that can reach a state satisfying P' by states satisfying P and transitions with events in Σ'. It is inductively defined as follows:

1. $P' \wedge P \equiv false \implies \mathcal{CR}(\mathbf{G}, P', \Sigma', P) \equiv false$.

2. $q \models P' \wedge P \implies q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$.

3. $q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ & $q' \models P$ & $\sigma \in \Sigma'$ & $\delta(q',\sigma)!$ & $\delta(q',\sigma) = q \implies q' \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$.

4. No other states satisfy $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$.

This means that a state $q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ if and only if there exists a path from q to a state satisfying P' in $\mathbf{G}$ and each state in that path satisfies P and each transition event σ is in Σ'.

By comparing the definitions of $\mathcal{CR}$ and CR, we note that $\mathcal{CR}(\mathbf{G}, P_m, \Sigma, P) \equiv CR(\mathbf{G}, P)$.

9.2 Symbolic Representation

In this section, we present the symbolic representation for states and transitions in our $||_{SD}$ setting. Specifically, we discuss how to use logic formulas to represent state subsets and transitions for our $||_{SD}$ system. We have based our work on the symbolic

representation of the SD setting given in Wang (2009), who in turn borrowed it from Song (2006).

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD}\, \ldots \,||_{SD}\, \mathbf{G}_n$ be constructed by synchronizing component TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ where $i = 1, 2, \ldots, n$, using the SD synchronous product operator. For any state $q \in Q$, by the definition of Q in the $||_{SD}$ operator (Definition 4.1.1), we have $q = (q_1, q_2, \ldots, q_n)$, where $q_i \in Q_i$.

It is worth pointing out that TDES $\mathbf{G}$ might contain some unreachable states. However, checking for unreachable states while verifying different properties of the TDES is expensive, and does not seem to provide any benefit as these unreachable states do not contribute towards the closed and marked behaviour of $\mathbf{G}$, i.e. $L(\mathbf{G})$ and $L_m(\mathbf{G})$. As a result, the property is first checked (possibly including unreachable states) and then a reachability check is performed over the entire system, and any unreachable states are excluded from the results. This also allows us to do one reachability check, and share the results across several algorithms.

9.2.1 State Subsets

Definition 9.2.1. For TDES $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD}\, \ldots \,||_{SD}\, \mathbf{G}_n$, let $i = 1, 2, \ldots, n$ and $q_i \in Q_i$. The *state variable* v_i for the i^{th} component TDES $\mathbf{G}_i$ is a variable of domain Q_i. If v_i is assigned the value q_i, then $v_i = q_i$ returns T, otherwise it returns F.

Please note that '=' has been used to test if v_i has been assigned the value q_i.

Definition 9.2.2. For TDES $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD}\, \ldots \,||_{SD}\, \mathbf{G}_n$, the *state variable vector* $\mathbf{v}$ is a vector $[v_1, v_2, \ldots, v_n]$ of state variables v_i from each component TDES $\mathbf{G}_i$, where $i = 1, 2, \ldots, n$. For state subset $A \subseteq Q$, the predicate P_A for A can be written as:

$$P_A(\mathbf{v}) := \bigvee_{q \in A} (v_1 = q_1 \wedge v_2 = q_2 \wedge \ldots \wedge v_n = q_n)$$

For convenience, instead of $P_A(\mathbf{v})$, we will simply write P_A if $\mathbf{v}$ is understood.

9.2.2 Transitions

Definition 9.2.3. For TDES $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD}\, \ldots \,||_{SD}\, \mathbf{G}_n$, let $\Sigma_{hib} \subset \Sigma$ and $\sigma \in \Sigma$. A *transition predicate* $N_\sigma : Q \times Q \to \{T, F\}$ is a boolean function that identifies all the transitions for σ in $\mathbf{G}$ and is defined as follows:

$$(\forall q, q' \in Q)\, N_\sigma(q, q') := \begin{cases} T & \text{if } \delta(q,\sigma)! \;\&\; \delta(q,\sigma) = q' \;\&\; \big((\sigma \neq \tau) \text{ OR} \\ & \quad ((\sigma = \tau) \;\&\; (\forall \sigma' \in \Sigma_{hib})\, \neg\, \delta(q, \sigma')!)\big) \\ F & \text{otherwise} \end{cases}$$

For each TDES, two different sets of state variables are needed to distinguish between source and destination states of transitions. These state variables and their corresponding vectors are defined below.

Definition 9.2.4. For TDES $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD}\, \ldots \,||_{SD}\, \mathbf{G}_n$, let $i = 1, 2, \ldots, n$. For each component TDES $\mathbf{G}_i$, we have the *normal state variable* v_i (source state)

and the *prime state variable* v_i' (destination state), both with domain Q_i. For $\mathbf{G}$, we have the *normal state variable vector* $\mathbf{v} = [v_1, v_2, \ldots, v_n]$ and the *prime state variable vector* $\mathbf{v}' = [v_1', v_2', \ldots, v_n']$.

For each $\sigma \in \Sigma$, the transition predicate for σ, N_σ, can be written as follows:

$$N_\sigma(\mathbf{v}, \mathbf{v}') := \begin{cases} \bigwedge\limits_{\{1 \le i \le n\}} \left(\bigvee\limits_{\{q_i, q_i' \in Q_i | \delta_i(q_i, \sigma) = q_i'\}} (v_i = q_i) \wedge (v_i' = q_i') \right) & \text{if } \mathbf{X} \\ F & \text{otherwise} \end{cases}$$

where $\mathbf{X} = (\sigma \neq \tau)$ OR $\left(\sigma = \tau \;\&\; (\forall \sigma' \in \Sigma_{hib}) \neg \left(\bigwedge\limits_{\{1 \le i \le n\}} \left(\bigvee\limits_{\{q_i \in Q_i | \delta_i(q_i, \sigma')!\}} (v_i = q_i) \right) \right) \right)$

Essentially, it says that for each $\sigma \in \Sigma - \{\tau\}$, if we set $\mathbf{v} = q$ and $\mathbf{v}' = q'$ such that $\delta(q, \sigma) = q'$, then $N_\sigma(\mathbf{v}, \mathbf{v}')$ will return T. However, for $\sigma = \tau$, $N_\sigma(\mathbf{v}, \mathbf{v}')$ will return T only if we set $\mathbf{v} = q$ and $\mathbf{v}' = q'$ such that $\delta(q, \sigma) = q'$ and for all events $\sigma' \in \Sigma_{hib}$, $\neg\, \delta(q, \sigma')!$.

When a TDES is designed as several smaller component TDES, designers often model these components over different event sets. In order to use the above-mentioned formula for N_σ, selfloops need to be added at every state of the component TDES for events that are missing from their event sets. This makes the transition predicate a lot more complicated and cluttered. In order to resolve this issue, the following version of N_σ has been defined.

Definition 9.2.5. To represent the transition for a given $\sigma \in \Sigma$, we use the *transition tuple* $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ such that $\mathbf{v}_\sigma := \{v_i \in \mathbf{v} \,|\, \sigma \in \Sigma_i\}, \mathbf{v}'_\sigma := \{v_i' \in \mathbf{v}' \,|\, \sigma \in \Sigma_i\}$ and N_σ is defined as:

$$N_\sigma(\mathbf{v}, \mathbf{v}') := \begin{cases} \bigwedge\limits_{\{1 \le i \le n | \sigma \in \Sigma_i\}} \left(\bigvee\limits_{\{q_i, q_i' \in Q_i | \delta_i(q_i, \sigma) = q_i'\}} (v_i = q_i) \wedge (v_i' = q_i') \right) & \text{if } \mathbf{X} \\ F & \text{otherwise} \end{cases}$$

where $\mathbf{X} = (\sigma \neq \tau)$ OR $\left(\sigma = \tau \,\&\, (\forall \sigma' \in \Sigma_{hib}) \neg \left(\bigwedge\limits_{\{1 \le i \le n | \sigma' \in \Sigma_i\}} \left(\bigvee\limits_{\{q_i \in Q_i | \delta_i(q_i, \sigma')!\}} (v_i = q_i) \right) \right) \right)$

It is noteworthy that although selflooped transitions are not explicitly specified in the above definition, the tuple still expresses the selfloop information. This implies that this definition can be used to create transition tuples for selflooped components as well.

9.3 Symbolic Computation

By using the logic formula representation for state subsets and transitions of our $||_{SD}$ system defined in the previous section, this section discusses the symbolic computation of transitions, inverse transitions and predicate transformers with respect to our $||_{SD}$

setting. We have built our work on the symbolic computation work done by Song (2006), which was used for the SD setting by Wang (2009).

9.3.1 Transitions and Inverse Transitions

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD} \,\ldots\, ||_{SD}\, \mathbf{G}_n$ be constructed by synchronizing component TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ where $i = 1, 2, \ldots, n$, using the SD synchronous product operator.

For any state $q \in Q$ and event $\sigma \in \Sigma$, we want to compute the transition $\delta(q, \sigma)$. An efficient way to compute transitions is to compute the predicate of the set of next states from the predicate of the set of current states.

For $P \in Pred(Q)$, we can directly compute the function $\hat{\delta} : Pred(Q) \times \Sigma \rightarrow Pred(Q)$ which is defined as follows:

$$(\forall P \in Pred(Q))\,(\forall \sigma \in \Sigma)\,\hat{\delta}(P, \sigma) := pr(\{q' \in Q \,|\, (\exists q \models P)\,\delta(q, \sigma) = q' \wedge \big((\sigma \neq \tau) \vee ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib})\,\neg\,\delta(q, \sigma')!)\big)\})$$

In order to compute the predicate of the set of source states from the predicate of the set of destination states, we define the inverse function $\hat{\delta}^{-1} : Pred(Q) \times \Sigma \rightarrow Pred(Q)$ as follows:

$$(\forall P \in Pred(Q))\,(\forall \sigma \in \Sigma)\,\hat{\delta}^{-1}(P, \sigma) := pr(\{q \in Q \,|\, \delta(q, \sigma) \models P \wedge \big((\sigma \neq \tau) \vee ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib})\,\neg\,\delta(q, \sigma')!)\big)\})$$

As BDD (Bryant, 1986, 1992) does not support first order logic by itself, Song (2006) has used the *existential quantifier elimination method for finite domain* (Arnon, 1988) to compute $\hat{\delta}(P, \sigma)$ and $\hat{\delta}^{-1}(P, \sigma)$. We will use the same method to compute our functions $\hat{\delta}$ and $\hat{\delta}^{-1}$ in the $||_{SD}$ setting.

Definition 9.3.1. For TDES $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD} \,\ldots\, ||_{SD}\, \mathbf{G}_n$, let $\sigma \in \Sigma$ and $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. For $i = 1, 2, \ldots, n$, if $v_i \in \mathbf{v}_\sigma$ and $v'_i \in \mathbf{v}'_\sigma$, then $\exists v_i N_\sigma$ and $\exists v'_i N_\sigma$ are defined as follows:

$$\exists v_i N_\sigma := \bigvee_{q_i \in Q_i} N_\sigma[q_i/v_i] \qquad\qquad \exists v'_i N_\sigma := \bigvee_{q_i \in Q_i} N_\sigma[q_i/v'_i]$$

Here, $N_\sigma[q_i/v_i]$ is the resulting predicate with each term v_i of N_σ substituted by q_i, and $N_\sigma[q_i/v'_i]$ is the resulting predicate with each term v'_i of N_σ substituted by q_i. In simple words, $\exists v_i$ and $\exists v'_i$ eliminate the variables v_i and v'_i respectively from N_σ.

For $\sigma \in \Sigma$, let $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. For $k \in \{1, 2, \ldots, n\}$, let $\mathbf{v}_\sigma = \{\hat{v_1}, \hat{v_2}, .., \hat{v_k}\}$ and $\mathbf{v}'_\sigma = \{\hat{v'_1}, \hat{v'_2}, \ldots, \hat{v'_k}\}$.

For convenience, we write $\exists \mathbf{v}_\sigma N_\sigma$ to represent $\exists \hat{v_1}(\exists \hat{v_2} \ldots (\exists \hat{v_k} N_\sigma) \ldots)$. The resulting logic formula $\exists \mathbf{v}_\sigma N_\sigma$ contains only the prime variables in $\mathbf{v}'_\sigma$. If we substitute all the prime variables by normal variables, denoted as $\exists \mathbf{v}_\sigma N_\sigma[\mathbf{v}'_\sigma \rightarrow \mathbf{v}_\sigma]$, then the resulting predicate represents the set of *destination states* for σ transitions in $\mathbf{G}$. This means that each state in this set has a σ transition entering it, as defined by our N_σ.

This variable substitution is required because normal variables are used to express the logic formula of a state subset predicate.

Likewise, for convenience, we write $\exists \mathbf{v}'_\sigma N_\sigma$ to represent $\exists \hat{v'_1}(\exists \hat{v'_2} \ldots (\exists \hat{v'_k} N_\sigma) \ldots)$. The resulting logic formula $\exists \mathbf{v}'_\sigma N_\sigma$ contains only the normal variables in $\mathbf{v}_\sigma$, therefore no variable substitution is required in this case. $\exists \mathbf{v}'_\sigma N_\sigma$ represents the predicate for the set of *source states* for σ transitions in $\mathbf{G}$. This means that each state in this set has a σ transition leaving it, as defined by our N_σ.

From the above description, it is obvious that $\exists \mathbf{v}_\sigma N_\sigma[\mathbf{v}'_\sigma \rightarrow \mathbf{v}_\sigma]$ computes the predicate representing the set of destination states $\{q' \in Q \,|\, (\exists q \in Q)\, \delta(q, \sigma) = q' \wedge ((\sigma \neq \tau) \vee ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib})\, \neg\delta(q, \sigma')!))\}$. Similarly, $\exists \mathbf{v}'_\sigma N_\sigma$ computes the predicate representing the set of source states $\{q \in Q \,|\, \delta(q, \sigma)! \wedge ((\sigma \neq \tau) \vee ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib})\, \neg\delta(q, \sigma')!))\}$.

By using the existential quantifier elimination method, we can now compute $\hat{\delta}$ and $\hat{\delta}^{-1}$ symbolically as follows.

Definition 9.3.2. For TDES $\mathbf{G} = \mathbf{G}_1 ||_{SD} \mathbf{G}_2 ||_{SD} \ldots ||_{SD} \mathbf{G}_n$, let $\sigma \in \Sigma$, $P \in Pred(Q)$ and $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. Then, $\hat{\delta}(P, \sigma)$ is computed as follows:

$$\hat{\delta}(P, \sigma) := (\exists \mathbf{v}_\sigma (N_\sigma \wedge P))\, [\mathbf{v}'_\sigma \rightarrow \mathbf{v}_\sigma]$$

In the above definition, by first computing $N_\sigma \wedge P$, we are restricting σ transitions to only those source states that satisfy P.

Definition 9.3.3. For TDES $\mathbf{G} = \mathbf{G}_1 ||_{SD} \mathbf{G}_2 ||_{SD} \ldots ||_{SD} \mathbf{G}_n$, let $\sigma \in \Sigma$, $P \in Pred(Q)$ and $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. Then, $\hat{\delta}^{-1}(P, \sigma)$ is computed as follows:

$$\hat{\delta}^{-1}(P, \sigma) := \exists \mathbf{v}'_\sigma (N_\sigma \wedge (P[\mathbf{v}_\sigma \rightarrow \mathbf{v}'_\sigma]))$$

In this definition, $P[\mathbf{v}_\sigma \rightarrow \mathbf{v}'_\sigma]$ returns predicate P with its normal variables substituted by prime variables. As prime variables represent destination states, this has the effect of restricting σ transitions to only those destination states that satisfy P.

9.3.2 Predicate Transformers

In order to compute the predicate transformers R and $\mathcal{CR}$ defined in Section 9.1.2, Algorithms 9.1 and 9.2 have been taken from Song (2006). Please refer to Song (2006) for a detailed description of these algorithms.

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 ||_{SD} \mathbf{G}_2 ||_{SD} \ldots ||_{SD} \mathbf{G}_n$ be constructed by synchronizing component TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ where $i = 1, 2, \ldots, n$, using the SD synchronous product operator. Let $P \in Pred(Q)$.

Algorithm 9.1 $R(\mathbf{G}, P)$

1: $P_1 \leftarrow P \wedge pr(\{q_o\})$
2: **repeat**
3: $\quad P_2 \leftarrow P_1$
4: $\quad$ **for** $i \leftarrow 1$ to n **do**
5: $\quad\quad P_3 \leftarrow false$
6: $\quad\quad$ **repeat**
7: $\quad\quad\quad P_{new} \leftarrow P_1 - P_3$
8: $\quad\quad\quad P_3 \leftarrow P_1$
9: $\quad\quad\quad P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma_i} (\hat{\delta}(P_{new}, \sigma) \wedge P) \right)$
10: $\quad\quad$ **until** $P_1 \equiv P_3$
11: $\quad$ **end for**
12: **until** $P_1 \equiv P_2$
13: **return** P_1

Reachability Check

Algorithm 9.1[1] computes $R(\mathbf{G}, P)$ by taking two parameters as input, a TDES $\mathbf{G}$ and a predicate P. It then computes and returns a predicate P_1 containing the set of states in $\mathbf{G}$ that are reachable by the initial state q_o via states satisfying P.

It is interesting to note that this algorithm has been used in the SD setting by Wang (2009), and we will also use the same algorithm in our $||_{SD}$ setting without any modification. Although the steps of the algorithm are the same, it will most likely give different results in the SD and the $||_{SD}$ setting.

In the SD setting, the input to Algorithm 9.1 is TDES $\mathbf{G}$ that represents the closed-loop system formed by combining plant and supervisor models defined over the same event set using the synchronous product. For this input, it returns a predicate representing the set of reachable states of this closed-loop system. This predicate is primarily computed at **line 9** by using the definition of $\hat{\delta}$ that is specified for the SD setting.

In the following sections, while discussing symbolic verification of our $||_{SD}$ setting, we will use Algorithm 9.1 to perform reachability check on the TDES that represents our closed-loop system. In this case, the input to this algorithm will be TDES $\mathbf{G}$ that represents the SD synchronous product of plant and supervisor models, and its output will be the predicate containing the set of reachable states of our closed-loop system. As we are using this algorithm in the $||_{SD}$ setting, it is implicit that the algorithm will perform all computations based on the function $\hat{\delta}$ that we have defined for our

[1]Readers will find some differences between Algorithm 6.3 given in Song (2006) and our Algorithm 9.1. This is because of the logical errors that were present in the original algorithm and we have fixed those errors in this version.

Algorithm 9.2 $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$

1: $P_1 \leftarrow P' \wedge P$
2: **repeat**
3: $P_2 \leftarrow P_1$
4: **for** $i \leftarrow 1$ to n **do**
5: **repeat**
6: $P_3 \leftarrow P_1$
7: $P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma' \cap \Sigma_i} (\hat{\delta}^{-1}(P_1, \sigma) \wedge P) \right)$
8: **until** $P_1 \equiv P_3$
9: **end for**
10: **until** $P_1 \equiv P_2$
11: **return** P_1

$||_{SD}$ setting. Please recall that the function $\hat{\delta}(P, \sigma)$ (Definition 9.3.2) relies on N_σ to compute the predicate, and the definition of N_σ in our $||_{SD}$ setting (Definition 9.2.5) is different from the SD setting (Definition B.1.3).

Therefore, due to different ways of constructing the input TDES $\mathbf{G}$ that is passed in to this algorithm, **line 9** uses different underlying definitions of $\hat{\delta}$ in the SD and $||_{SD}$ settings. For this reason, the seemingly same looking Algorithm 9.1 will potentially generate different results in the two different settings.

Coreachability Check

Algorithm 9.2 computes and returns predicate P_1 containing the set of states of input TDES $\mathbf{G}$ that can reach a state satisfying P' by states satisfying P and transitions with events in Σ'.

Like Algorithm 9.1, we are using Algorithm 9.2 of the SD setting unchanged in our $||_{SD}$ setting. As explained above, the only difference is the TDES $\mathbf{G}$ that we provide as an input to this algorithm. Based on how TDES $\mathbf{G}$ has been constructed, Algorithm 9.2 uses the corresponding definition of function $\hat{\delta}^{-1}$, which in turn relies on N_σ, to compute the required predicate in the SD and the $||_{SD}$ setting, as appropriate.

Please note that we will use this algorithm to compute $CR(\mathbf{G}, P)$ which is equivalent to $\mathcal{CR}(\mathbf{G}, P_m, \Sigma, P)$.

9.4 Construction of Closed-Loop System

In our $||_{SD}$ setting, we construct the closed-loop system by synchronizing TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ using the SD synchronous product operator, i.e. $\mathbf{S} ||_{SD} \mathbf{G}$. Instead of designing monolithic TDES, if $\mathbf{G}$ and $\mathbf{S}$ are modelled in a modular fashion,

then we assume that these component plant and supervisor models are independently combined using product operator to form $\mathbf{G}$ and $\mathbf{S}$ respectively.

In case, if plant and supervisor components are combined using synchronous product, then we can simply add selfloops at every state of the component TDES for events that are missing from their event sets to obtain our $\mathbf{G}$ and $\mathbf{S}$.

For TDES plant components $\mathbf{G}'_i = (Y_i, \Sigma_i, \delta_i, y_{o,i}, Y_{m,i})$ and $\mathbf{G}' = \mathbf{G}'_1 \,||\, \mathbf{G}'_2 \,||\, \ldots \,||\, \mathbf{G}'_k$, where $i = 1, 2, \ldots, k$, let $\mathbf{G}_i = \mathbf{selfloop}(\mathbf{G}'_i, \Sigma - \Sigma_i)$. The TDES plant $\mathbf{G}$ is then defined as follows:

$$\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times \ldots \times \mathbf{G}_k = (Y, \Sigma, \delta, y_o, Y_m)$$

For modular TDES supervisors $\mathbf{S}'_j = (X_j, \Sigma_j, \xi_j, x_{o,j}, X_{m,j})$ and $\mathbf{S}' = \mathbf{S}'_1 \,||\, \mathbf{S}'_2 \,||\, \ldots \,||\, \mathbf{S}'_n$, where $j = 1, 2, \ldots, n$, let $\mathbf{S}_j = \mathbf{selfloop}(\mathbf{S}'_j, \Sigma - \Sigma_j)$. The TDES supervisor $\mathbf{S}$ is then defined as follows:

$$\mathbf{S} = \mathbf{S}_1 \times \mathbf{S}_2 \times \ldots \times \mathbf{S}_n = (X, \Sigma, \xi, x_o, X_m)$$

Using this approach, both $\mathbf{G}$ and $\mathbf{S}$ are now defined over the same event set Σ. Finally, we construct our closed-loop system, $\mathbf{G}_{cl}$, as follows:

$$\mathbf{G}_{cl} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Q, \Sigma, \eta, q_o, Q_m)$$

Here, all five elements of $\mathbf{G}_{cl}$'s tuple are defined as per Definition 4.1.1 of the SD synchronous product operator. Please note that at this stage, $\mathbf{G}_{cl}$ might contain some unreachable states.

Next, we borrow some definitions of the SD setting from Wang (2009). As our strategy of constructing $\mathbf{G}$ and $\mathbf{S}$ from component TDES is same as the SD setting, these definitions work well in our $||_{SD}$ setting just by changing the way of constructing the closed-loop system.

Definition 9.4.1. Let $\mathbf{G}_{cl} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Q, \Sigma, \eta, q_o, Q_m)$, where $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times \ldots \times \mathbf{G}_k = (Y, \Sigma, \delta, y_o, Y_m)$ and $\mathbf{S} = \mathbf{S}_1 \times \mathbf{S}_2 \times \ldots \times \mathbf{S}_n = (X, \Sigma, \xi, x_o, X_m)$. For a given event $\sigma \in \Sigma$, the *σ plant transition predicate* $N_{\mathbf{G},\sigma} : Q \times Q \to \{T, F\}$ can be expressed as follows:

$$N_{\mathbf{G},\sigma}(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \le i \le k\}} \left(\bigvee_{\{y_i, y'_i \in Y_i \mid \delta_i(y_i, \sigma) = y'_i\}} (v_i = y_i) \wedge (v'_i = y'_i) \right)$$

Likewise, the *σ supervisor transition predicate* $N_{\mathbf{S},\sigma} : Q \times Q \to \{T, F\}$ can be expressed as follows:

$$N_{\mathbf{S},\sigma}(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \le j \le n\}} \left(\bigvee_{\{x_j, x'_j \in X_j \mid \xi_j(x_j, \sigma) = x'_j\}} (v_{j+k} = x_j) \wedge (v'_{j+k} = x'_j) \right)$$

It is noteworthy that $N_{\mathbf{G},\sigma}$ and $N_{\mathbf{S},\sigma}$ are defined on $Q \times Q$ and use the variables $\mathbf{v}$ and $\mathbf{v}'$ like N_σ. We will use $N_{\mathbf{G},\sigma}$ to determine if there is a σ transition defined at the plant portion of the indicated states. Similarly, $N_{\mathbf{S},\sigma}$ will be used to determine if there is a σ transition defined at the supervisor portion of the indicated states. They must be defined over $Q \times Q$ so that we can compare and combine their results with other state predicates on Q.

Definition 9.4.2. For TDES plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and some $\sigma \in \Sigma$, let $N_{\mathbf{G},\sigma}$ be the σ plant transition predicate. For $P \in Pred(Q)$, the function $\hat{\delta}_{\mathbf{G}} : Pred(Q) \times \Sigma \to Pred(Q)$ is defined as follows:

$$\hat{\delta}_{\mathbf{G}}(P, \sigma) := (\exists \mathbf{v}(N_{\mathbf{G},\sigma} \wedge P))[\mathbf{v}' \to \mathbf{v}]$$

The inverse function $\hat{\delta}^{-1}_{\mathbf{G}} : Pred(Q) \times \Sigma \to Pred(Q)$ is defined as follows:

$$\hat{\delta}^{-1}_{\mathbf{G}}(P, \sigma) := \exists \mathbf{v}'(N_{\mathbf{G},\sigma} \wedge (P[\mathbf{v} \to \mathbf{v}']))$$

Definition 9.4.3. For TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ and some $\sigma \in \Sigma$, let $N_{\mathbf{S},\sigma}$ be the σ supervisor transition predicate. For $P \in Pred(Q)$, the function $\hat{\xi} : Pred(Q) \times \Sigma \to Pred(Q)$ is defined as follows:

$$\hat{\xi}(P, \sigma) := (\exists \mathbf{v}(N_{\mathbf{S},\sigma} \wedge P))[\mathbf{v}' \to \mathbf{v}]$$

The inverse function $\hat{\xi}^{-1} : Pred(Q) \times \Sigma \to Pred(Q)$ is defined as follows:

$$\hat{\xi}^{-1}(P, \sigma) := \exists \mathbf{v}'(N_{\mathbf{S},\sigma} \wedge (P[\mathbf{v} \to \mathbf{v}']))$$

9.5 Symbolic Verification

In this section, we discuss predicate-based algorithms from Wang (2009) that we have modified to verify various properties of our $||_{SD}$ system. Please note that due to space limitations, we will not provide a detailed explanation for the unchanged parts of these modified algorithms. Please refer to Wang (2009) for the complete logical description of all algorithms. The unmodified algorithms that can be used to check other $||_{SD}$ properties are included in Appendix B for the sake of completeness.

With respect to the unchanged algorithms given in Appendix B, we wish to point out that although the steps of these algorithms remain unaltered, the input TDES that we pass in to these algorithms for verification are certainly different than the ones assumed in the SD setting. Also, the underlying definitions for some variables and functions used by these algorithms have changed with respect to our $||_{SD}$ setting. Therefore, in order to use these algorithms to verify properties in our $||_{SD}$ setting, it is an implicit assumption that these algorithms operate on our input, and use the variable and function definitions that we have specified in this chapter for our $||_{SD}$ setting.

Precisely, algorithms for the following properties remain unchanged in our $||_{SD}$ setting. Please refer to Sections B.2 and B.3 for further details.

1. Nonblocking (Algorithm B.1)
2. Activity-loop-free (ALF) (Algorithm B.2)
3. Proper time behaviour (Algorithm B.3)
4. **S**-singular prohibitable behaviour with $||_{SD}$ (Algorithm B.5: lines 12-16)
5. SD controllability with $||_{SD}$
   i. Point ii (Algorithms B.4, B.5, B.6, B.7, B.8)

ii. Point iii (Algorithm B.9)

Now we will discuss our modified algorithms for the $||_{SD}$ setting. With TDES plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, our closed-loop system is $\mathbf{G}_{cl} = \mathbf{S} ||_{SD} \mathbf{G} = (Q, \Sigma, \eta, q_o, Q_m)$. As per the definition of state set Q in the $||_{SD}$ operator, for every state $q \in Q$, there must exist a state $x \in X$ and $y \in Y$ such that $q = (x, y)$.

The system event set Σ is defined as $\Sigma = \Sigma_{hib} \dot\cup \Sigma_u \dot\cup \{\tau\}$, where Σ_{hib} and Σ_u represent the set of prohibitable events and uncontrollable events of $\mathbf{G}_{cl}$ respectively. The set of controllable events is $\Sigma_c = \Sigma_{hib} \dot\cup \{\tau\}$, and the set of activity events is $\Sigma_{act} = \Sigma_{hib} \dot\cup \Sigma_u$.

9.5.1 Plant Completeness with $||_{SD}$

According to Definition 4.4.1 of plant completeness with $||_{SD}$ property, the states of $\mathbf{G}_{cl}$, where a prohibitable event is enabled at the corresponding state in $\mathbf{S}$ but it is not possible at the corresponding state in $\mathbf{G}$, are the *incomplete* states that cause this property to fail. We can express the set of these states, $Q_{incomplete}$, and its corresponding predicate $P_{incomplete} := pr(Q_{incomplete})$ as follows:

$$Q_{incomplete} := \{q = (x, y) \in Q \mid (\exists \sigma \in \Sigma_{hib})\, \xi(x, \sigma)! \;\&\; \neg\, \delta(y, \sigma)!\}$$

$$P_{incomplete} := \bigvee_{\sigma \in \Sigma_{hib}} \left(\hat{\xi}^{-1}(true, \sigma) \wedge \neg\, \hat{\delta}^{-1}_{\mathbf{G}}(true, \sigma) \right)$$

Here, $\hat{\xi}^{-1}$ (Definition 9.4.3) and $\hat{\delta}^{-1}_{\mathbf{G}}$ (Definition 9.4.2) are the inverse functions for supervisor $\mathbf{S}$ and plant $\mathbf{G}$ respectively.

Our $\mathbf{G}$ is considered to be complete with $||_{SD}$ for $\mathbf{S}$ if none of the states of $Q_{incomplete}$ are reachable, i.e. $Q_{incomplete} \cap Q_{reach} = \emptyset$, where Q_{reach} is the set of reachable states of $\mathbf{G}_{cl}$. This implies that $P_{incomplete} \wedge P_{reach} \equiv false$, where $P_{reach} := pr(Q_{reach})$ is the predicate representing the set of states in Q_{reach}. Otherwise, $P_{incomplete} \wedge P_{reach}$ contains the set of states that cause this property to fail.

This is the logic used by Algorithm 9.3[2] to verify plant completeness with $||_{SD}$ property. It is notable that our plant completeness with $||_{SD}$ definition is similar to the plant completeness property (Definition 2.3.13) except for the actual supervisor TDES and the way of constructing the closed-loop system. Therefore, we are passing our TDES supervisor $\mathbf{S}$ of the $||_{SD}$ setting as input to Algorithm 9.3. Also, at **line 5** of Algorithm 9.3, we are performing reachability check on our closed-loop system "$\mathbf{S} ||_{SD} \mathbf{G}$" instead of the closed-loop system of the SD setting "$\mathbf{G} \times \boldsymbol{\mathcal{S}}$" that was used in the original algorithm. The rest of the algorithm steps are essentially unaltered.

[2]The value returned by this algorithm is a boolean, $True$ or $False$, instead of a state predicate.

Algorithm 9.3 CheckPlantCompleteness(**G**, **S**)

1: $P_{incomplete} \leftarrow false$
2: **for all** $(\sigma \in \Sigma_{hib})$ **do**
3: $\quad P_{incomplete} \leftarrow P_{incomplete} \vee (\hat{\xi}^{-1}(true, \sigma) \wedge \neg\, \hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma))$
4: **end for**
5: $P_{incomplete} \leftarrow P_{incomplete} \wedge R(\mathbf{S}\,||_{SD}\,\mathbf{G}, true)$
6: **if** $(P_{incomplete} \not\equiv false)$ **then**
7: $\quad$ **return** $False$
8: **end if**
9: **return** $True$

9.5.2 Untimed Controllability with $||_{SD}$

The standard untimed controllability property (Definition 2.2.15) gets redefined as part of the timed controllability with $||_{SD}$ definition (Definition 4.4.3). Therefore, its corresponding algorithm needs to be amended for use in the $||_{SD}$ setting.

According to Definition 4.4.4 for untimed controllability with $||_{SD}$, if an uncontrollable event is possible at a state in **G** but it is not possible at the corresponding composite state in $\mathbf{G}_{cl}$, then this composite state of $\mathbf{G}_{cl}$ is considered *bad* as it will make our **S** uncontrollable with $||_{SD}$ with respect to our **G**. The set of these bad states, Q_{bad}, and its corresponding predicate $P_{bad} := pr(Q_{bad})$ can be expressed as follows:

$$Q_{bad} := \{q = (x, y) \in Q \mid (\exists \sigma_u \in \Sigma_u)\, \delta(y, \sigma_u)! \;\&\; \neg\, \eta(q, \sigma_u)!\}$$

$$P_{bad} := \bigvee_{\sigma_u \in \Sigma_u} \left(\hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma_u) \wedge \neg\, \hat{\delta}^{-1}(true, \sigma_u) \right)$$

Here, $\hat{\delta}_{\mathbf{G}}^{-1}$ (Definition 9.4.2) and $\hat{\delta}^{-1}$ (Definition 9.3.3) are the inverse functions for **G** and $\mathbf{G}_{cl}$ respectively.

In order for **S** to be untimed controllable with $||_{SD}$ for **G**, none of the Q_{bad} states should be reachable, i.e $Q_{bad} \cap Q_{reach} = \emptyset$, where Q_{reach} is the set of reachable states of $\mathbf{G}_{cl}$. This implies that $P_{bad} \wedge P_{reach} \equiv false$, where $P_{reach} := pr(Q_{reach})$ is the predicate representing the set of states in Q_{reach}. Otherwise, $P_{bad} \wedge P_{reach}$ holds the set of states where $\mathbf{G}_{cl}$ is not allowing an uncontrollable event that is possible at the corresponding state in **G**.

Algorithm 9.4 essentially makes use of the above-mentioned logic to verify the untimed controllability with $||_{SD}$ property in our $||_{SD}$ setting. In addition to passing our $||_{SD}$ supervisor **S** as an input, our algorithm differs from the original algorithm of the SD setting at **lines 5** and **3**, where we use our closed-loop system, $\mathbf{S}\,||_{SD}\,\mathbf{G}$, and its inverse function, $\hat{\delta}^{-1}$, respectively.

Algorithm 9.4 CheckUntimedControllability($\mathbf{G}$, $\mathbf{S}$)

1: $P_{bad} \leftarrow false$
2: **for all** $(\sigma_u \in \Sigma_u)$ **do**
3: $\quad P_{bad} \leftarrow P_{bad} \vee (\hat{\delta}^{-1}_{\mathbf{G}}(true, \sigma_u) \wedge \neg\, \hat{\delta}^{-1}(true, \sigma_u))$
4: **end for**
5: $P_{bad} \leftarrow P_{bad} \wedge R(\mathbf{S}\,||_{SD}\,\mathbf{G}, true)$
6: **if** $(P_{bad} \not\equiv false)$ **then**
7: $\quad$ **return** $False$
8: **end if**
9: **return** $True$

9.5.3 SD Controllability with $||_{SD}$

All algorithms that contribute in verifying the property of SD controllability with $||_{SD}$ assume that $\mathbf{G}$ has proper time behaviour (Algorithm B.3) and $\mathbf{G}_{cl}$ is ALF (Algorithm B.2). These algorithms make use of several variables and functions to verify the SD controllability with $||_{SD}$ property. We would like to clarify two points about these variables and functions.

1. $\mathbf{G}_{cl}$ stated in these variables and functions refer to our closed-loop system, i.e. $\mathbf{G}_{cl} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, which is different from the original $\mathbf{G}_{cl}$ of the SD setting that was assumed to be constructed as $\mathbf{G} \times \boldsymbol{\mathcal{S}}$.
2. In our $||_{SD}$ setting, the underlying definitions for some of these variables and functions are different from the original ones that were defined in the SD setting. We will explicitly highlight them in our discussion. Since we are using these algorithms in our $||_{SD}$ setting, it is obvious that all variables and functions will be evaluated using the definitions given in this chapter for our $||_{SD}$ setting.

The following variables and functions are used in the upcoming algorithms:

- P_{reach}: The predicate of the set of reachable states of $\mathbf{G}_{cl}$.
- P_{SF}: The predicate of the set that contains sampled states of $\mathbf{G}_{cl}$ found by the algorithm.
- Z_{SP}: This set contains the predicates of sampled states in $\mathbf{G}_{cl}$ found and not yet analyzed by the algorithm.
- $N_{\mathbf{G},\sigma}, N_{\mathbf{S},\sigma}$: Transition predicates for σ for $\mathbf{G}$ and $\mathbf{S}$ respectively, as in Definition 9.4.1.
- N_σ: Transition predicate for σ for $\mathbf{G}_{cl}$, as in Definition 9.2.5. Please note that our definition for N_σ is different from N_σ of the SD setting (Definition B.1.3).
- $\hat{\delta}$: Transition function for state predicates for $\mathbf{G}_{cl}$, as in Definition 9.3.2. Please recall that $\hat{\delta}$ relies on N_σ which makes the computation logic of $\hat{\delta}$ different in the SD and the $||_{SD}$ setting.

Algorithm 9.5 CheckSDControllability($\mathbf{G}, \mathbf{S}$)

1: $\mathbf{G}_{cl} \leftarrow \mathbf{S} ||_{SD} \mathbf{G}$
2: $P_{reach} \leftarrow R(\mathbf{S} ||_{SD} \mathbf{G}, true)$
3: **if** (CheckSDCPointi($\mathbf{G}, \mathbf{S}, P_{reach}$) = $False$) **then**
4: **return** $False$
5: **end if**
6: $SDControllable \leftarrow True$
7: $P_{SF} \leftarrow pr\{q_o\}$
8: $Z_{SP} \leftarrow \{pr\{q_o\}\}$
9: $pNerFail \leftarrow \emptyset$
10: **while** ($Z_{SP} \neq \emptyset$) **do**
11: $P_{ss} \leftarrow$ Pop(Z_{SP})
12: $SDControllable \leftarrow$ AnalyzeSampledState($\mathbf{G}, \mathbf{S}, P_{SF}, Z_{SP}, P_{reach}, P_{ss}, pNerFail$)
13: **if** ($\neg\, SDControllable$) **then**
14: **return** $False$
15: **end if**
16: **end while**
17: **if** ($pNerFail \neq \emptyset$) **then**
18: $SDControllable \leftarrow$ RecheckNerodeCells($pNerFail$)
19: **if** ($\neg\, SDControllable$) **then**
20: **return** $False$
21: **end if**
22: **end if**
23: **if** ($\neg$ CheckSDCPointiii(P_{reach})) **then**
24: **return** $False$
25: **end if**
26: **return** $True$

- $\hat{\delta}_{\mathbf{G}}$: Transition function for state predicates for $\mathbf{G}$ only, as in Definition 9.4.2.
- $\hat{\xi}$: Transition function for state predicates for $\mathbf{S}$ only, as in Definition 9.4.3.
- $pNerFail$: This set $pNerFail \subseteq$ Pwr($Pred(Q)$) is a set of sets of predicates that stores information where Point ii.2 of SD controllability with $||_{SD}$ property may have failed.
- $SDControllable$: This flag asserts if $\mathbf{S}$ is SD controllable with $||_{SD}$ with respect to $\mathbf{G}$.

Algorithm 9.5 serves as the entry point for checking various points of the SD controllability with $||_{SD}$ property. At **line 3**, it calls Algorithm 9.6 to verify Point i of SD controllability with $||_{SD}$ definition. It is note worthy that Point i essentially represents the timed controllability with $||_{SD}$ property which includes the untimed controllability with $||_{SD}$ check. Since we have already discussed Algorithm 9.4 for

verifying untimed controllability with $||_{SD}$, we will not discuss it again. For the same reason, untimed controllability with $||_{SD}$ check is not showing up in Algorithm 9.6.

In order to verify Point ii of SD controllability with $||_{SD}$, processing starts with the initial state of $\mathbf{G}_{cl}$ which is always a sampled state (**line 7**). As verification proceeds, a reachability tree is created for a given sampling period, and required checks including the check of **S**-singular prohibitable behaviour with $||_{SD}$ are performed (**line 12**). If any of the desired properties fails, the algorithm terminates, except for Point ii.2 where the algorithm continues after recording the problematic nerode cells. These cells and Point ii.2 is then tested again afterwards (**line 18**).

Finally, the algorithm verifies Point iii of SD controllability with $||_{SD}$ at **line 23** by making use of Algorithm B.9.

Point i

Algorithm 9.6 verifies the timed controllability part of Point i of SD controllability with $||_{SD}$. We have derived this algorithm from Algorithm 9.7 that was developed by Wang (2009) to verify Point ii of SD controllability (Definition 3.5.1) in the SD setting.

At **lines 2-5**, Algorithm 9.7 checks the forward implication ($\Rightarrow$) of Point ii of SD controllablility. It determines if there exists a reachable state in $\mathbf{G}_{cl} = \mathbf{G} \times \boldsymbol{\mathcal{S}}$ where both *tick* and prohibitable events are enabled. If such a such exists, Point ii ($\Rightarrow$) fails, and the algorithm returns $False$.

Please recall that Point ii ($\Rightarrow$) of SD controllability does not exist in our definition of SD controllability with $||_{SD}$, and we are not required to check this condition explicitly in our $||_{SD}$ setting. We are able to get rid of this explicit check because of the distinct synchronization mechanism of our $||_{SD}$ operator that guarantees to automatically disable a *tick* event in the closed-loop system $\mathbf{G}_{cl} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, if both *tick* and a prohibitable event is possible in $\mathbf{G}$ and enabled by $\mathbf{S}$. This means that our $||_{SD}$ operator will never enable both *tick* and prohibitable event at any state of $\mathbf{G}_{cl}$ while synchronizing $\mathbf{G}$ and $\mathbf{S}$ to form the closed-loop system. Since this condition is automatically satisfied in our $||_{SD}$ setting, we do not need **lines 2-5** of Algorithm 9.7 and did not include them in our Algorithm 9.6.

At **lines 6-9**, Algorithm 9.7 checks the reverse implication ($\Leftarrow$) of Point ii of SD controllability. It determines if there exists a reachable state in $\mathbf{G}_{cl} = \mathbf{G} \times \boldsymbol{\mathcal{S}}$ where no prohibitable event is eligible, and *tick* is possible in $\mathbf{G}$ but disabled by $\boldsymbol{\mathcal{S}}$. If such a such exists, Point ii ($\Leftarrow$) fails, and the algorithm returns $False$. This is essentially the timed part of the timed controllability definition (Definition 2.3.2) used in the SD setting.

In our Algorithm 9.6, we have modified this logic to check our corresponding condition by using our closed-loop system $\mathbf{G}_{cl} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ instead of their supervisor $\boldsymbol{\mathcal{S}}$. This change is in line with the timed part of our timed controllability with $||_{SD}$ property (Definition 4.4.3) that we have defined for our $||_{SD}$ setting.

Algorithm 9.6 CheckSDCPointi($\mathbf{G}, \mathbf{S}, P_{reach}$)

1: $P_{q-hib} \leftarrow \bigvee_{\sigma \in \Sigma_{hib}} \exists \mathbf{v}' N_\sigma$
2: $P_{bad} \leftarrow \exists \mathbf{v}' N_{\mathbf{G},tick} \wedge \neg (\exists \mathbf{v}' N_{tick}) \wedge \neg P_{q-hib}$
3: **if** $(P_{bad} \wedge P_{reach} \not\equiv false)$ **then**
4: **return** $False$
5: **end if**
6: **return** $True$

Algorithm 9.7 CheckSDContii($\mathbf{G}, \boldsymbol{\mathcal{S}}, P_{reach}$)

1: $P_{q-hib} \leftarrow \bigvee_{\sigma \in \Sigma_{hib}} \exists \mathbf{v}' N_\sigma$
2: $P_{bad} \leftarrow \exists \mathbf{v}' N_{tick} \wedge P_{q-hib}$
3: **if** $P_{bad} \wedge P_{reach} \not\equiv false$ **then**
4: **return** $False$
5: **end if**
6: $P_{bad} \leftarrow \exists \mathbf{v}' N_{\mathbf{G},tick} \wedge \neg (\exists \mathbf{v}' N_{\boldsymbol{\mathcal{S}},tick}) \wedge \neg P_{q-hib}$
7: **if** $P_{bad} \wedge P_{reach} \not\equiv false$ **then**
8: **return** $False$
9: **end if**
10: **return** $True$

Line 1 of Algorithm 9.6 identifies the states of $\mathbf{G}_{cl}$ that have one or more prohibitable events defined. **Line 2** determines if there exists a *bad* state in $\mathbf{G}_{cl}$ where neither *tick* nor a prohibitable event is eligible in $\mathbf{G}_{cl}$, but *tick* is possible at the corresponding state in $\mathbf{G}$. If such a bad state exists and is reachable in $\mathbf{G}_{cl}$ (**line 3**), then the timed check of Point i of our SD controllability with $||_{SD}$ fails, and the algorithm returns $False$ at **line 4**. Otherwise, it returns $True$ at **line 6**.

Point ii

In order to verify Point ii of SD controllability with $||_{SD}$, we will reuse several variables from Wang (2009). Please note that we have redefined two variables, Σ_{poss} and B_{conc}, to make the corresponding algorithms compatible with our $||_{SD}$ setting.

- Σ_{Elig}: The set of prohibitable events eligible in both $\mathbf{G}$ and $\mathbf{S}$ at q_{ss}, where q_{ss} is the sampled state in $\mathbf{G}_{cl}$ that we are processing.
- P_q: The predicate of current state in $\mathbf{G}_{cl}$.
- Σ_{poss}: Wang (2009) defines this variable to be the set of events eligible in both $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ at predicate P_q of current state in $\mathbf{G}_{cl} = \mathbf{G} \times \boldsymbol{\mathcal{S}}$. This is because every event that is possible in $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ will be enabled in $\mathbf{G}_{cl}$ by the product operator. However, this might not be true for our $||_{SD}$ operator with respect to the *tick* event.

  In our $||_{SD}$ setting, we define Σ_{poss} to be the set of events that are eligible in

$\mathbf{G}_{cl} = \mathbf{S} ||_{SD} \mathbf{G}$ at predicate P_q of current state in $\mathbf{G}_{cl}$. Therefore, this set contains activity events that are eligible in both $\mathbf{G}$ and $\mathbf{S}$ at predicate P_q of current state in $\mathbf{G}_{cl}$. Additionally, it also contains a *tick* event if it is eligible in $\mathbf{G}$ and $\mathbf{S}$, and no prohibitable event is possible at predicate P_q of current state in $\mathbf{G}_{cl}$ to preempt the *tick*.

- $\Sigma_{\mathbf{G}poss}$: The set of prohibitable events eligible in $\mathbf{G}$ at predicate P_q of current state in $\mathbf{G}_{cl}$.
- $nextLabel$: This number represents the next unused node in B_{map}. It is used to name newly discovered nodes of the reachability tree.
- B_{map}: This partial function $B_{map} : \mathcal{N} \rightarrow Pred(Q)$ maps the nodes of the reachability tree to the predicates of the states of $\mathbf{G}_{cl}$ that the nodes represent. This function will sometimes be treated like the set $B_{map} \subseteq \mathcal{N} \times Pred(Q)$. Note that $\mathcal{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers.
- B_p: This is the set of nodes pending to be expanded in the reachability tree.
- B_{conc}: The set $B_{conc} \subseteq \mathcal{N} \times Pred(Q)$ contains the nodes that represent concurrent strings and the sampled states the strings lead to.

  In Wang (2009), for $(b, q) \in B_{conc}$, node b is a node at which *tick* is eligible in $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$, and q is the sampled state of $\mathbf{G}_{cl} = \mathbf{G} \times \boldsymbol{\mathcal{S}}$ that the *tick* leads to.

  In our $||_{SD}$ setting, for $(b, q) \in B_{conc}$, we define b to be a node at which *tick* is eligible in $\mathbf{G}_{cl} = \mathbf{S} ||_{SD} \mathbf{G}$, and q is the sampled state of $\mathbf{G}_{cl}$ that the *tick* leads to.
- Occu_B: The partial function $\text{Occu}_B : \mathcal{N} \rightarrow \text{Pwr}(\Sigma)$ maps the nodes of the reachability tree to the occurrence image of the string that they represent. This function will sometimes be treated like the set $\text{Occu}_B \subseteq \mathcal{N} \times \text{Pwr}(\Sigma)$.

The actual algorithm steps to verify Points ii.1, ii.2 and iii of SD controllability with $||_{SD}$ are essentially the same as Wang (2009). Please refer to Section B.3 to get an overview of these unmodified algorithms. This includes any function calls in Algorithm 9.5 that have not been discussed in this chapter. Please note that in order to verify Points ii and iii of SD controllability with $||_{SD}$ property, all these algorithms make use of the variable and function definitions specified in this chapter for our $||_{SD}$ setting.

Chapter 10

Flexible Manufacturing System

In this chapter, we present an example of a Flexible Manufacturing System (FMS) to demonstrate the application, utilization and benefits of our SD synchronous product operator and the $||_{SD}$ setting. This is the same TDES example that has been discussed in Wang (2009); Wang and Leduc (2012) to illustrate the SD supervisory control methodology, who in turn based it on the untimed FMS example given in Hill (2008). We have intentionally selected the same system so that we could clearly compare and discuss the complexity of designing modular TDES supervisors by hand and the size of resultant supervisor models in the SD and $||_{SD}$ setting, i.e. in the absence and presence of our $||_{SD}$ operator.

We begin this chapter by describing the structure and workflow of the FMS. Then, we provide its various TDES plant components. After that, we analyze the original design of each modular TDES supervisor developed in the SD setting and discuss how it gets simplified in our $||_{SD}$ setting in the presence of the $||_{SD}$ operator. Finally, we close this chapter by presenting a comprehensive discussion on our software implementation and verification results for the FMS example.

10.1 System Structure

The Flexible Manufacturing System (FMS), shown in Figure 1.1, consists of six machines and five buffers, where each buffer has the capacity to hold a single part. These buffers are treated as specifications and it is desired that buffers never overflow or underflow.

The basic idea of the FMS is that a part enters the system via conveyor Con2 and passes to a handling Robot via buffer B2. The Robot then passes the part to Lathe via buffer B4. The Lathe can generate two types of parts, A and B. Once the Robot receives the part back from Lathe via B4, it sends the part either to buffer B6 or B7 depending upon the part type. Precisely, type A part goes to B6 while type B part goes to B7. From B7, part B goes to a painting machine PM via conveyor Con3 and

Table 10.1: Meaning and Shorthand for Event Labels of FMS

Label	Meaning	Shorthand	Label	Meaning	Shorthand
921	Part enters system	pt_ent_sys	922	Part enters B2	pt_ent_B2
933	Robot takes from B2	R_from_B2	934	Robot to B4	R_to_B4
937	B4 to Robot for B6	B4_to_R_for_B6	938	Robot to B6	R_to_B6
939	B4 to Robot for B7	B4_to_R_for_B7	930	Robot to B7	R_to_B7
951	B4 to Lathe (A)	B4_to_L_A	952	Lathe to B4 (A)	L_to_B4_A
953	B4 to Lathe (B)	B4_to_L_B	954	Lathe to B4 (B)	L_to_B4_B
961	Initialize AM	init_AM	963	B6 to AM	B6_to_AM
964	Finished from B6	fin_from_B6	965	B7 to AM	B7_to_AM
966	Finished from B7	fin_from_B7	971	B7 to Con3	B7_to_C3
no921	No part enters system	no_pt_ent_sys	972	Con3 to B8	C3_to_B8
no963a	No B6 to AM (a)	no_B6_to_AM_a	973	B8 to Con3	B8_to_C3
no963b	No B6 to AM (b)	no_B6_to_AM_b	974	Con3 to B7	C3_to_B7
no965a	No B7 to AM (a)	no_B7_to_AM_a	981	B8 to PM	B8_to_PM
no965b	No B7 to AM (b)	no_B7_to_AM_b	982	PM to B8	PM_to_B8

buffer B8. After completing its operation, PM returns the part to B7 via the same route. From B6 and B7, the part goes to the finishing machine AM, from where the finished part finally exits the system.

Table 10.1 shows the mapping of numeric event labels, used in Wang and Leduc (2012), to their meaning. Odd numbered event labels represent prohibitable events, whereas even numbered labels represent unconrollable events. Instead of the numeric labels, we will use the meaningful shorthand event labels in our TDES models for the sake of readability and comprehension. Our shorthand corresponding to each event label in given in Table 10.1.

It is notable that there are five event labels in Table 10.1 that are prefixed by "no". These labels do not represent any physical events of the FMS. Rather, they are *prohibitable expansion events* that were introduced by Wang and Leduc (2012) to aid in communication between various modular TDES supervisors in order to satisfy the properties of the SD supervisory control methodology. We will discuss these events further in our subsequent sections.

Please recall from Section 2.3 that in the graphical TDES models, an event name given in italics and preceded by "!" indicates an uncontrollable event, a double circle represents the initial state, and a filled circle shows that the state is marked.

10.2 Plant Components

The FMS consists of six plant components: two conveyors **Con2** and **Con3**, **Robot**, **Lathe**, **PM** and **AM**. Their TDES models are shown in Figures 10.1-10.6. One more TDES plant model, **SysDownNup**, is given in Figure 10.7. This plant component

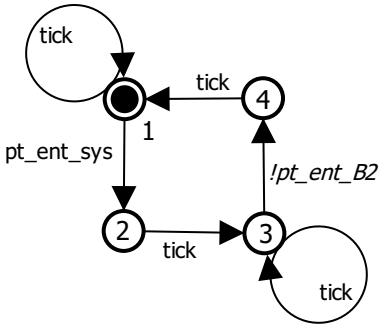


Figure 10.1: Conveyor **Con2**

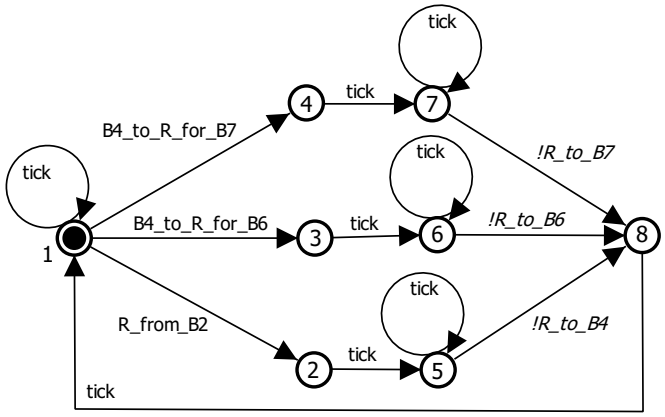


Figure 10.2: **Robot**

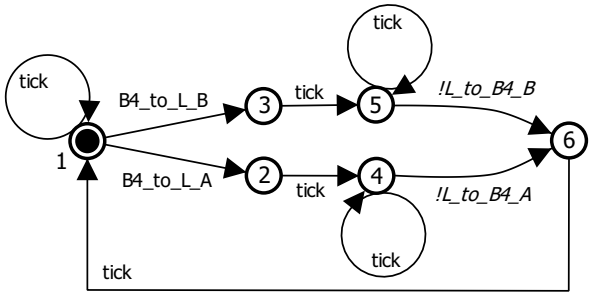


Figure 10.3: **Lathe**

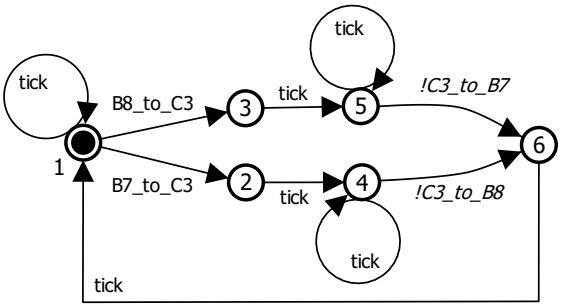


Figure 10.4: Conveyor **Con3**

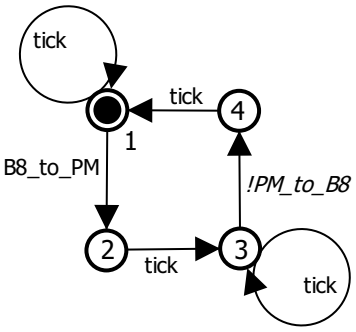


Figure 10.5: Painting Machine **PM**

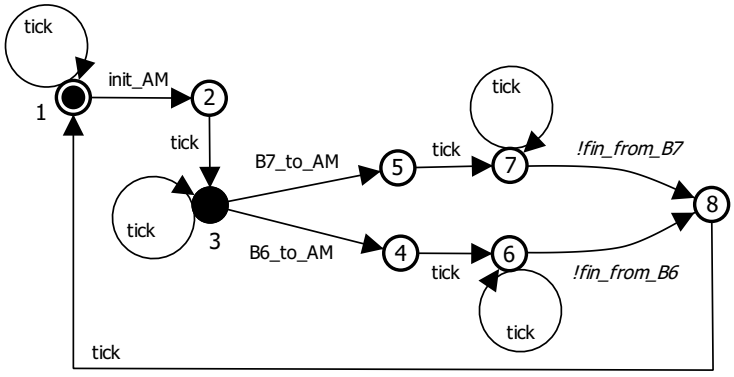


Figure 10.6: Finishing Machine **AM**

is added to introduce a shutdown mechanism in the FMS. This could correspond to a physical switch to turn off/restart the system. When the *shutdown* event occurs, Con2 stops accepting new parts and all existing parts exit the system after being processed. In the shutdown state, all components of the physical system go idle, i.e. return to their marked states in the corresponding TDES models. The *restart* event brings the system back up again.

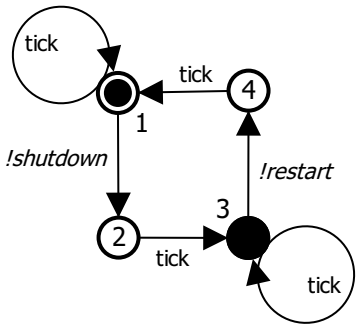


Figure 10.7:
SysDownNup

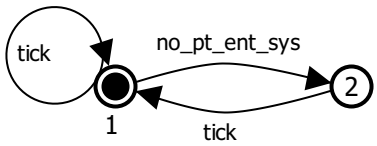


Figure 10.8:
AddNoPtEntSys

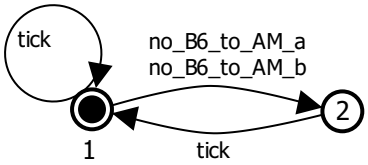


Figure 10.9:
AddNoB6toAM

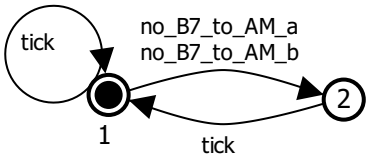


Figure 10.10:
AddNoB7toAM

As all these plant models represent the actual uncontrolled behaviour of the physical system, they are the same in the SD and $||_{SD}$ setting. This will make it easy for us to compare the design of modular TDES supervisors in the two settings, since in both cases, supervisors needed to be designed for the same TDES plant components and the same specifications.

In order to introduce five prohibitable expansion events to the system, three additional plant components were added by Wang and Leduc (2012) as part of the supervisor design. These plant TDES are shown in Figures 10.8-10.10. We will examine them further in the next section while discussing the design of modular supervisors.

10.3 Modular Supervisors

Now we will discuss the design of modular TDES supervisors for FMS in the $||_{SD}$ setting. Our approach is to first present the modular TDES supervisors that were originally designed in the SD setting, and then discuss how they get simplified in the presence of our $||_{SD}$ operator by comparing them with the TDES supervisor models that we have designed for our $||_{SD}$ setting. Essentially, there are two key takeaways from our discussion presented in this section:

1. It is noticeable how easy it becomes for the designers to design modular TDES supervisors in the presence of our $||_{SD}$ operator and satisfy the same system specifications. This is because they no longer need to manually keep track of the enablement/disablement of *tick* and prohibitable events, nor incorporate this logic

explicitly in various supervisor models.

2. It is striking how the size and logical complexity of many of the modular supervisors get reduced in the presence of our $||_{SD}$ operator. The reason is that different supervisor models do not need to communicate with each other and keep track of each other's behaviour about enablement/disablement of *tick* event and forcing of prohibitable events. Also, as we will see in Section 10.4, these simplifications make it easy and efficient to verify different properties of the closed-loop system.

Please note that in order to clearly differentiate between TDES models of the SD supervisory control and our $||_{SD}$ setting, names of TDES plants and supervisors that are used in the SD setting but are removed or modified in the $||_{SD}$ setting, will be stated in ***bold italics***. We will refer to TDES supervisors that appear in the $||_{SD}$ setting using **bold text** only.

10.3.1 Buffer Supervisors

Buffer supervisors control the flow of parts in and out of the buffers. They are primarily responsible for making sure that buffers do not overflow or underflow. Please note that while discussing buffer supervisor **B2**, we will also examine supervisor **HndlSysDwn**, as these two supervisors are closely related with respect to the shutdown/restart mechanism of the FMS.

B2 and HndlSysDwn

Supervisor ***B2***, shown in Figure 10.11, is designed in the SD setting to make sure that buffer B2 does not overflow or underflow. It guarantees this by watching the part's progress once a new part enters the system (*pt_ent_sys*). ***B2*** first waits for the part to enter buffer B2 by keeping track of event *pt_ent_B2*, and then it allows the Robot to take the part from B2 by enabling the prohibitable event *R_from_B2*. This prevents the underflow of buffer B2. ***B2*** also ensures that another part does not enter the system (*pt_ent_sys*) until the previous part has been removed from buffer B2 (*R_from_B2*), thus preventing overflow.

Another crucial task performed by ***B2*** is to decide when to force the prohibitable event *pt_ent_sys*. As soon as the system is turned on, ***B2*** causes Con2 to accept a new part into the system by enabling and forcing *pt_ent_sys*. Please recall that in order to force a prohibitable event in the SD supervisory control theory, *tick* event must be explicitly disabled by the supervisor to satisfy Point ii ($\Rightarrow$) of SD controllability. For this reason, *tick* has been disabled at state 0 of ***B2*** to force *pt_ent_sys*.

By looking at supervisor ***B2***, we observe that as soon as Robot takes the part (*R_from_B2*) and buffer B2 becomes empty, ***B2*** allows a new part to enter the system by forcing *pt_ent_sys*. This behaviour is acceptable as long as the system is up and running. However, once the system is shutdown, then Con2 must stop accepting

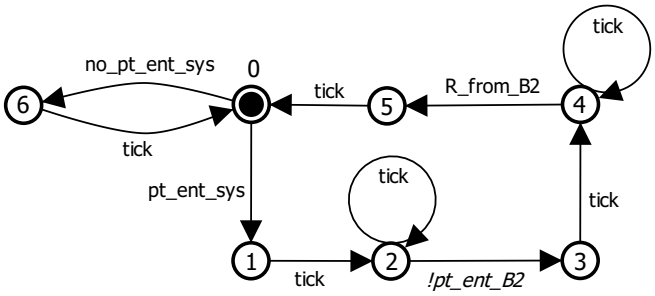


Figure 10.11: Supervisor ***B2***

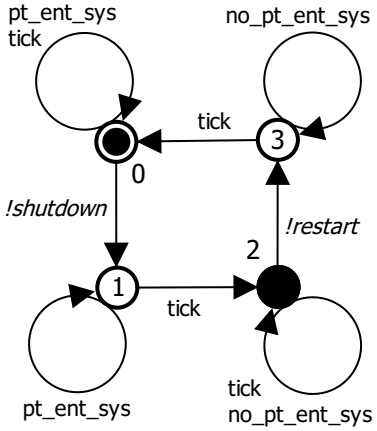


Figure 10.12: Supervisor ***HndlSysDwn***

new parts and system must empty out after processing the existing parts so that all machines can go to their idle (marked) states, as desired by the system specifications.

This means that after shutdown, *pt_ent_sys* needs to be disabled and must not be forced anymore, until the system is restarted. Since *tick* has already been disabled at state 0 to force *pt_ent_sys*, designers must figure out some other way to stop forcing *pt_ent_sys* without duplicating information from other parts of the system. Moreover, if *pt_ent_sys* is not forced at state 0, some other prohibitable event needs to be forced in the absence of an eligible *tick* event. Otherwise, the system becomes uncontrollable.

In order to resolve this issue in the SD setting, a prohibitable expansion event *no_pt_ent_sys* is introduced to the system by designing and including an additional plant TDES, ***AddNoPtEntSys*** (Figure 10.8). At state 0 of supervisor ***B2***, a loop of concurrent string "*no_pt_ent_sys*−*tick*" is added that allows ***B2*** to force *no_pt_ent_sys* when *pt_ent_sys* needs to be disabled to achieve the desired behaviour, while keeping the system controllable and not "stopping the clock".

Another supervisor ***HndlSysDwn***, shown in Figure 10.12, is designed in the SD setting to make sure that the two events, *pt_ent_sys* and *no_pt_ent_sys*, are enabled and disabled at the right time. Specifically, when the system is initially turned on or restarted, ***HndlSysDwn*** enables *pt_ent_sys* and disables *no_pt_ent_sys* to allow new parts to enter the system for processing. When the system is shutdown, it enables *no_pt_ent_sys* and disables *pt_ent_sys* to stop Con2 from accepting new parts into the system.

This discussion clearly shows that forcing a prohibitable event by explicitly disabling *tick*, along with making sure that system does not become uncontrollable is not a straightforward and trouble-free task. In this simple example, when prohibitable event *pt_ent_sys* is under the control of only two modular supervisors, designers have to add one extra plant TDES, a prohibitable expansion event and several additional transitions in the supervisor models to specify the correct forcing mechanism.

All this extra design effort is required because the logic for forcing a prohibitable

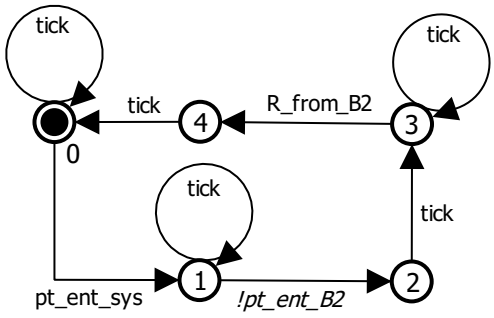


Figure 10.13: Supervisor **B2**

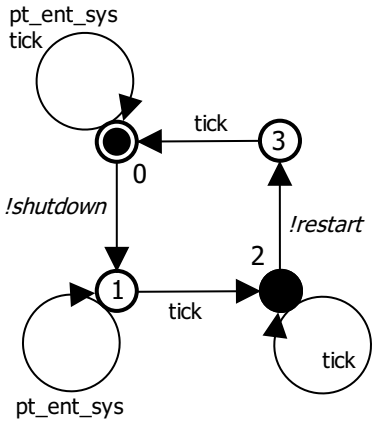


Figure 10.14: Supervisor **HndlSysDwn**

event needed to be manually specified in the supervisor model, and *tick* event was explicitly disabled at one state of the supervisor in order to force the prohibitable event. Certainly, this situation becomes more complicated when the prohibitable event to be forced is under the control of several modular supervisors. Not to mention, the behaviour of the plant model also needs to be considered to make sure that prohibitable event is possible in the plant when supervisor models are collectively trying to force it. We will see a glimpse of this intricate situation later in Section 10.3.4.

Now we will discuss our buffer supervisor **B2**, shown in Figure 10.13, that we have designed in our $||_{SD}$ setting. Precisely, we have derived **B2** from ***B2*** by trimming away its extra design logic that is not required in our $||_{SD}$ setting.

In the $||_{SD}$ setting, we do not need to manually decide when to force a prohibitable event, nor incorporate this logic explicitly in any of the supervisor models. Rather, we can simply enable a prohibitable event to indicate that we want this event to occur, without disabling the *tick* and the $||_{SD}$ operator will force the event automatically (by deleting the *tick*) as soon as event is enabled by all supervisors, and the event is possible in the plant. That is why, we have enabled both *tick* and prohibitable event *pt_ent_sys* at state 0 of our supervisor **B2**. Also, the synchronization logic of our $||_{SD}$ operator guarantees that the property checked by Point ii ($\Rightarrow$) of SD controllability will always be satisfied at every state of the closed-loop system.

Since **B2** is not disabling *tick* at state 0, we do not need to worry about the logic of figuring out how to keep our system controllable with $||_{SD}$ if *pt_ent_sys* cannot or should not be forced. In other words, we are not required to have any alternative expansion event to force in place of *pt_ent_sys* in order to make sure that we do not "stop the clock". This implies that the above-mentioned issue, that designers had to face and resolve in order to explicitly force a prohibitable event while designing supervisors in the SD setting, does not exist in our $||_{SD}$ setting. The development and use of the $||_{SD}$ synchronization operator has completely and permanently resolved this issue in our $||_{SD}$ setting.

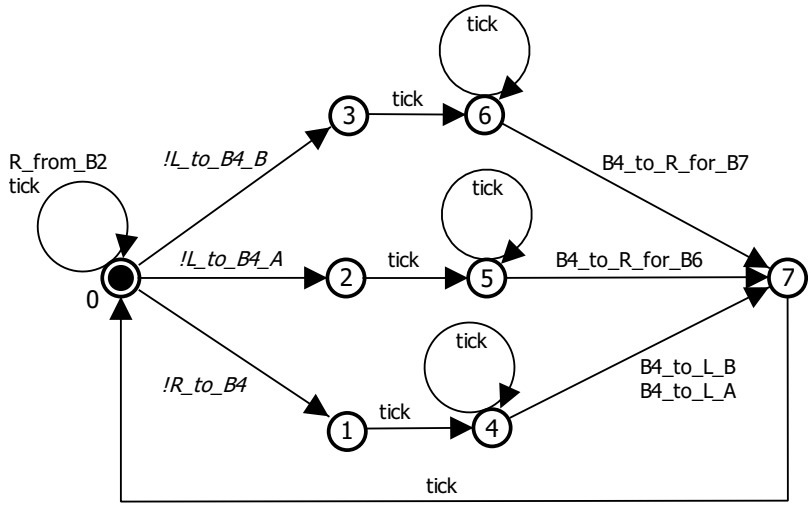


Figure 10.15: Supervisor **B4**

As a result, we have altogether removed the expansion event *no_pt_ent_sys* from our FMS plant and supervisor models designed in the $||_{SD}$ setting. Specifically, we are able to remove plant TDES ***AddNoPtEntSys*** from the system. In supervisor **B2**, we have not defined the concurrent string of "*no_pt_ent_sys* − *tick*" at state 0. Also, our supervisor **HndlSysDwn** shown in Figure 10.14, that we have developed corresponding to supervisor ***HndlSysDwn*** of the SD setting, does not include any transitions to enable *no_pt_ent_sys* once the system has been shutdown.

Another simplification is that we have removed the state changing *tick* transition between events *pt_ent_sys* and *pt_ent_B2* of ***B2***, and only included a selfloop of *tick* event at state 1 in our supervisor **B2**. This is because our plant model **Con2** already guarantees that these two events cannot occur in the same sampling period. Therefore, there is no need to replicate this logic in **B2**. Due to the same system specifications, the rest of the logic of our supervisors **B2** and **HndlSysDwn** is the same as their corresponding supervisors ***B2*** and ***HndlSysDwn*** of the SD setting.

B4

Supervisor **B4**, shown in Figure 10.15, has been designed in the SD setting to fulfill the specification that buffer B4 never overflows or underflows. It ensures this by enabling/disabling related events at the right time. Since this supervisor does not force any prohibitable event, its design remains unchanged in our $||_{SD}$ setting.

An additional role performed by supervisor **B4** is to ensure that once a part enters buffer B4, the correct follow-up action is performed to take it out of B4. To do this, it first makes sure that once a part is moved from buffer B2 to B4, it does go to Lathe for processing. This is ensured by enabling events *B4_to_L_A*/*B4_to_L_B* after *R_to_B4*. Also, supervisor **B4** assures that after being processed by Lathe, the part goes to the correct buffer, B6 or B7, depending upon its type. It guarantees this by enabling

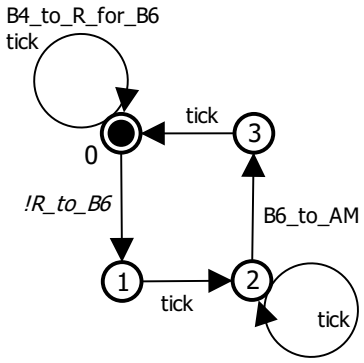


Figure 10.16: Supervisor **B6**

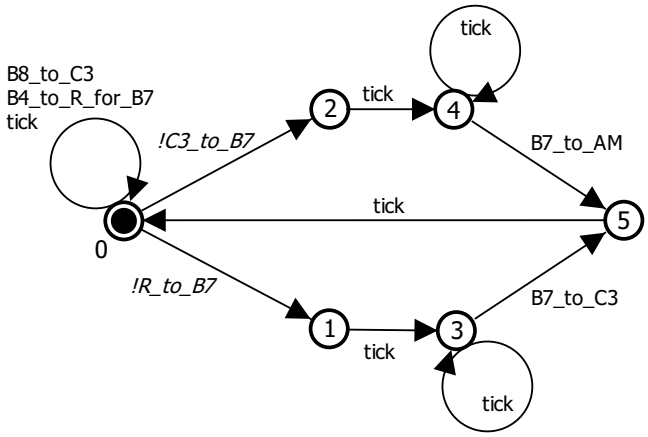


Figure 10.17: Supervisor **B7**

event *B4_to_R_for_B6* after a type A part is generated by Lathe and put into buffer B4 (*L_to_B4_A*), and enabling event *B4_to_R_for_B7* after a type B part is produced by Lathe and placed into buffer B4 (*L_to_B4_B*). We will need this information while discussing supervisors in Sections 10.3.2 and 10.3.3.

B6 and B7

In order to prevent the overflow and underflow of buffers B6 and B7, TDES supervisors **B6** (Figure 10.16) and **B7** (Figure 10.17) have been designed in the SD setting. These supervisors are strictly responsible for enabling/disabling prohibitable events to manage their respective buffers. Since they do not force any prohibitable event by explicitly disabling the *tick* event, they remain unchanged for our $||_{SD}$ setting.

B8

Figure 10.18 shows buffer supervisor ***B8*** of the SD setting. ***B8*** not only prevents the overflow and underflow of buffer B8, it also controls the flow of parts once the part arrives at buffer B7 (*R_to_B7*), goes to PM and then comes back to B7. It does this by watching the part's progress and then forcing prohibitable events *B7_to_C3*, *B8_to_PM* and *B8_to_C3* as needed, by explicitly disabling *tick* at states 2, 6 and 10 respectively to manually satisfy Point ii ($\Rightarrow$) of SD controllability.

It is notable that prohibitable events *B7_to_C3* and *B8_to_C3* are also under the control of supervisor **B7**. This means that ***B8*** needs to make sure that these events must be enabled by **B7** and possible in plant TDES **Con3** before it tries to force them by disabling the *tick*, so that supervisor model does not become uncontrollable with respect to **G**.

In order to force the prohibitable event *B7_to_C3*, ***B8*** needs to know that the part has arrived at buffer B7. This is achieved by replicating the logic of supervisor **B7**

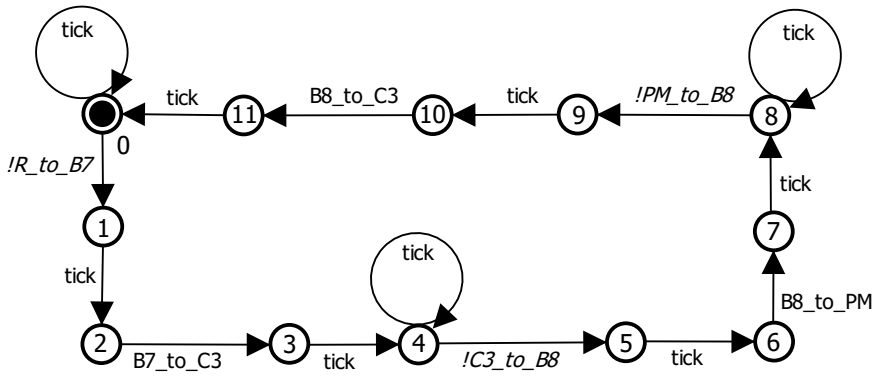


Figure 10.18: Supervisor ***B8***

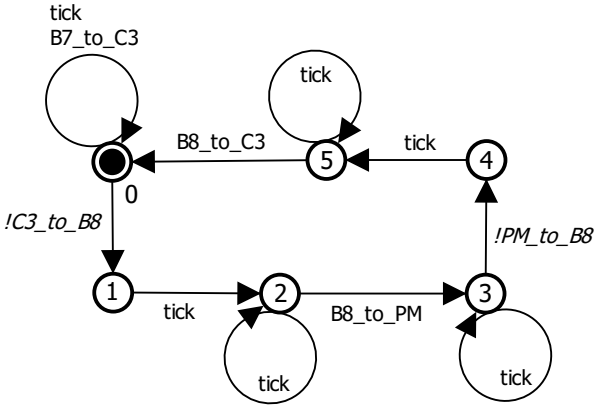


Figure 10.19: Supervisor **B8**

into ***B8***, i.e. by repeating the sequence of events "*R_to_B7* − *tick*" in ***B8***. The fact that ***B8*** cannot just enable a prohibitable event without knowing the part's progress and other supervisor's current behaviour, and needs to explicitly disable *tick* at the right time to force the prohibitable event has made things overly complicated and redundant. This point is also highlighted by Wang and Leduc (2012) while discussing the design of their FMS supervisors.

Figure 10.19 shows buffer supervisor **B8** that we have designed for our $||_{SD}$ setting, with its state size being half (6 states) as compared to the original supervisor ***B8*** (12 states). This is because in the presence of the $||_{SD}$ operator, **B8** can simply enable prohibitable events without explicitly deciding when to force them. Therefore, **B8** does not need to have redundant logic to keep track of the part's progress and supervisor **B7**'s behaviour. Consequently, **B8** gets simplified in two major ways.

First, we have not duplicated the related logic of supervisor **B7** in **B8** by excluding the uncontrollable event *R_to_B7* from **B8**. Second, since **B8** does not need to explicitly decide when to force prohibitable events, we have enabled both *tick* and prohibitable events *B7_to_C3*, *B8_to_PM* and *B8_to_C3* at states 0, 2 and 5 respectively of **B8**. Our $||_{SD}$ operator will automatically disable *tick* and force the appropriate

prohibitable event when it is enabled by all concerned supervisors and possible in the plant model **G**, thus keeping our system controllable with $||_{SD}$.

It is worth-mentioning that although we have added a selfloop of prohibitable event *B7_to_C3* at state 0 of supervisor **B8**, this prohibitable event cannot happen more than once in the same sampling period. This is because our **G** is required to have **S**-singular prohibitable behaviour with $||_{SD}$ with respect to our supervisor model **S**. Moreover, once *B7_to_C3* has occurred in the given sampling period, it will be disabled by supervisor **B7** anyway.

The fact that we are able to enable both *tick* and prohibitable event *B7_to_C3* at state 0 of supervisor **B8** due to our $||_{SD}$ operator has also allowed us to remove two explicit state changing *tick* transitions that were present in the original supervisor ***B8***, and include only a selfloop of *tick* event at state 0 in **B8**. First, we have omitted the state changing *tick* transition between events *B7_to_C3* and *C3_to_B8*. The reason being that our plant model **Con3** makes sure that *tick* always happens between these two events, and **B8** is not preventing this *tick* from occurring by explicitly forcing any event. Second, we have eliminated the state changing *tick* transition after event *B8_to_C3*. This is due to the fact that supervisor **B7** and plant component **Con3** already ensure that *B8_to_C3* and *B7_to_C3* do not happen after one another in the same sampling period. **Con3** also guarantees that *B8_to_C3* and *C3_to_B8* occur in different sampling periods. Therefore, there is no need to replicate this logic in **B8**. We have also removed the redundant logic of state changing *tick* transition between *B8_to_PM* and *PM_to_B8* of ***B8*** and replaced it with a selfloop of *tick* event at state 3 in **B8** due to the plant TDES **PM**.

10.3.2 Robot to B4 to Lathe Path

In order to resolve some nonblocking and concurrency issues along the Robot to B4 to Lathe path of the FMS, three supervisors are designed in the SD setting: ***TakeB2***, ***B4Path*** and ***LathePick***. We will discuss them one by one along with the simplifications that we have made while redesigning them for our $||_{SD}$ setting.

TakeB2

In the FMS, the Robot is responsible for serving buffers B2 and B4. Since both buffers cannot be served at the same time, it is essential to dictate the order in which Robot should provide service to these buffers without blocking the system or starving any one of them. This order of service is specified by supervisor ***TakeB2*** of the SD setting, shown in Figure 10.20.

TakeB2 forces the Robot to first serve buffer B2, followed by buffer B4, and then alternate between the two. It waits until there is a part in buffer B2 by watching event *pt_ent_B2*, after which it moves the part to buffer B4 by forcing the prohibitable event *R_from_B2* and disabling *tick* at state 2 (see Section 1.5.1 for the explanation of its forcing logic). It does not allow the Robot to serve B2 again, i.e. force another

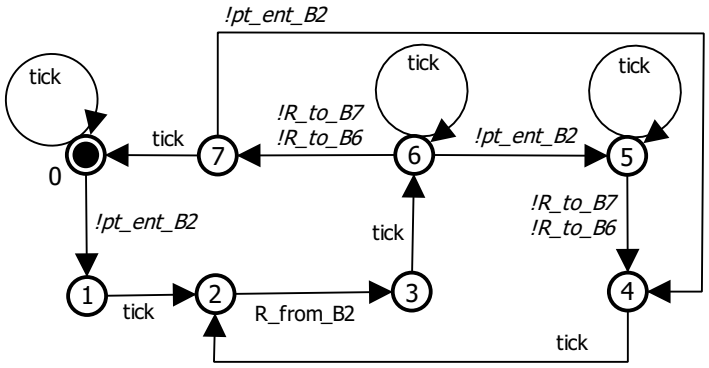


Figure 10.20: Supervisor ***TakeB2***

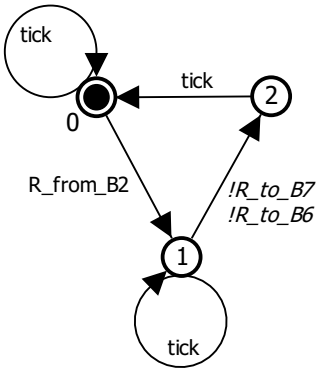


Figure 10.21: Supervisor **TakeB2**

R_from_B2, until Robot has moved the previous part to either buffer B6 (*R_to_B6*) or B7 (*R_to_B7*) from B4 after being processed by Lathe.

This alternate order of serving buffers B2 and B4 also prevents the potential blocking issue that is likely to happen if Robot is allowed to serve buffer B2 two times in a row. In this case, Robot might move second part from B2 to now empty buffer B4 while first part is being processed by Lathe, thus leaving no place for the first part to return to B4.

Figure 10.21 shows the supervisor **TakeB2** that we have designed in our $||_{SD}$ setting. Using our approach, **TakeB2** can simply enable the prohibitable event *R_from_B2* without explicitly deciding when to force it. That is why, we have enabled both *tick* and *R_from_B2* at state 0 of **TakeB2**, leaving it up to our $||_{SD}$ operator to automatically disable *tick* and make the forcing decision for us when *R_from_B2* is possible in TDES plant model **Robot** (Figure 10.2) and enabled by supervisors **B2** (Figure 10.13), **B4** (Figure 10.15), **B4Path** (Figure 10.23) and **TakeB2**.

As **TakeB2** is not explicitly disabling *tick* while enabling *R_from_B2*, this implies that it neither needs to have knowledge about the behaviour of other TDES models, nor does it have to keep track of the part's progress. Therefore, **TakeB2** does not need to duplicate the design logic ("*pt_ent_B2* − *tick*") of **B2**. In fact, there is no need to include the uncontrollable event *pt_ent_B2* in **TakeB2** at all. As a result, all concerns and issues (highlighted in Section 1.5.1) that designers had to deal with after including the event *pt_ent_B2* in ***TakeB2*** automatically vanish in our $||_{SD}$ setting. Also, it is easy to see that **TakeB2** specifies the same order for serving buffers B2 and B4 by Robot as ***TakeB2*** and addresses the blocking issue but in a much simplified way, i.e. reducing 8-state supervisor ***TakeB2*** to 3-state **TakeB2**.

It is notable that ***TakeB2*** also makes sure that events *pt_ent_B2* and *R_from_B2* do not occur in the same sampling period. This is already ensured by supervisor **B2**, therefore we have not cloned this logic in **TakeB2**. Also, we have replaced the explicit state changing *tick* transition between *R_from_B2* and *R_to_B6*/*R_to_B7* of ***TakeB2*** with a selfloop of *tick* at state 1 in **TakeB2** due to the plant model **Robot**.

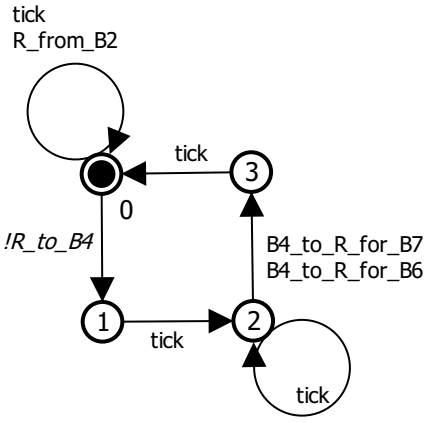


Figure 10.22: Supervisor ***B4Path***

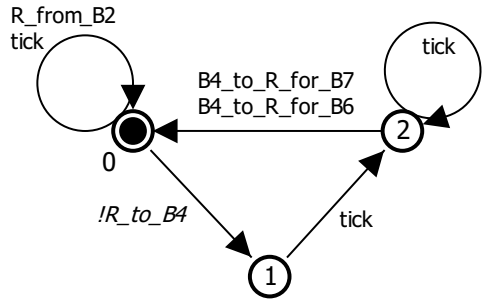


Figure 10.23: Supervisor **B4Path**

B4Path

Supervisor ***B4Path***, given in Figure 10.22, works with buffer supervisor **B4** (Figure 10.15) to ensure proper behaviour on the Robot−B4−Lathe path. It contributes to the correct behaviour of the system by disabling *R_from_B2* once a part is moved to buffer B4 from B2 (*R_to_B4*). Also, only after moving the part from B2 to B4, it enables *B4_to_R_for_B6* and *B4_to_R_for_B7*.

Figure 10.23 shows our supervisor **B4Path** that fulfills the same specification as ***B4Path***. The only way in which the two supervisors differ is that unlike ***B4Path***, **B4Path** does not contain an explicit state changing *tick* transition after event *B4_to_R_for_B6*/*B4_to_R_for_B7*. We are able to skip this transition because our plant model **Robot** and buffer supervisor **B4** already ensure that these two events occur in different sampling periods than events *R_from_B2* and *R_to_B4*, as desired.

LathePick

Supervisor ***LathePick***, shown in Figure 10.24, specifies the order for producing two types of parts by Lathe. It forces Lathe to start with type A part, then produce type B part, and then alternate between the two. It does this by forcing prohibitable events *B4_to_L_A* and *B4_to_L_B* at states 2 and 6 respectively.

As supervisor **B4** is also in charge of enabling/disabling events *B4_to_L_A* and *B4_to_L_B*, ***LathePick*** needs to have knowledge about the behaviour of **B4** so that it can make its forcing decisions correctly. We note that **B4** enables these two events after the occurrence of event *R_to_B4*. Therefore, ***LathePick*** replicates this logic from **B4** and waits for the occurrence of event *R_to_B4*. Once *R_to_B4* happens, then ***LathePick*** forces *B4_to_L_A*/*B4_to_L_B* to avoid any controllability issues.

Figure 10.25 illustrates our 3-state supervisor **LathePick** that does the same job as the 8-state ***LathePick*** supervisor. In the $||_{SD}$ setting, since our supervisors are not required to decide precisely when to force a prohibitable event, therefore we have

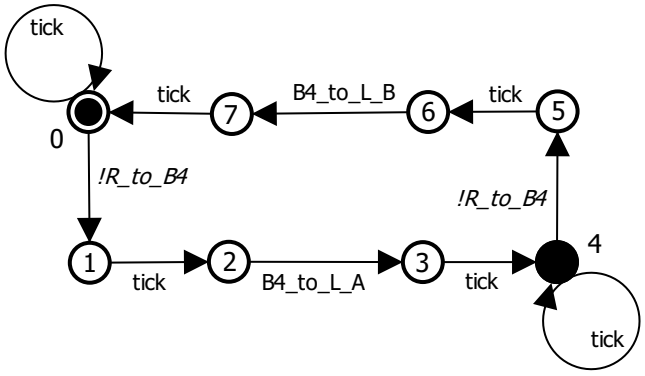


Figure 10.24: Supervisor ***LathePick***

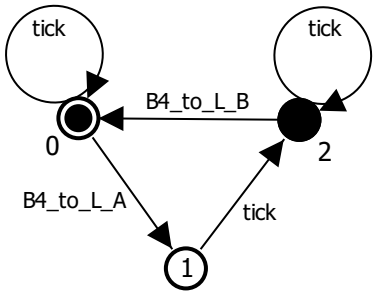


Figure 10.25: Supervisor **LathePick**

not included event *R_to_B4* in **LathePick**. Also, **LathePick** enables both *tick* and prohibitable events *B4_to_L_A* and *B4_to_L_B* at states 0 and 2 respectively, leaving it up to the $||_{SD}$ operator to make the forcing decision while keeping the supervisor controllable with $||_{SD}$ with respect to **G**.

It is notable that supervisor **B4** and plant model **Lathe** guarantee that events *B4_to_L_B* and *B4_to_L_A* occur in different sampling periods. That is why, we have not added an explicit state changing *tick* transition after *B4_to_L_B* in **LathePick**.

10.3.3 Moving Parts from B4 to B6/B7

In order to resolve some nonblocking and concurrency issues associated with moving parts from buffer B4 to B6 and B7, two supervisors, ***TakeB4PutB6*** and ***TakeB4PutB7***, are designed in the SD setting. Below, we discuss the original design of these supervisors followed by their remodelling for our $||_{SD}$ setting.

TakeB4PutB6

The primary purpose of designing supervisor ***TakeB4PutB6***, shown in Figure 10.26, in the SD setting is to decide when to force event *B4_to_R_for_B6*. As this prohibitable event is under the control of three other modular supervisors, **B4**, ***B4Path*** and **B6**, ***TakeB4PutB6*** must not try to force *B4_to_R_for_B6* when it is disabled by any of the other supervisors, or not possible in plant TDES **Robot**.

In order to have knowledge about the behaviour of the other models, ***TakeB4PutB6*** duplicates the logic by watching for event *L_to_B4_A*. As soon as type A part enters buffer B4 from Lathe (*L_to_B4_A*), ***TakeB4PutB6*** forces *B4_to_R_for_B6* to initiate the movement of part A from B4 to B6. It then waits for event *B6_to_AM*, signalling that the part has been moved from B6 to AM and now B6 is ready to accept another part A. ***TakeB4PutB6*** also makes sure that *L_to_B4_A* interleaves properly with *B6_to_AM* by specifying the logic for these events to occur in any order.

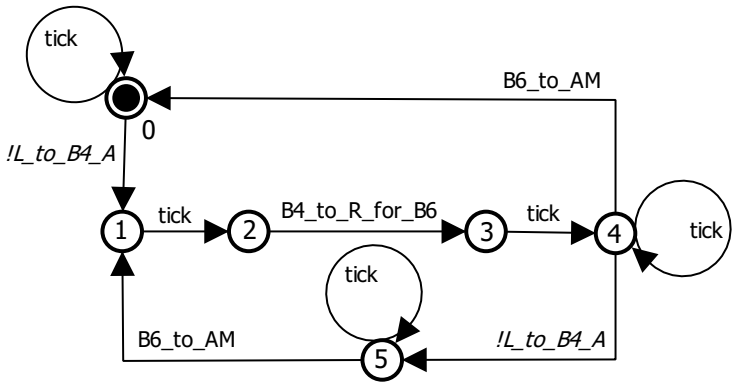


Figure 10.26: Supervisor ***TakeB4PutB6***

In our $||_{SD}$ setting, we do not need to design and include any supervisor corresponding to ***TakeB4PutB6*** because of the following two reasons: 1) In the presence of the $||_{SD}$ operator, we are not required to explicitly decide and specify when to force *B4_to_R_for_B6* by keeping track of other supervisors' behaviour. In fact, when *B4_to_R_for_B6* is possible in **Robot** and enabled by supervisors **B4**, ***B4Path*** and **B6**, the $||_{SD}$ operator will automatically disable *tick* to force *B4_to_R_for_B6* in the closed-loop system. 2) Our buffer supervisor **B6** already ensures that Robot cannot begin to move type A part from buffer B4 to B6 (*B4_to_R_for_B6*) until the previous part has been taken out of B6 and moved to AM (*B6_to_AM*). **B6** guarantees this by disabling event *B4_to_R_for_B6* once it has happened, and re-enables it only after event *B6_to_AM* has occurred.

As a result, we do not need to specify/replicate any logic and no supervisor exists in our $||_{SD}$ setting corresponding to supervisor ***TakeB4PutB6*** of the SD setting.

TakeB4PutB7

Supervisor ***TakeB4PutB7*** designed in the SD setting is shown in Figure 10.27. Besides deciding when to force the prohibitable event *B4_to_R_for_B7* to initiate the movement of type B part from buffer B4 to B7, ***TakeB4PutB7*** also handles a potential blocking issue as part B moves along the B7−PM−B7 path.

In order to determine when to force the prohibitable event *B4_to_R_for_B7*, ***TakeB4PutB7*** must take into account the behaviour of supervisors **B4**, ***B4Path*** and **B7**, and plant model **Robot**, as these models are also in charge of enabling/disabling *B4_to_R_for_B7*. Therefore, event *L_to_B4_B* is added to ***TakeB4PutB7*** to replicate the related logic from supervisor **B4** and determine the right time for forcing *B4_to_R_for_B7*. As soon as *L_to_B4_B* occurs, ***TakeB4PutB7*** disables *tick* to force *B4_to_R_for_B7* at state 2.

When part B is placed in buffer B7 from B4, it first goes to PM for processing. It is possible that another part B is put in the now empty buffer B7 by Robot, leaving no place for the returning part, thus blocking the system. Supervisor ***TakeB4PutB7***

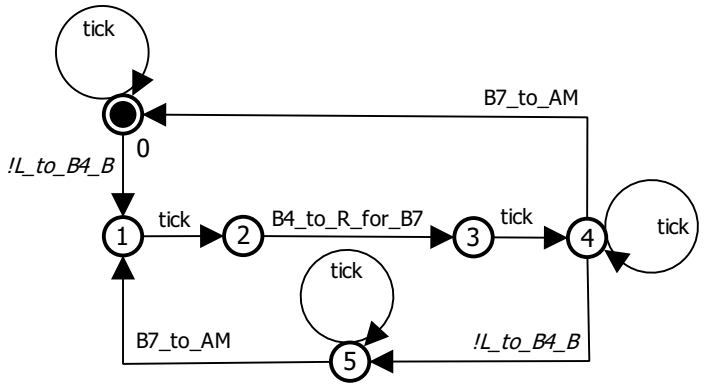


Figure 10.27: Supervisor ***TakeB4PutB7***

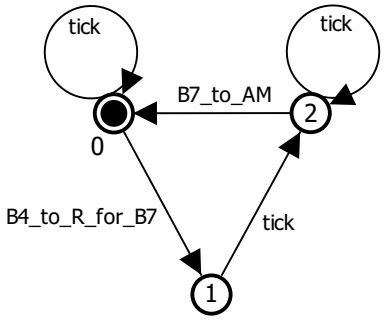


Figure 10.28: Supervisor **TakeB4PutB7**

prevents this situation from happening by waiting for the part to return to buffer B7 from PM and then moved to AM (*B7_to_AM*), before allowing the Robot to take another part B from B4 (*B4_to_R_for_B7*) to be placed into B7. The design logic for proper interleaving of events *L_to_B4_B* and *B7_to_AM* is also specified in ***TakeB4PutB7***.

To fulfill these specifications, our 3-state supervisor **TakeB4PutB7** designed for the $||_{SD}$ setting is given in Figure 10.28. Since we do not need to manually decide when to force *B4_to_R_for_B7*, we have neither added the logic for keeping track of other supervisors' behaviour and the plant model, nor forcing of event *B4_to_R_for_B7* in **TakeB4PutB7**. As a result, our supervisor does not contain the waiting event *L_to_B4_B*, and enables both *tick* and prohibitable event *B4_to_R_for_B7* at state 0.

Once *B4_to_R_for_B7* has occurred, **TakeB4PutB7** disables this event to avoid the above-mentioned blocking issue. This event is re-enabled after the occurrence of *B7_to_AM*, i.e. when part B returning from PM is moved from buffer B7 to AM. Also, we note that buffer supervisor **B7** already guarantees that events *B7_to_AM* and *B4_to_R_for_B7* always occur in different sampling periods. For this reason, we have not added an explicit state changing *tick* transition after *B7_to_AM* in **TakeB4PutB7**, as present in supervisor ***TakeB4PutB7*** of the SD setting.

10.3.4 B6/B7 to AM to Exit Path

Now we will discuss the movement of parts from buffers B6 and B7 to finishing machine AM, from where finished parts finally exit the system. In order to resolve several concurrency issues along this path, supervisors ***ForceB6toAM***, ***ForceB7toAM***, ***ForceInitAM*** and ***AMChooser*** have been designed in the SD setting. These supervisors are heavily dependent upon one another and work closely together to make several decisions. We will analyze them one by one, and then discuss how their design and logic get simplified in the presence of our $||_{SD}$ operator.

Parts are moved from buffers B6 and B7 to AM using prohibitable events *B6_to_AM*

and *B7_to_AM* respectively. Before accepting and processing any part, AM needs to initialize, which is indicated by prohibitable event *init_AM*. This means the first thing that needs to determined and specified along this path is when to force these three prohibitable events by explicitly disabling *tick* in order to satisfy Point ii ($\Rightarrow$) of the SD controllability property. We must also make sure that when one modular supervisor is trying to force a prohibitable event, it must be enabled by all the concerned supervisors and possible in plant TDES **AM**, to keep the system controllable.

This is further complicated by the fact that parts might be waiting in both buffers B6 and B7 to go to AM for processing. This implies that another decision that needs to be made is to determine which buffer to service first. Ideally, these decisions should be reflected in the supervisor models without significant reuse of logic which seemed non-obvious, as stated in Wang and Leduc (2012).

The solution devised in the SD setting to address the above-mentioned issues is to introduce four new prohibitable expansion events that provide communication between the modular supervisors. These expansion events are *no_B6_to_AM_a*, *no_B6_to_AM_b*, *no_B7_to_AM_a* and *no_B7_to_AM_b*. They are introduced to the system by designing two additional plant TDES, ***AddNoB6toAM*** (Figure 10.9) and ***AddNoB7toAM*** (Figure 10.10).

ForceB6toAM and ForceB7toAM

Supervisor ***ForceB6toAM***, shown in Figure 10.29, is designed in the SD setting to force the prohibitable event *B6_to_AM*. As *B6_to_AM* is under the control of supervisors **B6** and ***TakeB4PutB6***, ***ForceB6toAM*** replicates the design logic from **B6** by adding the watch event *R_to_B6*. ***ForceB6toAM*** waits for the occurrence of *R_to_B6*, signalling that there is a part in buffer B6 waiting to go to AM. It then forces *B6_to_AM* by explicitly disabling *tick* at state 2 in accordance with Point ii ($\Rightarrow$) of SD controllability.

However, if *B6_to_AM* is currently not possible in the plant TDES **AM** (Figure 10.6) or disabled by other supervisors, ***ForceInitAM*** (Figure 10.31) and ***AMChooser*** (Figure 10.33) that are also in control of *B6_to_AM*, then ***ForceB6toAM*** has no way of knowing this. In this case, system will become uncontrollable because *B6_to_AM* could not be forced and *tick* is already disabled by ***ForceB6toAM***.

This issue is handled by adding a loop of concurrent string "$no\_B6\_to\_AM\_a/no\_B6\_to\_AM\_b - tick$" at state 2 of supervisor ***ForceB6toAM***. The idea is to use expansion events, *no_B6_to_AM_a* or *no_B6_to_AM_b*, as alternative forcing options when *B6_to_AM* could not be forced. Since enablement information needs to be coordinated between three supervisors, that is why the designers have added two expansion events, '*a*' and '*b*'.

As *no_B6_to_AM_a* and *no_B6_to_AM_b* are meant to be used as substitute forcing options for *B6_to_AM*, they must only be enabled when it is not possible to force *B6_to_AM*. Also, it is important to make sure that only one of these three prohibitable

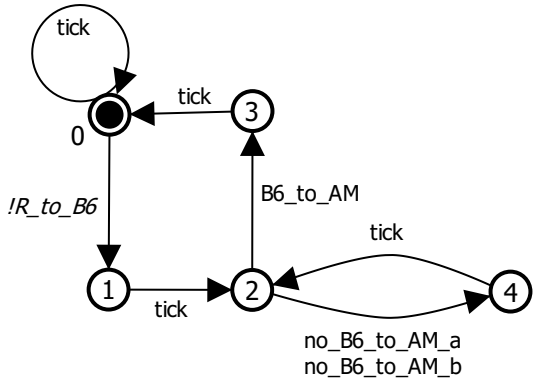


Figure 10.29: Supervisor ***ForceB6toAM***

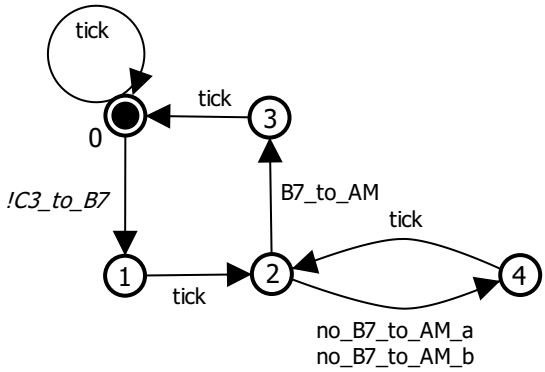


Figure 10.30: Supervisor ***ForceB7toAM***

events is possible in the system at a given time. Supervisors ***ForceInitAM*** and ***AMChooser*** contain the logic to fulfill these two requirements.

In order to force the prohibitable event *B7_to_AM*, supervisor ***ForceB7toAM***, shown in Figure 10.30, has been designed in the SD setting. As *B7_to_AM* is under the control of supervisors **B7** and ***TakeB4PutB7***, therefore ***ForceB7toAM*** duplicates the design logic from supervisor **B7** by including the watch event *C3_to_B7*. The rest of the logic of ***ForceB7toAM*** is same as ***ForceB6toAM***. Also, ***ForceB7toAM*** communicates with plant component **AM**, and supervisors ***ForceInitAM*** and ***AMChooser*** in a similar fashion as ***ForceB6toAM***.

ForceInitAM

Figure 10.31 shows supervisor ***ForceInitAM*** of the SD setting that is primarily responsible for deciding when to force prohibitable event *init_AM*. By disabling *tick* at state 0, it forces *init_AM* right away. After AM has processed the part received from buffer B6 or B7 (*B6_to_AM*/*B7_to_AM*), this supervisor then waits for the finished part to leave the system (*fin_from_B6*/*fin_from_B7*) before forcing another *init_AM*.

Another task performed by ***ForceInitAM*** is to make sure that '*a*' and '*b*' expansion events are never eligible in the system at the same time. It ensures this by enabling '*a*' events when *B6_to_AM*/*B7_to_AM* are not possible in the plant TDES **AM**. When they are possible in **AM**, ***ForceInitAM*** enables '*b*' events instead.

As supervisor ***AMChooser*** (Figure 10.33) ignores '*a*' events, this guarantees that '*a*' events will never be disabled when ***ForceInitAM*** needs them. As ***ForceInitAM*** never disables '*b*' events when *B6_to_AM*/*B7_to_AM* are possible in **AM**, this ensures that '*b*' events will never be disabled when ***AMChooser*** needs them.

By manually devising and explicitly incorporating this intricate logic in the supervisor models, designers made sure that the two supervisors do not interfere with each other with respect to these expansion events.

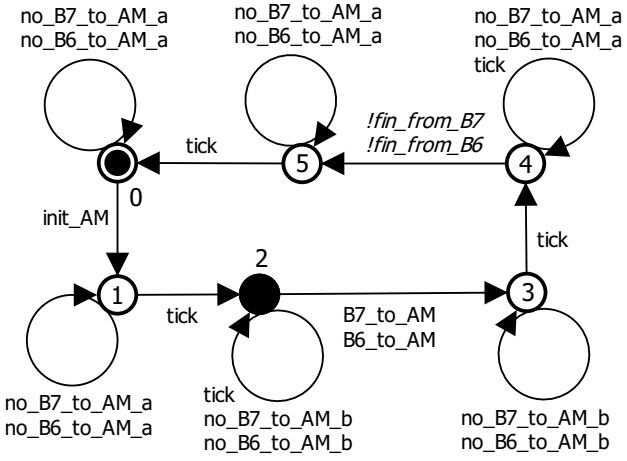


Figure 10.31: Supervisor ***ForceInitAM***

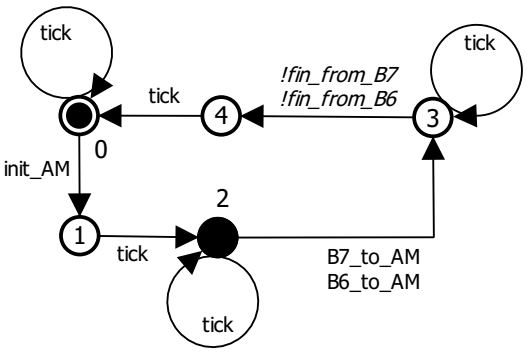


Figure 10.32: Supervisor **InitAM**

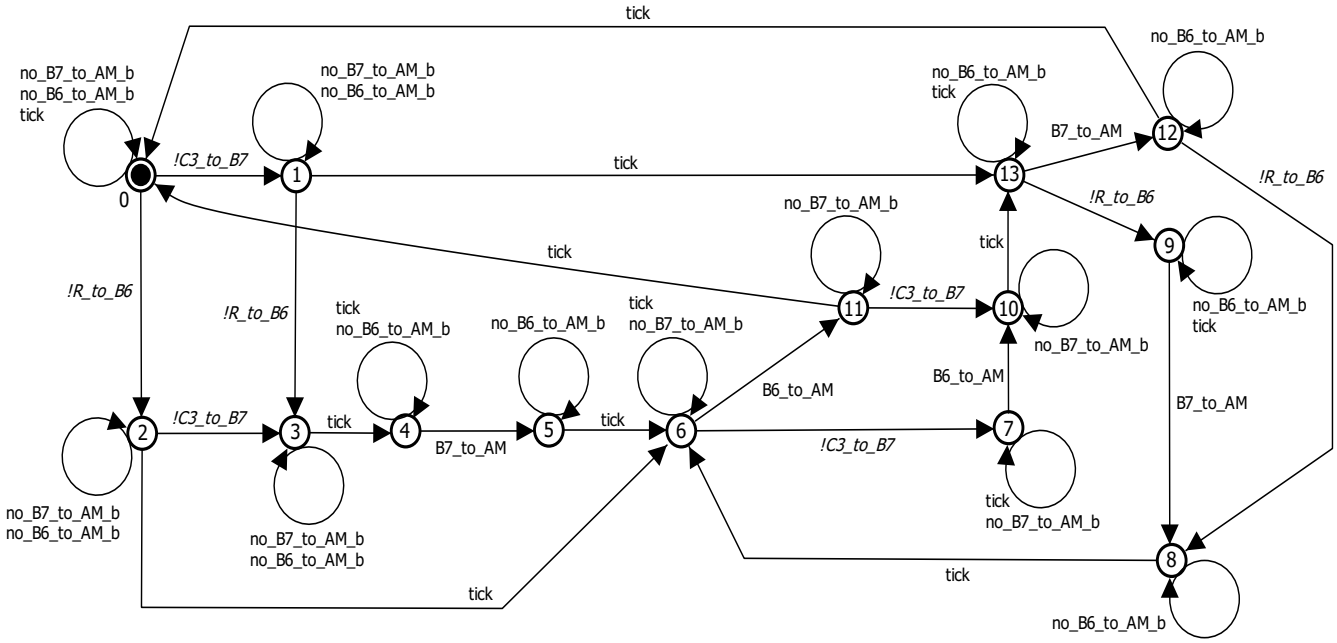


Figure 10.33: Supervisor ***AMChooser***

AMChooser

The primary purpose of the supervisor ***AMChooser***, given in Figure 10.33, of the SD setting is to dictate the order in which AM accepts the parts from buffers B6 and B7, when both buffers have a part waiting to be processed by AM. If parts A and B arrive in both buffers (*R_to_B6*, *C3_to_B7*) in the same sampling period, then ***AMChooser*** forces AM to first take the part from B7 (*B7_to_AM*), and then from B6 (*B6_to_AM*). The reason is that there are more machines along the B7−PM−B7 path that should be kept busy. If only one buffer has a part waiting, then this part

is taken by AM for processing.

In order to enforce this order for processing parts, ***AMChooser*** sometimes has to disable prohibitable events *B6_to_AM* and *B7_to_AM*. In such cases, it enables the appropriate '*b*' expansion events as a forcing alternative. This also guarantees that substitute forcing options of *B6_to_AM* and *no_B6_to_AM_b* are never enabled at the same time. The same is true for *B7_to_AM* and *no_B7_to_AM_b*.

Remodelling of Modular Supervisors for the $||_{SD}$ Setting

In the SD setting, the only reason for introducing four prohibitable expansion events was to aid in communication between various modular supervisors in order to specify the explicit forcing decisions, while making sure that all desired properties and system specifications are satisfied. In the $||_{SD}$ setting, since modular supervisors are only concerned about their own behaviour, this eradicates the need to add expansion events to our $||_{SD}$ system. As a result, we can exclude all these expansion events and their corresponding plant models, ***AddNoB6toAM*** and ***AddNoB7toAM***, from our set of FMS plant components for the $||_{SD}$ setting.

In the SD setting, the primary purpose of designing supervisors ***ForceB6toAM*** and ***ForceB7toAM*** was to decide when to force prohibitable events *B6_to_AM* and *B7_to_AM* respectively. Since we have the $||_{SD}$ operator that automatically makes these forcing decisions for us, we will not include these two supervisors in our set of FMS modular supervisors. Please note that the order of occurrence of events enforced by these two supervisors is already present in the corresponding buffer supervisors, **B6** and **B7**.

Our $||_{SD}$ supervisor **InitAM**, shown in Figure 10.32, manages the initialization of AM and the movement of parts along the B6/B7−AM−exit path. Since we are not required to explicitly force prohibitable event *init_AM*, we have enabled both *tick* and *init_AM* at state 0. Also, plant TDES **AM** already makes sure that events *B6_to_AM*/*B7_to_AM* and *fin_from_B6*/*fin_from_B7* occur in different sampling periods. Therefore, we have not duplicated this logic in **InitAM**, as specified by ***ForceInitAM*** in the SD setting.

It is worth noting how simple this supervisor's design and logic has become in the absence of all expansion events and their transitions, as compared to its corresponding supervisor ***ForceInitAM*** of the SD setting. With no expansion events, we do not need to do any extra design effort to figure out how many other supervisors **InitAM** will communicate with, determine the number of expansion events required for communication, and then enable/disable all expansion events at the right time by keeping track of plant and other supervisors' behaviours.

Figure 10.34 illustrates our supervisor **AMChooser** of the $||_{SD}$ setting. Essentially, we have derived it from supervisor ***AMChooser*** of the SD setting after removing all expansion event transitions. **AMChooser** enforces the same order on AM for accepting parts from buffers B6 and B7 as ***AMChooser***. However, unlike ***AMChooser***,

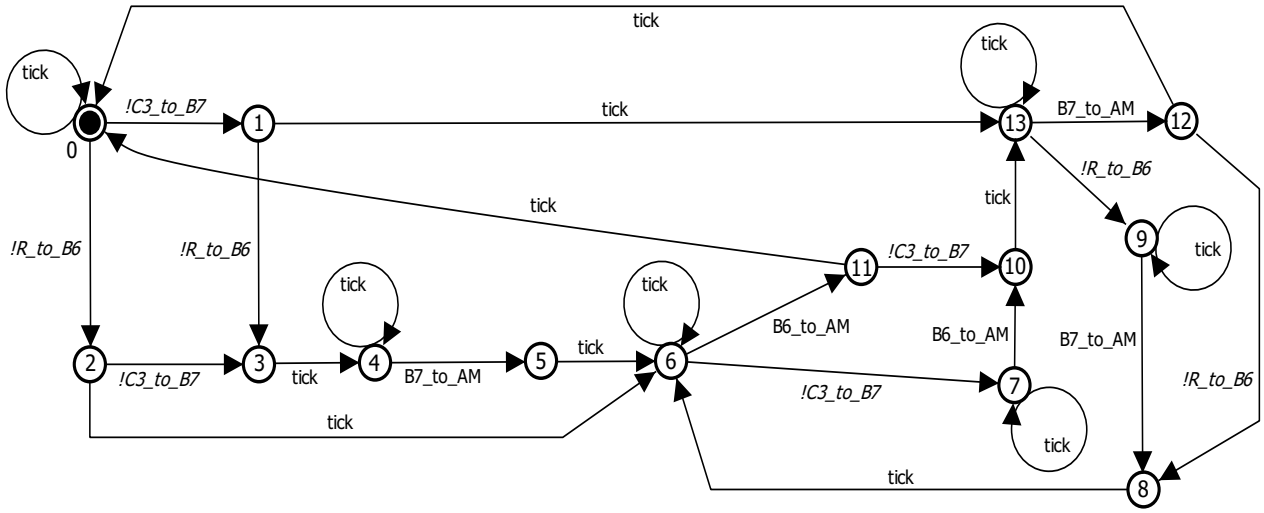


Figure 10.34: Supervisor **AMChooser**

our supervisor **AMChooser** does not need to keep track of the plant model and other supervisors' behaviour to enable/disable appropriate expansion events at the right time. This results in greatly reducing the complexity of its design and logic.

10.4 Results and Discussion

Now it is time to present and discuss our results for the FMS example. Our complete results are shown in Table 10.2. In order to be clear and precise in our discussion, we will refer to the FMS TDES models designed in the SD setting as the "*SD system*", and the simplified FMS TDES models designed for our $||_{SD}$ setting as the "$||_{SD}$ *system*".

10.4.1 Theoretical TDES

By looking at the theoretical TDES models discussed in the previous section, we note that for the same FMS specifications, we are able to model our $||_{SD}$ system by designing fewer plant components and modular supervisors. This is because, unlike the SD system, we did not have to introduce and manage five prohibitable expansion events to aid in communication between different modular supervisors and make explicit forcing decisions in our $||_{SD}$ system. These results are summarized in the topmost section of Table 10.2.

10.4.2 Verification Results

In order to evaluate the performance of verifying our $||_{SD}$ properties, we implemented the SD synchronous product operator and our tweaked algorithms (presented in Chapter 9) as part of the DES research tool, DESpot (2023). Our code is based on the

Table 10.2: FMS Example Results in the SD and $||_{SD}$ Setting

	SD System		$||_{SD}$ **System**	
Plant Components	10		7	
Modular Supervisors	15		12	
System Events	31		26	
Supervisor Properties	*Verification Time (seconds)*			
CS Deterministic	< 1		< 1	
Non-Selfloop ALF	< 1		< 1	
Closed-Loop System Properties	**SD Algorithms**	$||_{SD}$ **Algorithms**	**SD Algorithms**	$||_{SD}$ **Algorithms**
Nonblocking	< 1	< 1	< 1	< 1
Untimed Controllability	< 1	< 1	< 1	< 1
Timed Controllability	< 1	< 1	< 1	< 1
Proper Time Behaviour	< 1	< 1	< 1	< 1
Plant Completeness	< 1	< 1	< 1	< 1
ALF	2	2	1	1
SD Controllability & **S**-Singular Prohibitable Behaviour	30	26	False	7
Check All	32	28	False	8
State Size	***82,608***	***82,608***	***56,244***	***49,020***

source code written by Wang (2009) that verifies the SD supervisory control methodology. The code uses the *BuDDy* package (Lind-Nielsen, 2002), a C++ library that implements standard BDD structures and operations.

In the rest of this discussion, we will refer to the algorithms implemented by Wang (2009) as the "*SD algorithms*". These SD algorithms check various properties in the SD setting (Chapter 3) and rely on the standard synchronous product operator to form the closed-loop system. On the other hand, the adapted algorithms that we have implemented for our $||_{SD}$ setting will be referred to as the "$||_{SD}$ *algorithms*". These $||_{SD}$ algorithms verify the $||_{SD}$ version of the properties (introduced in Chapter 4) and use our SD synchronous product operator to construct the closed-loop system. Please recall that, as mentioned in Chapter 9, although we are reusing some algorithms of the SD setting in our $||_{SD}$ setting without modifying their steps, still these algorithms are actually different in the two settings because of their way of constructing the closed-loop system in order to verify the desired properties.

In order to analyze and compare the FMS SD and $||_{SD}$ systems in detail, we decided to run SD and $||_{SD}$ algorithms on both systems. In other words, we not only verified our $||_{SD}$ system in our $||_{SD}$ setting by running our $||_{SD}$ algorithms, but we also tested it in the SD setting by running SD algorithms to find out which properties does it fail to satisfy (algorithms return "$False$") in the SD setting in the absence of our $||_{SD}$

operator. Similarly, besides running SD algorithms on the SD system, we also ran our $||_{SD}$ algorithms on the SD system to evaluate its performance in our $||_{SD}$ setting.

Table 10.2 shows our verification results for running various SD and $||_{SD}$ algorithms on the SD and $||_{SD}$ systems. These tests are performed on a machine running Windows 10 with 16GB of RAM and 2.6GHz Intel 6-core processor.

Supervisor Properties

As the ultimate goal of designing TDES supervisors in the SD supervisory control theory is to generate the corresponding SD controller, we started by verifying two properties that play an important role in this translation process. These two properties are CS deterministic and non-selfloop ALF supervisors. As shown in Table 10.2, TDES supervisors of both SD and $||_{SD}$ systems passed each of these checks in less than 1 second.

Our next step is to verify various properties of the SD and $||_{SD}$ closed-loop systems by running the SD and $||_{SD}$ algorithms. Before we analyze the results of these tests in detail, first we wish to highlight some important points about state sizes of the two systems that are constructed by the SD and $||_{SD}$ algorithms.

State Space Size of Closed-Loop System

By looking at the state size of the SD system given in Table 10.2, we observe that although SD and $||_{SD}$ algorithms use different synchronization operators to construct the closed-loop system, state size of the SD system is same in both cases, i.e. 82,608. The reason is that the SD system was originally designed for the SD setting, where designers are responsible for manually satisfying Point ii ($\Rightarrow$) of SD controllability. In this case, the $||_{SD}$ operator does not find any states where it has to disable *tick* in the presence of an enabled prohibitable event while constructing the closed-loop system. Consequently, the synchronization mechanism of the $||_{SD}$ operator essentially becomes equivalent to the standard synchronous product operator. This results in having the same state space for the SD system in both cases.

On the contrary, SD and $||_{SD}$ algorithms specify different state sizes for our $||_{SD}$ system. Specifically, our $||_{SD}$ closed-loop system constructed by the SD algorithms has 56,244 states, whereas $||_{SD}$ algorithms construct the state space of 49,020 states. This is because, keeping in view the synchronization mechanism of the $||_{SD}$ operator, we have enabled both *tick* and prohibitable events at various states of the TDES supervisors while modelling our $||_{SD}$ system. As the synchronous product operator is not capable of automatically disabling *tick* event in the presence of enabled prohibitable events while forming the closed-loop system, this is why the SD algorithms construct a bigger state space for our $||_{SD}$ system than our $||_{SD}$ algorithms.

In essence, the key point to note is that state size for the FMS example has reduced from 82,608 states in the SD setting to 49,020 states in our $||_{SD}$ setting. This represents a reduction of 40% in the overall state space of the FMS closed-loop system.

This decrease in the state space is because of two reasons. First, in the presence of our $||_{SD}$ operator, less number of TDES plant and supervisor components are required to model the same system specifications. Second, as evident in Section 10.3, the size and logical design complexity of most of the modular supervisors of the SD system have been greatly reduced for our $||_{SD}$ system.

Next, we discuss our results of verifying various properties of the SD and $||_{SD}$ systems by running SD and $||_{SD}$ algorithms.

Closed-Loop System Properties

As shown in Table 10.2, both SD and $||_{SD}$ systems satisfy the properties of nonblocking, untimed controllability, timed controllability, proper time behaviour and plant completeness in both settings. Each of these checks is completed individually in less than 1 second.

By running the ALF test, both SD and $||_{SD}$ systems are found to be ALF. However, this check was completed for the SD system in 2 seconds, whereas our $||_{SD}$ system took only 1 second to pass this test in the SD and $||_{SD}$ settings. Given the fact that we have reused the ALF algorithm of the SD setting in our $||_{SD}$ setting without changing its steps, this difference in verification time can be attributed to different state sizes of the SD and $||_{SD}$ systems. As our $||_{SD}$ system has a reduced state space as compared to the SD system, this property gets verified more efficiently for our $||_{SD}$ system in both settings.

Currently, in DESpot, the check for **S**-singular prohibitable behaviour is implemented as part of the SD controllability test. Therefore, we verified these two properties together for the SD and $||_{SD}$ systems. Using SD algorithms, it took 30 seconds to verify these properties for the SD system. However, when we checked these properties of the SD system using our $||_{SD}$ algorithms, verification time dropped to 26 seconds. This is because our $||_{SD}$ algorithm tests the property of SD controllability with $||_{SD}$, which does not include an explicit check for Point ii ($\Rightarrow$) of SD controllability. This saves time by performing one less check in the presence of our $||_{SD}$ operator as compared to the SD setting.

When we tried to verify the property of SD controllability for our $||_{SD}$ system using SD algorithms, the algorithms returned $False$. The reason is quite obvious. Since we have not explicitly disabled *tick* while enabling prohibitable events at various states of the modular $||_{SD}$ supervisors, our $||_{SD}$ system fails to satisfy the constraint imposed by Point ii ($\Rightarrow$) of SD controllability in the SD setting.

In order to test the properties of SD controllability with $||_{SD}$ and **S**-singular prohibitable behaviour with $||_{SD}$ for the $||_{SD}$ system, we ran our corresponding $||_{SD}$ algorithms. Our $||_{SD}$ system not only passed these checks but the verification process was completed within 7 seconds, as opposed to the SD system that took 30 seconds to pass these tests in the SD setting.

This indicates that for the FMS example, we have verified these two properties of

our $||_{SD}$ system 4x faster as compared to the SD system. In other words, we recorded a 76.6% reduction in verification time and more than 300% increase in performance in our $||_{SD}$ setting as compared to the SD setting with respect to the verification of these two properties.

This significant reduction in verification time is primarily due to the smaller state size of our $||_{SD}$ system as compared to the SD system, that we are able to achieve due to the automatic *tick* disablement mechanism of the $||_{SD}$ operator. Moreover, our $||_{SD}$ algorithms check one less condition as part of the SD controllability with $||_{SD}$ property in the presence of the $||_{SD}$ operator.

In DESpot, we also have an option to check the desired system properties all at once (Check All). In the SD setting, it took 32 seconds to verify all properties of the SD system, while our $||_{SD}$ system passed all tests in 8 seconds in our $||_{SD}$ setting. On the whole, our results demonstrate a time reduction of 75% and performance increase of exactly 300% in our $||_{SD}$ setting compared to the SD setting. This shows that we are able to do complete verification of our FMS $||_{SD}$ system 4x faster than its corresponding FMS SD system.

10.4.3 Miscellaneous Discussion

We will close this section by discussing two important points.

1. It is worth-mentioning that, apparently, our $||_{SD}$ operator does more work than the synchronous product while synchronizing plant and supervisor models to construct the closed-loop system. This is because our $||_{SD}$ operator is required to figure out whether to enable/disable *tick* event at every state of the closed-loop system using a more complex logic than synchronous product.

Nevertheless, we are still able to notice a visible decline in the overall verification time of the FMS example in our $||_{SD}$ setting as compared to the SD setting. This suggests that the slightly complicated synchronization logic of our $||_{SD}$ operator has not adversely affected the overall performance of our $||_{SD}$ algorithms.

2. We would like to point out that we ran the SD algorithms on our $||_{SD}$ system because we wanted to inspect that other than manually satisfying Point ii ($\Rightarrow$) of SD controllability, is there any other reason/significance which necessitates the design of a complicated SD system instead of modelling a simpler $||_{SD}$ system? As shown in Table 10.2, our results indicate that our $||_{SD}$ system satisfies all properties in the SD setting except for Point ii ($\Rightarrow$) of SD controllability.

This makes it evident that the design and verification of the FMS example got a lot more complicated in the SD setting just because this one property needed to be satisfied manually. This clearly shows the significance of our $||_{SD}$ operator that has made the design and verification process of our FMS $||_{SD}$ system relatively simple, easy and efficient, by providing a guarantee to automatically satisfy this intricate property in our $||_{SD}$ setting.

Chapter 11

Introduction to Moore FSM to TDES Translation

In this chapter, we present a novel approach for the automatic translation of Moore synchronous Finite State Machines (FSM) into TDES supervisors. In devising this approach, we utilize the structural similarity created by the SD supervisory control theory between the two models (Wang, 2009; Wang and Leduc, 2012; Leduc *et al.*, 2014). In this chapter, we will explain our translation approach with the help of an illustrative example. In Chapter 12, we will formalize the approach by providing algorithms.

In Wang (2009); Wang and Leduc (2012), the authors provide a formal and structured way of translating a TDES supervisor into an SD controller, modelled as a Moore FSM (Brown and Vranesic, 2013). This TDES-FSM translation method, briefly discussed in Section 3.7, has been implemented in the DES research tool, DESpot (2023), by Hamid (2014). Following a two-step approach, Hamid has developed algorithms to convert a TDES supervisor into a Moore FSM, which is then used to generate Verilog code for the digital logic implementation (Brown and Vranesic, 2013).

While developing our FSM-TDES translation approach, one of our major goals is to make our approach consistent and compatible with the existing TDES-FSM translation method and its corresponding DESpot implementation. This provides an additional benefit by enabling the system designers to go back and forth between the two models using the two translation approaches, i.e. easily translate one model into the other, regardless of whether they started with a TDES supervisor or a Moore FSM.

It is obvious that Hamid's TDES-FSM implementation approach was not developed taking into consideration the possibility of doing the reverse (FSM-TDES) translation. For this reason, we needed to make some changes in the existing TDES-FSM translation approach of DESpot in order to build compatibility between the two translation approaches. We will discuss some of these changes in this chapter, but

please refer to Appendix C for details on how we have updated Hamid's TDES-FSM translation algorithms to implement these changes.

We begin this chapter by describing the input structure for Moore FSM that we will use for our FSM-TDES translation approach, and the changes we have made to the FSM output format of Hamid (2014). We will discuss them at a high level here, but leave the detailed description for Appendix C. Next, we list down some consistency and design requirements that must be satisfied by the input Moore FSM before they could be considered "valid" for the FSM-TDES translation. This is followed by an in-depth explanation of our complete FSM-TDES translation approach. We demonstrate our translation method step by step with the help of an example.

Note: From now on, we will refer to "Moore synchronous FSM" as "Moore FSM" or simply as "FSM" for conciseness.

11.1 Moore System as an Input

For any physical system, the designers can choose to design either a single, more complex TDES supervisor or multiple modular supervisors. Hamid's (2014) DESpot TDES-FSM implementation method works well in both cases by generating one *individual Moore FSM* corresponding to each TDES supervisor. Besides these individual FSM, his translation algorithm also generates one *central FSM* for the system. The central FSM file defines the global input and output signals, and contains information about each individual Moore FSM. The central and individual FSM are stored in the XML file format (Ray, 2003).

Just like TDES designers, the control practitioners may wish to model a system controller either as a monolithic Moore FSM or as multiple individual FSM. Our FSM-TDES translation approach, presented in this chapter, is capable of handling both possibilities, and generates one TDES supervisor corresponding to each input Moore FSM.

For a given control system, the input to our FSM-TDES translation method is a Moore system that consists of: 1) one *central FSM* file for the system, and 2) one or more *individual FSM* files, where each individual file contains one Moore FSM's specifications. We require that all FSM files must be expressed in XML format. Basically, we have customized Hamid's output XML file format to suit our needs and make our stuff compatible with what already exists.

Below, we describe the input format for central and individual FSM, as well as the changes that we have made. Please note that in Chapter 13, we apply our FSM-TDES translation approach to an example of a 4-bit Combination Lock. We will use a small portion of this example to explain our input structure.

The 4-bit Combination Lock is a digital lock system that uses a 4-bit passcode to provide secured access to authentic users, and has three user buttons: **Enter**, **Change** and **Reset**. Users can unlock the combination lock, either to *open* the door

or *change* the currently saved passcode, by entering the existing passcode and pressing the appropriate button (**Enter** or **Change**). However, if the user enters an incorrect passcode, the *alarm* goes off. *Alarm* can only be cancelled by pressing the **Reset** button, which will also reset the currently saved passcode to its default value. Please refer to Section 13.1 for a thorough explanation of the complete system specification and functionality.

11.1.1 Individual Moore FSM

Figure 11.1 shows the **OpenLock** Moore FSM from the Combination Lock example. We will use this graphical FSM to illustrate our discussion. Please see Section C.1.1 in Appendix C for a description of the XML file for **OpenLock**. Our FSM-TDES translation method requires the designers to specify one XML file corresponding to each individual Moore FSM that needs to be translated.

The **OpenLock** FSM consists of two states, "1" and "2", and three signals, *open*, *enter* and *equal*. At each state of **OpenLock**, the value of the output *open* is indicated. The FSM arrows represent next state boolean conditions in terms of inputs, *enter* and *equal*. In the boolean expressions, "$\cdot$" symbol represents the **AND** operator, "+" represents the **OR** operator, and "!" represents the **NOT** operator. The arrow labelled **Reset** indicates the reset state of **OpenLock**. Please note that in our figures, we will write " . " instead of "$\cdot$", as " . " is easier to produce. Please refer to Section 13.2.1 to gain familiarity with the complete graphical notation of a Moore FSM.

The **OpenLock** FSM, as the name suggests, controls the functionality of unlocking the Combination Lock. In the lock state (state "1", where output of signal *open* = 0), if the user enters the correct passcode (indicated by the input signal, *equal*) and presses the **Enter** button (represented by the input signal, *enter*), the door opens

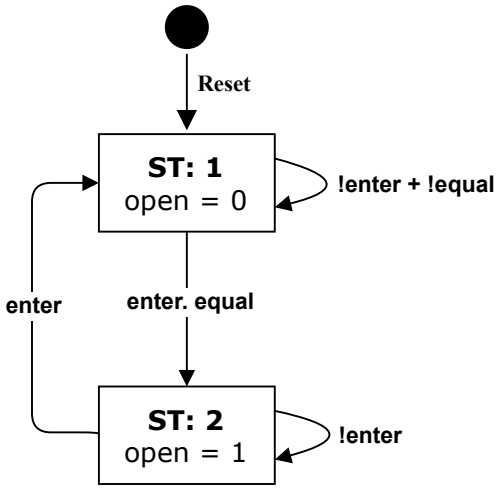


Figure 11.1: Moore FSM **OpenLock**

(state "2", where *open* = 1). The door stays open until the user presses **Enter** again, after which the system returns back to its locked state. Please see Section 13.2.1 to gain an insight into the functional design of this FSM.

Our FSM-TDES translation method requires that the next state conditions of the FSM must be specified either: 1) as *boolean expressions*, or 2) using one of our three reserved *keywords*.

1) Boolean Expressions

Currently, Hamid's (2014) TDES-FSM translation method, implemented in DESpot (2023), generates the next state conditions as boolean expressions in the *sum-of-products (SOP)* form (Brown and Vranesic, 2013). Therefore, we also require that all boolean expressions must be expressed in our input FSM only as SOP. This is not a hard requirement as any boolean function can be represented in both the SOP and *product-of-sums (POS)* form. Also, we can convert a boolean expression expressed in one form into the other, either manually, or using any automated tool or website. In future, it would be beneficial to incorporate the support for handling the next state conditions expressed as POS boolean expressions in our translation method.

It is notable that a boolean expression may be specified either in a *minimal* or *non-minimal* form. Presently, DESpot's TDES-FSM translation method generates boolean expressions only in a non-minimal form. However, the control designers might find it convenient to write compact boolean expressions in a minimal form while designing their individual FSM by hand. Keeping this in view, our FSM-TDES translation method is capable of handling both minimal and non-minimal boolean expressions. In the **OpenLock** FSM (Figure 11.1), that we have manually designed, we have specified all the next state conditions as minimal boolean expressions in the SOP form.

While expressing next state conditions as boolean expressions, each FSM input can take on the value of either: **i)** $True$ (1), **ii)** $False$ (0), or **iii)** *Don't Care* (d). In a boolean expression, if we want to specify the value of '1' for an input, we write the signal name in the uncomplemented form. For example, the next state condition of "$enter \cdot equal$" at state "1" of **OpenLock** means that this condition will be $True$ if both *enter* and *equal* are '1'. As per the SD supervisory control theory, this means that both *enter* and *equal* have occurred in the clock period that has just ended.

Writing an input name in the complemented form in a boolean expression specifies the value of '0' for the corresponding signal. For example, at state "1" of **OpenLock**, the boolean expression of "!*enter*" means that this next state condition will be $True$ if *enter* signal has the value of '0'. In other words, if *enter* did not occur in the previous clock period.

If any FSM input is not included in a boolean expression, this means we are treating this signal as a *Don't Care* (abbreviated as "DC," from now on). For example, the next state condition of "$enter \cdot equal$" means that *open* signal has

been specified as a DC and has the value of '*d*'. Basically, "$enter \cdot equal$" is the minimal form of two non-minimal boolean expressions, "$enter \cdot equal \cdot open$" and "$enter \cdot equal \cdot !open$". At state "1" of **OpenLock**, this next state condition means that FSM goes from state "1" to state "2" if both *enter* and *equal* are '1'. It does not matter whether *open* is '0' or '1', i.e. we *don't care* about the occurrence of *open*, hence *open* is a '*d*'.

2) Reserved Keywords

In order to enable the designers to conveniently specify some generic next state conditions in their individual FSM, we allow the use of the following three keywords as shorthand notations. As we are introducing these keywords for our FSM-TDES translation approach, we need to modify DESpot's existing TDES-FSM translation algorithms and add support for these keywords to make the two translation approaches compatible with each other. Please refer to Section C.2 in Appendix C to see our modified algorithms.

i) Tick: <TICK>

One of the possible next state conditions that needs to be specified at every state of an FSM is that all FSM input signals have the value of '0', i.e. none of the FSM inputs occurred in the previous clock period. For example, if an FSM has three inputs, $open, enter$ and $equal$, the boolean expression to represent this next state condition is "$!open \cdot !enter \cdot !equal$". It is obvious that as the signals of an FSM increase in number, this boolean expression becomes lengthier.

As this next state condition is generic for all individual FSM, i.e. '0' for all FSM inputs regardless of how many signals an FSM have, we introduce the keyword of <**TICK**> to represent it. We are using **TICK** as our keyword because this next state condition of the FSM corresponds to a concurrent string of the TDES supervisor that does not contain any activity events, i.e. a concurrent string that only contains a *tick* event.

ii) Global Don't Care: <GDC>

While defining the next state conditions of an FSM, designers may wish to specify an input combination at an FSM state where every signal of the FSM is a DC. We refer to this next state condition as the "*Global Don't Care (GDC)*" condition. The reason is that the FSM does not care what the value of each signal is. Rather, it always goes to the same next state, which could even be the current state of the FSM, depending upon its design logic. This means that at a given FSM state, all the specified input combinations have the very same destination state.

As discussed earlier, if we want to specify an FSM signal as a DC, we can simply exclude this signal from our boolean expression to make it compact. However, expressing GDC by following this strategy becomes complicated, as every signal of the FSM is a DC. To handle this situation, we introduce the keyword of <**GDC**>.

It is noteworthy that **GDC** covers all possible next state conditions that could be defined at an FSM state. This means that if we specify **GDC** at any given

state of the FSM, then it must be the *only* transition defined at this state and no other next state conditions could be specified at this FSM state. Otherwise, our FSM might become non-deterministic.

iii) Default: <DEF>

In the TDES-FSM translation approach, Wang (2009) introduces the shorthand notation of a *default transition*, abbreviated as **DEF**. This is because the transition function of a TDES supervisor is a partial function, whereas the next state function of a Moore FSM must be a total function. Wang added a selfloop of **DEF** transition at every state of the translated FSM in order to cover the next state conditions (input combinations) that are not explicitly specified in the TDES supervisor. In other words, Wang used a selfloop of **DEF** in the translated FSM to represent *invalid* transitions of the supervisor. By this, we mean transitions that the supervisor asserts cannot occur.

At any given state of the FSM, **DEF** is equivalent to taking the logical **OR** of all the next state conditions that are explicitly defined at this state, and then negating the result. It is notable that at any FSM state, if the explicitly specified next state conditions cover all possible input combinations, then **DEF** will be empty at this state.

While implementing Wang's TDES-FSM translation approach in DESpot (2023), Hamid (2014) interpreted the **DEF** transition in a completely different and contradictory way. In his translated FSM, Hamid uses **DEF** to represent all the input combinations that do not cause a state change, i.e. **DEF** represents all the next state conditions that are selflooped. Hamid did not take into account whether these next state conditions were explicitly defined in the supervisor or not, and treated both *valid* and *invalid* selflooped next state conditions in the same way.

In our FSM-TDES translation approach, we will use **DEF** transition with its original and correct meaning, as defined by Wang (2009). Please refer to Section C.2 to see our modified TDES-FSM translation algorithms of DESpot that generate the output FSM XML file with correct meaning of the **DEF** transition.

It is worth clarifying that we discourage the use of **DEF** while manually designing FSM because it is really easy to misinterpret and misuse the **DEF** transition and introduce design errors. For instance, since **DEF** is always defined as a selfloop, it looks very tempting just to define one **DEF** transition at every state of the FSM to cover all the next state conditions that do not cause an explicit state change (as Hamid (2014) did for his translated FSM). However, by doing so, designers make the *valid* selflooped next state conditions of their FSM *invalid* by merging them together and using **DEF** to represent all of them.

In this case, since the manually designed FSM does not specify the correct next state conditions that designers actually wanted to specify, it is obvious that the corresponding TDES supervisor generated by our FSM-TDES translation method will be incorrect. Precisely, valid transitions will be missing from the translated supervisor because valid selflooped next state conditions were merged with **DEF**

in the input FSM, and our translation method does not generate anything corresponding to **DEF**. Since the translated supervisor is logically incorrect, this in turn makes it more likely that it will fail the desired TDES and $||_{SD}$ properties (Chapter 4).

The only situation when designers must write a **DEF** transition in the input FSM is if no valid next state conditions exist at an FSM state. In this case, designers must specify a selfloop of **DEF** transition in order to fulfill the requirement of making the FSM's next state function a total function.

11.1.2 Central FSM

Our FSM-TDES translation approach requires the designers to specify one central FSM file corresponding to the control system whose one or more individual FSM need to be translated into TDES supervisor(s). We use the same XML file format for specifying our central FSM that Hamid (2014) has defined while implementing his TDES-FSM translation method in DESpot (2023). As a result, the corresponding DESpot algorithm to generate the central FSM XML file remains unmodified. Please refer to Section C.1.2 in Appendix C to see a discussion on the central FSM XML file of our Combination Lock example.

While implementing the TDES-FSM translation method in DESpot, Hamid allows an event to belong to the project that may or may not belong to the event set of any TDES supervisor. This event is not under the control of any supervisor and is assumed to be always allowed by the supervisor model. Consequently, in the translated FSM, its corresponding signal appears in the global list of signals in the central FSM, but does not belong to any individual FSM. If this is an output signal, this means its value will always be set to $True$ (1).

For our FSM-TDES translation approach, this assumption does not cause any issues with respect to input signals (uncontrollable events). However, it is not suitable for our translation approach with respect to output signals (prohibitable events). The reason being, this allowance might cause our translated supervisors to fail the desired $||_{SD}$ properties, especially the property of plant completeness with $||_{SD}$ (Definition 4.4.1) and **S**-singular prohibitable behaviour with $||_{SD}$ (Definition 4.4.2).

In order to make it more likely that our translated supervisors satisfy the desired $||_{SD}$ properties, we require that every output signal included in the global list of signals in the central FSM must belong to at least one individual FSM. Please refer to Section C.2 to see how we incorporate this constraint in our DESpot's TDES-FSM translation algorithm.

11.2 FSM-TDES Translation Prerequisites

In this section, we list down some consistency and design requirements that must be satisfied by the Moore FSM designers when specifying the central and individual

FSM for translation. Our FSM-TDES translation method (described in Section 11.3) and its corresponding algorithms (presented in Chapter 12) are specifically designed to be implemented in DESpot (2023). For this reason, we have also included some consistency requirements with respect to the central FSM and the DESpot project. This DESpot project is the one within which our FSM-TDES translation process is being performed. We assume that the current DESpot project already contains the TDES plant models of the physical system for which the controllers have been designed as Moore FSM.

It is worth clarifying that although software designers can use our translation method to automatically generate TDES supervisors from Moore FSM, the TDES plant models for the given control system must be available to perform the desired system verification checks, and might need to be designed manually. However, it is generally believed that designing plant models is much more straightforward and less challenging than designing supervisors. This is because the plant TDES simply represents the existing uncontrolled system behaviour as it is. On the other hand, in order to design the supervisor, control designers need to create the desired specifications from scratch. Then, they need to model these specifications in such a way that the resultant supervisor model should work correctly with the plant, and should be able to control the physical system as expected while satisfying all the desired $||_{SD}$ properties.

Moreover, even if the plant models are designed manually, they need not be modelled by the same designers who developed the Moore FSM. It is quite possible that there is only one person in the design team who is proficient in formal methods, hence they could model the plant TDES. The rest of the team members, who do not have expertise in formal methods, could simply design the Moore FSM.

It is also possible that the plant models could potentially be developed by an external formal methods expert, or might even be provided by the control system manufacturer. In either case, we anticipate that our FSM-TDES translation approach should facilitate software designers, with limited or no knowledge of formal methods, in the formal representation and verification of their control systems, just by expressing their controllers in a way that they are familiar with, i.e. as Moore FSM.

Please note that we are stating and checking only those requirements that are vital to make our translation method work. Other DESpot specific checks, like using only valid characters that DESpot allows to name the FSM, states and signals, or following certain naming conventions, can easily be incorporated while implementing our algorithms in DESpot (which is left as future work due to time constraints). For this reason, our translation method and algorithms do not focus on these DESpot specific implementation checks.

11.2.1 Consistency Requirements

One of the prerequisites for our FSM-TDES translation method is that the information specified in the central FSM, all individual FSM and the current DESpot project must be consistent. Specifically, we impose the following consistency checks on the input FSM XML files and the current DESpot project. We are labelling these consistency requirements as **CR-1**, **CR-2**, ... , **CR-8** and will use these labels to refer to them in our later sections and chapters.

Central FSM and DESpot Project

CR-1: The name of the central FSM must be same as the name of the DESpot project within which the FSM-TDES translation process is taking place. We impose this check to make sure that the translation process is initiated within the correct DESpot project.

CR-2: The number of global output signals listed in the central FSM must be equal to the number of prohibitable events in the DESpot project. Also, the names of all global output signals and prohibitable events must match.

CR-3: The number of global input signals listed in the central FSM must be equal to the number of uncontrollable events in the DESpot project. Also, the names of all global input signals and uncontrollable events must match.

Requirements **CR-2** and **CR-3** ensure that TDES plant models present in the current DESpot project match with the input Moore FSM that the designers wish to translate.

CR-4: Every global output signal listed in the central FSM must be present as an input-output (IO) signal in at least one individual FSM specified in the central FSM, and vice versa.

CR-5: In the central FSM, every input signal of an individual FSM must belong to the list of global input signals.

Requirements **CR-4** and **CR-5** are required by our translation method. They will be helpful in generating the TDES supervisors that, when synchronized with the TDES plant models, are more likely to satisfy the desired $||_{SD}$ properties.

Central FSM and Individual FSM

CR-6: The number of Moore FSM listed in the central FSM must be equal to the number of FSM specified individually. Also, the FSM names specified in the central and individual FSM XML files must match.

CR-7: For every Moore FSM, the list of local IO signals specified in the central and individual FSM must be the same.

CR-8: For every Moore FSM, the list of local input signals specified in the central and individual FSM must be the same.

11.2.2 Design Requirements

Every individual FSM, that the designers wish to translate to a supervisor, must satisfy the following design requirements. We will use the labels of these design requirements (**DR-1**, **DR-2**, ... , **DR-13**) to refer to them in our discussion later on.

DR-1: Every individual FSM must have a unique name.

As we will use the FSM name as the name of the translated supervisor, it is advisable to use only those characters and letters in the FSM name that are allowed in the supervisor's name by DESpot. Otherwise, DESpot will reject the translated supervisor. Please refer to the help documentation of DESpot (2023) for further details on valid/invalid characters and letters. Also, our reserved keywords (introduced in Section 11.1.1), and special characters and delimiters used in XML input files (e.g. '.', '!', '=', '<', '>', etc.) must not be used in the FSM name.

Please note that this requirement also applies to the names of the FSM states, as they will be used as the state names in the translated supervisor.

DR-2: The name of every IO and input signal must be unique.

DR-3: For every individual FSM, the list of states must not be empty. Also, at every state of the FSM, its corresponding output information (represented as *outputvector* in the XML file (Section C.1.1)) must be specified.

DR-4: At every state of the FSM, the signals included in the output information must be listed as IO signals in the individual FSM's list of signals.

DR-5: At every state of the FSM, at least one transition must be defined to specify the next state logic (NSL). If no valid next state conditions exist at a state, a **DEF** transition must be specified to make the FSM's next state function a total function.

DR-6: At every state of the FSM, all possible next state conditions must be specified to make the FSM's next state function a total function. If all next state conditions are not covered by the explicitly specified transitions, then a **DEF** transition must be included to satisfy this requirement.

DR-7: The **DEF** transition must always be defined as a selfloop transition.

DR-8: Every transition must specify valid next state condition(s), either as a boolean expression or using one of our reserved keywords. The transition's next state conditions (represented by *inputvector* in the XML file (Section C.1.1)) must not be left empty.

DR-9: For next state conditions that are expressed as boolean expressions, the signals included in every boolean expression must be listed in the FSM's list of signals.

DR-10: At any given state of the FSM, if **GDC** transition is specified, then it must be the only transition specified at this state. No other valid next state conditions could be specified at this state.

DR-11: At any given state of the FSM, if the output of an IO signal is set to $True$ (1), then this signal must show up, either as occurring (1) or as a DC (d), in at least one of the "valid" next state conditions represented by the transitions specified at this state. However, if an IO signal is not occurring (0) in any valid transition, then the output of this signal must be set to $False$ (0) at this state.

DR-12: The end state of every specified transition must be listed in the FSM's list of states.

DR-13: At every state of the FSM, all the specified NSL must be deterministic, i.e. there must be only one destination state for any given next state condition.

11.3 FSM-TDES Translation Method

Now we are ready to present our FSM-TDES translation method in detail. In order to perform the translation, we assume that the user initiates the FSM-TDES translation process in DESpot (2023) by providing one central FSM and one or more individual Moore FSM XML files as an input with respect to the current DESpot project. Of these files, we use only the individual Moore FSM files to perform the actual translation process, and generate one TDES supervisor corresponding to each Moore FSM. However, both central and individual FSM XML files are used to verify the desired consistency and design requirements.

Our translation method begins by examining the central FSM file, individual FSM files and the DESpot project for verifying requirements **CR-1−8** and **DR-1−2**, listed in Section 11.2. If any of these requirements is not satisfied, our translation process terminates immediately by generating an appropriate error message for the user.

Otherwise, our translation method proceeds to analyze the individual Moore FSM files in more detail. Specifically, we perform the remaining design checks, **DR-3−12**, on each FSM file. Again, the translation process terminates if any of these checks fails. However, if all of these checks pass, then we process each FSM's information to convert it into a form that we can directly use to perform the actual translation.

Once an individual Moore FSM has been successfully processed, we perform the actual translation process and convert this FSM to its corresponding TDES supervisor representation. We do this by creating and populating the supervisor's quintuple, $\mathbf{S}_i = (X_i, \Sigma_i, \xi_i, x_{o,i}, X_{m,i})$, where $0 \leq i < n$, and n is the total number of individual Moore FSM that are provided as input for the translation. Here, X_i is the *state set*, Σ_i is the *event set*, $\xi_i : X_i \times \Sigma_i \rightarrow X_i$ is the *partial transition function*, $x_{o,i}$ is the *initial state*, and $X_{m,i}$ is the *set of marked states* for the i^{th} TDES supervisor that is translated from the i^{th} Moore FSM.

It is notable that DESpot requires every supervisor to have a unique name within

the project. Therefore, our translation method uses the name of each individual FSM as the name of the corresponding translated supervisor.

After generating all supervisors, we assume that these translated supervisor models get added to the current DESpot project. This will enable the designers to synchronize these translated supervisors with the TDES plant models using our $||_{SD}$ operator. The designers can then verify the desired properties of their closed-loop system by running the $||_{SD}$ algorithms (discussed in Chapter 9).

We would like to mention here that one of our major goals while devising our FSM-TDES translation approach is that our translation method should generate supervisors that are more likely to satisfy the required $||_{SD}$ properties. This point should become more clear as we will introduce our translation rules and unfold our translation method step by step in the following sections.

Below, we describe the FSM-TDES translation method by explaining how our translation method uses the information specified in the XML file of the individual Moore FSM to populate each element of the supervisor's quintuple. We will use the **OpenLock** FSM (Figure 11.1, XML Input File C.1 from Appendix C) of the 4-bit Combination Lock as an example. For simplicity, we assume that **OpenLock** is the first FSM of the system that is being translated, i.e. $i = 1$, and our translation method populates the quintuple of $\mathbf{S}_1 = (X_1, \Sigma_1, \xi_1, x_{o,1}, X_{m,1})$ for the **OpenLock** TDES supervisor.

Please recall from Chapter 3 that in the SD supervisory control theory (Wang, 2009; Leduc *et al.*, 2014), a Moore synchronous FSM is used to model an SD controller. Therefore, we will use several concepts related to SD controllers while describing our translation method. Please refer to Chapter 3 to refresh your memory about these concepts.

11.3.1 Create State Set

In order to create a state set for each translated supervisor, we start by analyzing the behaviour of an SD controller. In the SD supervisory control theory, an SD controller changes state only on the clock edge, which is equivalent to the occurrence of a *tick* event in the TDES theory. This essentially means that the states of a Moore FSM correspond to the sampled states of a TDES supervisor.

Keeping this in view, our translation method adds the states of each FSM to the state set of its corresponding translated supervisor. For example, the **OpenLock** Moore FSM (Figure 11.1) has two states, "1" and "2". Thus, our translation method adds both of these states to the state set of the translated supervisor **OpenLock**, i.e. $X_{samp,1} = \{1, 2\} \subseteq X_1$.

Next, we note that unlike an SD controller, a supervisor knows immediately when an event occurs, and then it updates its enablement and forcing information right away, potentially by changing its state. This indicates that, in most of the cases, a supervisor will probably have more states than its corresponding FSM representation.

This implies that in addition to sampled states, our translation method also needs to add one or more *non-sampled* or *intermediate states* to the translated supervisor. These intermediate states are needed to represent the transitions of individual events (*tick* and activity events) that make up the concurrent strings which are defined at each sampled state.

In our translation method, we add intermediate states to the translated supervisor related to each state of the FSM. The number of intermediate states that we add depends upon the output information and "valid" next state conditions that are defined at each FSM state. Precisely, for a given FSM state q (that does not have a **GDC** transition), we can use the formula given below to calculate the number of intermediate states that our translation method adds to the translated supervisor. The reason our formula has $2^n - 1$ and not $2^n - 2$ (i.e. subtracting for the start and end states that are sampled states) is to account for the *tick* transition to terminate the concurrent string.

> Number of intermediate states $= 2^n - 1$,
> where n = number of IO signals whose outputs are set to $True$ (1) at state q + number of unique input signals that show up either as '1' or '*d*' in at least one of the valid next state conditions specified at q.

However, if a **GDC** transition is specified at q, then we can use the following formula to calculate the number of intermediate states that our translation method adds to the state set of the translated supervisor:

> Number of intermediate states for <**GDC**> $= 2^n - 1$,
> where n = number of IO signals whose outputs are set to $True$ (1) at state q.

In the case of a **GDC** transition, we are not considering the input signals of the FSM. The reason being, by our **GDC** definition (Section 11.1.1), all input signals have the value of '*d*' in the **GDC** next state condition. We will further elaborate this point in Section 11.3.5 while discussing the construction of transition function for the translated supervisor.

Please note that in the above-mentioned formulas, we are considering the IO and input signals that belong to the individual Moore FSM that we are currently translating, rather than the global list of output and input signals.

The **OpenLock** FSM, shown in Figure 11.1, has three signals: one IO signal (*open*) and two input signals (*enter* and *equal*). At state "1" of the FSM, the output of *open* is set to $False$ (0). For the input signals, we note that both of these signals appear in the uncomplemented form in the boolean expression "$enter \cdot equal$", i.e. they both have the value of '1' in this next state condition. This means the number of intermediate states that our translation method will add to the translated supervisor corresponding to state "1" of the FSM is:

$$2^{0+2} - 1 = 2^2 - 1 = 4 - 1 = 3$$

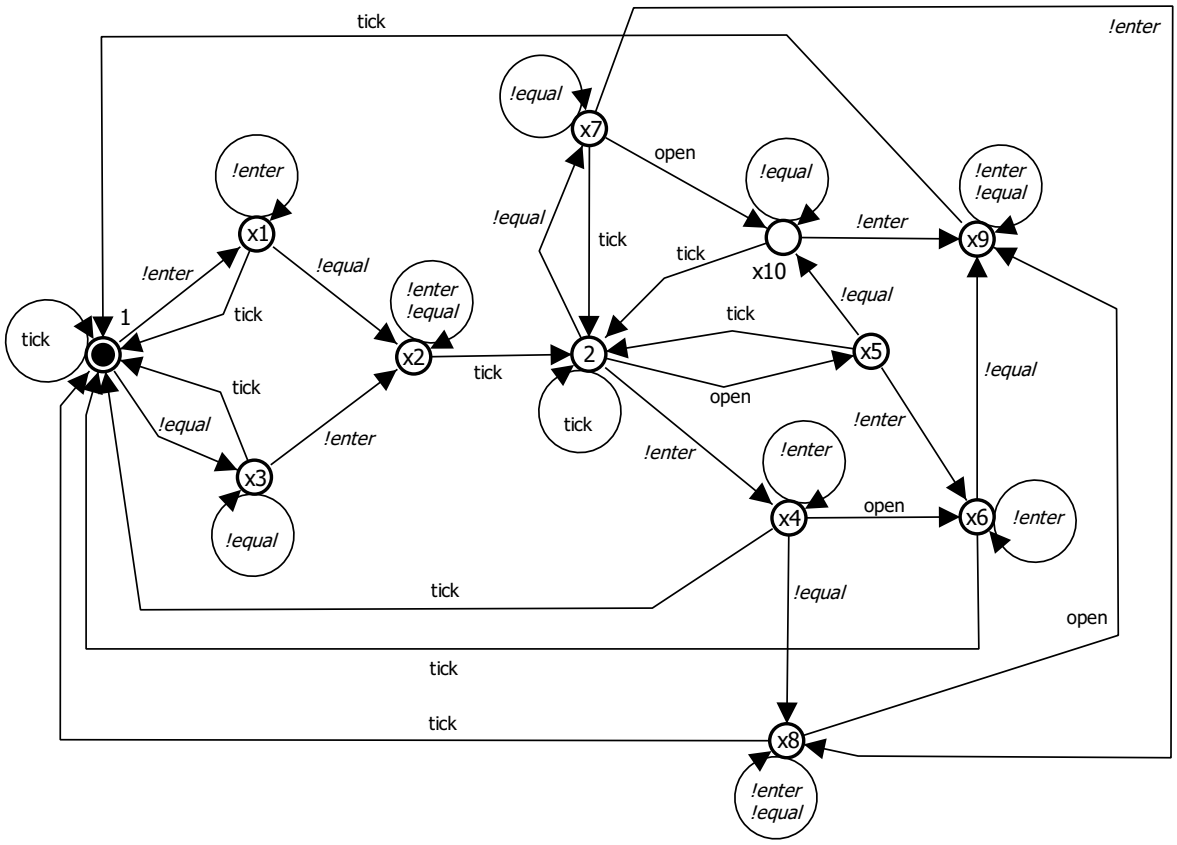


Figure 11.2: Translated TDES Supervisor **OpenLock**

Likewise, we can calculate the number of intermediate states that will be added to the state set of the translated supervisor corresponding to state "2" of **OpenLock**. At state "2", the output of IO signal *open* is set to $True$ (1). In the "*enter*" next state condition specified at state "2", *enter* has the value of '1', whereas *equal* is a '*d*'. Thus, the number of intermediate states is:

$$2^{1+2} - 1 = 2^3 - 1 = 8 - 1 = 7$$

This means that our translation method adds $3+7 = 10$ intermediate states to the translated supervisor related to the two states of the **OpenLock** FSM. Please recall that we also have two sampled states in the event set of the translated supervisor. Hence, in total, our translation method creates a state set of $10 + 2 = 12$ states for the translated supervisor **OpenLock**.

Figure 11.2 shows the complete TDES supervisor **OpenLock** that we have generated corresponding to the **OpenLock** FSM by applying our complete FSM-TDES translation method. We will only focus on the states of this translated supervisor for now. Please recall from Section 2.3 that in the graphical TDES models, an event name given in italics and preceded by "!" indicates an uncontrollable event, a double circle represents the initial state, and a filled circle shows that the state is marked.

Also, please note that "!" symbol used in a TDES model has a different meaning than "!" of an FSM, where it represents a **NOT** operator in the latter.

The translated supervisor has two sampled states ("1" and "2") that correspond to the two states of the **OpenLock** FSM. The concurrent strings defined at sampled state "1" use three intermediate states (x1-x3) to represent transitions of the individual events. Similarly, the concurrent strings defined at sampled state "2" make use of seven intermediate states (x4-x10) to define the individual event transitions. Overall, the state set of the translated supervisor that our translation method has created is $X_1 = \{1, 2, \text{x1}, \text{x2}, \ldots, \text{x10}\}$. For simplicity, we are ignoring the issue that state labels x1, x2, ... , x10 might have already existed in the input FSM. In such a situation, we would have to modify the names of intermediate states to make them unique.

It is worth noting that for the sampled states of the translated supervisor **OpenLock**, we have used the same name as the FSM state names. For naming the intermediate states, we use characters and letters that are allowed by DESpot as part of the state name. Please refer to Algorithm 12.8, discussed in Section 12.2.3, to see how our translation method names the non-sampled states that get added to the translated supervisor.

11.3.2 Populate Event Set

Our translation method uses the list of signals specified in each individual FSM to create and populate the event set of the corresponding translated supervisor. Precisely, the IO signals of the FSM become the prohibitable events of the supervisor, whereas input signals become the uncontrollable events.

Besides, we also add a *tick* event to the event set of each translated supervisor. This is because the clock edge, that an SD controller uses for sampling its inputs, changing its state, and updating its outputs, is associated with the occurrence of a *tick* event in the SD theory.

The **OpenLock** FSM (Figure 11.1) has one IO signal (*open*) and two input signals (*enter, equal*). This means the event set of the translated supervisor **OpenLock** (Figure 11.2) is $\Sigma_1 = \{open, enter, equal, tick\}$, where $\Sigma_{hib,1} = \{open\}$ and $\Sigma_{u,1} = \{enter, equal\}$.

11.3.3 Assign Initial State

In our translation method, the reset/initial state of each individual FSM becomes the initial state of the corresponding supervisor. The reset state of the **OpenLock** FSM (Figure 11.1) is state "1". This implies the initial state of the translated supervisor **OpenLock** (Figure 11.2) is $x_{o,1} = 1$.

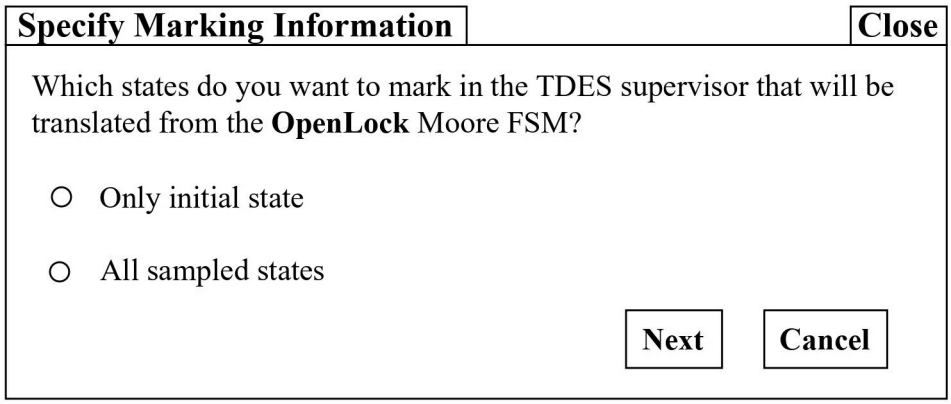


Figure 11.3: Dialog Box for Taking Input About Set of Marked States

11.3.4 Generate Set of Marked States

It is interesting to note that, unlike TDES theory, there is no concept of *marked behaviour* (Definition 2.2.7) in the SD controllers and Moore FSM. However, in order to make it possible to satisfy the property of nonblocking (Definition 2.2.8), one or more states of each translated supervisor must be marked.

In order to do that, when designers initiate the FSM-TDES translation process in DESpot (2023), we ask them to provide this marking information separately. We assume that while reading each input Moore FSM XML file, DESpot displays a dialog box to the designers asking them to specify the set of states they want to mark in each translated supervisor.

The dialog box for taking the marking information for **OpenLock** FSM is shown in Figure 11.3. The dialog box gives two options to the designers to choose from: 1) mark the initial state, or 2) mark all sampled states. We intend to take this marking information corresponding to each Moore FSM. This will enable the designers to select different options for each translated supervisor. This information should then be stored in the appropriate data structures that are populated by DESpot after reading each input Moore FSM XML file (see Section 12.2 for details).

Please note that we are providing only two marking options to the designers to keep things simple. Once the translation process is complete, the designers always have the option of opening the translated supervisors in the DESpot editor and customize marking information (mark/unmark any state) as they want.

It is worth clarifying that, hypothetically, we could have provided a third option of "mark all states" to the designers. However, we excluded this option intentionally because we strive to develop a translation method that should be able to generate supervisors with increased likelihood of satisfying the desired $||_{SD}$ properties. This is to satisfy Point iii of the SD controllability with $||_{SD}$ definition (Definition 4.5.1) which requires that all marked strings in the closed-loop system must be sampled strings. Otherwise, our supervisor will not be SD controllable with $||_{SD}$. This implies that only sampled states should be marked at a minimum.

For the **OpenLock** FSM, let us assume that designers choose to mark only the initial state. Therefore, the set of marked states of the **OpenLock** supervisor (Figure 11.2) generated by our translation method is $X_{m,1} = \{1\}$.

11.3.5 Construct Transition Function

In order to construct a transition function for each translated supervisor, our translation method uses the NSL (next state conditions and end states) that are specified at each state of the input FSM. Specifically, we make use of the next state conditions to generate occurrence images of the concurrent strings that need to be defined at the sampled states of the translated supervisor. Then, we use these occurrence images to define transitions for individual events at the sampled and non-sampled states of the supervisor.

We have designed our translation method in such a way that it first generates a supervisor only with respect to explicit state changing transitions. Once we have defined all the state changing transitions and added them to the transition function, then we add selfloops of uncontrollable events and/or *tick* event at the appropriate states of the translated supervisor.

By using **OpenLock** FSM (Figure 11.1) as an example, below we describe six steps for generating occurrence images from the given next state conditions and constructing the translated supervisor's transition function. While explaining these steps, we will also discuss how our FSM-TDES translation rules construct supervisors in such a way that increase the probability of the generated supervisor models to satisfy the desired $||_{SD}$ properties.

Step 1: Generate Hybrid Next State Logic

In order to generate occurrence images from the next state conditions, we start by generating *hybrid next state logic* at each state of a given Moore FSM. We do this by converting each next state condition into a *hybrid vector*. A hybrid vector h is similar to a boolean vector (defined in Section 3.6), except that in addition to the boolean values of '0' ($False$) and '1' ($True$), it can also store the value of 'd' which is a shorthand notation for a "*Don't Care*" condition.

Let H be the set of possible hybrid vectors that the inputs of the FSM can take on. For an individual FSM, each hybrid vector $h \in H$ is of size s, i.e. each hybrid vector has s elements, where s is the number of IO and input signals of the FSM. Every element of h corresponds to a unique FSM signal such that:

$$h = [h[0], h[1], \ldots, h[s-1]], \quad h[j] \in \{0, 1, d\}, \text{ where } j = 0, 1, \ldots, s-1$$

For example, the **OpenLock** FSM (Figure 11.1) has three signals, *open*, *enter* and *equal*. Therefore, the hybrid vectors that we generate corresponding to the next state conditions of **OpenLock** has three elements, i.e. $s = 3$. For simplicity, we use the *order* of the FSM signal specified in the Moore FSM XML file as the index

Table 11.1: Hybrid Next State Logic for **OpenLock** Moore FSM

Row No.	**Start State**	**Hybrid Vectors**			**End State**
		open	***enter***	***equal***	
R-1	**1**	*d*	*1*	*1*	**2**
R-2	**1**	*d*	*0*	*d*	**1**
R-3	**1**	*d*	*d*	*0*	**1**
R-4	**2**	*d*	*1*	*d*	**1**
R-5	**2**	*d*	*0*	*d*	**2**

of each FSM signal in the hybrid vector. This implies that in the hybrid vectors of **OpenLock** (XML Input File C.1 in Appendix C), the IO signal *open* is at index 0, and input signals *enter* and *equal* are at indexes 1 and 2 respectively.

As stated in Section 11.1.1, the next state conditions of a Moore FSM can be specified either using boolean expressions or one of our three reserved keywords. Now we will describe how our translation method generates hybrid vectors corresponding to each of these four next state conditions.

1) Boolean Expressions

Our translation method generates one hybrid vector corresponding to each boolean expression that represents the next state condition(s) at a given state of the FSM. We use the interpretation of boolean expressions, given in Section 11.1.1, to assign values to each individual element of the hybrid vectors.

In a given boolean expression, if the name of an FSM signal appears in the uncomplemented form, we assign the value of '1' to that signal in the hybrid vector. However, if a signal appears in the complemented form, then hybrid vector stores the value of '0' at the corresponding index. If a signal name does not appear in the boolean expression, this means the signal has been treated as a DC. In this case, we add '*d*' to the appropriate element of the hybrid vector.

Table 11.1 shows the hybrid NSL that our translation method generates corresponding to the **OpenLock** FSM (Figure 11.1). At state "1" of **OpenLock**, the next state conditions are specified using three boolean expressions. That is why we have three hybrid vectors that have state "1" as the *start state* in the table.

In Table 11.1, the row *R-1* represents the hybrid NSL that we have generated corresponding to the first boolean expression "$enter \cdot equal$". Since both *enter* and *equal* appear in the boolean expression in the uncomplemented form, we have stored the value of '1' for these two signals in the generated hybrid vector. The IO signal *open* has the value of '*d*', since it is not present in this boolean expression.

The row *R-2* represents the hybrid NSL corresponding to the second boolean expression, "*!enter*". Since *enter* shows up in the complemented form, that is why we have assigned the value of '0' to the *enter* signal in the hybrid vector. The other two signals, *open* and *equal*, are DC ('*d*'). The same logic applies to the third boolean expression, "*!equal*", specified at state "1" of **OpenLock**. Its corresponding hybrid NSL is shown in the row *R-3* of Table 11.1.

After generating hybrid NSL corresponding to state "1", our translation method moves on to state "2" of the **OpenLock** FSM. Using the same logic discussed above, we generate hybrid vectors for the boolean expressions that are specified at state "2". These hybrid NSL are given in rows *R-4* and *R-5* of Table 11.1.

2) <**TICK**>
By the definition of **TICK** (Section 11.1.1), it is the shorthand notation for a boolean expression in which every IO and input signal of an FSM appears in the complemented form. Therefore, the hybrid vector that our translation method generates corresponding to the **TICK** next state condition will have the value of '0' for all FSM signals.

3) <**GDC**>
As **GDC** must be the only valid transition specified at an FSM state, we generate one hybrid vector corresponding to **GDC**. By definition, the **GDC** transition represents a next state condition when every signal at the given state of an FSM is a DC. We treat the IO signals of the **GDC** transition in a standard way, and assign the value of '*d*' to each element of the hybrid vector that corresponds to an IO signal.

However, we plan to handle the input signals of **GDC** in a slightly different way while generating transitions of the translated supervisor. This is because we wish to translate the **GDC** transition of the FSM into corresponding TDES transitions in an efficient way, and have a compact TDES representation for the FSM's **GDC** transition. Specifically, instead of generating the state changing transitions, we want our translation method to add only selfloop transitions to the supervisor corresponding to the input signals of **GDC**. This point will become more clear at **Step 6**.

In order to do that, instead of assigning the value of '*d*', we assign the value of '0' to each element of the hybrid vector that corresponds to the input signal of the FSM. Also, we associate a flag with each state of the FSM and set this flag to *True* where a **GDC** transition is defined. At **Step 6**, we will use this flag to identify the presence of a **GDC** transition at the FSM state(s). We will then use a different translation strategy to add TDES transitions corresponding to these input signals of the **GDC** than for the input signals of the rest of the next state conditions.

4) <**DEF**>
The **DEF** transition is a shorthand notation to cover all *invalid* next state conditions that cannot occur at a given state in the physical system. Since the transition function of a TDES supervisor is a partial function, we do not need to generate any transitions in the translated supervisor corresponding to the invalid next state conditions of the FSM (see Section 11.1.1 for the related discussion). Therefore, our translation method simply ignores the **DEF** transition, and does not generate any hybrid vector corresponding to **DEF**.

Table 11.2: Boolean Next State Logic for **OpenLock** Moore FSM

Row No.	**Start State**	**Boolean Vectors**			**End State**
		open	***enter***	***equal***	
R-1.1	**1**	*0*	*1*	*1*	**2**
R-1.2	**1**	*1*	*1*	*1*	**2**
R-2.1	**1**	*0*	*0*	*0*	**1**
R-2.2	**1**	*0*	*0*	*1*	**1**
R-2.3	**1**	*1*	*0*	*0*	**1**
R-2.4	**1**	*1*	*0*	*1*	**1**
R-3.1	**1**	*0*	*0*	*0*	**1**
R-3.2	**1**	*0*	*1*	*0*	**1**
R-3.3	**1**	*1*	*0*	*0*	**1**
R-3.4	**1**	*1*	*1*	*0*	**1**
R-4.1	**2**	*0*	*1*	*0*	**1**
R-4.2	**2**	*0*	*1*	*1*	**1**
R-4.3	**2**	*1*	*1*	*0*	**1**
R-4.4	**2**	*1*	*1*	*1*	**1**
R-5.1	**2**	*0*	*0*	*0*	**2**
R-5.2	**2**	*0*	*0*	*1*	**2**
R-5.3	**2**	*1*	*0*	*0*	**2**
R-5.4	**2**	*1*	*0*	*1*	**2**

Step 2: Generate Boolean Next State Logic

The next step of our translation method is to convert the hybrid NSL into *boolean NSL*. In other words, we translate each hybrid vector into its equivalent *boolean vector* representation. Please recall that in Wang's (2009) boolean vectors (see Section 3.6), every individual element can be assigned the value of either '0' ($False$) or '1' ($True$). This implies that we need to process the DC value that is represented by '*d*' in our hybrid vectors, and express it in terms of '0' and '1' only.

To do this, we split each hybrid vector with a '*d*' into two or more boolean vectors. The number of boolean vectors that we generate corresponding to each hybrid vector depends upon the number of '*d*' values that we have in one hybrid vector. Precisely, if a hybrid vector has n elements that have the value of '*d*', then our translation method generates 2^n boolean vectors corresponding to this hybrid vector.

For the **OpenLock** FSM (Figure 11.1), Table 11.2 shows the boolean NSL that are generated by our translation method. For the hybrid NSL given in *R-1* of Table 11.1, we generate two boolean NSL, as shown in rows *R-1.1* and *R-1.2* of Table 11.2. This is because there is only one element, *open*, that has the value of '*d*' in the hybrid vector, hence $2^1 = 2$ boolean vectors.

In the generated boolean vectors, *open* takes on the value of '0' in one boolean vector (*R-1.1*), and '1' in the other boolean vector (*R-1.2*). The values for the other

two signals, *enter* and *equal*, as well as the end state remain unchanged in the two boolean NSL.

The hybrid NSL shown in *R-2* of Table 11.1 has the value of '*d*' for two elements. Its corresponding $2^2 = 4$ boolean NSL are given in rows *R-2.1*−*R-2.4* of Table 11.2. For the two '*d*' elements of *open* and *equal* in the hybrid vector, we generate all possible boolean combinations of 00 (*R-2.1*), 01 (*R-2.2*), 10 (*R-2.3*) and 11 (*R-2.4*) in the boolean vectors. The value of *enter* remains unmodified. Other hybrid NSL of Table 11.1 are converted into boolean NSL by applying the same technique, as shown in Table 11.2.

Step 3: Identify Nondeterministic Next State Logic

At this point, we note that the process of converting hybrid NSL into boolean NSL might result in the generation of duplicate boolean vectors that have the same start state. This is evident in Table 11.2, as rows *R-2.1* and *R-3.1* represent the same boolean vector of 000 at state "1" of the **OpenLock** FSM. Likewise, *R-2.3* and *R-3.3* specify the identical boolean vector of 100 at state "1".

The reason for getting these duplicate boolean vectors is that starting at state "1" of **OpenLock**, the next state conditions of "*!enter*" and "*!equal*" represented by the hybrid vectors of rows *R-2* and *R-3* respectively in Table 11.1 are *overlapping*, i.e. at state "1", both of these hybrid vectors cover the next sate conditions of "$!open \cdot !enter \cdot !equal$" (000) and "$open \cdot !enter \cdot !equal$" (100). However, this overlapping of the next state conditions is not really obvious in the hybrid NSL, and becomes crystal clear in the boolean NSL. This kind of overlapping is quite common when designers prefer to write boolean expressions in a simplified form.

Taking this into consideration, after generating the boolean NSL, our translation method looks for duplicate boolean vectors that have the same start state. If we are able to find such boolean vectors, then we need to examine their end states. This is because if two or more identical boolean vectors starting at the same state have different end states, this implies that designers have specified *nondeterministic* NSL in the input FSM. As our translation method only focuses on deterministic Moore FSM (**DR-13**), this will be considered as a discrepancy and our translation method will terminate immediately by generating an appropriate error message for the designers.

On the other hand, if two or more identical boolean vectors have the same start states as well as the same end states, this means that we have multiple instances of this NSL in the input FSM. In this case, our translation method retains only one instance of this boolean NSL, and removes all the duplicate instances. We are able to do this because there is no need to process the same NSL multiple times while generating transitions of the translated supervisor.

By looking at Table 11.2, we note that the boolean NSL specified at *R-2.1* and *R-3.1* are identical, i.e. not only their start states and boolean vectors, but their end states are also same. Therefore, our translation method removes the second instance

Table 11.3: Unique Boolean Next State Logic for **OpenLock** Moore FSM

Row No.	Start State	Boolean Vectors			End State
		open	*enter*	*equal*	
R-1.1	**1**	*0*	*1*	*1*	**2**
R-1.2	**1**	*1*	*1*	*1*	**2**
R-2.1	**1**	*0*	*0*	*0*	**1**
R-2.2	**1**	*0*	*0*	*1*	**1**
R-2.3	**1**	*1*	*0*	*0*	**1**
R-2.4	**1**	*1*	*0*	*1*	**1**
R-3.2	**1**	*0*	*1*	*0*	**1**
R-3.4	**1**	*1*	*1*	*0*	**1**
R-4.1	**2**	*0*	*1*	*0*	**1**
R-4.2	**2**	*0*	*1*	*1*	**1**
R-4.3	**2**	*1*	*1*	*0*	**1**
R-4.4	**2**	*1*	*1*	*1*	**1**
R-5.1	**2**	*0*	*0*	*0*	**2**
R-5.2	**2**	*0*	*0*	*1*	**2**
R-5.3	**2**	*1*	*0*	*0*	**2**
R-5.4	**2**	*1*	*0*	*1*	**2**

of this boolean NSL by eliminating the row *R-3.1*. Similarly, since the boolean NSL represented by *R-2.3* and *R-3.3* are also identical, we delete *R-3.3* from the set of boolean NSL for the **OpenLock** FSM. Table 11.3 shows the boolean NSL after removing duplicate instances.

Step 4: Remove Invalid Next State Logic

While designing the individual Moore FSM to provide as an input to our translation method, designers need to specify the output information and NSL at every state of the FSM. Here, it is worth remembering that an IO signal occurs in a composite SD controller only when its global output is set to $True$ (1). The global output for an IO signal is determined by ANDing the local outputs of all the individual SD controllers that have this signal in their list of signals. This implies that the global output of an IO signal can never be '1' at a state if an individual SD controller sets its local output for this IO signal to '0' ($False$).

Since the output information of an SD controller remains constant for the entire clock period, this implies that an IO signal with global output of '0' cannot occur until the next clock edge. This in turn means that all the next state conditions that have the value of '1' for this IO signal become *invalid*, as they cannot occur at the current state of the FSM.

Since the transition function of the TDES supervisor is a partial function, it seems logical and reasonable that our translation method does not generate any concurrent string transitions at the given sampled state in the supervisor corresponding to these

Table 11.4: Valid Boolean Next State Logic for **OpenLock** Moore FSM

Row No.	Start State	Boolean Vectors			End State
		open	*enter*	*equal*	
R-1.1	**1**	*0*	*1*	*1*	**2**
R-2.1	**1**	*0*	*0*	*0*	**1**
R-2.2	**1**	*0*	*0*	*1*	**1**
R-3.2	**1**	*0*	*1*	*0*	**1**
R-4.1	**2**	*0*	*1*	*0*	**1**
R-4.2	**2**	*0*	*1*	*1*	**1**
R-4.3	**2**	*1*	*1*	*0*	**1**
R-4.4	**2**	*1*	*1*	*1*	**1**
R-5.1	**2**	*0*	*0*	*0*	**2**
R-5.2	**2**	*0*	*0*	*1*	**2**
R-5.3	**2**	*1*	*0*	*0*	**2**
R-5.4	**2**	*1*	*0*	*1*	**2**

invalid next state conditions of the FSM. Therefore, we can simply remove these invalid next state conditions from the set of boolean NSL, as we do not need to process them any further.

The **OpenLock** FSM (Figure 11.1) sets the local output of the IO signal *open* to '0' at state "1". Since the global output of *open* cannot be '1' in the composite SD controller at this point, all the next state conditions specified in **OpenLock** at this state that have *open* as '1' become invalid. We can simply remove these invalid NSL from **OpenLock**, as there is no way they could occur in the physical system.

By looking at the boolean NSL of Table 11.3, we note that four invalid next state conditions are represented by boolean vectors of rows *R-1.2, R-2.3, R-2.4* and *R-3.4*, as they contain a '1' for *open*. After removing these four invalid boolean vectors, the remaining boolean NSL are shown in Table 11.4. In the next step, our translation method will use these boolean NSL to generate the concurrent string transitions at sampled state "1" of the translated supervisor.

It is worth clarifying that we have not eliminated any boolean NSL that are specified at state "2" of **OpenLock**. Since the output of the *open* signal is set to '1' at state "2", the boolean vectors that have *open* as '1' seem to represent *valid* next state conditions. Also, we cannot remove the boolean vectors that have *open* as '0'. This is because **OpenLock** does not have any information about the behaviour of other individual FSM. Hence, it is possible that the output of *open* might be set to '0' by some other FSM. If this happens, the global output of *open* will be set to '0', and then the next state conditions that will be valid in the physical system would be those that have *open* as '0'. In this case, if we eliminate the next state conditions with *open* as '0' from **OpenLock** and another FSM set the output of *open* to '0', there will be no valid next state conditions possible at the current state and our system will block.

Step 5: Generate State Changing TDES Transitions

Now we are ready to construct the transition function of the TDES supervisor by utilizing the valid NSL of the Moore FSM. Our translation method processes one state of the FSM at a time. Then, it uses the boolean vectors that start at this FSM state one by one to define all the concurrent string transitions at the corresponding sampled state in the supervisor.

General Concepts

Essentially, our translation method treats the boolean vectors of the NSL as occurrence images of the concurrent strings that need to be defined at the corresponding sampled states of the supervisor. The FSM signals that appear as '1' in a boolean vector correspond to activity events that make up the concurrent string. Please recall that a concurrent string always ends with a *tick*. Therefore, we will always include a *tick* event in the occurrence image of any given concurrent string.

Our translation method adds a *state changing transition* (as opposed to a selfloop transition) to the supervisor for each individual event of the concurrent string that we need to define in our translated supervisor. The only exception to this is a *tick* only concurrent string (i.e. occurrence image = $\{tick\}$), that has the same start and end state. For example, the boolean NSL specified at *R-2.1* and *R-5.1* of Table 11.4 have all FSM signals as '0', and have the same start and end states. In this case, our translation method adds a selfloop of *tick* event instead of specifying it as a state changing transition.

For each boolean vector, our translation method defines "$n!$" (i.e. $n \times (n-1) \times \ldots \times 1$) concurrent strings with the same occurrence image at the source sampled state of the supervisor. Here, n is the number of FSM signals that appear as '1' in the boolean vector (number of activity events that make up the concurrent string), and '!' represents the *factorial* operation. In simple words, we generate all possible sequences of activity events that are present in the occurrence image of a given concurrent string.

By generating all possible sequences of activity events, we essentially try to make sure that Point ii.1 of the SD controllability with $||_{SD}$ property (Definition 4.5.1) is satisfied by our translated supervisors. Specifically, we ensure that when a prohibitable event is possible in a clock period, it must be possible immediately after the *tick* and stay possible for the clock period until it occurs. Also, it guarantees that an event possible in a given clock period should be able to interleave with the other events occurring in the same sampling period. This is important as we do not know which concurrent string will actually be possible in the plant TDES that designers intend to use for $||_{SD}$ verification.

Applying to OpenLock FSM

For the **OpenLock** FSM (Figure 11.1), let us consider that our translation method starts by taking state "1" of the FSM. It will then generate concurrent string transitions at sampled state "1" of the supervisor corresponding to the four valid boolean

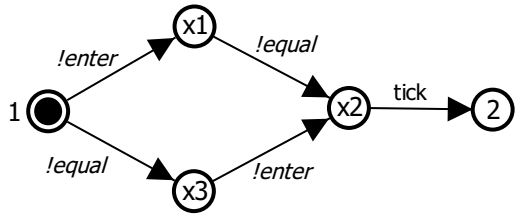


Figure 11.4: TDES Supervisor **OpenLock** After Translating Boolean NSL of *R-1.1*

NSL that start at the FSM state "1" in Table 11.4.

The boolean vector specified in row *R-1.1* of Table 11.4 has the value of '1' for two FSM signals, *enter* and *equal*. This means our translation method will generate $2! = 2$ concurrent strings with the occurrence image of {*enter, equal, tick*} at sampled state "1" of the supervisor. Specifically, we generate the concurrent strings of "*enter-equal-tick*" and "*equal-enter-tick*" that take the translated supervisor from sampled state "1" to sampled state "2", as shown in Figure 11.4. In this way, we generate all possible strings containing *enter* and *equal* exactly once, but in any order, ending with a *tick* event.

It is worth-mentioning that our translation method always generates a *deterministic* TDES supervisor (Definition 2.2.5). We guarantee this by making sure that any event, that we want to define at a given state of the supervisor, is not already defined at this state. Specifically, if an event's transition already exists at the given state, then our translation method will simply reuse the already defined transition to keep our TDES deterministic. Otherwise, we define a new transition for this event at the given state and add this newly defined transition to the transition function. Please refer to Algorithm 12.8 (discussed in Section 12.2.3) to see how we have concretely realized this strategy and other translation rules of our FSM-TDES translation method.

Processing State "1"

Now we will discuss how our translation method constructs the portion of the translated supervisor, shown in Figure 11.4, corresponding to the boolean NSL of *R-1.1* of Table 11.4. At sampled state "1" of the supervisor, the first concurrent string that we want to define is "*enter-equal-tick*". Since we have just started the translation process and no event is already defined at state "1", we define a new *enter* transition at state "1" that takes us to a new non-sampled state, say x1. As this is a new transition that we have defined at state "1" of the supervisor, our translation method adds this transition $(1, \mathit{enter}, \text{x1})$ to the transition function ξ_1.

After *enter*, the next activity event that we need to define as part of the current concurrent string is *equal*. We will define *equal* at state x1. As *equal* transition does not already exist at x1, our translation method adds a new *equal* transition that takes the translated supervisor to a new non-sampled state, say x2. We also add this new transition $(\text{x1}, \mathit{equal}, \text{x2})$ to ξ_1.

After defining all the activity event transitions, now our translation method should

add a *tick* transition at the end to complete the definition of this concurrent string. Since *tick* is the last event of this concurrent string, it should take us to the destination sampled state "2", specified as the end state in the boolean NSL, *R-1.1*. Therefore, we define a *tick* transition at state x2, and add $(\text{x2}, tick, 2)$ to ξ_1. In this way, our translation method completes the definition of the first concurrent string "*enter-equal-tick*" at sampled state "1" of the supervisor, and also updates the transition function ξ_1 with respect to this concurrent string.

The second concurrent string that we need to define at sampled state "1" of the supervisor corresponding to the same boolean vector of *R-1.1* is "*equal-enter-tick*". Our translation method uses the same strategy discussed above to define the *equal* transition at state "1", and adds the transition $(1, equal, \text{x3})$ to ξ_1.

After *equal*, the remaining part of this concurrent string that we need to define is "*enter-tick*". As *enter* is not already defined at state x3, our translation method defines a new *enter* transition at x3. While deciding about the destination state of the *enter* transition, we note that there already exists a state in the supervisor that has the same set of events possible which we want to define after the *enter* transition, i.e. state x2 has a *tick* event possible. Moreover, the *tick* transition defined at x2 goes to sampled state "2", which is the destination state of the current concurrent string. This implies that the existing state x2 has the same future that the current concurrent string has once we have defined the *enter* transition. We thus simply reuse the existing non-sampled state x2 as the destination state of the *enter* transition, and add $(\text{x3}, enter, \text{x2})$ to ξ_1. This completes the second concurrent string "*equal-enter-tick*" of the translated supervisor. In this way, our translation method has constructed the chunk of the supervisor shown in Figure 11.4 corresponding to the boolean NSL of *R-1.1*.

More examples of reusing non-sampled states can be seen at state "2" of the partially translated supervisor **OpenLock**, shown in Figure 11.8. For the boolean vector of *R-4.4* of Table 11.4, our translation method generates $3! = 6$ concurrent strings with the same occurrence image that start at sampled state "2" and end at sampled state "1" in the translated supervisor.

While generating these six concurrent strings, our translation method starts reusing the non-sampled states as soon as it has defined different sequences of any two activity events. For example, starting at sampled state "2", we converge "*open-enter*" and "*enter-open*" to a single non-sampled state x6. This is because we know that after defining *open* and *enter* transitions in a different order, the set of events that are still possible in these two concurrent strings are the same (*equal* and *tick*), as overall they have the same occurrence image. Therefore, our translation method converges them to a single state x6, and then defines a single *equal* transition followed by a *tick* to complete the definition of these two concurrent strings. Reusing states like this guarantees that two or more activity event strings, containing the same activity events but in different order, always converge to a single non-sampled state in the translated supervisor, given that these activity event strings are part of two or more

concurrent strings that start at the same sampled state and have the same occurrence image.

Non-Sampled State Reusability Issue

We would now like to clarify that our translation method will *not* reuse any non-sampled states while defining concurrent strings that correspond to two different boolean vectors. The reason being, since these concurrent strings have different activity events possible in a given clock period, this might result in the violation of given control specifications. In simple words, this might result in the enablement of an activity event in one concurrent string that is supposed to be disabled as per its occurrence image. This could happen because this activity event is enabled in the other concurrent string with a different occurrence image and we are reusing one or more non-sampled states in both concurrent strings. As a result, this activity event now becomes possible in both concurrent strings after reaching the non-sampled state(s) that we have reused.

While defining concurrent strings from different boolean vectors, the only scenario in which our translation method reuses non-sampled states is when an event is already defined at a given state of the supervisor. In this case, we do not want to define another transition for the same event at the given state, otherwise the translated supervisor becomes non-deterministic.

However, it is important to point out that starting at the source sampled state, even if we reuse some non-sampled states while defining the initial activity event transitions of the concurrent strings to keep the TDES deterministic, these two concurrent strings will certainly diverge at some later point. The reason being, since they have different occurrence images, hence different activity event transitions need to be defined overall. Once they go to different paths, our translation method never converges them back to a single non-sampled state with the purpose of reusing any existing non-sampled state of the supervisor.

Finishing State "1"

Now we will resume our discussion about the step by step translation of **OpenLock** FSM into its corresponding supervisor. After translating the boolean NSL of *R-1.1* of Table 11.4, our translation method takes the subsequent NSL specified at *R-2.1*. As all FSM signals have the value of '0' in this boolean vector, we generate $0! = 1$ concurrent string with the occurrence image of $\{tick\}$ corresponding to this boolean vector. Since *tick* is not already defined at sampled state "1" of the translated supervisor, our translation method adds a *tick* selfloop transition at state "1". We also add the transition $(1, tick, 1)$ to ξ_1. Figure 11.5 shows the addition of this *tick* transition to the previously translated supervisor of Figure 11.4.

After that, our translation method translates the third NSL specified at *R-2.2* of Table 11.4. As only one FSM signal, *equal*, has the value of '1' in this boolean vector, our translation method will generate $1! = 1$ concurrent string, "*equal-tick*", that has

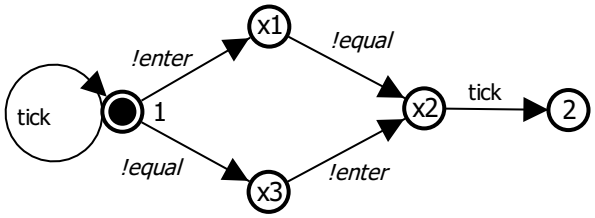


Figure 11.5: TDES Supervisor **OpenLock** After Translating Boolean NSL of *R-2.1*

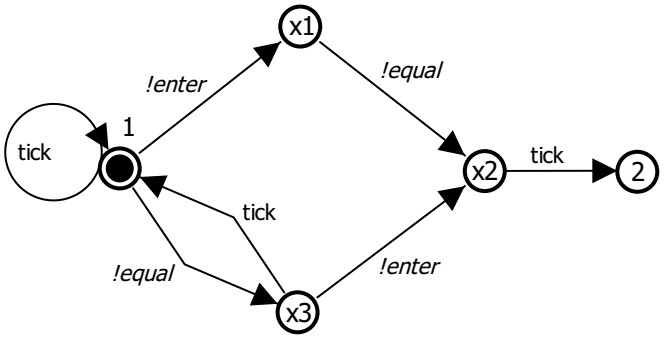


Figure 11.6: TDES Supervisor **OpenLock** After Translating Boolean NSL of *R-2.2*

the occurrence image of $\{equal, tick\}$. In order to define this concurrent string, we need to define the *equal* transition at sampled state "1" of the supervisor. However, as stated earlier, before defining a new transition, our translation method always checks to find out if an event is already defined at the given state or not.

While analyzing the events that are already defined at state "1", we discover that an *equal* transition already exists at state "1" (Figure 11.5). In this case, our translation method simply reuses and follows the existing *equal* transition. Also, since this transition is already defined in the supervisor, we know that it must exist in the transition function. Hence, we do not need to add any new transition to ξ_1. We then determine the destination state of this *equal* transition.

As *equal* is the only activity event in the current concurrent string, we now need to add a *tick* transition at the end to complete the definition of "*equal-tick*". Since *tick* is not already defined at x3, our translation method adds a *tick* transition at x3 that takes the supervisor to the destination sampled state "1". We also add this transition $(x3, tick, 1)$ to ξ_1. Figure 11.6 shows the translated supervisor after defining the concurrent string "*equal-tick*".

Our translation method handles the boolean NSL *R-3.2* of Table 11.4 in a similar way. Figure 11.7 shows the result after this translation. This completes the translation of NSL that are specified at state "1" of the **OpenLock** FSM.

After that, our translation method proceeds to the next FSM state and begins to translate the eight boolean NSL that start at state "2", as shown in Table 11.4. By applying the same translation procedure and rules that we have described above for state "1", we generate all concurrent string transitions at sampled state "2" of the

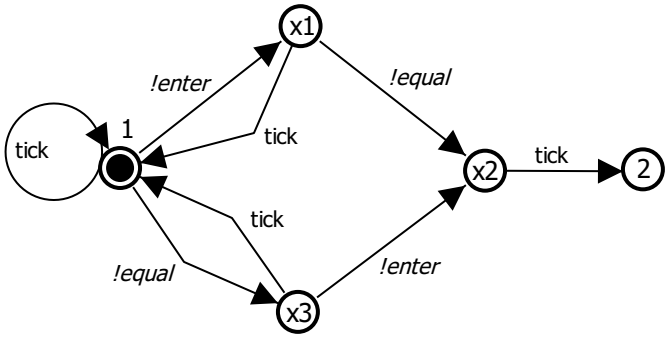


Figure 11.7: TDES Supervisor **OpenLock** After Translating Boolean NSL of *R-3.2*

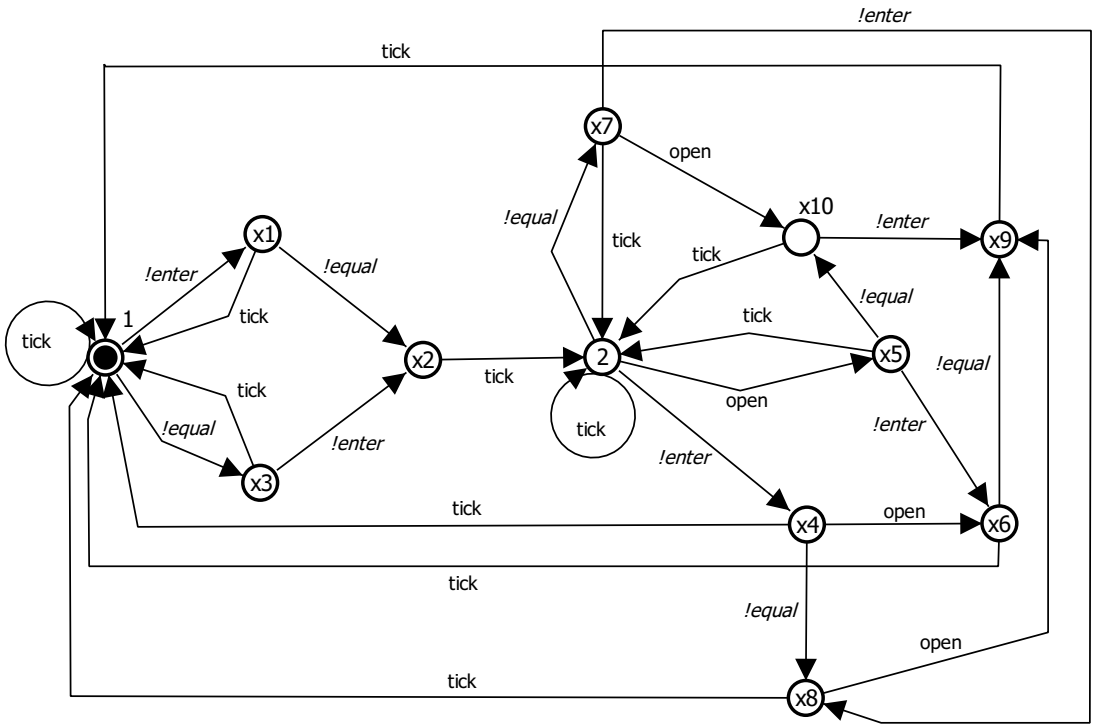


Figure 11.8: TDES Supervisor **OpenLock** After Translating Boolean NSL of Table 11.4

translated supervisor. The resultant supervisor **OpenLock** is shown in Figure 11.8.

Step 6: Add Appropriate Selfloop Transitions

This is the last step for constructing the transition function of the translated supervisor. At this step, our translation method adds transitions for the uncontrollable events and/or the *tick* event to the supervisor, if needed. We add these transitions to increase the likelihood of satisfying the desired $||_{SD}$ properties by the translated supervisors and the resulting closed-loop system constructed using our $||_{SD}$ operator.

The property of untimed controllability with $||_{SD}$ (Definition 4.4.4), which is also a part of Point i of the SD controllability with $||_{SD}$ definition (Definition 4.5.1), requires that an uncontrollable event possible in the plant TDES must be allowed to occur in the closed-loop system. This implies that the translated supervisor models must allow an uncontrollable event to occur whenever it is possible in the plant. Also, we cannot predict how many times an uncontrollable event is going to occur in a given sampling period, i.e. it may occur multiple times.

In order to generate supervisors that are more likely to be untimed controllable with $||_{SD}$ with respect to the plant TDES, our translation method adds selfloops of uncontrollable events that could occur in a given sampling period, i.e. input signals that show up as '1' in the given boolean vector. We add these selfloop transitions at those non-sampled states that are part of the concurrent string transition(s) which were defined at **Step 5** corresponding to the given boolean vector. Also, we add these selfloops only if the uncontrollable event transition is not already defined at the non-sampled state, in order to keep the translated supervisor deterministic.

It is notable that we need to add these selfloop transitions only at the non-sampled states of the supervisor, and not at the source sampled state for the given boolean vector. The reason is that our translation method generates all possible sequences of activity events that could occur in a given sampling period. Therefore, an uncontrollable event that appears as '1' in the given boolean vector always gets defined as a state changing transition at the source sampled state by our translation method at **Step 5**. The only exception to this is a **GDC** transition, that we will discuss shortly.

Applying to OpenLock FSM

For the **OpenLock** FSM, the boolean vector of *R-1.1* of Table 11.4 has the value of '1' for two input signals, *enter* and *equal*. We have three non-sampled states, x1, x2 and x3, as part of the two concurrent strings, "*enter-equal-tick*" and "*equal-enter-tick*", that get defined in the supervisor as part of processing this boolean vector (Figure 11.8). This means that our translation method needs to define selfloop transitions of *enter* and *equal* at x1, x2 and x3, if these uncontrollable event transitions do not already exist at these states.

By looking at the translated supervisor of Figure 11.8, constructed at **Step 5**, we note that *enter* needs to be selflooped at x1 and x2, whereas *equal* should be selflooped at x2 and x3. Hence, our translation method adds the transitions of (x1, *enter*, x1), (x2, *enter*, x2), (x2, *equal*, x2) and (x3, *equal*, x3) to the transition function ξ_1. The resulting translated supervisor **OpenLock** is shown in Figure 11.9.

For the boolean vector specified by *R-2.1*, since none of the input signals appear as '1', our translation method does not generate any new transitions. For *R-2.2*, the input signal *equal* has the value of '1'. The only non-sampled state that is related to the concurrent string "*equal-tick*" of this boolean vector is x3 (Figure 11.9). Since the *equal* selfloop transition already exists at x3 (was added while processing the boolean vector of *R-1.1* above), we do not add new transition to the translated supervisor.

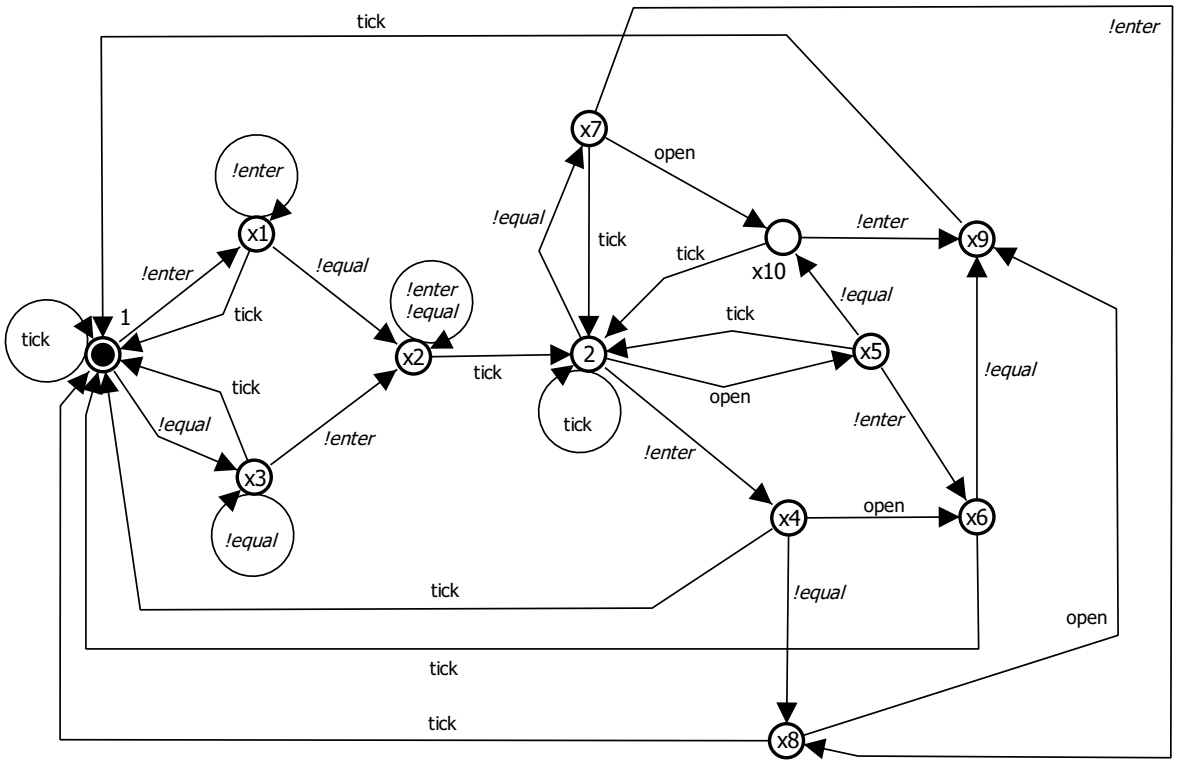


Figure 11.9: TDES Supervisor **OpenLock** After Adding Selfloop Transitions of Uncontrollable Events for Boolean NSL of *R-1.1*

The same logic applies to *R-3.2*'s boolean vector, as *enter* transition already exists at x1 as a selfloop.

By using the aforementioned logic, our translation method adds selfloop transitions of uncontrollable events corresponding to the rest of the boolean NSL of Table 11.4. Figure 11.2 shows our resultant translated supervisor **OpenLock**. It is worth-mentioning that while realizing **Steps 5** and **6** in our translation Algorithm 12.8, we do not traverse the boolean vectors multiple times. Rather, we gather all the required information to perform **Step 6** while traversing the boolean NSL at **Step 5**. This is done to improve the efficiency of our translation method and algorithm.

Processing <GDC> Transitions

In order to identify the presence of a **GDC** transition at any sampled state of the supervisor, our translation method uses the flag that we set at **Step 1**. First, it is worth clarifying that for the **GDC** transition, our translation method will not add any selfloop transitions for prohibitable events in the supervisor. This is because of one of the SD assumptions which says that controllers allow prohibitable events to occur only once per sampling period (Section 3.3). That is why, our translation method generates state changing transitions for prohibitable events at **Step 5**, which completes the translation of the supervisor with respect to the prohibitable event

transitions.

As for the uncontrollable events, our translation method adds selfloops for *all* uncontrollable events that belong to the current supervisor at the source sampled state of the **GDC** transition. Also, we will add these selfloop transitions at all of the non-sampled states that are part of the concurrent string transition(s) which are defined corresponding to the boolean vector(s) of the **GDC** transition at **Step 5**.

Please recall that **GDC** is the only valid transition defined at an FSM state (design requirement **DR-10**), and none of the uncontrollable event transitions were defined for **GDC** at **Step 5**. For this reason, we could simply add the selfloop transitions at these states without even checking whether or not an uncontrollable event's transition already exists at the state.

Adding *tick* Transitions

Another important property that we want our translated supervisors to satisfy is timed controllability with $||_{SD}$ (Definition 4.4.3). Please note that this property is also part of Point i of the SD controllability with $||_{SD}$ definition. This property requires that if *tick* event is possible in the plant TDES and no prohibitable event is possible in the closed-loop system, then *tick* must be enabled in the closed-loop system. For our purposes, we need to make sure that *tick* must be enabled in the translated supervisor if no prohibitable event is possible in the closed-loop system.

In order to ensure this, at each sampled state of the supervisor where *tick* event is not already defined at **Step 5**, our translation method adds a *tick* selfloop transition. Also, at every non-sampled state where *tick* transition does not already exist, we add a *tick* transition from the non-sampled state back to its source sampled state. Please recall that we are not reusing non-sampled states while defining concurrent strings that correspond to two different boolean vectors (see discussion at **Step 5**). Even if we reuse non-sampled states across two different boolean vectors in order to keep our TDES deterministic, these boolean vectors will certainly start at the same sampled state. As a result, each non-sampled state has only one source sampled state.

Lastly, please note that our translation method adds a *tick* transition at every sampled and non-sampled state of the supervisor where *tick* is not already defined. This might result in having one or more states in the translated supervisor where both *tick* and prohibitable events are enabled. This is not going to violate any of the desired $||_{SD}$ properties or create any other issue. The reason being, our $||_{SD}$ operator will automatically remove the unnecessary *tick* events while constructing the closed-loop system, in order to force the enabled prohibitable event(s) in the current sampling period.

This completes **Step 6**, and thus our FSM-TDES translation method. In the next section, we discuss a final step that can be performed to make the translated supervisors more compact.

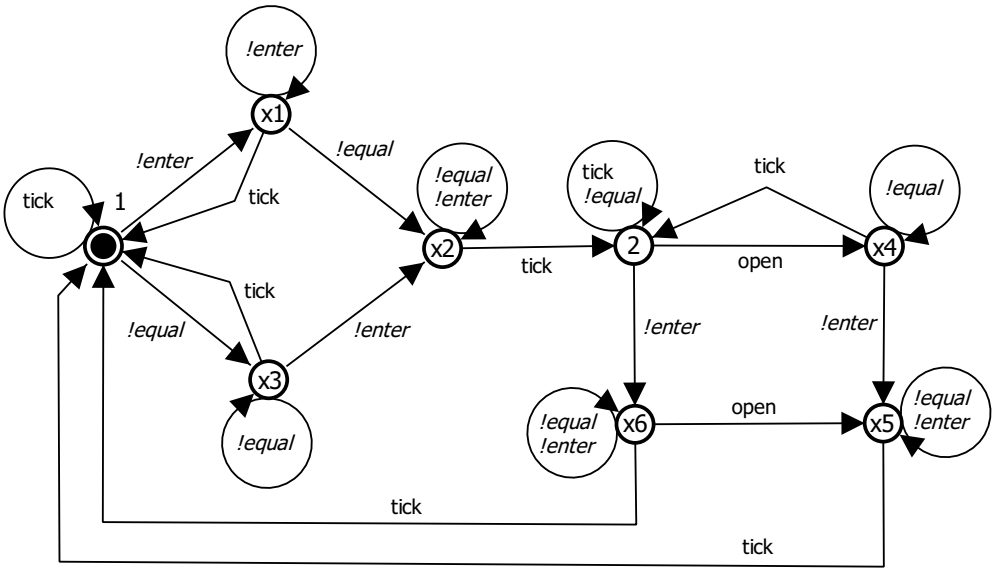


Figure 11.10: Minimal TDES Supervisor **OpenLock**

11.3.6 Make Translated Supervisor More Compact

While devising our FSM-TDES translation method and rules, our primary goal is to generate a "correct" TDES supervisor from the input Moore FSM without violating any of the given control specifications. Therefore, it is probable that the supervisor generated by our translation method might have one or more λ-equivalent states (Definition 2.2.9). In other words, the translated supervisor might not be in its minimal form (Definition 2.2.10).

Keeping this in view, the last step of our translation approach is to obtain the minimal version of the translated supervisor by performing the state space minimization process (described in Section 6.2). In future, this approach and its corresponding algorithms (Algorithms 6.1 and 6.2) should be implemented in DESpot (2023) while implementing our translation approach and algorithms.

In order to obtain the minimum-state version of our translated supervisors for this thesis, we used the **minstate** procedure of the XPTCT software package, which is a computation tool for supervisory control synthesis (Feng and Wonham, 2006). The completely translated supervisor **OpenLock** is shown in Figure 11.2, and its minimal version generated by XPTCT is shown in Figure 11.10.

After minimizing the **OpenLock** supervisor, we observe a reduction in the number of states as well as the number of transitions. Precisely, the number of states decreased from 12 states in the translated supervisor to 8 states in its minimal version. The number of transitions declined from 40 to 26.

This reduction in the overall size and complexity of the supervisor is primarily because of the fact that our translation method is generating state changing transitions at **Step 5** from the given boolean vectors. We are generating state changing

transitions in order to make sure that prohibitable events occur only once during a sampling period. Also, adding selfloops of uncontrollable events instead of state changing transitions while generating the supervisor might result in missing some sequences of activity events from the translated supervisor. This might generate a supervisor that is not SD controllable with $||_{SD}$ and fails Point ii.1 of this desired property.

Another reason for this room for minimization in the translated supervisor is that we are not reusing any non-sampled states while defining concurrent strings that correspond to two different boolean vectors, except to keep our TDES deterministic. As discussed at **Step 5**, we have developed our translation method this way because reusing non-sampled states *during* the translation process might result in the violation of the given control specifications and generation of a supervisor with undesired and incorrect system language.

In future, it would be advantageous to improve our FSM-TDES translation approach in such a way that there is no need to apply a state space minimization algorithm separately at the end to make the translated supervisor more compact. In other words, the translation method itself should generate a supervisor that does not only satisfy all the desired $||_{SD}$ properties, but is also in its minimal form.

Chapter 12

Moore FSM to TDES Translation Algorithms

In this chapter, we present the algorithms that we have developed to realize the Moore FSM-TDES translation approach described in Chapter 11. We begin this chapter by introducing the notation that we use in our algorithms. Next, we present the main algorithm that serves as the entry point to our proposed FSM-TDES translation approach for DESpot (2023). Then, we provide algorithms to perform various consistency checks on the DESpot project, central FSM and individual Moore FSM.

This is followed by the algorithms that we have developed to verify the design of each individual Moore FSM and process its information to convert it into a form that we can directly use to perform the actual FSM-TDES translation. After that, we describe the translation algorithm to generate a TDES supervisor corresponding to each input Moore FSM. Finally, we close this chapter by analyzing the time complexity of the FSM-TDES translation algorithm. Please note that due to time constraints, we have left the implementation of these algorithms in DESpot as future work.

12.1 Algorithmic Notation

In our algorithms, we will be dealing primarily with sets, sets of tuples and sets of sets of tuples. Besides the common symbols and operations used in set theory, we will use the following notation to access and manipulate the sets, and their tuples and elements in our algorithms.

12.1.1 Size Function

In our algorithms, we will use a *size function* to get the size of a set. For example, for a set named $mainFSMS$, we write $|mainFSMS|$ to get the number of elements of this set. If we pass a string to this function, then we assume that it returns the

length of the string, i.e. $|s|$ returns the length of string s. We could also pass a hybrid/boolean vector to this function, in which case it returns the size (number of elements) of the given hybrid/boolean vector.

12.1.2 Subscript Notation

While managing a set of tuples, sometimes we need to access the individual tuples of the set in a specific sequence/order. In order to be able to do that, we will treat our set like a *list*, and assume that each individual tuple of the list can be accessed using an index. We will write the index of the tuple that we want to access using the *subscript notation*.

For example, let $mainFSMS$ be a set of tuples that we will treat like a list. We write $mainFSMS_i$ to access the i^{th} tuple of this list, where $0 \leq i < |mainFSMS|$.

12.1.3 Dot Notation

In our algorithms, we will work with n-element tuples, where $n > 0$. We will use the *dot notation* to access an individual element of the tuple. Let $f = (fsmName, \Sigma_{locIO}, \Sigma_{locIn})$ be a 3-element tuple. We write $f.fsmName$ to access the first element of the tuple f. Likewise, we write $f.\Sigma_{locIO}$ and $f.\Sigma_{locIn}$ to refer to the second and third elements of this tuple respectively.

Please note that in order to refer to an element of a tuple, we are using its name (not the index) to keep things simple and memorable. Also, an individual element of a tuple could itself be a set or set/list of tuples.

Sometimes, in our algorithms, we will use the dot notation together with the subscript notation. Let $mainFSMS$ be a set/list of 3-element tuples, where each tuple is of the form $(fsmName, \Sigma_{locIO}, \Sigma_{locIn})$. Then, $mainFSMS_i.fsmName$ refers to the first element of the i^{th} tuple of this list, where $0 \leq i < |mainFSMS|$.

12.1.4 Bracket Notation

We will use the *bracket notation* for managing strings, hybrid vectors and boolean vectors in our algorithms. We will access an individual character of a string by specifying its index inside the brackets. For example, for string $s =$ "$enter \cdot equal$", we write $s[i]$ to refer to the i^{th} character of s, where $0 \leq i < |s|$.

For hybrid vectors, we will refer to an individual element of the vector by writing its index inside the brackets. Let h be a hybrid vector of size s. We write $h[i]$ to refer to the i^{th} element of h, where $0 \leq i < s$. This notation for hybrid vectors also applies to boolean vectors.

Please note that in our algorithms, we will often use this bracket notation in combination with other notations described above, as needed. For example, we could write $HQ_i.h[j]$ to refer to the j^{th} element of the hybrid vector h, such that h is an

element of the i^{th} tuple in the list of tuples HQ, where $0 \leq i < |HQ|$ and $0 \leq j < |h|$. In simple words, we have a list of tuples HQ, where each tuple of HQ has a hybrid vector as one of its elements. HQ_i refers to the i^{th} tuple of the list HQ. We then access the hybrid vector h of this i^{th} tuple by writing $HQ_i.h$, and the j^{th} element of this hybrid vector h by writing $HQ_i.h[j]$.

12.2 Main Algorithm

Algorithm 12.1 (MFSMtoTDES_Main) serves as the main entry point to perform the translation of Moore FSM into TDES supervisors for DESpot (2023). This algorithm makes a call to the other algorithms that we have developed as part of realizing our FSM-TDES translation approach (Chapter 11).

Algorithm 12.1 Assumptions

Algorithm 12.1 makes the following three assumptions:

1. As discussed in Section 11.2, the FSM-TDES translation process should be initiated in a DESpot project that already contains the TDES plant models of the physical system for which the controllers have been designed as Moore FSM. We assume that the information of the current DESpot project is available to us while performing the translation process.
2. We assume that the input XML files for the central and individual Moore FSM (see Section C.1 for details) have already been read and parsed, and the information is available to us in the form of variables. These variables could be sets, sets of tuples, or sets of sets of tuples (described shortly). Please note that the development of these algorithms to read and parse the XML files has been left as future work due to time constraints. We require that the information read from the XML files has been stored in the variables only once, i.e. there are no duplicate elements.
3. In our FSM-TDES translation algorithms, we have added some error messages. In case of an error during the translation process, we assume that DESpot displays the appropriate message to the users in a dialog box. Please note that these error messages are generic and have been added primarily to improve the comprehension and readability of the algorithms. While implementing these algorithms in DESpot, these error messages could be made more informative and helpful by adding specific details about the error causing FSM, etc.

XML Variables for FSM

Below, we list and define the variables that we use in Algorithm 12.1. We assume that these variables have already been populated by the algorithms that read and parse the input XML files of the central and individual Moore FSM.

1. The information about the current DESpot project is accessible to us in the form of the following variables:

- $projName$: A string variable that contains the name of the current DESpot project.
- Σ_{hib}: The set of prohibitable events in the project.
- Σ_{u}: The set of uncontrollable events in the project.

2. The information specified in the central FSM XML file (described in Section C.1.2) is available to us by means of the following variables:

- $mainFSM$: A string variable that stores the name of the central FSM.
- Σ_{gout}: The set of global output signals specified in the central FSM.
- Σ_{gin}: The set of global input signals specified in the central FSM.
- $mainFSMS$: A set of all the individual Moore FSM that are listed in the central FSM XML file. Each element of this set stores information about one individual FSM, and is a 3-tuple of the form $(fsmName, \Sigma_{locIO}, \Sigma_{locIn})$. Here, $fsmName$ refers to the name of the individual FSM, Σ_{locIO} is a set of the FSM's local IO signals, and Σ_{locIn} is a set of the FSM's local input signals. Sometimes, we will treat this set like a list in our algorithms (see Section 12.1.2 for details).

3. While reading and parsing the XML files of the individual Moore FSM (described in Section C.1.1), the information is stored for us in the following variables:

- $FSMS$: This set stores all the individual Moore FSM that have been provided as an input to the FSM-TDES translation process. Each element of this set is a tuple that stores information about the individual FSM specified in the given XML file. Each tuple consists of the variables defined below.
  - $name$: A string variable that stores the name of the individual Moore FSM.
  - $resetState$: A string variable containing the initial/reset state of the FSM.
  - Σ_{IO}: The set of IO signals of the FSM.
  - Σ_{In}: The set of input signals of the FSM.
  - Q: The set of states of the FSM. For simplicity, we assume that all states in Q are labelled in the form of q0, q1, q2, ... This is to ensure that these names are distinct from the non-sampled state names we will generate in Algorithm 12.8 of the form x0, x1, x2, ...
  - QZ: Let Z be a set of strings. The set $QZ \subseteq Q \times Z$ contains the states of the FSM and their corresponding output information. For $(q, outputVector) \in QZ$, $outputVector$ is a string that contains a comma separated list of the IO signals whose outputs are set to $True$ at state q of the FSM.
  - Q_{invect}: Before defining Q_{invect}, we will first introduce the variable IQ.

Let IS be a set of strings. The set $IQ \subseteq IS \times Q$ contains the possible next state conditions of the FSM and their corresponding destination states. For $(inputVector, q') \in IQ$, $inputVector$ is a string that represents a next state condition, and q' is the destination state reached by this next state condition. Please note that the next state condition can either be expressed as a boolean expression or using one of our keywords (<**TICK**>, <**GDC**> or <**DEF**>).

The partial function $Q_{invect} : Q \rightarrow \text{Pwr}(IQ)$ maps the states of an FSM to the next state logics (NSL) that are specified at these states. In simple words, for a given state q of the FSM, this function returns a set of tuples of the form $(inputVector, q')$, i.e. a set of all the next state conditions that are defined at q, and their corresponding destination states. This function will sometimes be treated like the set $Q_{invect} \subseteq Q \times \text{Pwr}(IQ)$.

– *marking*: An enumerated data type to specify the marking information for the TDES supervisor that will be generated corresponding to the individual Moore FSM. Currently, we provide the following two marking options to the designers to choose from:

i) *Initial*: Mark only the initial state of the translated supervisor.

ii) *Sampled*: Mark all sampled states of the translated supervisor, and nothing else.

FSM-TDES Variables

We need to process the information that is available to us for each individual Moore FSM and convert it into a form that we can directly use to perform the FSM-TDES translation. We have developed several algorithms to do the required processing and then store the processed information in our desired variables. Since we have declared the new variables in Algorithm 12.1, we will define and describe these variables in this section. Specifically, by using the information stored in the above-mentioned variables for each individual FSM, our algorithms populate the following variables that we will ultimately use to perform the translation process.

- Q_{Elig}: The partial function $Q_{Elig} : Q \rightarrow \text{Pwr}(\Sigma_{IO})$ maps the states of an individual Moore FSM to the set of IO signals that are eligible to occur at these states, i.e. their outputs are set to $True$. Sometimes, in our algorithms, we will treat this function like the set $Q_{Elig} \subseteq Q \times \text{Pwr}(\Sigma_{IO})$. Algorithm D.3 (given in Appendix D) populates the set Q_{Elig} by processing the information stored in the set QZ.

- Q_{nsz}: Before defining Q_{nsz}, we will first introduce the variable HQ.

  Let H be a set of hybrid vectors. The set $HQ \subseteq H \times Q$ contains the possible hybrid vectors of the Moore FSM, and their corresponding destination states. For $(hybridVector,\ q') \in HQ$, $hybridVector$ represents a next state condition, and q' is the destination state reached by this next state condition. Sometimes, we will treat this set like a list in our algorithms.

The partial function $Q_{nsz} : Q \rightarrow \text{Pwr}(HQ)$ maps the states of an FSM to the NSL that are specified at these states. In simple words, for a given state q of the FSM, this function returns a set of tuples of the form $(hybridVector, q')$, i.e. a set of hybrid vectors that represents the next state conditions defined at q, and their corresponding destination states. Sometimes, we will treat this function like the set $Q_{nsz} \subseteq Q \times \text{Pwr}(HQ)$. Algorithm 12.2 populates the set Q_{nsz} by processing the information stored in the set Q_{invect}.

- Q_{DC}: The partial function $Q_{DC} : Q \rightarrow Boolean$ maps the states of an FSM to the boolean value of $True$ or $False$. Precisely, for a given state q of the Moore FSM, this function returns $True$ if the next state condition specified at q is a global don't care, i.e. a **GDC** transition is defined at q. Otherwise, it returns $False$.

  In our algorithms, we will sometimes treat this function like the set $Q_{DC} \subseteq Q \times Boolean$. We will use the information stored in Q_{invect} to populate the set Q_{DC} in Algorithm 12.2.

- Q_{DEF}: The partial function $Q_{DEF} : Q \rightarrow Boolean$ maps the states of an FSM to the boolean value of $True$ or $False$. Precisely, for a given state q of the Moore FSM, this function returns $True$ if the set of next state conditions defined at q includes the **DEF** transition. If **DEF** is missing at q, this function returns $False$.

  Sometimes, we will treat this function like the set $Q_{DEF} \subseteq Q \times Boolean$. We will use the information stored in Q_{invect} to populate the set Q_{DEF} in Algorithm 12.2.

In order to store the translated TDES supervisors, we define the following variables in Algorithm 12.1:

- $supName$: A string variable to store the name of the supervisor that is currently being translated. We will use the name of the individual Moore FSM as the name of the corresponding supervisor.

  To keep things simple, we will ignore the issue that the name of each TDES in the DESpot project must be unique, and the project could already contain a TDES with the same name as the FSM. We leave the resolution of such a conflict (such as popping up a dialog box to the user) as an implementation detail.

- $\mathbf{S}$: This variable is used to store the TDES supervisor that is currently being translated. $\mathbf{S}$ is a quintuple of the form $\mathbf{S} = (X, \Sigma_{\mathbf{S}}, \xi, x_o, X_m)$. Here, X is the state set, $\Sigma_{\mathbf{S}}$ is the event set, $\xi : X \times \Sigma_{\mathbf{S}} \rightarrow X$ is the partial transition function, $x_o \in X$ is the initial state, and $X_m \subseteq X$ is the set of marked states of the translated supervisor.

- $TDES$: The set $TDES \subseteq String \times \mathbf{S}$ stores the names of the translated supervisors and their corresponding quintuples. For $(supName, \mathbf{S}) \in TDES$, $supName$ is the name of the translated supervisor whose quintuple is stored in $\mathbf{S}$. We will use "Push" method to add the tuples of the translated supervisors to $TDES$.

Besides these variables, we will use an implementation-independent method "*print*()" in our algorithms to indicate that we wish to display a message to the DESpot users in a dialog box. The message to be printed is enclosed in quotation marks, i.e. print("*message*").

Algorithm 12.1

Algorithm 12.1 (MFSMtoTDES_Main) calls several other algorithms to perform the FSM-TDES translation process. Specifically, it calls Algorithm D.1 (VerifyCentralFSM), Algorithm D.2 (VerifyIndividualFSM), Algorithm D.3 (GenerateEnablementInfo), Algorithm 12.2 (GenerateHybridNextStateLogic), Algorithm 12.3 (GenerateBooleanNextStateLogic) and Algorithm 12.8 (GenerateTDESSupervisor).

Please note that Algorithms D.1−D.3 are presented in Appendix D. These algorithms do the required setup and perform some consistency checks. We have moved them to Appendix D so we can focus on the algorithms that implement the key part of our FSM-TDES translation approach described in Chapter 11.

Algorithm 12.1 begins by verifying the consistency requirement **CR-1** at **lines 1-4**. At **lines 5-7**, Algorithm 12.1 calls Algorithm D.1 to perform some consistency and design checks using the information specified in the central FSM. If any of the checks fail, Algorithm D.1 returns $False$ and the FSM-TDES translation process terminates immediately by returning $False$ to the caller algorithm.

At **lines 8-10**, we call Algorithm D.2 to analyze the information specified in the individual Moore FSM and perform the required consistency checks. Again, we end the translation process at once if any of the consistency requirements is not satisfied by the input Moore FSM.

At **lines 12-26**, we loop through all individual Moore FSM that are stored in the $FSMS$ variable. By taking one individual Moore FSM at a time, Algorithm 12.1 calls other algorithms to do the desired design checks, process the FSM's information, translate it into a TDES supervisor, and finally add the name of the translated supervisor along with its quintuple to the $TDES$ variable. Once all Moore FSM have been successfully translated into supervisors, we notify DESpot users about the successful completion of the translation process at **line 27**.

Since our FSM-TDES translation approach uses the FSM name as the name of the translated supervisor, we assign the name of the FSM that we are currently processing to the $supName$ variable at **line 14**. At **lines 15-17**, we call Algorithm D.3 to process the output information of the current Moore FSM and generate the corresponding enablement information for the supervisor. This algorithm also evaluates the related design requirements that must be satisfied by the FSM.

At **lines 18-20**, we call Algorithm 12.2 to generate hybrid NSL from the NSL specified in the current Moore FSM. This hybrid NSL is then converted to the boolean NSL by calling Algorithm 12.3 at **lines 21-23**. **Algorithms 12.2** and **12.3** also perform some related design checks on the current FSM while processing its NSL. If any of these algorithms finds any discrepancy in the NSL of the current FSM, they

Algorithm 12.1 MFSMtoTDES_Main()

1: **if** $(mainFSM \neq projName)$ **then**
2: print("*Error! Project and central FSM names do not match.*")
3: **return** $False$
4: **end if**
5: **if** $(\neg$ VerifyCentralFSM$(mainFSMS, \Sigma_{gout}, \Sigma_{gin}, \Sigma_{hib}, \Sigma_u))$ **then**
6: **return** $False$
7: **end if**
8: **if** $(\neg$ VerifyIndividualFSM$(mainFSMS, FSMS))$ **then**
9: **return** $False$
10: **end if**
11: $TDES \leftarrow \emptyset$
12: **for all** $(fsm \in FSMS)$ **do**
13: $Q_{Elig}, Q_{nsz}, Q_{DEF}, Q_{DC}, \mathbf{S} \leftarrow \emptyset$
14: $supName \leftarrow fsm.name$
15: **if** $(\neg$ GenerateEnablementInfo$(fsm, Q_{Elig}))$ **then**
16: **return** $False$
17: **end if**
18: **if** $(\neg$ GenerateHybridNextStateLogic$(fsm, Q_{nsz}, Q_{DEF}, Q_{DC}))$ **then**
19: **return** $False$
20: **end if**
21: **if** $(\neg$ GenerateBooleanNextStateLogic$(fsm, Q_{nsz}, Q_{Elig}, Q_{DEF}, Q_{DC}))$ **then**
22: **return** $False$
23: **end if**
24: GenerateTDESSupervisor $(fsm, Q_{nsz}, Q_{DC}, \mathbf{S})$
25: Push$(TDES, (supName, \mathbf{S}))$
26: **end for**
27: print("*Success! The Moore FSM-TDES supervisors translation process is complete.*")
28: **return** $True$

return $False$ and Algorithm 12.1 immediately suspends the translation process.

After processing the NSL of the current FSM, we call the translation Algorithm 12.8 at **line 24** to generate a TDES supervisor corresponding to the current Moore FSM. Finally, **at line 25**, we save the name and quintuple of the translated supervisor in the $TDES$ variable before proceeding to the translation of the next Moore FSM.

12.2.1 Generate Hybrid Next State Logic

Algorithm 12.2 processes the NSL of an individual Moore FSM ($fsm.Q_{invect}$) to generate the corresponding hybrid NSL (Q_{nsz}). Specifically, we generate hybrid vectors

from the given next state conditions (input vectors) of the FSM. This algorithm primarily realizes **Step 1** of Section 11.3.5.

While processing the NSL, Algorithm 12.2 also performs the related design checks on the current FSM. If the FSM fails to satisfy any of these design requirements, we return $False$ to Algorithm 12.1 after printing the appropriate error message.

The following local variables are defined and used by Algorithm 12.2. Please refer to Section 12.2 to see the definition of other variables used in the algorithm.

- *signalCount*: This integer variable stores the total number of IO and input signals that belong to the current FSM.
- IQ: A set of tuples of the form ($inputVector$, q'), where $inputVector$ is a string that represents a next state condition, and $q' \in Q$ is the destination state reached by this next state condition.
- $hybridVector[signalCount]$: A hybrid vector $hybridVector \in H$ that has *signalCount* elements, i.e. the size of the $hybridVector$ is equal to *signalCount*. The indexes for the $hybridVector$ elements range from 0 to $signalCount - 1$.
- HQ: A set of tuples of the form ($hybridVector$, q'), where $hybridVector \in H$ is a hybrid vector representing a next state condition, and $q' \in Q$ is the destination state reached by this next state condition.
- *isDisabled*: This flag is set to $True$ if the currently processed signal appears in the current input vector in the complemented form, i.e. !<*signalname*>.
- Σ_{dis}: This set stores the IO and input signals that appear in the currently processed input vector in the complemented form.
- Σ_{en}: This set stores the IO and input signals that appear in the currently processed input vector in the uncomplemented form, i.e. without "!".
- *signal*: This string variable temporarily holds the characters, as we read the name of a signal (one character at a time) from the input vector that we are currently processing.
- μ: The *signal mapping function* for an individual Moore FSM $fsm \in FSMS$ is defined to be a bijective map $\mu : fsm.\Sigma_{IO} \cup fsm.\Sigma_{In} \rightarrow \{0, 1, \ldots, s-1\}$, where $s = |fsm.\Sigma_{IO} \cup fsm.\Sigma_{In}|$. In simple words, this function maps the signals of the given Moore FSM to their corresponding indexes in the hybrid vector $hybridVector \in H$ of size s.

At **line 1** of Algorithm 12.2, we populate the *signalCount* variable by assigning to it the total number of IO and input signals of the FSM that we are currently translating. At **lines 2-78**, we loop through the state set of the current FSM ($fsm.Q$). By taking one FSM state q at a time (**line 2**), we retrieve the set of all NSL that are defined at q by using the partial function Q_{invect}, and store it in IQ (**line 3**). At **lines 4-7**, we perform design check **DR-5** which requires that at least one NSL must be specified at every state of the FSM. We do this by making sure that the set IQ is

Algorithm 12.2 GenerateHybridNextStateLogic($fsm, Q_{nsz}, Q_{DEF}, Q_{DC}$) Part I

```
1: signalCount ← |fsm.Σ_IO ∪ fsm.Σ_In|
2: for all (q ∈ fsm.Q) do
3:     IQ ← fsm.Q_invect(q)
4:     if (IQ = ∅) then
5:         print("Error! At every state of the FSM, at least one next state condition
           must be specified. Please specify a <DEF> transition, if no valid next
           state conditions exist at a state.")
6:         return False
7:     end if
8:     HQ ← ∅
9:     Push(Q_DEF, (q, False))
10:    Push(Q_DC, (q, False))
11:    for all (nsl ∈ IQ) do
12:        hybridVector[signalCount]
13:        isDisabled ← False
14:        Σ_dis, Σ_en ← ∅
15:        signal ← ""
16:        if (nsl.inputVector = "") then
17:            print("Error! In the FSM, the transition input vectors cannot be left
               empty. Please specify valid next state conditions in the empty input
               vectors or remove the entire transition of the empty input vectors.")
18:            return False
19:        else if (nsl.inputVector = <DEF>) then
20:            if (nsl.q′ ≠ q) then
21:                print("Error! In the FSM, the <DEF> transition must always
                   be specified as a selfloop.")
22:                return False
23:            end if
24:            Q_DEF(q) ← True
25:        else
26:            if (nsl.inputVector = <TICK>) then
27:                for (i ← 0 to signalCount − 1) do
28:                    hybridVector[i] ← 0
29:                end for
30:            else
31:                for (i ← 0 to signalCount − 1) do
32:                    hybridVector[i] ← d
33:                end for
34:                if (nsl.inputVector = <GDC>) then
35:                    Q_DC(q) ← True
```

Algorithm 12.2 GenerateHybridNextStateLogic($fsm, Q_{nsz}, Q_{DEF}, Q_{DC}$) Part II

```
36:             else
37:                 for (i ← 0 to |nsl.inputVector| − 1) do
38:                     if (nsl.inputVector[i] = "!") then
39:                         isDisabled ← True
40:                     else if (nsl.inputVector[i] = ".") then
41:                         if (isDisabled) then
42:                             Σ_dis ← Σ_dis ∪ {signal}
43:                         else
44:                             Σ_en ← Σ_en ∪ {signal}
45:                         end if
46:                         isDisabled ← False
47:                         signal ← ""
48:                     else
49:                         signal ← signal + nsl.inputVector[i]
50:                     end if
51:                 end for
52:                 if (isDisabled) then
53:                     Σ_dis ← Σ_dis ∪ {signal}
54:                 else
55:                     Σ_en ← Σ_en ∪ {signal}
56:                 end if
57:                 for all (σ ∈ Σ_dis ∪ Σ_en) do
58:                     if (σ ∉ fsm.Σ_IO ∪ fsm.Σ_In) then
59:                         print("Error! The signals specified in the transition input
                           vectors must be listed in the FSM's list of signals.")
60:                         return False
61:                     end if
62:                     if (σ ∈ Σ_dis) then
63:                         hybridVector[μ(σ)] ← 0
64:                     else
65:                         hybridVector[μ(σ)] ← 1
66:                     end if
67:                 end for
68:             end if
69:         end if
70:         if (nsl.q′ ∉ fsm.Q) then
71:             print("Error! The states specified as the end states of the transi-
                   tions must be listed in the FSM's list of states.")
72:             return False
73:         end if
74:         HQ ← HQ ∪ {(hybridVector, nsl.q′)}
```

Algorithm 12.2 GenerateHybridNextStateLogic($fsm, Q_{nsz}, Q_{DEF}, Q_{DC}$) Part III

```
75:         end if
76:     end for
77:     Push(Q_nsz, (q, HQ))
78: end for
79: return True
```

not empty.

At **line 9**, we add a tuple of the current state q along with the boolean value of $False$ to the set Q_{DEF}. This is because by default, we assume that a **DEF** transition is not defined at q and all the possible next state conditions have been explicitly specified at q. However, while processing the NSL defined at q, if we come across a **DEF** transition afterwards, then we update this boolean value to $True$ at **line 24**.

Similarly, at **line 10**, we update the set Q_{DC}. Initially, we are assuming that **GDC** transition is not defined at q. Hence, we add a tuple of the current state q along with the boolean value of $False$ to Q_{DC}. However, later on, if we detect a <**GDC**> keyword in the next state condition(s) specified at q, then we update this boolean value to $True$ at **line 35**.

At **lines 11-76**, we loop through the set IQ to process all the NSL that are defined at q. We take one NSL (nsl) at a time (**line 11**). In order to process the current next state condition represented by the input vector ($nsl.inputVector$), we define an **if-elseif-else** block at **lines 16-75**.

First, at **line 16**, we use the **if** statement to verify design requirement **DR-8** by making sure that the input vector of the current NSL is not empty. If **DR-8** is satisfied, the **if** condition evaluates to $False$.

In this case, the algorithm proceeds to the **else if** block at **lines 19-24**. At **line 19**, we determine if the current next state condition defines a **DEF** transition, i.e. if the current input vector contains the <**DEF**> keyword. If so, we verify design check **DR-7** at **lines 20-23**. **DR-7** requires that **DEF** transition must always be defined as a selfloop in the FSM. If **DR-7** passes, we update the tuple of the current state q in Q_{DEF} at **line 24** by replacing the default boolean value of $False$ with $True$. This indicates that **DEF** transition has been defined at state q in the FSM to cover the invalid next state conditions.

Please recall that our FSM-TDES translation method does not generate any transitions in the supervisor corresponding to invalid next state conditions covered by the **DEF** transition of the FSM. Therefore, in the case of **DEF**, the algorithm does not execute the **else** block to construct or populate any hybrid vector. It directly jumps to **lines 75-76**, and then proceeds to take the subsequent NSL from IQ at **line 11**.

However, if the current input vector does not define a **DEF** transition, then we are left with three valid possibilities, i.e. the input vector defines a **TICK**, **GDC** or a boolean expression. We assess and process these three valid possibilities inside the

else block starting at **line 25**.

At **line 26**, we determine if the current input vector defines a **TICK** transition. If so, at **lines 27-29**, we assign the value of '0' to all elements of the hybrid vector ($hybridVector$). On the other hand, if the possibility of **TICK** gets ruled out, then the current input vector should either represent a **GDC** transition or a boolean expression. In both cases, we initialize all elements of the hybrid vector to the '*d*' value at **lines 31-33**.

If the current input vector represents a **GDC** transition (**line 34**), then we update the tuple of the current state q in Q_{DC} by replacing the default boolean value of $False$ with $True$ (**line 35**). This indicates that a **GDC** transition exists at q. Please note that we do not make any changes in the hybrid vector corresponding to **GDC**, as every element of the hybrid vector already contains the value of '*d*'.

However, if the current input vector does not represent a **GDC** transition, then it should contain a next state condition expressed as a boolean expression. In this case, at **lines 36-68**, we read the boolean expression from the input vector, and generate the corresponding hybrid vector by updating the hybrid vector initialized at **lines 31-33**.

At **lines 37-51**, we read the characters of the current input vector one by one. If we read a complement symbol "!" (**line 38**), this means the signal that we are about to read appears in the complemented form in the input vector. In this case, we set the $isDisabled$ boolean variable to $True$ (**line 39**).

If we come across a period (".", **line 40**), this implies that we have read one complete signal name from the input vector. If this signal appears in the complemented form in the input vector, i.e. $isDisabled = True$, we add it to Σ_{dis} (**lines 41-42**). Otherwise, we add the signal to Σ_{en} (**lines 43-44**). As we will start reading a new signal name now, given that the input vector has not ended, we set $isDisabled$ to its default value of $False$ (**line 46**). Also, we clear out the string *signal* that we used to store the current signal name (**line 47**).

For all other characters that we read from the input vector, we simply append the character to the end of the *signal* string (**line 49**). Once we have read all characters of the input vector, depending upon the value of $isDisabled$, we add the last read signal name to Σ_{dis} or Σ_{en} at **lines 52-56**.

At **lines 57-67**, we loop through all signals of Σ_{dis} and Σ_{en} by taking one signal at a time. First, at **lines 58-61**, we verify design requirement **DR-9** by making sure that the current signal of the boolean expression belongs to the list of signals of the current FSM, i.e. it belongs to either $fsm.\Sigma_{IO}$ or $fsm.\Sigma_{In}$.

If **DR-9** is satisfied, then we update the value of the current signal in the hybrid vector at **lines 62-66**. Precisely, if the current signal belongs to Σ_{dis}, we assign the value of '0' to the element of the hybrid vector that corresponds to this signal (**lines 62-63**). Otherwise, we assign the value of '1' to the element of hybrid vector that corresponds to the current signal (**lines 64-65**).

It is worth clarifying that if an input vector contains a simplified boolean expression, then the signal(s) that have been treated as DC, will not appear in the given boolean expression. Hence, these signals belong to neither Σ_{dis} nor Σ_{en}. Please recall that we initialized all elements of the hybrid vector to 'd' at **lines 31-33**. As a result, the elements of the hybrid vector that correspond to the DC signals of the boolean expression already have the value of 'd' assigned to them, as desired.

After generating a hybrid vector corresponding to the current input vector, we perform the design check **DR-12** at **lines 70-73**. In simple words, we verify that the end (destination) state of the current input vector ($nsl.q'$) belongs to the list of states of the current FSM ($fsm.Q$).

At **line 74**, we add the hybrid NSL, i.e. the generated hybrid vector along with its destination state, to the set HQ. Once we have generated all hybrid NSL corresponding to the current FSM state q, and added them to HQ, we finally add this tuple of q and HQ to Q_{nsz} at **line 77**.

12.2.2 Generate Boolean Next State Logic

Algorithm 12.3 is primarily responsible for generating, processing and verifying the boolean NSL for the current Moore FSM. It does so by making use of several other algorithms that convert hybrid vectors into boolean vectors, process the boolean NSL, and perform various design checks. If any of the required design checks fails, Algorithm 12.3 returns $False$ to Algorithm 12.1 after displaying the appropriate error message. Specifically, Algorithm 12.3 (GenerateBooleanNextStateLogic) calls Algorithm 12.4 (ConvertHybridToBooleanNSL), Algorithm 12.5 (IdentifyNondeterministicNSL), Algorithm 12.6 (RemoveInvalidNSL) and Algorithm 12.7 (CheckForValidNSL).

In addition to the variables introduced in Section 12.2, Algorithm 12.3 defines the following local variables:

- $signalCount$: This integer variable stores the total number of IO and input signals that belong to the FSM that is being processed.
- Σ_{en}: This set contains the IO signals whose outputs are set to $True$ at the current state of the FSM. In terms of the supervisor, this set contains prohibitable events that are enabled at the corresponding sampled state of the supervisor.
- Σ_{dis}: This set contains the IO signals whose outputs are set to $False$ at the current state of the FSM. In terms of the supervisor, this set contains prohibitable events that are disabled at the corresponding sampled state of the supervisor.

Algorithm 12.3 begins by populating the $signalCount$ variable at **line 1**. At **lines 2-37**, we loop through the state set of the current FSM ($fsm.Q$). By taking one FSM state q at a time (**line 2**), we pop the set of hybrid NSL that are defined at q from Q_{nsz}, and store it in HQ (**line 3**).

Algorithm 12.3 GenerateBooleanNextStateLogic($fsm, Q_{nsz}, Q_{Elig}, Q_{DEF}, Q_{DC}$) Part I

```
1: signalCount ← |fsm.Σ_IO ∪ fsm.Σ_In|
2: for all (q ∈ fsm.Q) do
3:     HQ ← Pop(Q_nsz(q))
4:     if (Q_DC(q)) then
5:         if (|HQ| = 1) then
6:             for all (σ ∈ fsm.Σ_In) do
7:                 HQ_0.hybridVector[μ(σ)] ← 0
8:             end for
9:         else
10:            print("Error! At any given state of the FSM, if <GDC> is specified
               as the next state condition, then it must be the only next state con-
               dition specified at this state. No other valid next state conditions could
               be specified.")
11:            return False
12:        end if
13:    end if
14:    ConvertHybridToBooleanNSL(HQ, signalCount)
15:    if (¬ Q_DC(q)) then
16:        if (¬ IdentifyNondeterministicNSL(HQ, signalCount)) then
17:            return False
18:        end if
19:        if (¬ Q_DEF(q)) then
20:            if (|HQ| ≠ 2^signalCount) then
21:                print ("Error! At any given state of the FSM, if <DEF> transi-
                   tion is not specified, then the next state logic for all possible input
                   combinations must be explicitly specified/covered by the input
                   vectors.")
22:                return False
23:            end if
24:        end if
25:    end if
26:    Σ_en ← Q_Elig(q)
27:    Σ_dis ← fsm.Σ_IO − Σ_en
28:    if (Σ_dis ≠ ∅) then
29:        RemoveInvalidNSL(HQ, Σ_dis)
30:    end if
31:    if (¬ Q_DC(q) ∧ Σ_en ≠ ∅) then
32:        if (¬ CheckForValidNSL(HQ, Σ_en)) then
33:            return False
34:        end if
```

Algorithm 12.3 GenerateBooleanNextStateLogic($fsm, Q_{nsz}, Q_{Elig}, Q_{DEF}, Q_{DC}$) Part II

35: **end if**
36: Push($Q_{nsz}(q), HQ$)
37: **end for**
38: **return** $True$

At **line 4**, we check for a **GDC** transition at current state q of the FSM by using the partial function Q_{DC}. If **GDC** is defined at q, then we verify **DR-10** which requires that **GDC** must be the only valid NSL defined at q. We evaluate this at **line 5** by making sure that the set HQ contains only one hybrid NSL, i.e. it has only one hybrid vector. This hybrid vector stores the value of 'd' for every element, as per the logic of generating hybrid vectors described in Algorithm 12.2.

If **DR-10** is satisfied, then at **lines 6-8**, we replace the value of 'd' with '0' for each element of the hybrid vector that corresponds to the input signal of the FSM. Please refer to **Step 1** of Section 11.3.5 to gain an insight into why this replacement is required in the hybrid vector for the input signals of **GDC**.

At **line 14**, we call Algorithm 12.4 to generate boolean NSL from the hybrid NSL defined at state q. Specifically, we convert each hybrid vector into its corresponding boolean vector representation.

Lines 16-24 are executed only if a **GDC** transition is not defined at the current state q (**line 15**). First, at **lines 16-18**, we call Algorithm 12.5 that verifies **DR-13** by checking for nondeterministic boolean NSL defined at q. This algorithm is also responsible for removing any duplicate boolean NSL at q. Since **DR-10** requires that **GDC** must be the only valid transition defined at a given FSM state, we cannot have nondeterministic or duplicate boolean NSL in the case of **GDC**. That is why, we call Algorithm 12.5 only if a **GDC** transition does not exist at q.

After that, at **line 19**, we check to see whether or not a **DEF** transition is defined at q. If not, we perform design check **DR-6** at **lines 20-23**. Specifically, at **line 20**, we examine the number of unique boolean NSL defined at q ($|HQ|$) to make sure that all possible next state conditions ($2^{signalCount}$) are explicitly specified at q to make the FSM's next state function a total function.

Clearly, **DR-6** does not need to be verified if **DEF** is defined at q. This is because the main purpose of including **DEF** is to make sure that **DR-6** is always satisfied. In other words, **DR-6** can never fail in the presence of **DEF**, hence there is no need to do an explicit check. Also, we do not need to verify **DR-6** in the case of **GDC**. Since each element of the hybrid vector has a 'd' for **GDC**, therefore **GDC** automatically covers all possible next state conditions by definition.

Please note that it is acceptable to specify both **DEF** and **GDC** transitions at a given state of the input FSM. This is essentially the current behaviour of DESpot, as it generates a selfloop of **DEF** at every state of the FSM while performing the

TDES-FSM translation process. However, in this case, the **DEF** transition will be empty, as **GDC** covers all possible next state conditions.

At **lines 26-27**, we populate the local variables Σ_{en} and Σ_{dis} respectively by making use of the enablement information for the current state q ($Q_{Elig}(q)$) generated by Algorithm D.3 (given in Appendix D). At **lines 28-30**, we use the enablement information of q to remove the invalid NSL from the set of boolean NSL defined at q. Precisely, if one or more IO signals have the output of $False$ at state q (**line 28**), then we call Algorithm 12.6 to perform the desired task (**line 29**).

At **lines 32-34**, we call Algorithm 12.7 to verify design requirement **DR-11**. It is obvious that **DR-11** does not need to be verified if **GDC** exists at q or Σ_{en} is empty. For this reason, we call Algorithm 12.7 only if **GDC** is not defined at q and the output of at least one IO signal is set to $True$ at q (**line 31**).

After generating, processing and verifying the boolean NSL defined at q, we store the verified set of valid boolean NSL corresponding to state q (HQ) in Q_{nsz} at **line 36**.

Convert Hybrid to Boolean Next State Logic

Algorithm 12.4 performs the task of converting hybrid NSL into boolean NSL. Specifically, it translates each hybrid vector into its corresponding boolean vector representation. This algorithm primarily realizes **Step 2** of Section 11.3.5. Please refer to the description of **Step 2** to refresh your memory about the technique of converting hybrid vectors into boolean vectors.

Algorithm 12.4 makes use of the following local variables. Please refer to Algorithm 12.3 to see the definition of other variables used in Algorithm 12.4.

- *tmpnsl*: This is a 2-element tuple of the form $(hybridVector, q')$, where $hybridVector \in H$ is a hybrid vector that represents a next state condition, and q' is the destination state reached by this next state condition. We use the *tmpnsl* variable to temporarily store the hybrid NSL that we are currently processing.
- *tmpHQ*: A set of tuples of the form $(hybridVector, q')$. We use the *tmpHQ* variable to temporarily store the set of hybrid NSL after processing them.

In Algorithm 12.4, we loop through all the FSM signals that are represented by the elements of the hybrid vector (**lines 1-14**). For each element (i), we loop through all the hybrid NSL that are stored in HQ (**lines 3-10**). By taking one hybrid NSL (HQ_j) at a time, we examine its hybrid vector ($HQ_j.hybridVector$) to see if the current element of this hybrid vector ($HQ_j.hybridVector[i]$) has the value of 'd' (**line 4**). If not, we proceed to the next hybrid NSL of HQ to evaluate the value stored at the current element of its hybrid vector.

However, if the current element of the currently processed hybrid vector contains the value of 'd', this means we need to break down this 'd' value into its corresponding boolean values of '1' and '0'. In other words, we need to break down this hybrid vector

Algorithm 12.4 ConvertHybridToBooleanNSL($HQ, signalCount$)

```
1: for (i ← 0 to signalCount − 1) do
2:     tmpHQ, tmpnsl ← ∅
3:     for (j ← 0 to |HQ| − 1) do
4:         if (HQ_j.hybridVector[i] = d) then
5:             tmpnsl ← HQ_j
6:             tmpnsl.hybridVector[i] ← 1
7:             HQ_j.hybridVector[i] ← 0
8:             tmpHQ ← tmpHQ ∪ {tmpnsl}
9:         end if
10:    end for
11:    if (tmpHQ ≠ ∅) then
12:        HQ ← HQ ∪ {tmpHQ}
13:    end if
14: end for
```

into two vectors, where 'd' should be replaced by '1' in one vector, and '0' in the other vector.

In order to do that, we copy the current hybrid NSL to *tmpnsl* (**line 5**). Then, we replace 'd' with '1' in the current element of *tmpnsl*'s hybrid vector (**line 6**). Likewise, we overwrite 'd' with '0' in the current element of HQ_j's hybrid vector that we are currently processing (**line 7**). After that, we add the hybrid NSL of *tmpnsl* to *tmpHQ* (**line 8**).

We repeat these steps (**lines 5-8**) for each hybrid NSL whose hybrid vector has the value of 'd' for the current element. Once we get rid of d's in all the hybrid vectors for the current element, we add the set of converted hybrid NSL of *tmpHQ* to HQ (**line 12**). We do this only if at least one hybrid vector had 'd' for the current element, i.e. the set *tmpHQ* is not empty (**line 11**).

After processing all hybrid NSL of HQ for the current element, we proceed to process the next element of the hybrid vectors. Once we have processed all the elements, HQ will contain only boolean NSL when the algorithm ends.

Identify Nondeterministic Next State Logic

Algorithm 12.5 realizes **Step 3** of Section 11.3.5. Specifically, this algorithm performs design check **DR-13** to identify the presence of any nondeterministic boolean NSL at a given state of the FSM. Also, it detects and removes any duplicate boolean NSL.

In addition to the variables defined by Algorithm 12.3, Algorithm 12.5 makes use of one local variable.

- *identical*: This boolean flag is set to *False* if two boolean vectors that we are currently comparing are not identical. Otherwise, it retains its default value of

Algorithm 12.5 IdentifyNondeterministicNSL($HQ, signalCount$)

```
1: for (i ← 0 to |HQ| − 1) do
2:     for (j ← (|HQ| − 1) down to (i + 1)) do        // looping backwards, j − 1 in
                                                      // every iteration
3:         identical ← True
4:         for (k ← 0 to signalCount − 1) do
5:             if (HQ_i.hybridVector[k] ≠ HQ_j.hybridVector[k]) then
6:                 identical ← False
7:             end if
8:         end for
9:         if (identical) then
10:            if (HQ_i.q′ ≠ HQ_j.q′) then
11:                print ("Error! At any given state of the FSM, the next state
                   conditions specified by all input vectors must be deterministic.")
12:                return False
13:            else
14:                HQ ← HQ − HQ_j
15:            end if
16:        end if
17:    end for
18: end for
19: return True
```

$True$.

In order to detect whether or not the set HQ contains nondeterministic or duplicate boolean NSL, we need to compare the boolean vectors as well as their destination states. Algorithm 12.5 uses nested **for** loops to perform this comparison.

The outer **for** loop starts by taking the first boolean NSL that is present at index 0 in HQ, and loops forward in every iteration (**lines 1-18**). The inner **for** loop begins with the last element of HQ, and loops backward in every iteration (**lines 2-17**). Before comparing any two boolean vectors, we set the *identical* boolean flag to $True$, assuming that the boolean vectors we are about to compare are identical (**line 3**).

We loop through all elements of the two boolean vectors that we need to compare (**lines 4-8**). We compare the boolean vectors by comparing the values of their corresponding elements (**line 5**). If they have different values for at least one corresponding element, we set *identical* to $False$ (**line 6**). This indicates that the currently compared boolean vectors are not identical.

After comparing all elements of the boolean vectors, if *identical* remains $True$, this implies that the two boolean vectors are identical. If so (**line 9**), we compare the destination states of these identical boolean vectors. If their destination states are different (**line 10**), this means we have nondeterministic NSL in HQ and the

Algorithm 12.6 RemoveInvalidNSL(HQ, Σ_{dis})

```
1: for all (σ ∈ Σ_dis) do
2:     for (i ← (|HQ| − 1) down to 0) do        // looping backwards, i − 1 in
                                                // every iteration
3:         if (HQ_i.hybridVector[μ(σ)] = 1) then
4:             HQ ← HQ − HQ_i
5:         end if
6:     end for
7: end for
```

design requirement **DR-13** fails. In this case, we return $False$ to Algorithm 12.3 after displaying an error message (**lines 11-12**).

However, if two identical boolean vectors have the same destination states, this indicates that we have deterministic but duplicate boolean NSL in HQ. Therefore, we remove the latter instance of this duplicate boolean NSL from HQ (**line 14**). In this way, at the successful completion of this algorithm, the set HQ contains only unique boolean NSL.

Remove Invalid Next State Logic

Algorithm 12.6 performs **Step 4** of Section 11.3.5 by removing the invalid NSL at a given state of the FSM. Please refer to Algorithm 12.3 for the definition of variables used in Algorithm 12.6.

Algorithm 12.6 uses a nested **for** loop to perform the desired task. The outer **for** loop loops through the set Σ_{dis} that contains IO signals whose outputs are set to $False$ at the current FSM state (**lines 1-7**). By taking one disabled IO signal at a time, the inner **for** loop loops through all the boolean NSL of the set HQ (**lines 2-6**). This **for** loop begins with the last element of HQ, and loops backward in every iteration.

If the current IO signal appears as '1' in the boolean vector of the currently processed NSL (**line 3**), we remove this boolean NSL from HQ (**line 4**). After removing all the invalid NSL from HQ for the currently disabled IO signal, we take the next disabled IO signal from Σ_{dis}, and repeat the same steps. At the successful termination of this algorithm, HQ contains only valid boolean NSL that could occur in the physical system.

Check for Valid Next State Logic

Algorithm 12.7 is developed to verify design requirement **DR-11** at a given state of the FSM. In addition to the variables defined by Algorithm 12.3, Algorithm 12.7 uses the following local variable:

Algorithm 12.7 CheckForValidNSL(HQ, Σ_{en})

```
1: for all (σ ∈ Σ_en) do
2:     hasNSL ← False
3:     for (i ← 0 to |HQ| − 1) do
4:         if (HQ_i.hybridVector[μ(σ)] = 1) then
5:             hasNSL ← True
6:         end if
7:     end for
8:     if (¬ hasNSL) then
9:         print ("Error! At any given state of the FSM, if the output of an IO
           signal is set to True, then it must show up, either as occurring (1) or
           as a Don't Care (d), in at least one of the valid next state conditions,
           that are represented by the input vectors specified at that state.")
10:        return False
11:    end if
12: end for
13: return True
```

- $hasNSL$: This flag is set to $True$ if an IO signal, whose output is set to $True$ at an FSM state, shows up as '1' in at least one of the boolean vectors that are defined at this state. Otherwise, it retains its default value of $False$.

Algorithm 12.7 uses a nested **for** loop to check **DR-11**. The outer **for** loop loops through the set Σ_{en} that contains IO signals whose outputs are set to $True$ at the current FSM state (**lines 1-12**). Before checking **DR-11** for each IO signal, we set the $hasNSL$ flag to $False$, assuming that the current IO signal fails **DR-11** (**line 2**).

For each IO signal in Σ_{en}, the inner **for** loop loops through all the boolean NSL stored in HQ (**lines 3-7**). If the current IO signal appears as '1' in the boolean vector of the currently processed NSL (**line 4**), we set $hasNSL$ to $True$ (**line 5**). This indicates that the current IO signal satisfies **DR-11**.

However, after looping through all the NSL of HQ, if $hasNSL$ remains $False$, this implies that the currently enabled IO signal does not occur in any of the boolean NSL of HQ. If so (**line 8**), **DR-11** fails, and we return $False$ to Algorithm 12.3 after printing an error message (**lines 9-10**).

If **DR-11** is satisfied for the current IO signal, we take the next enabled IO signal from Σ_{en}, and repeat the same steps. The successful execution of this algorithm indicates that **DR-11** is satisfied at the current state of the FSM.

12.2.3 Generate TDES Supervisor

Algorithm 12.8 performs the process of generating a TDES supervisor from an individual Moore FSM. It does so by creating and populating the quintuple of the supervisor $\mathbf{S} = (X, \Sigma_{\mathbf{S}}, \xi, x_o, X_m)$, where X is the *state set*, $\Sigma_{\mathbf{S}}$ is the *event set*, $\xi : X \times \Sigma_{\mathbf{S}} \to X$ is the *partial transition function*, x_o is the *initial state*, and X_m is the *set of marked states*. Specifically, Algorithm 12.8 realizes the FSM-TDES translation logic described in Sections 11.3.1−11.3.4 and **Steps 5-6** of Section 11.3.5. Please refer to these sections to refresh your memory about the translation method, as we will not restate any part of this method here.

Local Variables

In addition to the variables introduced in Section 12.2, Algorithm 12.8 defines and uses the following local variables to generate a supervisor:

- *stateNumber*: This integer variable is used while creating names for new non-sampled (intermediate) states that we add to the supervisor during the translation process.
- $X_{related}$: This set keeps track of the non-sampled states of the supervisor that are *related* to the currently processed boolean vector. Basically, these are the non-sampled states that get processed (added/reused) while we are generating all possible sequences of concurrent string transitions corresponding to the current boolean vector.
- X_{pend}: This set contains the states of the supervisor that are *pending* to be processed, as we are in the process of generating all possible sequences of concurrent string transitions from the current boolean vector.
- T_{poss}: The partial function $T_{poss} : X \to \text{Pwr}(\Sigma_{\mathbf{S}})$ maps the states of a supervisor to the events that are *possible* at these states with respect to the currently processed boolean vector. Sometimes, we will treat this function like the set $T_{poss} \subseteq X \times \text{Pwr}(\Sigma_{\mathbf{S}})$.

  We will use the information stored in T_{poss} to define the transitions in the supervisor. Also, T_{poss} will enable us to reuse non-sampled states, while we are in the process of generating all possible sequences of concurrent strings from the boolean vector that we are currently processing.
- T_{def}: The partial function $T_{def} : X \to \text{Pwr}(\Sigma_{\mathbf{S}})$ maps the states of a supervisor to the events that are *defined* at these states. In other words, for a given state x of the supervisor, this function returns the set of events for which the transitions already exist at x. This function will sometimes be treated like the set $T_{def} \subseteq X \times \text{Pwr}(\Sigma_{\mathbf{S}})$.

  We will use the information stored in T_{def} to make sure that the supervisor remains deterministic, and we do not add any nondeterministic or duplicate transitions to the transition function ξ.

- Σ_{uncont}: This set contains the uncontrollable events that are present in the occurrence image of the concurrent string that we are currently generating. In terms of an FSM, Σ_{uncont} stores the input signals that show up as '1' in the currently processed boolean vector.
- T_{uncont}: The partial function $T_{uncont} : X \rightarrow \text{Pwr}(\Sigma_u)$ maps the states of a supervisor to the uncontrollable events that should be selflooped at these states, if they are not already defined. We will sometimes treat this function like the set $T_{uncont} \subseteq X \times \text{Pwr}(\Sigma_u)$.

  If the currently processed boolean vector is not part of a **GDC** transition, we will use Σ_{uncont} to populate T_{uncont} for each of the related non-sampled states. In case of **GDC**, we will populate T_{uncont} for the source sampled state and the related non-sampled states using $fsm.\Sigma_{In}$.
- T_{tick}: Let $X_{samp} \subseteq X$ be the set of sampled states of the supervisor. This partial function $T_{tick} : X_{samp} \rightarrow \text{Pwr}(X)$ maps the sampled states of a supervisor to their related non-sampled states. We will sometimes treat this function like the set $T_{tick} \subseteq X_{samp} \times \text{Pwr}(X)$.

  At the end of the translation process, if a *tick* transition does not already exist at any non-sampled state, we will use the information stored in T_{tick} to define a *tick* transition at the non-sampled state that goes back to its source sampled state. Please refer to **Step 6** of Section 11.3.5 for details.
- $addedToPend$: This boolean flag is set to $True$ if a non-sampled destination state $x' \in X$ is recently added to X_{pend}, as we are in the process of defining a $\sigma \in \Sigma_{\mathbf{S}}$ transition at state $x \in X$ in the supervisor.
- $futureExists$: This boolean flag is set to $True$ if there exists a non-sampled state $x' \in X_{related}$ that has the same future, i.e. same set of events possible, as the destination state of the currently processed $\sigma \in \Sigma_{\mathbf{S}}$ transition. In this case, instead of adding a new non-sampled state to the supervisor, we will reuse x' by making it the destination state for the σ transition.

Initializing the Translation Process

For the current Moore FSM, Algorithm 12.8 begins to generate a TDES supervisor by populating its event set $\Sigma_{\mathbf{S}}$ at **line 4**. We assign the initial state x_o of the supervisor at **line 5**. At **lines 6-10**, we generate the set of marked states X_m of the supervisor according to the marking option selected by the designers while initiating the FSM-TDES translation process in DESpot (see Section 11.3.4 for details).

At **lines 11-110**, we loop through all states of the current FSM ($fsm.Q$). By taking one FSM state q at a time, we create the state set X, define all possible sequences of the concurrent string transitions, and construct the transition function ξ of the supervisor by processing the boolean NSL defined at q.

At **lines 12-14**, we add the currently processed FSM state q to the state set X, if

Algorithm 12.8 GenerateTDESSupervisor $(fsm, Q_{nsz}, Q_{DC}, \mathbf{S})$ Part I

```
1: X_related, X_pend, T_poss, T_def, T_tick, T_uncont, Σ_uncont ← ∅
2: X, Σ_S, ξ, x_o, X_m ← ∅
3: stateNumber ← 0
4: Σ_S ← fsm.Σ_IO ∪ fsm.Σ_In ∪ {τ}
5: x_o ← fsm.resetState
6: if (fsm.marking = Initial) then
7:     X_m ← {x_o}
8: else if (fsm.marking = Sampled) then
9:     X_m ← fsm.Q
10: end if
11: for all (q ∈ fsm.Q) do
12:     if (q ∉ X) then
13:         Push(X, q)
14:     end if
15:     Push(T_poss, (q, ∅))
16:     Push(T_def, (q, ∅))
17:     Push(T_tick, (q, ∅))
18:     if (Q_DC(q)) then
19:         Push(T_uncont, (q, fsm.Σ_In))
20:     end if
21:     HQ ← Q_nsz(q)
22:     for (i ← 0 to |HQ| − 1) do
23:         for all (σ ∈ fsm.Σ_IO ∪ fsm.Σ_In) do
24:             if (HQ_i.hybridVector[μ(σ)] = 1) then
25:                 T_poss(q) ← T_poss(q) ∪ {σ}
26:                 if (σ ∈ fsm.Σ_In) then
27:                     Σ_uncont ← Σ_uncont ∪ {σ}
28:                 end if
29:             end if
30:         end for
31:         Push(X_pend, q)
32:         while (X_pend ≠ ∅) do
33:             x ← Pop(X_pend)
34:             if (x = q ∧ T_poss(x) = ∅) then
35:                 T_poss(x) ← {τ}
36:             end if
37:             for all (σ ∈ T_poss(x)) do
38:                 futureExists ← False
39:                 addedToPend ← False
40:                 if (σ ≠ τ) then
```

Algorithm 12.8 GenerateTDESSupervisor $(fsm, Q_{nsz}, Q_{DC}, \mathbf{S})$ Part II

41: **if** $(\sigma \in T_{def}(x))$ **then**
42: $x' \leftarrow \xi(x, \sigma)$
43: **if** $(x' \notin X_{related})$ **then**
44: Push$(X_{related}, x')$
45: Push(X_{pend}, x')
46: $addedToPend \leftarrow True$
47: **end if**
48: **else**
49: **for all** $(x' \in X_{related})$ **do**
50: **if** $(x \neq x' \ \wedge \ [(T_{poss}(x) - \{\sigma\} = T_{poss}(x')) \ \vee$ $(T_{poss}(x) - \{\sigma\} = \emptyset \ \wedge \ T_{poss}(x') = \{\tau\})])$ **then**
51: Push$(\xi, (x, \sigma, x'))$
52: $T_{def}(x) \leftarrow T_{def}(x) \cup \{\sigma\}$
53: $futureExists \leftarrow True$
54: **end if**
55: **end for**
56: **if** $(\neg\, futureExists)$ **then**
57: $x' \leftarrow$ "x"+$stateNumber$
58: $stateNumber \leftarrow stateNumber + 1$
59: Push(X, x')
60: Push$(\xi, (x, \sigma, x'))$
61: $T_{def}(x) \leftarrow T_{def}(x) \cup \{\sigma\}$
62: Push$(T_{poss}, (x', \emptyset))$
63: Push$(T_{def}, (x', \emptyset))$
64: Push$(T_{uncont}, (x', \emptyset))$
65: Push$(X_{related}, x')$
66: Push(X_{pend}, x')
67: $addedToPend \leftarrow True$
68: **end if**
69: **end if**
70: **if** $(addedToPend)$ **then**
71: **if** $(T_{poss}(x) - \{\sigma\} = \emptyset)$ **then**
72: $T_{poss}(x') \leftarrow \{\tau\}$
73: **else**
74: $T_{poss}(x') \leftarrow T_{poss}(x) - \{\sigma\}$
75: **end if**
76: **end if**
77: **else**
78: **if** $(HQ_i.q' \notin X)$ **then**
79: Push$(X, HQ_i.q')$
80: **end if**

Algorithm 12.8 GenerateTDESSupervisor $(fsm, Q_{nsz}, Q_{DC}, \mathbf{S})$ Part III

81: Push$(\xi, (x, \tau, HQ_i.q'))$
82: $T_{def}(x) \leftarrow T_{def}(x) \cup \{\tau\}$
83: **end if**
84: **end for**
85: **end while**
86: **if** $(Q_{DC}(q))$ **then**
87: **for all** $(x \in X_{related})$ **do**
88: $T_{uncont}(x) \leftarrow T_{uncont}(x) \cup fsm.\Sigma_{In}$
89: **end for**
90: **else if** $(\Sigma_{uncont} \neq \emptyset)$ **then**
91: **for all** $(x \in X_{related})$ **do**
92: **for all** $(\sigma \in \Sigma_{uncont})$ **do**
93: **if** $(\sigma \notin T_{uncont}(x))$ **then**
94: $T_{uncont}(x) \leftarrow T_{uncont}(x) \cup \{\sigma\}$
95: **end if**
96: **end for**
97: **end for**
98: **end if**
99: **for all** $(x \in X_{related})$ **do**
100: **if** $(x \notin T_{tick}(q))$ **then**
101: $T_{tick}(q) \leftarrow T_{tick}(q) \cup \{x\}$
102: **end if**
103: **end for**
104: $\Sigma_{uncont}, T_{poss}(q) \leftarrow \emptyset$
105: **for all** $(x \in X_{related})$ **do**
106: $T_{poss}(x) \leftarrow \emptyset$
107: **end for**
108: $X_{related} \leftarrow \emptyset$
109: **end for**
110: **end for**
111: **while** $(T_{uncont} \neq \emptyset)$ **do**
112: $(x, \Sigma_{uncont}) \leftarrow$ Pop(T_{uncont})
113: **for all** $(\sigma \in \Sigma_{uncont})$ **do**
114: **if** $(\sigma \notin T_{def}(x))$ **then**
115: Push$(\xi, (x, \sigma, x))$
116: $T_{def}(x) \leftarrow T_{def}(x) \cup \{\sigma\}$
117: **end if**
118: **end for**
119: **end while**

Algorithm 12.8 GenerateTDESSupervisor $(fsm, Q_{nsz}, Q_{DC}, \mathbf{S})$ Part IV

```
120: while (T_tick ≠ ∅) do
121:     (q, X_related) ← Pop(T_tick)
122:     for all (x ∈ X_related) do
123:         if (τ ∉ T_def(x)) then
124:             Push(ξ, (x, τ, q))
125:             T_def(x) ← T_def(x) ∪ {τ}
126:         end if
127:     end for
128: end while
129: for all (q ∈ fsm.Q) do
130:     if (τ ∉ T_def(q)) then
131:         Push(ξ, (q, τ, q))
132:         T_def(q) ← T_def(q) ∪ {τ}
133:     end if
134: end for
135: S ← (X, Σ_S, ξ, x_o, X_m)
```

q does not already exist in X. At **line 15**, we add a tuple for state q of the supervisor to the set T_{poss}. The second element of this tuple is an empty set. Later on (at **line 25**), while processing the boolean NSL defined at q, we will update the set of events that are possible at q in T_{poss}. Similarly, at **line 16**, we add a tuple to the set T_{def} corresponding to q. We initialize the second element of this tuple with an empty set, indicating that no events are currently defined at q in the supervisor. Likewise, at **line 17**, we add q's tuple to the set T_{tick} while initializing its second element with an empty set.

Lines 18-20 are executed only if a **GDC** transition exists at state q of the FSM, i.e. $Q_{DC}(q) = True$. Please recall that in the case of a **GDC** transition, our translation method (Section 11.3) does not define any state changing transitions for the input signals of the FSM. This implies that we need to add selfloops of all uncontrollable events (input signals) at state q of the supervisor. In order to do that at the end of this algorithm, we add a tuple of q along with the set of current FSM's input signals ($fsm.\Sigma_{In}$) to the set T_{uncont}.

At **line 21**, we retrieve the set of all NSL defined at q from Q_{nsz}, and store it in the HQ variable. At **lines 22-109**, we loop through the set HQ, take one NSL at a time, and then use it to generate all possible sequences of concurrent string transition(s) at state q in the supervisor.

Lines 23-30 update $T_{poss}(q)$ and populate Σ_{uncont} by looping through all signals of the FSM. Specifically, we take one signal σ at a time (**line 23**), and determine whether or not σ shows up in the boolean vector of the currently processed NSL as a '1'. If it does (**line 24**), we add σ to the set of events that are possible at q (**line 25**).

Moreover, if σ is an input signal (uncontrollable event, **line 26**), we add σ to Σ_{uncont} (**line 27**). Afterwards, we will use Σ_{uncont} to update T_{uncont}, only if **GDC** does not exist at q. Please recall from **Algorithm 12.3** that in the case of **GDC**, we assigned the value of '0' to each element of the vector that corresponds to the input signals of the FSM. Therefore, Σ_{uncont} will be empty for **GDC** anyway.

At **line 31**, we add the sampled state q of the supervisor to X_{pend}. This indicates that q is pending to be processed by the translation algorithm, i.e. we need to define appropriate transitions at q by using the information stored in $T_{poss}(q)$. **Lines 32-85** loop through the set X_{pend} until it becomes empty. At **line 33**, we remove one state x from X_{pend} for processing.

Lines 34-36 are executed if the boolean vector that we are currently processing represents a **TICK** (*tick* only) transition. At **line 34**, we use an **if** statement to determine if we have a **TICK** transition by checking two conditions: 1) the state x that we are currently processing is a sampled state (since **TICK** transition can be defined only at a sampled state), and 2) no activity events are possible at x ($T_{poss}(x)$ will be empty corresponding to **TICK** of an FSM). If both conditions evaluate to $True$, we add a *tick* event to $T_{poss}(x)$ to indicate that *tick* is the only event that is possible at x (**line 35**).

At **lines 37-84**, we loop through each event σ that is possible at state x of the supervisor. Before initiating the process to define a σ transition at x, we set the boolean flags, $futureExists$ and $addedToPend$, to $False$ (**lines 38-39**). If σ is an activity event (**line 40**), we execute **lines 41-76** to define a σ transition at x. Otherwise (**line 77**), we execute **lines 78-82** to define a *tick* transition at x.

Defining a non-*tick* Transition

Our strategy to define a non-*tick* σ transition at x is to first determine whether or not a σ transition already exists at x. If it does, we simply reuse the existing σ transition. However, if σ is not already defined at x, then we define a new σ transition at x. We do this either by using an existing non-sampled state as the destination state for the σ transition, or by adding a new non-sampled state to the supervisor. Now we will discuss how the algorithm realizes this logic to define a non-*tick* σ transition at x.

In order to evaluate whether or not a σ transition exists at x, we use the set T_{def}. If a σ transition is already defined at x (**line 41**), we execute **lines 42-47**. At **line 42**, we get the destination state of this σ transition from the partial transition function ξ and store it in x'. Then, we determine whether or not this non-sampled destination state x' is present in $X_{related}$. If not (**line 43**), we add x' to sets $X_{related}$ and X_{pend} (**lines 44-45**). As we have added a new element to X_{pend}, we update the value of the boolean flag $addedToPend$ to $True$ (**line 46**).

Please note that due to the check performed at **line 43**, we add a non-sampled state x' to $X_{related}$ and X_{pend} only when we come across x' for the first time while processing the current boolean vector. If we encounter x' multiple times while defining different possible sequences of the concurrent string with the same occurrence image

corresponding to the current boolean vector, we do not add x' again to neither $X_{related}$ nor X_{pend}.

It is also notable that at **line 43**, we are only checking for the existence of x' in $X_{related}$, and not in X_{pend}. The reason is that we always add a non-sampled state x' to $X_{related}$ and X_{pend} together. However, we remove states one by one from X_{pend} for processing. If we check X_{pend} for the existence of x', we might end up adding x' to X_{pend} again, even though we have already processed x' after removing it from X_{pend} earlier. But this is not the case with $X_{related}$. Since we do not remove any non-sampled states from $X_{related}$ while processing the current boolean vector, the set $X_{related}$ keeps track of all the non-sampled states that we have encountered till now while processing the current boolean vector. This is regardless of whether these non-sampled states have already been processed or are pending to be processed by the algorithm.

Lines 49-68 are executed if a σ transition does not already exist at x (**line 48**). Before defining a σ transition at x, we first need to decide about its destination state, i.e. do we need to add a new non-sampled state to the supervisor as the destination state for σ, or does there already exist a non-sampled state that we can reuse as the destination state for the σ transition?

In order to answer these questions, we loop through the set $X_{related}$ (**lines 49-55**). For each state x' of $X_{related}$, we compare the future of x' with the future of the destination state for the σ transition (**line 50**). In order to do this comparison, we first make sure that we are not comparing the currently processed state x to itself, i.e. $x \neq x'$. After that, we specify two main conditions in the **if** statement to compare the futures. If any of these conditions evaluates to $True$, this means the futures of x and x' are same, and we can reuse the existing state x' as the destination state for the σ transition.

The first condition of the **if** statement after the AND operator ($\wedge$) will be $True$ if the set of events possible at x' ($T_{poss}(x')$) is equal to the set of events that will be possible at the destination state for σ transition. We determine the set of events that are possible at the destination state for σ transition by excluding σ from the set of events that are possible at x ($T_{poss}(x) - \{\sigma\}$).

It is noteworthy that if σ is the only activity event possible at x, i.e. $T_{poss}(x) = \{\sigma\}$, then we have $T_{poss}(x) - \{\sigma\} = \emptyset$. In this case, we do not need to find a state $x' \in X_{related}$ that does not have any events possible, i.e. $T_{poss}(x') = \emptyset$. Rather, we should look for some x' that has only *tick* event possible, i.e. $T_{poss}(x') = \{\tau\}$. This is because a concurrent string always ends with a *tick*. If any such x' exists in $X_{related}$, we should reuse this x' as the destination state for the σ transition. We evaluate this scenario by defining a composite condition after the OR operator ($\vee$) in the **if** statement.

Please note that $X_{related}$ contains the non-sampled states that are related only to the *current* boolean vector. Therefore, while reusing x' in the above-mentioned scenario, there is no need to examine the destination state of the *tick* transition that

is possible at x'. In other words, it is guaranteed that this *tick* will always take us to the correct destination sampled state.

If we find a state x' in $X_{related}$ that we can reuse as the destination state for the σ transition (**line 50**), then we execute **lines 51-53** to perform the desired steps. Precisely, we define a σ transition at x and add the transition (x, σ, x') to the transition function ξ (**line 51**). As we have defined a new σ transition at x, we add this information to T_{def} by updating the set of events for which the transitions exist at x (**line 52**). Accordingly, we update the $futureExists$ boolean variable to $True$ (**line 53**).

Lines 56-68 are executed if $futureExists = False$, i.e. there does not exist a non-sampled state in $X_{related}$ that has the same future as the destination state for the σ transition. In this case, we need to add a new non-sampled state to the supervisor. At **line 57**, we construct the name of this new state by appending $stateNumber$ to "x", and store it in x'. For example, if we are adding the first non-sampled state to the supervisor, then we create the state name "$x0$", and assign this name to x'. At **line 58**, we increment the $stateNumber$ variable by 1 in order to use the subsequent integer value while creating the name for the next non-sampled state.

At **line 59**, we add the newly created non-sampled state x' to X. At **line 60**, we define a σ transition at x, and add the transition (x, σ, x') to ξ. **Line 61** adds the information about the definition of this σ transition to T_{def} by updating the set corresponding to x in T_{def}. As we have added a new state x' to the supervisor, we add new tuples for x' to sets T_{poss}, T_{def}, and T_{uncont} at **lines 62-64**. Also, we add x' to $X_{related}$ and X_{pend} at **lines 65-66**. Accordingly, we update the boolean flag $addedToPend$ to $True$ at **line 67**.

Now that we have completed the processing for defining the non-*tick* σ transition at x, we need to check the value of the $addedToPend$ flag. This is required in order to determine whether or not we have come across the destination state x' for the first time while processing the current boolean vector. If this is is the first time that we are dealing with x' while processing the current boolean vector, i.e. $addedToPend = True$ (**line 70**), we should specify the future of x' by updating the set of events that are possible at x' in T_{poss}. We do this at **lines 71-75**.

We consider two cases while specifying the future of x'. If σ was the only event possible at x, then we have $T_{poss}(x) - \{\sigma\} = \emptyset$ (**line 71**). This indicates that we have defined all activity events in the concurrent string that we are generating corresponding to the current boolean vector. In this case, we add a *tick* event to $T_{poss}(x')$ to signify that only *tick* is possible at x' (**line 72**). On the other hand, there could be some other activity events possible at x besides σ. If so (**line 73**), we specify the set of events possible at x' by excluding σ from the set of events that are possible at x (**line 74**).

Defining a *tick* Transition

If the currently processed event is a *tick* event, i.e. $\sigma = \tau$ (**line 77**), then we define

a *tick* transition at x by performing the required steps at **lines 78-82**. It is notable that a concurrent string always ends with a *tick*. Therefore, the destination state of the current boolean vector becomes the destination state of this *tick* transition.

At **lines 78-80**, we add the destination sampled state of the *tick* transition ($HQ_i.q'$) to X, if it does not already exist in X. At **line 81**, we define a *tick* transition at x and add the transition $(x, \tau, HQ_i.q')$ to ξ. At **line 82**, we update $T_{def}(x)$ by adding a *tick* event to the set of events that are defined at x.

It is worth clarifying that we are defining a *tick* transition at x without checking whether or not a *tick* transition already exists at x. This is because of the fact that our translation method (Section 11.3), and hence this translation algorithm, does not reuse any non-sampled states while defining concurrent string transitions corresponding to two different boolean vectors defined at x. The only exception to this is when we have to reuse a non-sampled state in order to keep the TDES deterministic. Even in that case, we cannot have a *tick* transition already possible/defined at x that is going to the same/different destination sampled state. Please refer to **Step 5** of Section 11.3.5 for details.

After processing the current event σ, we proceed to take the next event that is possible at x from $T_{poss}(x)$ to generate its transition at x. Once we have defined transitions for all the events that are possible at x, this completes the processing of the current state x. After that, we extract another state from X_{pend} for processing. We then repeat the steps discussed above to define transitions for the events that are possible at this subsequent state taken from X_{pend}. We repeat this process until all states of the supervisor that are related to the current boolean vector have been processed, i.e. $X_{pend} = \emptyset$.

At this point, $X_{pend} = \emptyset$ signifies that we have generated all possible sequences of concurrent string transitions in the supervisor corresponding to the current boolean vector. It is notable that if the current boolean vector represents a non-**TICK** transition, then we have defined only state changing transitions in the supervisor till now with respect to the current boolean vector.

Preparing to Add Selfloop Transitions

Please recall that at **Step 6** of Section 11.3.5, our translation method adds transitions for *tick* and uncontrollable events at the appropriate states of the supervisor, if needed. In order to perform this step at the end of this algorithm, we update our sets at **lines 86-103** to store the required information corresponding to the currently processed boolean vector.

At **lines 86-98**, we update the set T_{uncont}. Specifically, for each non-sampled state that gets processed as part of the current boolean vector, i.e. for each state $x \in X_{related}$, we update its corresponding set of uncontrollable events in T_{uncont}.

In order to correctly update T_{uncont}, we begin by checking whether or not **GDC** is defined at the source sampled state q of the current boolean vector. If **GDC** exists at q (**line 86**), we execute the **for** loop to update T_{uncont} (**lines 87-89**). Precisely, for

each non-sampled state x of $X_{related}$, we add the set of current FSM's input signals ($fsm.\Sigma_{In}$) to $T_{uncont}(x)$.

However, if **GDC** does not exist at q, we update the set of uncontrollable events in T_{uncont} for the related non-sampled states at **lines 91-97**. We perform these steps if at least one input signal appears as '1' in the current boolean vector, i.e. the set Σ_{uncont} is not empty (**line 90**). In this case, we use a nested **for** loop to loop through the set of related states ($X_{related}$), and the set of uncontrollable events (Σ_{uncont}) that might need to be selflooped at these related states at the end of this algorithm (**lines 91-97**). For each state x of $X_{related}$, we add an event σ from Σ_{uncont} to $T_{uncont}(x)$, if σ does not already exist in $T_{uncont}(x)$. This makes sure that we do not add any duplicate instances of uncontrollable events to $T_{uncont}(x)$.

At **lines 99-103**, we update the set T_{tick} corresponding to the source sampled state q of the current boolean vector. Please recall that $X_{related}$ contains all the non-sampled states that get processed as part of the current boolean vector. Therefore, we add each state x of $X_{related}$ to $T_{tick}(q)$. We do this only if x does not already belong to $T_{tick}(q)$ in order to guarantee that $T_{tick}(q)$ does not contain any duplicate non-sampled states.

After updating our sets with respect to the current boolean vector, we now reset some of our variables at **lines 104-108**. We do this in order to prepare these variables for the processing of the next boolean vector. First, we empty out the set Σ_{uncont} at **line 104**. Since our translation method does not reuse any non-sampled states across different boolean vectors, we need to reset the set of events that are possible at a given state in T_{poss}. Precisely, we reset the set of events that are possible at q in T_{poss} at **line 104**. At **lines 105-107**, we empty out the set of events in T_{poss} for all the non-sampled states that get processed as part of the current boolean vector, i.e. for all states of $X_{related}$. Finally, we remove all non-sampled states from $X_{related}$ at **line 108**.

After resetting the variables, we take the next boolean NSL defined at q from HQ and start processing it. Once we have processed all the NSL corresponding to state q, we proceed to the next FSM state. In other words, we take the next FSM state from $fsm.Q$, and then process its boolean NSL one by one in order to generate the corresponding concurrent string transitions in the supervisor. After processing all FSM states, we now have a supervisor that is complete with respect to the state changing transitions of the activity events.

Adding Selfloops of Uncontrollable and *tick* Events

The rest of this algorithm uses the information stored in our sets to realize **Step 6** of Section 11.3.5. At **lines 111-119**, we loop through the set T_{uncont} to add selfloops of uncontrollable events in the supervisor, if needed. At **line 112**, we extract one element of T_{uncont}. For every uncontrollable event σ that we need to add as a selfloop at state x (**line 113**), we first determine whether or not σ is already defined at x by using the set $T_{def}(x)$. If σ transition does not exist at x (**line 114**), we define a

selfloop of σ at x, and add this transition (x, σ, x) to the transition function ξ (**line 115**). As we have defined a new σ transition at x, we update $T_{def}(x)$ accordingly.

At **lines 120-128**, we loop through the set T_{tick}. We use the elements of T_{tick} to define a *tick* transition at each of the non-sampled states of the supervisor where *tick* event is not already defined. We define this *tick* transition from a non-sampled state x back to its source sampled state q. We add the transition (x, τ, q) to ξ (**line 124**), and update $T_{def}(x)$ accordingly (**line 125**).

At **lines 129-134**, we loop through the set $fsm.Q$ to add a selfloop of *tick* event at every sampled state q of the supervisor where *tick* event is not already defined. We define a *tick* transition at q by adding the transition (q, τ, q) to ξ (**line 131**). Then, we update $T_{def}(q)$ accordingly (**line 132**).

This completes the translation of the current Moore FSM into a TDES supervisor, as per the FSM-TDES translation method described in Section 11.3. Finally, we end the algorithm by assigning the five elements $(X, \Sigma_{\mathbf{S}}, \xi, x_o, X_m)$ that we have populated during this algorithm to the quintuple of the supervisor **S**.

12.3 Complexity Analysis

Now we are ready to analyze Algorithm 12.8 and determine its worst case time complexity for translating an individual Moore FSM into a TDES supervisor. Please note that in the following analysis, we will only focus on the significant operations performed by the algorithm, and ignore the constant time operations.

In order to make our discussion concise and clear, we will use the following variables as shorthand notations to refer to the variables used in Algorithm 12.8 for an individual Moore FSM:

1. Let n be the number of FSM states, i.e. $n = |fsm.Q|$.
2. Let u be the number of IO signals, $u = |fsm.\Sigma_{IO}|$.
3. Let i be the number of input signals, $i = |fsm.\Sigma_{In}|$.
4. Let s be the number of FSM signals, $s = u + i = |fsm.\Sigma_{IO} \cup fsm.\Sigma_{In}|$.

By looking at Algorithm 12.8, we note that it consists of four loops that are executed sequentially. Specifically, the algorithm has two **for** loops at **lines 11-110** (**FL-1**) and **129-134** (**FL-2**), and two **while** loops at **lines 111-119** (**WL-1**) and **120-128** (**WL-2**). We will analyze each of these additive loops one by one.

Analyzing FL-1

In Algorithm 12.8, the outermost **for** loop starting at **line 11** executes n times. Inside this loop, the nested **for** loop beginning at **line 22** runs at most 2^s times. Please recall that the variable HQ contains all the boolean NSL that are defined at an FSM state. Hence, for an FSM having s number of signals, the number of possible boolean NSL defined at an FSM state is bounded by 2^s.

Within this **for** loop, we have several **for** loops at **lines 23-30**, **87-89**, **91-97**, **99-103** and **105-107**, and a **while** loop at **lines 32-85**. Next, we analyze all these sequential loops one by one.

The **for** loop at **lines 23-30** has the worst case time complexity of $O(s)$. The two **for** loops at **lines 99-103** and **105-107** have the complexity of $O(2^s)$ each. Please recall that $X_{related}$ stores the non-sampled states of the supervisor that are *related* to the currently processed boolean vector. As per the formula defined in Section 11.3.1, the number of non-sampled states that our translation method adds corresponding to an FSM state has an upper bound of $2^s - 1$, assuming that all output signals are enabled and every IO and input signal of the FSM shows up as '1' in the currently processed boolean vector. Ignoring the constant of 1, we get the complexity of $O(2^s)$ for each of these two **for** loops.

The **for** loop at **lines 87-89** is executed only in the case of a **GDC** transition. Please recall that we assign the value of '0' to each element of the boolean vector that corresponds to an input signal of the FSM. This implies the **for** loop starting at **line 87** runs at most 2^u times. Hence, it has the worst case complexity of $O(2^u)$.

The nested **for** loop at **lines 91-97** has the worst case complexity of $O(i{\cdot}2^s)$. This is because the **for** loop starting at **line 91** has the complexity of $O(2^s)$, and the **for** loop at **lines 92-96** gives the complexity of $O(i)$.

We note that $O(s) \leq O(2^u) \leq O(2^s) \leq O(i{\cdot}2^s)$. Since the five **for** loops analyzed above are additive, their worst case time complexity after ignoring the lesser terms is $O(i{\cdot}2^s)$.

Now we will analyze the **while** loop at **lines 32-85**. The **for** loop at **lines 49-55** has the complexity of $O(2^s)$. The **for** loop starting at **line 37** runs at most s times. This implies the **for** loop at **lines 37-84** has the complexity of $O(s{\cdot}2^s)$. The **while** loop starting at **line 32** is limited by $O(2^s)$. Therefore, the **while** loop at **lines 32-85** has the worst case time complexity of $O(s{\cdot}2^s)$.

It is notable that the **while** loop and five **for** loops analyzed above are additive. Hence, they have an overall worst case time complexity of $O(s{\cdot}2^s)$, since $O(i{\cdot}2^s) \leq O(s{\cdot}2^s)$.

Based on the foregoing analysis, we deduce that the **for** loop at **lines 22-109** has the complexity of $O(s{\cdot}2^s)$. This in turn implies that the worst case time complexity of the **for** loop at **lines 11-110** (**FL-1**) is $O(n{\cdot}s{\cdot}2^s)$.

Analyzing WL-1

The **for** loop at **lines 113-118** has the complexity of $O(i)$. The **while** loop starting at **line 111** is limited by $O(n{\cdot}2^s)$. This is because T_{uncont} contains information about the uncontrollable events that need to be selflooped at the end of the translation algorithm. This information is stored corresponding to each state of the supervisor. Please recall from Section 11.3.1 that our translation method adds a maximum of $2^s - 1$ non-sampled states corresponding to each FSM state. This gives us the worst case state space of $n{\cdot}2^s$ for the translated supervisor. As a result, the worst case time

complexity of the **while** loop at **lines 111-119** (**WL-1**) is $O(n \cdot i \cdot 2^s)$.

Analyzing WL-2

The **for** loop at **lines 122-127** has the complexity of $O(2^s)$. The **while** loop starting at **line 120** is executed n times. Therefore, the time complexity of the **while** loop at **lines 120-128** (**WL-2**) is $O(n \cdot 2^s)$.

Analyzing FL-2

The **for** loop at **lines 129-134** (**FL-2**) has the complexity of $O(n)$.

Overall Complexity

We note that the two **for** loops and two **while** loops at **lines 11-110** (**FL-1**), **111-119** (**WL-1**), **120-128** (**WL-2**) and **129-134** (**FL-2**) are sequential, hence additive. It is evident that $O(n) \leq O(n \cdot 2^s) \leq O(n \cdot i \cdot 2^s) \leq O(n \cdot s \cdot 2^s)$. Consequently, we conclude that the worst case time complexity of Algorithm 12.8 is $O(n \cdot s \cdot 2^s)$.

Chapter 13

Combination Lock Example

In this chapter, we present the example of a 4-bit Combination Lock to demonstrate the application of our Moore FSM-TDES translation approach presented in Chapters 11 and 12. This example is based on the Combination Lock example to which Hamid (2014) applied his DESpot (2023) algorithms for TDES-FSM translation, but with some refined functionality.

We begin this chapter by describing the overall structure and specifications of the 4-bit Combination Lock. Then, we discuss the design of modular controllers for this lock system and express them as individual Moore FSM. This is followed by the translation of these Moore FSM into TDES supervisors using our FSM-TDES translation approach. After that, we introduce TDES plant models for the Combination Lock and discuss verification results of the closed-loop system in our $||_{SD}$ setting. Finally, we close this chapter by presenting our conclusions with respect to the correctness of our FSM-TDES translation approach using the Combination Lock example. Please see Section E.4 (Appendix E) for details on how we arrived at these conclusions.

13.1 System Description

In this section, we introduce our 4-bit Combination Lock system by presenting its structure, specifications and components.

13.1.1 Structure and Specifications

The 4-bit Combination Lock is a digital lock system that uses a 4-bit passcode to provide secured access to authentic users. In addition to the numeric pad to enter the 4-bit passcode, the combination lock has three user buttons labelled as **Enter**, **Change** and **Reset**.

The users of the Combination Lock can perform two primary tasks: 1) open the lock, and 2) change the existing 4-bit passcode. To perform either of these two tasks,

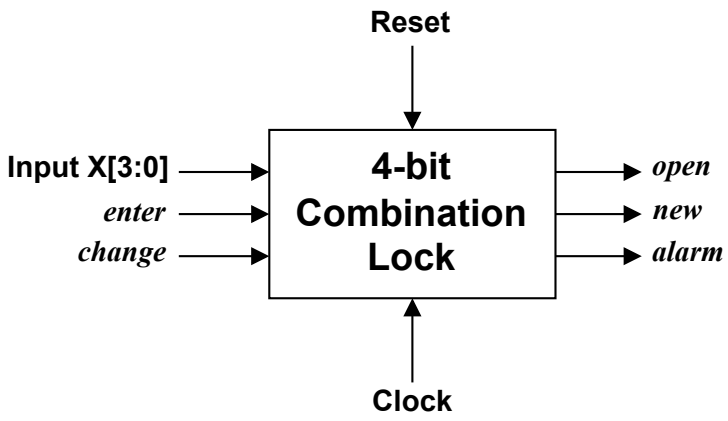


Figure 13.1: An Overview of 4-bit Combination Lock

the user must first enter the correct existing passcode and then press the appropriate button (**Enter** or **Change**). If user enters an incorrect passcode, the alarm goes off. Once the system goes to the alarm state, the normal system functionality becomes unavailable to the users. The alarm can only be cancelled by pressing the **Reset** button, which also resets the currently saved passcode to its default value.

An overview of the 4-bit Combination Lock is shown in Figure 13.1. At an abstract level, the system can receive three binary input signals from the users: *enter*, *change* and *reset*. The users need to press the appropriate buttons to provide these input signals to the system. The label of "Input X[3:0]" represents the 4-bit passcode that will be provided by the users as an input to perform their desired task. The system generates three output signals that are observable by the users: *open*, *new* and *alarm*.

Below, we describe the two functionalities of the 4-bit Combination Lock in detail.

1) Open the Lock

In order to open the combination lock, the user inputs the 4-bit passcode and sets the input signal *enter* to '1' for one clock cycle by pressing the **Enter** button. If the 4-bit passcode matches the stored combination, the system gets unlocked and the door opens. The opening of the door is represented by the output signal *open* being set to '1'. The door stays open until the user presses the **Enter** button again, i.e. sets *enter* signal to '1' for one clock cycle. At this point, the door closes (output of *open* signal is set to '0') and the system goes back to its locked state.

However, if a user tries to open the lock by entering the incorrect passcode, the alarm goes off. This is indicated by setting the output signal *alarm* to '1'. In the alarm state, the system ceases its normal operation and users cannot access the regular system functionality. The alarm can only be cancelled (*alarm* output set to '0') by pressing the **Reset** button, i.e. by setting the *reset* input signal to '1' for one clock cycle.

2) Change Saved Code

In order to change the existing passcode, the user inputs the 4-bit passcode and

sets the input signal *change* to '1' for one clock cycle by pressing the **Change** button. If the 4-bit passcode matches the stored combination, the system sets the *new* output signal to '1'. This is an indication to the user that system is ready to accept the new passcode. The user provides the new passcode and presses the **Change** button to set the *change* signal to '1' for one clock cycle. The system saves the new passcode and sets the output of *new* back to '0'.

However, if the user attempts to change the existing passcode by entering the incorrect passcode, then the alarm goes off as described above.

With respect to the input signals of the Combination Lock, we make two important assumptions. As stated earlier, users provide input signals to the system by pressing the appropriate buttons. In this context, we first assume that the input signals provided by the users have been preprocessed into *pulses* before they reach the system. In simple words, when user presses any button, its corresponding input signal should be set to '1' for exactly one clock cycle, and then it must be set to '0' for the next clock cycle. In this way, we ensure that the system sees the inputs only once rather than across multiple clock edges.

Secondly, we assume that the clock frequency is much higher than the rate at which the users could generate input signals by pressing the buttons. In other words, the clock must be fast enough that several clock cycles must have gone by between any two button presses. For example, while changing the existing passcode, the user enters the new passcode and presses the **Change** button. In this case, the clock must be sufficiently fast to detect the *change* input at the next clock edge, save the new passcode, and have the *change* signal set back to '0' for at least one clock cycle, before the user could press any other button to initiate the next task which will be detected on the following clock edge. This makes sure that the system responds correctly to the users' inputs and it does not miss anything.

13.1.2 System Components

Figure 13.2 shows the block diagram for the 4-bit Combination Lock system. In order to design the Combination Lock with the aforementioned specifications, we decompose the system into three interacting components. Below, we discuss each of these components one by one.

1) Controller expressed as a Moore FSM

The primary component of the 4-bit Combination Lock that we are interested in is the system controller. We will design and express this controller as a Moore FSM. Instead of designing a monolithic controller, we will develop three modular controllers that work together to provide the desired system functionality. We will discuss the design of each modular FSM in Section 13.2.1.

As shown in Figure 13.2, the inputs and outputs of the Combination Lock system (Figure 13.1) become the inputs and outputs of the Moore FSM. Besides

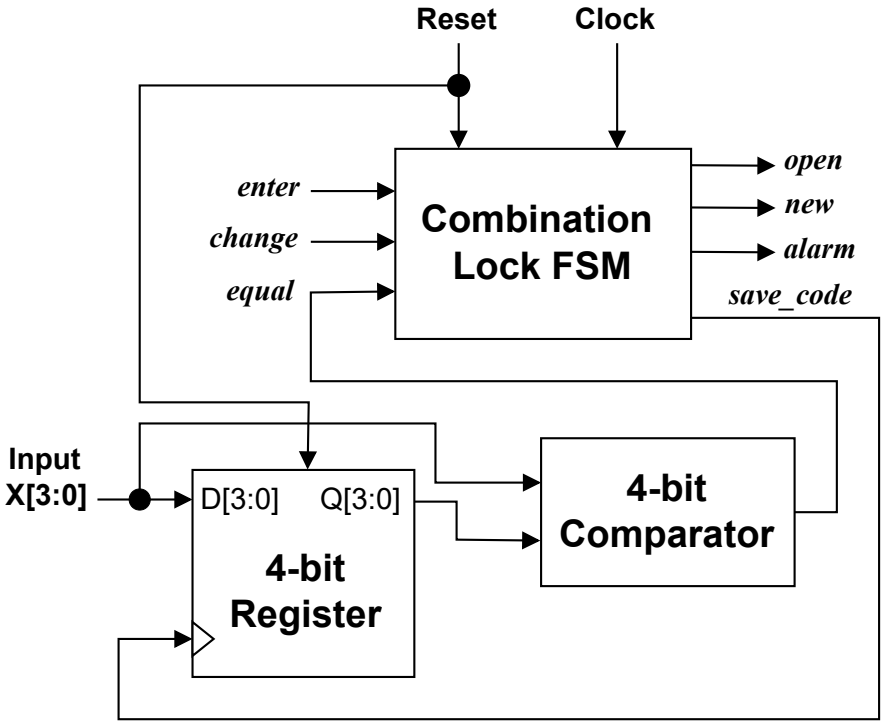


Figure 13.2: Block Diagram for 4-bit Combination Lock

these signals, we have two internal signals, *equal* and *save_code*, that facilitate interaction between the three system components.

In order to open the lock or change the current passcode, the user must first enter the system's existing passcode. Depending upon whether or not the input passcode (X[3:0]) matches the currently saved passcode, the system needs to take the appropriate future action. This matching/unmatching of the two passcodes is indicated by the *equal* signal. Precisely, if the two passcodes match (they are equal), the value of *equal* signal will be '1'. Otherwise, *equal* will be '0'. We feed this *equal* signal as an input to the FSM so that the system controller will act accordingly.

Besides the three output signals that are observable by the users, the FSM generates the output signal *save_code* that is internal to the system. While changing the existing passcode, when the user enters the new passcode and presses **Change**, the FSM sets the output signal *save_code* to '1'. This indicates to the system that the new passcode provided by the user as an input must be saved on the next clock edge. After saving the new passcode, the output of *save_code* is set back to '0'.

2) 4-bit Register

In order to store the 4-bit passcode for the Combination Lock system, we use a 4-bit register. A register is a standard digital logic element that is used to store information. Please refer to Brown and Vranesic (2013) for details.

We connect the FSM to a 4-bit register by feeding the output signal *save_code* of the FSM as an input to the register. This signifies that if the value of *save_code* is '1' on the clock edge, then the passcode present on the D input must be stored by the register.

Applying a *reset* signal to the 4-bit register clears the currently stored passcode and resets the register to its default value.

3) 4-bit Comparator

In order to compare the 4-bit passcode set as an input by the user (X[3:0]) to the passcode that is currently stored in the 4-bit register, we use a 4-bit comparator. A comparator is a standard digital logic element that is used to compare two binary numbers (Brown and Vranesic, 2013).

If the two passcodes that the 4-bit comparator is comparing in the current clock cycle are equal, the comparator sets the *equal* signal to '1'. Otherwise, the value of *equal* signal remains '0'. Please note that the *equal* signal generated by the 4-bit comparator is a repeating signal, rather than a one-time signal.

Please note that for our purposes, we are only interested in the design of the Combination Lock controller expressed as Moore FSM. Therefore, we will not explain the internal design and working of the other two components of the Combination Lock, i.e. 4-bit register and 4-bit comparator, and assume that these two components are already available to us for use.

13.2 Design of Controllers

In this section, we discuss the design of the controller for the 4-bit Combination Lock system. In order to apply our FSM-TDES translation approach, we model and express the controller as a complete Moore system. Specifically, the Moore system consists of: 1) one central FSM for the 4-bit Combination Lock, and 2) three individual Moore FSM to realize the system specifications described in Section 13.1. We will provide this Moore system as an input to our FSM-TDES translation method.

13.2.1 Individual Moore FSM

In the real world, control designers typically find it easier to design several modular controllers instead of developing a large monolithic system controller all at once. Keeping this in view, we design three modular controllers expressed as individual Moore FSM to realize the functionality of the 4-bit Combination Lock system.

The three modular controllers that we design are **OpenLock**, **ChangeCode** and **ActivateAlarm**. The **OpenLock** Moore FSM focuses on the functionality of opening the combination lock. The **ChangeCode** FSM contains the logic for changing the existing 4-bit passcode. The **ActivateAlarm** FSM specifies the sequence for activating the system alarm.

Below, we first introduce the graphical notation for expressing the Moore FSM. Then, we elaborate the design of the three individual Moore FSM for the 4-bit Combination Lock.

Graphical Notation for Moore FSM

For an individual Moore FSM, we represent the states as boxes, with the state name written inside the box. Each state name is preceded by the label "ST:". For example, "ST: 1" written inside a box means that the name of this FSM state is 1.

At each state of the FSM, we specify the output values for all the input-output (IO) signals that belong to this FSM. We state this output information inside the state box. The output value of '1' means that the output of the IO signal has been set to $True$ by the FSM at the current state. Whereas, the output value of '0' indicates that the output of the IO signal has been set to $False$, and this IO signal cannot occur at the current state of the FSM.

We use arrows to represent the transition of the FSM from one state to the next. Each arrow is labelled with one or more next state conditions that are expressed as either boolean expressions or using one of the three reserved keywords (introduced in Section 11.1.1). The FSM will move from one state to the other on the clock edge when a given next state condition is satisfied, i.e. it evaluates to $True$.

In the boolean expressions, "$\cdot$" symbol represents the **AND** operator, "+" represents the **OR** operator, and "!" represents the **NOT** operator. However, in the figures, we will write " . " instead of "$\cdot$", as " . " is easier to produce.

Every FSM has one initial/reset state. We represent this state with an incoming arrow that starts at a filled circle and is labelled with "Reset". Whenever the system is reset, all FSM go back to their initial/reset state regardless of their current state. It is notable that reset mechanism is inherent in the design of a Moore FSM. Hence, we will not explicitly show a reset transition at any state of the individual FSM.

Moore FSM-1: OpenLock

In order to model the functionality to lock and unlock the 4-bit Combination Lock, we design the Moore FSM **OpenLock** shown in Figure 11.1 (given in Section 11.1.1 on page 181). The **OpenLock** FSM has two states: state 1 that represents the lock state, and state 2 that depicts the unlock state. We assume that initially the **OpenLock** FSM will be in its lock state. That is why, we have selected state 1 as the initial/reset state for **OpenLock**. While designing **OpenLock**, we use three signals: one IO signal, *open*, and two input signals, *enter* and *equal*.

In the lock state, the door secured by the 4-bit Combination Lock cannot be opened, unless user unlocks the system. For this reason, we set the output of the IO signal *open* to '0' at state 1. In terms of the Moore FSM, *open* = 0 means that *open* cannot occur in the physical system while **OpenLock** is at state 1.

In order to unlock the Combination Lock and open the door, the user must input the correct existing passcode and press **Enter**. In the context of the FSM, this means that in the lock state, **OpenLock** must receive *enter* and *equal* signals in the same clock cycle in order to go from state 1 to state 2. We can express this condition as a boolean expression by writing $enter \cdot equal$. For all next state conditions other than $enter \cdot equal$, **OpenLock** must stay in the same state, i.e. state 1. In order to model this, we take the complement of the next state condition $enter \cdot equal$ and specify it as a selfloop transition at state 1, i.e. $!(enter \cdot equal) = !enter + !equal$ by DeMorgan's Theorem (Brown and Vranesic, 2013).

In this way, we have explicitly specified all possible next state conditions at state 1 of the **OpenLock** FSM. This provides the benefit that we do not not need to add a **DEF** transition to make the next state function a total function at state 1 of **OpenLock**. Please refer to Section 11.1.1 to see the related discussion on **DEF** transition and FSM's total function.

At state 2 of **OpenLock**, we set the output of *open* to '1'. This signifies that the **OpenLock** FSM allows the door to be opened while it is in the unlock state. The door should stay open until user presses the **Enter** button, after which the door closes and the system goes back to its lock state. In terms of the FSM, this means that **OpenLock** must go from state 2 to state 1 when it receives the *enter* signal, i.e. $enter = 1$. Otherwise, as long as $enter = 0$, **OpenLock** must stay at state 2.

We model this in **OpenLock** by specifying the boolean expression of *enter* that takes the FSM from state 2 to state 1. We use the complement of this next state condition, i.e. $!enter$ to indicate that **OpenLock** must stay at state 2 as long as the user does not press **Enter**. This completes the design of the **OpenLock** Moore FSM as per the system specifications given in Section 13.1. XML Input File E.1 (Section E.1.1 of Appendix E) represents the **OpenLock** FSM in our XML file format.

Moore FSM-2: ChangeCode

The second individual FSM that we have designed for the 4-bit Combination Lock is shown in Figure 13.3. The Moore FSM **ChangeCode** models the specifications for changing the existing 4-bit passcode of the Combination Lock. The **ChangeCode** FSM has three states labelled as 1, 2 and 3, with state 1 being the initial/reset state of the FSM. In order to realize the required functionality, we use four signals in **ChangeCode**: two IO signals, *new* and *save_code*, and two input signals, *change* and *equal*.

The **ChangeCode** FSM starts at its initial/reset state 1. At state 1, we set the outputs of the two IO signals, *new* and *save_code*, to '0'. This indicates that the process of changing the existing passcode has not yet been initiated by the user.

To change the existing 4-bit passcode, the user must enter the correct existing passcode and press the **Change** button. We model this in **ChangeCode** by specifying a transition from state 1 to state 2 with the next state condition of $change \cdot equal$.

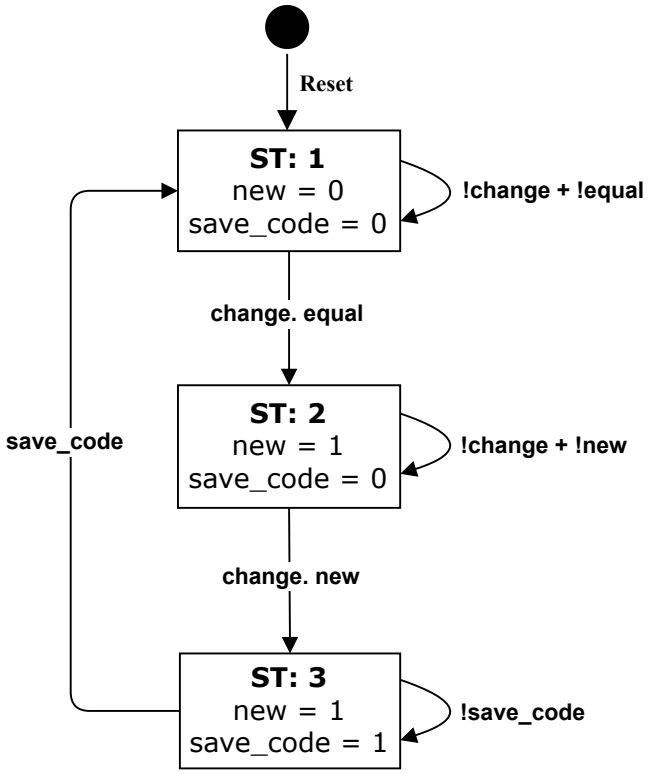


Figure 13.3: Moore FSM **ChangeCode**

For all the next state conditions other than $change \cdot equal$, **ChangeCode** must stay at state 1. We determine these other next state conditions as $!(change \cdot equal) = !change + !equal$ (DeMorgan's Theorem (Brown and Vranesic, 2013)). We add a self-loop transition at state 1 of **ChangeCode** and label it with this boolean expression, as shown in Figure 13.3.

At state 2 of **ChangeCode**, we set the output value of the IO signal *new* to '1'. This indicates that user has initiated the task of changing the existing passcode, and the **ChangeCode** FSM is now in the "new" state. In the new state, when user enters the new passcode and presses **Change**, the system should save the new passcode. We model this in **ChangeCode** by adding a state changing transition of $change \cdot new$ that goes from state 2 to state 3. At state 2, we specify the rest of all possible next state conditions by defining a selfloop transition labelled with $!(change \cdot new) = !change + !new$ (DeMorgan's Theorem (Brown and Vranesic, 2013)).

At state 3, we set the outputs of both IO signals, *new* and *save_code*, to '1'. This shows that in the process of changing the existing passcode, user has provided the new passcode and now it needs to be stored in the 4-bit register on the next clock edge. Please see the description of *save_code* signal in Section 13.1.2.

Once the new passcode is stored in the 4-bit register, the process of changing the existing 4-bit passcode is complete. We model this in **ChangeCode** by defining the state changing transition of *save_code* that goes from state 3 to state 1. We specify

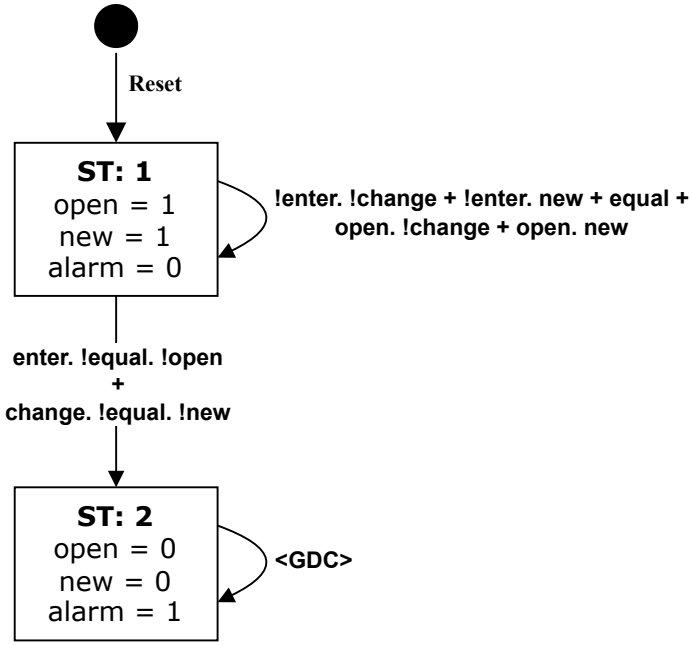


Figure 13.4: Moore FSM **ActivateAlarm**

the rest of the possible next state conditions at state 3 of **ChangeCode** by adding a selfloop transition of !*save_code*. XML Input File E.2 (Section E.1.1) represents the **ChangeCode** FSM in our XML file format.

Moore FSM-3: ActivateAlarm

Figure 13.4 shows the third and last individual Moore FSM **ActivateAlarm** for the 4-bit Combination Lock. The **ActivateAlarm** FSM is responsible for detecting if the user tries to access any system functionality by entering an incorrect passcode, i.e. does not match the saved passcode. In response, it activates the alarm and ceases the normal system operation, until user presses **Reset** to cancel the alarm and reset the system.

The **ActivateAlarm** FSM comprises of two states: state 1 that represents the initial/reset state, and state 2 that the FSM goes to once the alarm is activated. In order to model the required specifications, we use six signals in the **ActivateAlarm** FSM: three IO signals, *open*, *new* and *alarm*, and three input signals, *enter*, *change* and *equal*.

At state 1 of **ActivateAlarm**, we set the output of *alarm* IO signal to '0'. This is to specify that system alarm must remain deactivated while **ActivateAlarm** is at state 1. Since alarm has not been activated yet, users should be able to access the regular system functionality and operate the 4-bit Combination Lock as normal. We express this logic in **ActivateAlarm** by setting the outputs of *open* and *new* IO signals to '1'. Please note that the two system functionalities of opening the combination

lock and changing the existing 4-bit passcode are primarily managed by the other two FSM, **OpenLock** and **ChangeCode** respectively. Therefore, **ActivateAlarm** must not disable the outputs of *open* and *new* signals during the normal system operation.

In the normal mode, if user tries to perform any task by entering the incorrect existing passcode, then alarm must go off. We model this by adding a state changing transition at state 1 that takes **ActivateAlarm** to state 2. We label this transition by expressing the next state conditions as a boolean expression in the sum-of-products (SOP) form, $(enter \cdot !equal \cdot !open) + (change \cdot !equal \cdot !new)$.

For all the remaining next state conditions, alarm must remain deactivated and **ActivateAlarm** must stay in the same state, i.e. state 1. We determine these remaining next state conditions by taking the complement of the state changing next state conditions, i.e. $!((enter \cdot !equal \cdot !open) + (change \cdot !equal \cdot !new))$. By applying boolean algebra properties (Brown and Vranesic, 2013), we obtain a simplified version of this boolean expression in the SOP form and specify it as a selfloop transition at state 1 of **ActivateAlarm**, as shown in Figure 13.4. Please refer to Section E.2 to see the steps for deriving this simplified boolean expression.

State 2 of **ActivateAlarm** represents the alarm state of the system. At state 2, we set the *alarm* output to '1' to indicate that once system reaches this state, alarm must be activated. In the alarm state, the regular system functionality becomes unavailable to the users. We enforce this by setting the outputs of *open* and *new* to '0' at state 2. In other words, in the alarm state, the users will not be able to open the lock (since $open = 0$) or change the existing 4-bit passcode (since $new = 0$).

At state 2 of **ActivateAlarm**, we add a selfloop of the **GDC** transition (introduced in Section 11.1.1). This is to model the system requirement that no matter what input combination(s) the system receives in the alarm state, alarm must remain activated. The only way to cancel the alarm is by pressing the **Reset** button. XML Input File E.3 (Section E.1.1) represents the **ActivateAlarm** FSM in our XML file format.

13.2.2 Central FSM

XML Input File E.4 (Section E.1.2) shows the central FSM for the 4-bit Combination Lock system in our XML file format.

13.3 Translated TDES Supervisors

In this section, we present the results of applying our FSM-TDES translation algorithms (given in Chapter 12) to the Moore FSM of the 4-bit Combination Lock example. Please note that the Combination Lock system satisfies all FSM-TDES translation prerequisites stated in Section 11.2, as verified by our algorithms.

For the Combination Lock, our FSM-TDES translation algorithms generate three

modular TDES supervisors, i.e. one TDES supervisor corresponding to each individual Moore FSM. It does so by creating and populating each TDES supervisor's quintuple $\mathbf{S}_i = (X_i, \Sigma_i, \xi_i, x_{o,i}, X_{m,i})$, where $i = \{1, 2, 3\}$. Here, X_i is the *state set*, Σ_i is the *event set*, $\xi_i : X_i \times \Sigma_i \rightarrow X_i$ is the *partial transition function*, $x_{o,i}$ is the *initial state*, and $X_{m,i}$ is the *set of marked states* for the i^{th} TDES supervisor that is translated from the i^{th} Moore FSM.

13.3.1 Open Lock

While describing our FSM-TDES translation method in Section 11.3, we used the **OpenLock** Moore FSM (Figure 11.1) as an example. The result of this translation is a non-minimal TDES supervisor **OpenLock**, shown in Figure 11.2. The translated TDES has 12 states and 40 transitions. Its minimal version is presented in Figure 11.10, and has 8 states and 26 transitions.

13.3.2 Change Code

For the **ChangeCode** FSM (Figure 13.3), the translated non-minimal TDES supervisor is shown in Figure E.1 (Section E.3). Its minimal version is given in Figure 13.5. After minimization, the state space reduced from 28 states in the translated supervisor to 14 states in its minimal version. The number of transitions decreased from 104 to 51.

13.3.3 Activate Alarm

The non-minimal TDES supervisor translated from the **ActivateAlarm** FSM (Figure 13.4) is shown in Figure E.2 (Section E.3). Its minimal version is presented in Figure 13.6. After minimization, the state space reduced from 34 states in the translated supervisor to 14 states in its minimal version. The number of transitions declined from 169 to 73.

13.4 TDES Plant Models

In order to verify the desired properties of the closed-loop system by running our $||_{SD}$ verification checks (discussed in Chapter 9) in DESpot (2023), we need to synchronize the translated TDES supervisors with the TDES plant models using our $||_{SD}$ operator. Keeping this in view, we design the TDES plant components for the 4-bit Combination Lock system. Our plant TDES are shown in Figures 13.7-13.13.

By looking at the plant models, we note that each plant component models one activity (non-*tick*) event of the physical system. Each plant component allows the activity event to occur at most once per clock period. We are able to model the plant TDES this way because of our assumption (stated in Section 13.1.1) that the clock

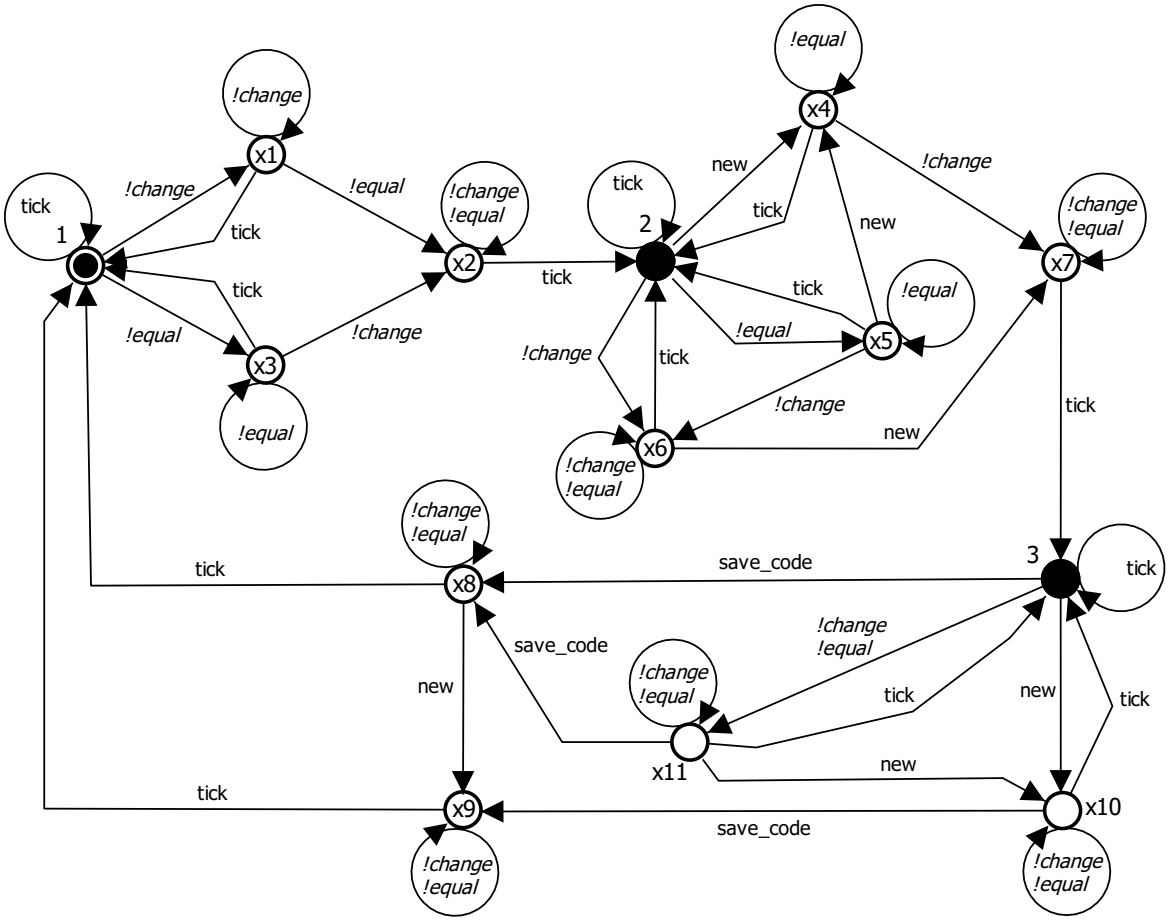


Figure 13.5: Minimal TDES Supervisor **ChangeCode**

frequency of the system is much higher than the rate at which users could generate the input signals (uncontrollable events). Moreover, our $||_{SD}$ setting (Chapter 4) assumes that controllers allow prohibitable events to occur once per clock period.

13.5 Verification Results

For any physical system, the ultimate goal of developing theoretical TDES supervisor models is to verify the desired properties of the closed-loop system before the actual implementation. Therefore, for the 4-bit Combination Lock system, we performed the desired $||_{SD}$ checks by utilizing our BDD $||_{SD}$ algorithms (Chapter 9) that we have implemented in DESpot (2023). In this section, we present and discuss the verification results of evaluating the $||_{SD}$ properties for the Combination Lock example.

Using our BDD $||_{SD}$ algorithms, we found that the Combination Lock system satisfies all of the desired $||_{SD}$ properties. These properties include CS deterministic

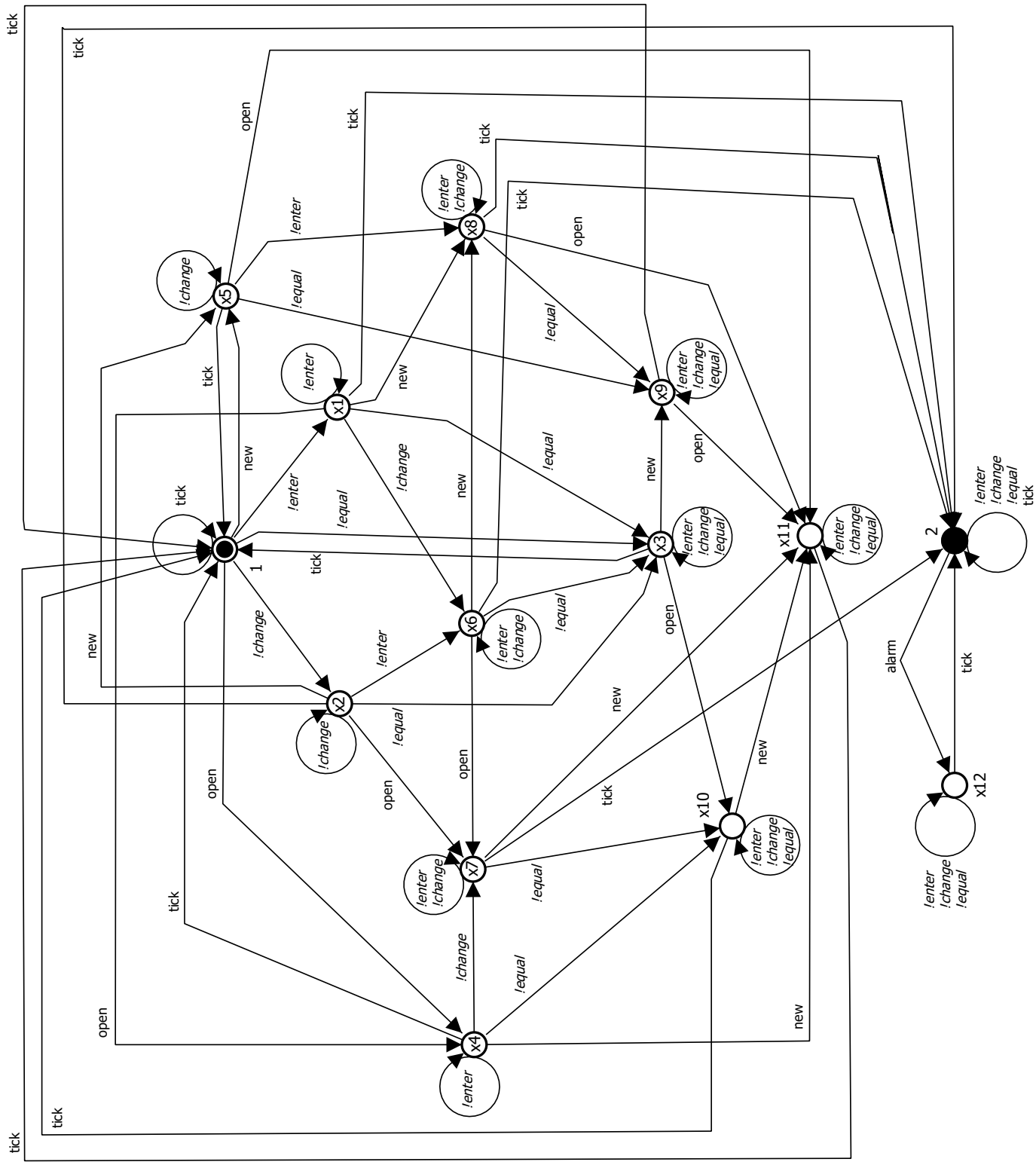


Figure 13.6: Minimal TDES Supervisor **ActivateAlarm**

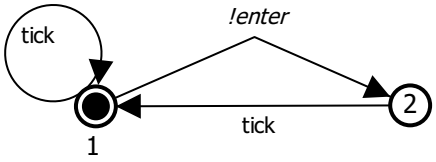


Figure 13.7: **Enter**

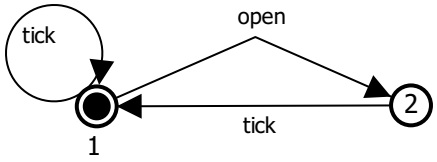


Figure 13.8: **Open**

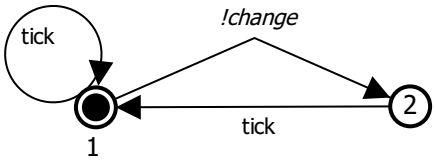


Figure 13.9: **Change**

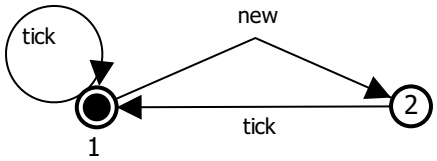


Figure 13.10: **New**

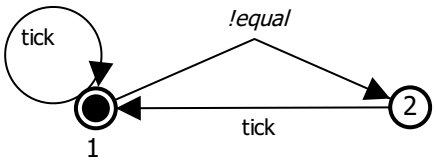


Figure 13.11: **Comparator**

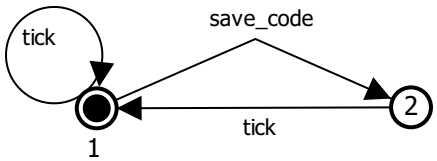


Figure 13.12: **Register**

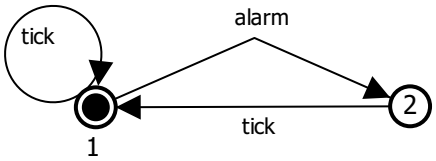


Figure 13.13: **Alarm**

supervisors, non-selfloop ALF supervisors, proper time behaviour, ALF, nonblocking, plant completeness with $||_{SD}$, **S**-singular prohibitable behaviour with $||_{SD}$ and SD controllability with $||_{SD}$ (that covers the untimed and timed controllability with $||_{SD}$ properties). The closed-loop system had 264 states, and verification took less than 1 second. We performed these tests on a machine running Windows 10 with 16GB of RAM and 2.6GHz Intel 6-core processor.

We would like to briefly discuss about the property of nonblocking (Definition 2.2.8), which is primarily determined by the marking information of a TDES. In other words, changing the marking of states in a TDES might cause this property to pass or fail. For example, if we change the marking of the TDES supervisor **ChangeCode** (Figure 13.5) from "mark all sampled states" to "mark only initial state", the closed-loop system, that is nonblocking in its current form, will start to block.

Therefore, if the closed-loop system blocks with the translated TDES supervisors, the designers might need to change the marking information to make the system

nonblocking. If designers are unsure about which marking option to select, we recommend that they should choose to mark all sampled states instead of just marking the initial state of each TDES supervisor. This increases the probability of the closed-loop system to satisfy the nonblocking property.

We will close this section by restating that one of our major goals while addressing this research problem was to develop an automatic FSM-TDES translation approach that should generate TDES supervisors that are more likely to satisfy the desired $||_{SD}$ properties. Our $||_{SD}$ verification results for the Combination Lock example and the discussion of this section makes it evident that our devised FSM-TDES translation approach is "capable" of generating TDES supervisors that satisfy the desired $||_{SD}$ properties. However, we cannot guarantee that this will always be the case for every control system.

13.6 Correctness of FSM-TDES Translation Approach

As mentioned in Section 11.3.6, our primary goal while devising the FSM-TDES translation approach is to formulate an automatic translation method that should be capable of generating a "correct" TDES supervisor from the Moore FSM without violating the given control specifications. By looking at the supervisors that our FSM-TDES translation approach has generated for the 4-bit Combination Lock, it is evident that the translated TDES supervisors fulfill the desired system specifications and abide by all the given control laws.

However, in order to rigorously verify the correctness of our FSM-TDES translation approach for the Combination Lock system, we decided to complete the cycle of FSM-TDES-FSM translation for this system. Specifically, we started our Combination Lock example by manually designing its controllers expressed as Moore FSM (Section 13.2). Then, we converted these Moore FSM into TDES supervisors by applying our FSM-TDES translation approach (Section 13.3). In Section E.4 of Appendix E, we convert the translated TDES supervisors back to Moore FSM using the TDES-FSM translation method defined by Wang (2009) (described in Section 3.7.2) and our modified TDES-FSM translation algorithms (presented in Section C.2).

The results of our translation cycle indicate that although our manually designed Moore FSM that we started with are not identical to the result of our FSM-TDES-FSM translation, they generated the same outputs and reacted the same to the valid next state conditions that could occur in the system. Hence, we conclude that, at least for this example, our FSM-TDES translation approach produced correct results.

Chapter 14

Conclusions and Future Work

This chapter presents our conclusions and gives some directions for further research related to this study.

14.1 Conclusions

The research work presented in this thesis focuses on the automatic translation of Moore synchronous Finite State Machines (FSM) (Brown and Vranesic, 2013) into Timed Discrete Event System (TDES) supervisors in order to facilitate software and hardware designers and practitioners in the formal representation and verification of their new and existing systems. We build our work on the sampled-data (SD) supervisory control theory (Wang, 2009; Wang and Leduc, 2012; Leduc *et al.*, 2014). Particularly, we make use of the structural similarity created by the SD theory between the two models while developing our automatic FSM-TDES translation approach.

The first part of this thesis (Chapters 4-10) focuses on presenting an approach to automate the mechanism of forcing eligible prohibitable events in the SD supervisory control framework. We introduced a new synchronization operator, called the *SD synchronous product* ($||_{SD}$), that provides a novel way of constructing closed-loop systems in the SD framework. Our $||_{SD}$ operator is smart enough to automatically disable a *tick* event in the closed-loop system, if both *tick* and a prohibitable event is possible in the plant TDES and enabled by all modular TDES supervisors. We adapted the TDES and SD properties of the existing SD supervisory control setting ("SD setting," for short) in order to make them compatible with our $||_{SD}$ operator.

Our approach provides twofold benefits: 1) In the existing SD setting, it liberates the designers from manually satisfying the intricate property of SD controllability Point ii ($\Rightarrow$) (Definition 3.5.1) while developing their TDES supervisors by hand. This results in simplifying the TDES modelling process and improving the ease of manually designing SD controllable TDES supervisors, thus making the SD framework more accessible to designers. 2) While devising this approach, we adopted the controllers

way of event forcing and applied it to our theoretical TDES setting. This essentially bridges the gap between TDES supervisors and physical controllers by making them behave in a similar way with respect to forcing of events. This proved to be advantageous for us in the development of our automatic FSM-TDES translation approach later in this study, by making it relatively uncomplicated and straightforward.

After formulating our $||_{SD}$ setting, we established logical equivalence between the existing SD setting and our proposed $||_{SD}$ setting with respect to their closed and marked languages, TDES and SD properties, and the SD controllers that are obtained by translating the TDES supervisors designed in the two settings. By making use of this equivalence, we formally verified our $||_{SD}$ setting with respect to the desired properties of controllability and nonblocking. Specifically, we proved that if designers create a theoretical TDES system in our $||_{SD}$ setting that is controllable, nonblocking and satisfy the required $||_{SD}$ properties, then the physical implementation will retain these properties and the system abides by the control laws. By proving this, we have essentially transferred all benefits of the SD setting to our $||_{SD}$ setting.

After theoretical verification, we focused on providing tool support for our $||_{SD}$ setting. We adapted the predicate-based algorithms of the SD setting (Wang, 2009) to verify the corresponding properties in our $||_{SD}$ setting. We implemented these algorithms and our $||_{SD}$ operator in the DES research tool, DESpot (2023). This tool support enables the designers to automatically verify the desired properties of their TDES systems that they design in our $||_{SD}$ setting.

Finally, we showed the application and strengths of our devised approach by applying it to an example of a Flexible Manufacturing System (FMS). By comparing the modular TDES supervisors designed for FMS in the SD and $||_{SD}$ settings, we demonstrated that our $||_{SD}$ operator has greatly simplified the design logic and reduced the state size of the TDES supervisors. Consequently, for the FMS example, we noted a reduction of 75% in the time taken by DESpot to run the $||_{SD}$ verification checks as compared to the corresponding SD checks.

In the second part of this thesis (Chapters 11-13), we presented a generic and structured approach to automatically translate Moore synchronous FSM into TDES supervisors. First, we specified the input structure for concretely expressing the system controllers that are represented as Moore FSM. Next, we identified a set of consistency and design requirements that the input Moore FSM must satisfy in order to be considered "valid" for translation. Then, we defined the translation steps and well-defined rules to automatically convert a Moore FSM into an equivalent TDES supervisor representation. We devised our FSM-TDES translation method in such a way that the translated TDES supervisors are more likely to satisfy the desired properties of the $||_{SD}$ setting. After that, we developed a set of algorithms to express our complete FSM-TDES translation approach algorithmically.

In order to demonstrate the application of our FSM-TDES translation approach, we translated the system controllers of a 4-bit Combination Lock, expressed as Moore

FSM, into TDES supervisors. We verified the correctness of our FSM-TDES translation method for the Combination Lock example by completing the FSM-TDES-FSM translation cycle for this example. This translation cycle also helped us in verifying the changes that we made in Hamid's (2014) TDES-FSM translation algorithms to make the existing TDES-FSM translation method consistent with our FSM-TDES translation approach. This compatibility between the two translation approaches of the SD theory allows the designers to translate one model into the other without experts' intervention, hence enhancing the utility of the SD theory.

We conclude this research by stating that our automatic FSM-TDES translation approach should be of great utility to software and hardware practitioners in the formal representation and verification of their existing systems, as well as new systems that are designed to be formally verified. The presented approach enables the designers to express their system controllers in a generic and standard way that they are familiar with, i.e. as Moore FSM. Hence, our approach should be particularly useful to practitioners that have limited or no knowledge of formal methods by enabling them to formally verify their systems and identify the potential design and implementation issues they might be having, without designing formal TDES supervisors by hand. It should also be beneficial to those with expertise in formal methods, since expressing a controller in terms of a Moore FSM is typically easier and more intuitive than developing the corresponding TDES supervisors. Overall, we anticipate this research work to be advantageous in increasing the adoption of SD supervisory control theory in particular, and formal methods in general, in the industry by simplifying the formal design and verification process of control systems.

14.2 Future Work

Some future research directions to extend the work presented in this thesis as well as to address its limitations follow.

Improvements for Automatic FSM-TDES Translation Approach

1. Our primary goal while devising our FSM-TDES translation approach was to generate a "correct" TDES supervisor from the input Moore FSM, without violating the given control specifications. Now that we have a correct translation method, a useful next step would be to improve the efficiency of the translation process and focus on generating a "compact" supervisor while retaining its logical design correctness.

   One feasible way to do this is to come up with the FSM-TDES translation rules that directly operate on the hybrid next state logic (NSL) for generating TDES transitions, instead of the boolean NSL of our current approach (Section 11.3.5). The elimination of boolean NSL should not only make the translation process efficient, but is also expected to pave the way for generating a TDES supervisor

in its minimal form (Definition 2.2.10). As a result, unlike our current translation approach, there would be no need to perform a separate TDES state space minimization step at the end of the translation process.

2. In this work, we have demonstrated the correctness of our FSM-TDES translation approach by completing the FSM-TDES-FSM translation cycle for our 4-bit Combination Lock example. However, in order to generalize and strengthen the claims that we made in Section 13.6, the correctness of our FSM-TDES translation method and the translated TDES supervisors should be proven formally.

3. With respect to the theoretical research work presented in this thesis, it would be beneficial to make the following implementation extensions in DESpot (2023):

- We have developed a set of algorithms to realize our FSM-TDES translation approach (Chapter 12) and the TDES state space minimization process (Section 6.2). However, the implementation of these algorithms has been left as future work due to time constraints. Therefore, it would be significantly useful to provide tool support for our FSM-TDES translation approach by implementing these algorithms in DESpot.
- In order to completely automate the FSM-TDES translation process in DESpot, algorithms should be developed and implemented to read our input XML files for the central and individual Moore FSM, parse the information, and populate the appropriate variables, as described in Section 12.2.
- For the purpose of making the existing TDES-FSM translation method (Wang, 2009) compatible with our FSM-TDES translation approach, we have presented two TDES-FSM translation algorithms in Section C.2. These algorithms are meant to replace Hamid's (2014) TDES-FSM translation algorithms in DESpot. Our algorithms should be implemented in DESpot in order to allow the designers to automatically translate one model into the other and take complete advantage of the SD supervisory control theory.

4. We have demonstrated the application of our FSM-TDES translation approach by applying it to the small example of a 4-bit Combination Lock (Chapter 13). The primary reason for choosing this small system is that, currently, we do not have tool support for our FSM-TDES translation approach. Once our approach is implemented in DESpot, it would be interesting to apply it to translate larger controllers and gain an insight into its performance and efficiency.

5. Our FSM-TDES translation approach relies on a generic input format, i.e. an XML file, for expressing the system controllers as Moore FSM. In future, it would be beneficial to provide implementation-specific support for our translation method, i.e. develop and implement methods to automatically generate our XML input files from existing controllers that have been implemented in C, PLC LD (Antonsen, 2021) or Verilog (Brown and Vranesic, 2013). This is expected to further facilitate the designers by liberating them from writing the XML input

files by hand, thus making our FSM-TDES translation method more accessible to them.

Improvements for Automatic Event Forcing/*tick* Disablement Approach

1. As part of this study, we implemented our $||_{SD}$ operator and related $||_{SD}$ properties by tweaking predicate-based algorithms of the SD setting that have been implemented by Wang (2009) in DESpot (2023). Wang's implemented algorithms perform a monolithic check to verify various TDES and SD properties of the closed-loop system. On the other hand, Hamid (2014) has implemented a modular method to verify these properties in DESpot. Therefore, it would be beneficial to implement our $||_{SD}$ operator and complete $||_{SD}$ setting using Hamid's modular verification method and gauge the performance gain.
2. We have verified our approach by applying it to an example of a Flexible Manufacturing System (Chapter 10). We noted improvement in the ease of manually designing SD controllable TDES supervisors, reduction in the state space, and decrease in verification time of the overall system. In future, it would be useful to check the efficacy of our approach and $||_{SD}$ setting, and reaffirm our results by applying it to TDES with larger state spaces.
3. Some interesting further studies include the extension of our approach and $||_{SD}$ setting to fault-tolerant supervisory control (Mulahuwaish, 2019) and hierarchical interface-based supervisory control (Leduc, 2001).

Bibliography

Adam, H.-J. and Adam, M. (2022). *PLC Programming in Instruction List According To IEC 61131-3: A Systematic and Action-Oriented Introduction in Structured Programming*. Springer, Berlin, Heidelberg, 1st edition.

Alur, R. and Dill, D. L. (1994). A Theory of Timed Automata. *Theoretical Computer Science*, **126**(2), 183–235.

Alves, M. R. C., Rudie, K., and Pena, P. N. (2022). A Security Testbed for Networked DES Control Systems. *IFAC-PapersOnLine*, **55**(28), 128–134.

Antonsen, T. M. (2021). *PLC Controls with Ladder Diagram (LD): IEC 61131-3 and Introduction to Ladder Programming*. Books on Demand.

Arinez, J., Benhabib, B., Smith, K., and Brandin, B. (1993). Design of a PLC-Based Supervisory Control System for a Manufacturing Workcell. In *Proceedings of the Canadian High Technology Show and Conference*, Canada.

Arnon, D. S. (1988). A Bibliography of Quantifier Elimination for Real Closed Fields. *Journal of Symbolic Computation*, **5**(1-2), 267–274.

Azkarate, I., Ayani, M., Mugarza, J. C., and Eciolaza, L. (2021). Petri Net-Based Semi-Compiled Code Generation for Programmable Logic Controllers. *Applied Sciences*, **11**(15).

Balemi, S. (1994). Input/Output Discrete Event Processes and Communication Delays. *Discrete Event Dynamic Systems*, **4**(1), 41–85.

Balemi, S., Hoffmann, G., Gyugyi, P., Wong-Toi, H., and Franklin, G. (1993). Supervisory Control of a Rapid Thermal Multiprocessor. *IEEE Transactions on Automatic Control*, **38**(7), 1040–1059.

Basile, F., Chiacchio, P., and Gerbasio, D. (2013). On the Implementation of Industrial Automation Systems Based on PLC. *IEEE Transactions on Automation Science and Engineering*, **10**(4), 990–1003.

Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M., and Stursberg, O. (2004). Verification of PLC Programs Given as Sequential Function Charts. In *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 517–540. Springer, Berlin, Heidelberg.

Bender, D. F., Combemale, B., Crégut, X., Farines, J. M., Berthomieu, B., and Vernadat, F. (2008). Ladder Metamodeling and PLC Program Validation through Time Petri Nets. In *Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA'08)*, pages 121–136, Berlin, Heidelberg. Springer.

Berthomieu, B., Ribet, P.-O., and Vernadat, F. (2004). The Tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, **42**(14), 2741–2756.

Bolton, W. (2015). *Programmable Logic Controllers*. Elsevier, 6th edition.

Brandin, B. A. (1993). *Real-Time Supervisory Control of Automated Manufacturing Systems*. Ph.D. Thesis, Department of Computer and Electrical Engineering, University of Toronto, Toronto, ON, Canada.

Brandin, B. A. (1996). The Real-Time Supervisory Control of an Experimental Manufacturing Cell. *IEEE Transactions on Robotics and Automation*, **12**(1), 1–14.

Brandin, B. A. and Wonham, W. M. (1994). Supervisory Control of Timed Discrete-Event Systems. *IEEE Transactions on Automatic Control*, **39**(2), 329–342.

Brown, S. and Vranesic, Z. (2013). *Fundamentals of Digital Logic with Verilog Design*. McGraw-Hill Education, New York, 3rd edition.

Bryant (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, **C-35**(8), 677–691.

Bryant, R. E. (1992). Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys (CSUR)*, **24**, 293–318.

Caldwell, B., Cardell-Oliver, R., and French, T. (2016). Learning Time Delay Mealy Machines From Programmable Logic Controllers. *IEEE Transactions on Automation Science and Engineering*, **13**(2), 1155–1164.

Canet, G., Couffin, S., Lesage, J.-J., Petit, A., and Schnoebelen, P. (2000). Towards the Automatic Verification of PLC Programs Written in Instruction List. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC'2000)*, pages 2449–2454, USA.

Cantarelli, M. (2006). *Control System Design Using Supervisory Control Theory: From Theory to Implementation*. Master's Thesis, University of Cagliari, Italy.

Cassandras, C. G. and Lafortune, S. (2008). *Introduction to Discrete Event Systems*. Springer US, New York, NY, 2nd edition.

Chandra, V., Huang, Z., and Kumar, R. (2003). Automated Control Synthesis for an Assembly Line using Discrete Event System Control Theory. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, **33**(2), 284–289.

Chang, N., Kwon, W. H., and Park, J. (1998). Hardware Implementation of Real-Time Petri-Net-Based Controllers. *Control Engineering Practice*, **6**(7), 889–895.

Crockett, D., Desrochers, A., DiCesare, F., and Ward, T. (1987). Implementation of a Petri Net Controller for a Machining Workstation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1861–1867, USA.

de Queiroz, M. H. and Cury, J. E. R. (2000). Modular Supervisory Control of Large Scale Discrete Event Systems. In *Discrete Event Systems*, pages 103–110. Springer, Boston, MA.

de Queiroz, M. H. and Cury, J. E. R. (2002). Synthesis and Implementation of Local Modular Supervisory Control for a Manufacturing Cell. In *Proceedings of the 6th International Workshop on Discrete Event Systems (WODES'02)*, pages 377–382, Spain.

DESpot (2023). www.cas.mcmaster.ca/∼leduc/DESpot.html.

Dietrich, P., Malik, R., Wonham, W. M., and Brandin, B. A. (2002). Implementation Considerations in Supervisory Control. In *Synthesis and Control of Discrete Event Systems*, pages 185–201. Springer US, Boston, MA.

Fabian, M. and Hellgren, A. (1998). PLC-Based Implementation of Supervisory Control for Discrete Event Systems. In *Proceedings of the 37th IEEE Conference on Decision and Control*, volume 3, pages 3305–3310, USA.

Feio, R., Rosas, J., and Gomes, L. (2017). Translating IOPT Petri Net Models into PLC Ladder Diagrams. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT'17)*, pages 1211–1216, Canada.

Feng, L. and Wonham, W. M. (2006). TCT: A Computation Tool for Supervisory Control Synthesis. In *Proceedings of the 8th International Workshop on Discrete Event Systems (WODES 2006)*, pages 388–389, Michigan, USA.

Forschelen, S. T. J., van de Mortel-Fronczak, J. M., Su, R., and Rooda, J. E. (2012). Application of Supervisory Control Theory to Theme Park Vehicles. *Discrete Event Dynamic Systems*, **22**(4), 511–540.

Fouquet, K. and Provost, J. (2017). A Signal-Interpreted Approach to the Supervisory Control Theory Problem. *IFAC-PapersOnLine*, **50**(1), 12351–12358.

Fujino, K., Imafuku, K., Yuh, Y., and Hirokazu, N. (2000). Design and Verification of the SFC Program for Sequential Control. *Computers & Chemical Engineering*, **24**(2-7), 303–308.

Gelen, G. and Uzam, M. (2014). The Synthesis and PLC Implementation of Hybrid Modular Supervisors for Real Time Control of an Experimental Manufacturing System. *Journal of Manufacturing Systems*, **33**(4), 535–550.

Gelen, G., Uzam, M., and Dalci, R. (2010). The Concept of Postponed Event in Timed Discrete Event Systems and its PLC Implementation. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC'10)*, pages 2753–2759, Turkey.

Ghasaei, A., Zhang, Z. J., Wonham, W. M., and Iravani, R. (2021). A Discrete-Event Supervisory Control for the AC Microgrid. *IEEE Transactions on Power Delivery*, **36**(2), 663–675.

Gouyon, D., Pétin, J.-F., and Gouin, A. (2004). A Pragmatic Approach for Modular Control Synthesis and Implementation. *International Journal of Production Research*, **42**(14), 2839–2858.

Hamid, A. (2014). *Implementation of Sampled-Data Supervisory Control*. Master's Thesis, Department of Computing and Software, McMaster University, Hamilton, ON, Canada.

Hasdemir, İ. T., Kurtulan, S., and Gören, L. (2008). An Implementation Methodology for Supervisory Control Theory. *The International Journal of Advanced Manufacturing Technology*, **36**(3), 373–385.

Heiner, M. and Menzel, T. (1998). A Petri Net Semantics for the PLC Language Instruction List. In *Proceedings of the 4th Workshop on Discrete Event Systems (WODES'98)*, pages 161–166, Italy.

Hellgren, A., Lennartson, B., and Fabian, M. (2002). Modelling and PLC-Based Implementation of Modular Supervisory Control. In *Proceedings of the 6th International Workshop on Discrete Event Systems (WODES'02)*, pages 371–376, Spain.

Hill, R. C. (2008). *Modular Verification and Supervisory Controller Design for Discrete-Event Systems Using Abstraction and Incremental Construction*. Ph.D. Thesis, Department of Mechanical Engineering, University of Michigan.

Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Menlo Park.

IEC (2013). IEC 61131-3: Third Edition. International Standard, International Electrotechnical Commission (IEC), Switzerland.

James, L. D., Teixeira, C. A., and Leal, A. B. (2019). Formal Design and Implementation of Supervisory Controller for a Didactic Manufacturing Cell. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT'19)*, pages 935–940, Australia.

Jimenez, I., Lopez, E., and Ramirez, A. (2001). Synthesis of Ladder Diagrams from Petri Nets Controller Models. In *Proceeding of the IEEE International Symposium on Intelligent Control (ISIC'01)*, pages 225–230, Mexico.

Kharrazi, A., Mishra, Y., and Sreeram, V. (2019). Discrete-Event Systems Supervisory Control for a Custom Power Park. *IEEE Transactions on Smart Grid*, **10**(1), 483–492.

Korssen, T., Dolk, V., van de Mortel-Fronczak, J., Reniers, M., and Heemels, M. (2018). Systematic Model-Based Design and Implementation of Supervisors for Advanced Driver Assistance Systems. *IEEE Transactions on Intelligent Transportation Systems*, **19**(2), 533–544.

Lauzon, S., Ma, A., Mills, J., and Benhabib, B. (1996). Application of Discrete-Event-System Theory to Flexible Manufacturing. *IEEE Control Systems Magazine*, **16**(1), 41–48.

Lauzon, S. C., Mills, J. K., and Benhabib, B. (1997). An Implementation Methodology for the Supervisory Control of Flexible Manufacturing Workcells. *Journal of Manufacturing Systems*, **16**(2), 91–101.

Leal, A., L. L. da Cruz, D., and Hounsell, M. (2012). PLC-Based Implementation of Local Modular Supervisory Control for Manufacturing Systems. In *Manufacturing System*, pages 159–182.

Leduc, R. J. (1996). *PLC Implementation of a DES Supervisor for a Manufacturing Testbed: An Implementation Perspective*. Master's Thesis, Department of Computer and Electrical Engineering, University of Toronto, Toronto, ON, Canada.

Leduc, R. J. (2001). *Hierarchical Interface-Based Supervisory Control*. Ph.D. Thesis, Department of Computer and Electrical Engineering, University of Toronto, Toronto, ON, Canada.

Leduc, R. J., Wang, Y., and Ahmed, F. (2014). Sampled-Data Supervisory Control. *Discrete Event Dynamic Systems*, **24**(4), 541–579.

Lind-Nielsen, J. (2002). BuDDy: Binary Decision Diagram Package. IT-University of Copenhagen (ITU).

Luo, J., Zhang, Q., Chen, X., and Zhou, M. (2018). Modeling and Race Detection of Ladder Diagrams via Ordinary Petri Nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, **48**(7), 1166–1176.

Ma, C. (2004). *Nonblocking Supervisory Control of State Tree Structures*. Ph.D. Thesis, Department of Computer and Electrical Engineering, University of Toronto, Toronto, ON, Canada.

Moon, I. (1994). Modeling Programmable Logic Controllers for Logic Verification. *IEEE Control Systems Magazine*, **14**(2), 53–59.

Moreira, M. V. and Basilio, J. C. (2014). Bridging the Gap Between Design and Implementation of Discrete-Event Controllers. *IEEE Transactions on Automation Science and Engineering*, **11**(1), 48–65.

Mulahuwaish, A. (2019). *Fault-Tolerant Supervisory Control*. Ph.D. Thesis, Department of Computing and Software, McMaster University, Hamilton, ON, Canada.

Peixoto, R. J. S., da Silva, L. D., and Perkusich, A. (2019). Model-Based Testing of Software for Automation Systems using Heuristics and Coverage Criterion. *Software and Systems Modeling (SoSyM)*, **18**(2), 797–823.

Prenzel, L. and Provost, J. (2018). PLC Implementation of Symbolic, Modular Supervisory Controllers. *IFAC-PapersOnLine*, **51**(7), 304–309.

Quezada, J. C., Medina, J., Flores, E., Seck Tuoh, J. C., and Hernández, N. (2014). Formal Design Methodology for Transforming Ladder Diagram to Petri Nets. *The International Journal of Advanced Manufacturing Technology*, **73**(5), 821–836.

Quezada, J. C., Flores, E., Baños, E., and Quezada, V. (2023). Petri Net Models of Discrete Logics Used in Control Algorithms Developed in Ladder Diagram Language. *The International Journal of Advanced Manufacturing Technology*, **124**(7), 2597–2612.

Ramadge, P. J. and Wonham, W. M. (1989). The Control of Discrete Event Systems. *Proceedings of the IEEE*, **77**(1), 81–98.

Ray, E. T. (2003). *Learning XML*. O' Reilly Media, Inc., 2nd edition.

Reijnen, F. F. H., Goorden, M. A., van de Mortel-Fronczak, J. M., and Rooda, J. E. (2017). Supervisory Control Synthesis for a Waterway Lock. In *Proceedings of the IEEE Conference on Control Technology and Applications (CCTA'17)*, pages 1562–1568, USA.

Rossi, O. and Schnoebelen, P. (2000). Formal Modeling of Timed Function Blocks for the Automatic Verification of Ladder Diagram Programs. In *Proceedings of 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM'2000)*, pages 177–182, Germany.

Schiffelers, R. R. H., Theunissen, R. J. M., van Beek, D. A., and Rooda, J. E. (2009). Model-Based Engineering of Supervisory Controllers using CIF. *Electronic Communication of the European Association of Software Science and Technology*, **21**(9), 1–10.

Seatzu, C., Silva, M., and van Schuppen, J. H., editors (2013). *Control of Discrete-Event Systems: Automata and Petri Net Perspectives*. Lecture Notes in Control and Information Sciences. Springer-Verlag, London.

Silva, D. B., Vieira, A. D., Loures, E. F. R., Busetti, M. A., and Santos, E. A. P. (2011). Dealing with Routing in an Automated Manufacturing Cell: A Supervisory Control Theory Application. *International Journal of Production Research*, **49**(16), 4979–4998.

Song, R. (2006). *Symbolic Hierarchical Interface-based Supervisory Control*. Master's Thesis, Department of Computing and Software, McMaster University, Hamilton, ON, Canada.

Swartjes, L., van Beek, D. A., Fokkink, W. J., and van Eekelen, J. A. W. M. (2017). Model-Based Design of Supervisory Controllers for Baggage Handling Systems. *Simulation Modelling Practice and Theory*, **78**, 28–50.

Szpak, R., de Queiroz, M. H., and Cury, J. E. R. (2020). Synthesis and Implementation of Supervisory Control for Manufacturing Systems Under Processing Uncertainities and Time Constraints. **53**(4), 229–234.

Thapa, D., Dangol, S., and Wang, G.-N. (2005). Transformation from Petri Nets Model to Programmable Logic Controller using One-to-One Mapping Technique. In *Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, pages 228–233, Austria.

Theunissen, R. J. M., Petreczky, M., Schiffelers, R. R. H., van Beek, D. A., and Rooda, J. E. (2014). Application of Supervisory Control Synthesis to a Patient Support Table of a Magnetic Resonance Imaging Scanner. *IEEE Transactions on Automation Science and Engineering*, **11**(1), 20–32.

Torta, E., Reniers, M., Kok, J., van de Mortel-Fronczak, J., and van de Molengraft, M. (2023). Synthesis-Based Engineering of Supervisory Controllers for ROS-Based Applications. *Control Engineering Practice*, **133**.

UPPAAL (2003). www.uppaal.com.

Uzam, M. (2012). A General Technique for the PLC-Based Implementation of RW Supervisors with Time Delay Functions. *The International Journal of Advanced Manufacturing Technology*, **62**(5), 687–704.

Uzam, M. and Gelen, G. (2009). The Real-Time Supervisory Control of an Experimental Manufacturing System Based on a Hybrid Method. *Control Engineering Practice*, **17**(10), 1174–1189.

Uzam, M. and Jones, A. H. (1998). Discrete Event Control System Design using Automation Petri Nets and their Ladder Diagram Implementation. *The International Journal of Advanced Manufacturing Technology*, **14**(10), 716–728.

Uzam, M. and Wonham, W. M. (2006). A Hybrid Approach to Supervisory Control of Discrete Event Systems Coupling RW Supervisors to Petri Nets. *The International Journal of Advanced Manufacturing Technology*, **28**(7), 747–760.

Uzam, M., Koç, İ. B., Gelen, G., and Aksebzeci, B. H. (2009a). Asynchronous Implementation of Discrete Event Controllers Based on Safe Automation Petri Nets. *The International Journal of Advanced Manufacturing Technology*, **41**(5), 595–612.

Uzam, M., Gelen, G., and Dalci, R. (2009b). A New Approach for the Ladder Logic Implementation of Ramadge-Wonham Supervisors. In *Proceedings of the 22nd International Symposium on Information, Communication and Automation Technologies (ICAT'09)*, pages 113–119, Sarajevo, Bosnia and Herzegovina.

Van der Sanden, B., Reniers, M., Geilen, M., Basten, T., Jacobs, J., Voeten, J., and Schiffelers, R. (2015). Modular Model-Based Supervisory Controller Design for Wafer Logistics in Lithography Machines. In *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 416–425, Canada.

Vieira, A., Santos, E., Hering de Queiroz, M., Leal, A., Neto, A., and Cury, J. (2017). A Method for PLC Implementation of Supervisory Control of Discrete Event Systems. *IEEE Transactions on Control Systems Technology*, **25**(1), 175–191.

Vieira, A. D., Cury, J. E. R., and de Queiroz, M. H. (2006). A Model for PLC Implementation of Supervisory Control of Discrete Event Systems. In *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, pages 225–232, Prague, Czech Republic.

Vogel-Heuser, B., Diedrich, C., Fay, A., Jeschke, S., Kowalewski, S., Wollschlaeger, M., and G, P. (2014). Challenges for Software Engineering in Automation. *Journal of Software Engineering and Applications*, **7**, 440–451.

Wang, Y. (2009). *Sampled-Data Supervisory Control*. Master's Thesis, Department of Computing and Software, McMaster University, Hamilton, ON, Canada.

Wang, Y. and Leduc, R. J. (2012). Sampled-Data Controller Implementation. *International Journal of Control*, **85**(9), 1343–1360.

Wightkin, N., Buy, U., and Darabi, H. (2011). Formal Modeling of Sequential Function Charts With Time Petri Nets. *IEEE Transactions on Control Systems Technology*, **19**(2), 455–464.

Wong, K. C. and Wonham, W. M. (1996). Hierarchical Control of Timed Discrete-Event Systems. *Discrete Event Dynamic Systems*, **6**(3), 275–306.

Wonham, W. M. and Cai, K. (2018). *Supervisory Control of Discrete-Event Systems*. Springer International Publishing.

Wonham, W. M. and Ramadge, P. J. (1987). On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal on Control and Optimization*, **25**(3), 637–659.

Wonham, W. M., Cai, K., and Rudie, K. (2018). Supervisory Control of Discrete-Event Systems: A Brief History. *Annual Reviews in Control*, **45**, 250–256.

Zaytoon, J. and Carre-Meneatrier, V. (2001). Synthesis of Control Implementation for Discrete Manufacturing Systems. *International Journal of Production Research*, **39**(2), 329–345.

Zaytoon, J. and Riera, B. (2017). Synthesis and Implementation of Logic Controllers – A Review. *Annual Reviews in Control*, **43**, 152–168.

Zhang, H., Feng, L., and Li, Z. (2020). Control of Black-Box Embedded Systems by Integrating Automaton Learning and Supervisory Control Theory of Discrete-Event Systems. *IEEE Transactions on Automation Science and Engineering*, **17**(1), 361–374.

Zhou, M., He, F., Gu, M., and Song, X. (2009). Translation-Based Model Checking for PLC Programs. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, pages 553–562, USA.

Zhou, M. C., Dicesare, F., and Rudolph, D. L. (1992). Design and Implementation of a Petri Net Based Supervisor for a Flexible Manufacturing System. *Automatica*, **28**(6), 1199–1208.

Zoubek, B., Roussel, J.-M., and Kwiatkowska, M. (2003). Towards Automatic Verification of Ladder Logic Programs. In *Proceedings of IMACS Multiconference on Computational Engineering in Systems Applications (CESA'03)*, France.

Appendix A

Miscellaneous Definitions

In this appendix, we give some miscellaneous definitions that are used in the fundamental DES concepts described in Chapter 2.

A.1 Equivalence Relation

Definition A.1.1. Let X be a nonempty set. Let $E \subseteq X \times X$ be a binary relation on X. The relation E is an *equivalence relation* on X if:

1. $(\forall x \in X)\, xEx$ (E is *reflexive*)
2. $(\forall x, x' \in X)\, xEx' \Rightarrow x'Ex$ (E is *symmetric*)
3. $(\forall x, x', x'' \in X)\, xEx' \;\&\; x'Ex'' \Rightarrow xEx''$ (E is *transitive*)

In this definition, we use the standard infix notation xEx' to represent the ordered pair $(x, x') \in E$. Instead of xEx', we shall often write $x \equiv x' (\text{mod } E)$.

A.2 Product Operator

Definition A.2.1. Let $\mathbf{G}_1 = (Q_1, \Sigma, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma, \delta_2, q_{o,2}, Q_{m,2})$ be two DES defined over the same event set Σ. The *product* of two DES is defined as:

$$\mathbf{G}_1 \times \mathbf{G}_2 := (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$$

where $\delta_1 \times \delta_2 : Q_1 \times Q_2 \times \Sigma \to Q_1 \times Q_2$ is given by:

$$(\delta_1 \times \delta_2)\,((q_1, q_2), \sigma) := (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$$

whenever $\delta_1(q_1, \sigma)!$ and $\delta_2(q_2, \sigma)!$.

By this definition, we have:

$$L(\mathbf{G}_1 \times \mathbf{G}_2) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2) \quad \text{and} \quad L_m(\mathbf{G}_1 \times \mathbf{G}_2) = L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2)$$

A.3 Meet Operator

Definition A.3.1. The *meet* of two DES $\mathbf{G}_1$ and $\mathbf{G}_2$, represented as $\mathbf{G} = \mathbf{meet}(\mathbf{G}_1, \mathbf{G}_2)$, is the reachable sub-DES of the product DES $\mathbf{G}_1 \times \mathbf{G}_2$.

A.4 Selfloop Operation

Definition A.4.1. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a DES defined over Σ. Let Σ' be another set of events such that $\Sigma \cap \Sigma' = \emptyset$. The *selfloop* operation on $\mathbf{G}$ is used to generate a new DES $\mathbf{G}'$ by selflooping each event in Σ' at every state of $\mathbf{G}$. Formally, this is expressed as:

$$\mathbf{G}' = \mathbf{selfloop}(\mathbf{G}, \Sigma') = (Q, \Sigma \cup \Sigma', \delta', q_o, Q_m)$$

where $\delta' : Q \times (\Sigma \cup \Sigma') \to Q$ is a partial function defined as:

$$\delta'(q, \sigma) := \begin{cases} \delta(q, \sigma) & \sigma \in \Sigma,\ \delta(q, \sigma)! \\ q & \sigma \in \Sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

A.5 Bijective Function

Definition A.5.1. A function $f : A \to B$ is:

1. **injective** if for every $x, y \in A, x \neq y \Rightarrow f(x) \neq f(y)$;
2. **surjective** if for every $b \in B$ there is an $a \in A$ with $f(a) = b$;
3. **bijective** if f is both injective and surjective.

Appendix B

Symbolic Verification

In this appendix, we provide some content related to symbolic verification from Wang (2009) for the sake of completeness. Specifically, we restate the definition for symbolic representation of transitions in the SD supervisory control theory that we have referred to in Chapter 9. Also, we present some predicate-based algorithms that we are reusing to verify some properties in our $||_{SD}$ setting. Please see Wang (2009) for complete details.

B.1 Symbolic Representation of Transitions

Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 \times \mathbf{G}_2 \times \ldots \times \mathbf{G}_n$ be the product TDES of component TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ for $i = 1, 2, \ldots, n$. For any state $q \in Q$, we have $q = (q_1, q_2, \ldots, q_n)$ where $q_i \in Q_i$.

Definition B.1.1. For $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times \ldots \times \mathbf{G}_n$, let $\sigma \in \Sigma$. A *transition predicate* $N_\sigma : Q \times Q \to \{T, F\}$ identifies all the transitions for σ in $\mathbf{G}$ and is defined as follows:

$$(\forall q, q' \in Q)\, N_\sigma(q, q') := \begin{cases} T & \text{if } \delta(q,\sigma)! \;\&\; \delta(q,\sigma) = q' \\ F & \text{otherwise} \end{cases}$$

In order to distinguish between source states and destination states, two different vectors of state variables are needed, as defined below.

Definition B.1.2. For $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times \ldots \times \mathbf{G}_n$, let $i = 1, 2, \ldots, n$. For each $\mathbf{G}_i$, we have the *normal state variable* v_i (source state) and the *prime state variable* v_i' (destination state), both with domain Q_i. For $\mathbf{G}$, we have the *normal state variable vector* $\mathbf{v} = [v_1, v_2, \ldots, v_n]$ and the *prime state variable vector* $\mathbf{v}' = [v_1', v_2', \ldots, v_n']$.

For each $\sigma \in \Sigma$, we can write the transition predicate for σ, N_σ, as below. Essentially, if we set $\mathbf{v} = q$ and $\mathbf{v}' = q'$ such that $\delta(q, \sigma) = q'$, then $N_\sigma(\mathbf{v}, \mathbf{v}')$ will return T.

Definition B.1.3. We use the *transition tuple* $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ to represent the transition

on σ, where $\mathbf{v}_\sigma := \{v_i \in \mathbf{v} \mid \sigma \in \Sigma_i\}, \mathbf{v}'_\sigma := \{v'_i \in \mathbf{v}' \mid \sigma \in \Sigma_i\}$, and

$$N_\sigma(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \leq i \leq n \mid \sigma \in \Sigma_i\}} \left(\bigvee_{\{q_i, q'_i \in Q_i \mid \delta_i(q_i, \sigma) = q'_i\}} (v_i = q_i) \wedge (v'_i = q'_i) \right)$$

B.2 Symbolic Verification of $||_{SD}$ Properties

In this section, we present some predicate-based algorithms from Wang (2009) that can be reused in our $||_{SD}$ setting without altering the actual algorithm steps. It is notable that we will only mention the underlying modifications while using these algorithms to verify properties in our $||_{SD}$ setting. Please see the complete description of these algorithms in Wang (2009).

B.2.1 Nonblocking

Algorithm B.1[1] checks the property of nonblocking on the input TDES $\mathbf{G}$. As this property has not changed for our $||_{SD}$ setting, we can use this algorithm from the SD setting to check our $||_{SD}$ system.

Algorithm B.1 Nonblocking($\mathbf{G}$)

1: $P_{reach} \leftarrow R(\mathbf{G}, true)$
2: $P_{coreach} \leftarrow \mathcal{CR}(\mathbf{G}, P_m, \Sigma, P_{reach})$
3: **if** $(P_{reach} \wedge \neg P_{coreach} \not\equiv false)$ **then**
4: **return** $False$
5: **end if**
6: **return** $True$

As we are interested in making sure that our closed-loop system is nonblocking, we will provide our closed-loop system of the $||_{SD}$ setting as an input to this algorithm, which is different from the closed-loop system used in the SD setting. Also, R (Algorithm 9.1) and $\mathcal{CR}$ (Algorithm 9.2) will be computed using our function definitions, as explained in Section 9.3.2.

B.2.2 Activity Loop Free

As the original activity-loop-free (ALF) property remains unchanged in our $||_{SD}$ setting, we can use Algorithm B.2 of the SD setting without changing its steps.

The only difference is the TDES $\mathbf{G}$ that we provide as an input to this algorithm. In our $||_{SD}$ setting, parameter $\mathbf{G}$ will represent our closed-loop system that we want

[1]Line 2 of this algorithm is different from the corresponding line of Algorithm 6.5 given in Wang (2009). This is due to the incorrect number of parameters specified in the original algorithm. We have fixed this error in this version.

Algorithm B.2 ALF(**G**)

1: $P_{chk} \leftarrow R(\mathbf{G}, true)$
2: $P_{tmp} \leftarrow false$
3: **for** $(q \models P_{chk})$ **do**
4: $\quad P_{visit} \leftarrow \left(\bigvee_{\sigma \in \Sigma_{act}} \hat{\delta}(pr(\{q\}), \sigma) \right) \wedge P_{chk}$
5: $\quad overlap \leftarrow False$
6: $\quad P_{next} \leftarrow P_{visit}$
7: $\quad$ **repeat**
8: $\quad\quad P_{next} \leftarrow \left(\bigvee_{\sigma \in \Sigma_{act}} \hat{\delta}(P_{next}, \sigma) \right) \wedge P_{chk}$
9: $\quad\quad P_{tmp} \leftarrow P_{visit}$
10: $\quad\quad$ **if** $(P_{visit} \wedge P_{next} \not\equiv false)$ **then**
11: $\quad\quad\quad overlap \leftarrow True$
12: $\quad\quad$ **end if**
13: $\quad\quad P_{visit} \leftarrow P_{visit} \vee P_{next}$
14: $\quad\quad$ **if** $(q \models P_{visit})$ **then**
15: $\quad\quad\quad$ **return** $False$
16: $\quad\quad$ **end if**
17: $\quad$ **until** $(P_{visit} \equiv P_{tmp})$
18: $\quad P_{chk} \leftarrow P_{chk} - pr(\{q\})$
19: $\quad$ **if** $(\neg\, overlap)$ **then**
20: $\quad\quad P_{chk} \leftarrow P_{chk} - P_{visit}$
21: $\quad$ **end if**
22: **end for**
23: **return** $True$

to make sure is ALF. As our input **G** is constructed in a different way than the closed-loop system of the SD setting, Algorithm B.2 will use our R (Algorithm 9.1) at **line 1**, and our Definition 9.3.2 of the function $\hat{\delta}$ at **lines 4** and **8** to compute the required predicates while verifying the ALF property in our $||_{SD}$ setting. Please recall that $\hat{\delta}$ relies on N_σ, and the definition of N_σ in our $||_{SD}$ setting (Definition 9.2.5) is different from the SD setting (Definition $B.1.3$).

B.2.3 Proper Time Behaviour

The property of proper time behaviour is defined only in terms of TDES plant **G**. As **G** is same in the SD and $||_{SD}$ settings, Algorithm B.3 remains unmodified with respect to its steps, underlying functions, and input **G**.

Algorithm B.3 ProperTimeBehaviour($\mathbf{G}$)

1: $P_1 \leftarrow \bigvee_{\sigma \in \Sigma_u \cup \{\tau\}} \hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma)$
2: $P_2 \leftarrow R(\mathbf{G}, true)$
3: **if** $(P_2 - P_1 \not\equiv false)$ **then**
4: **return** $False$
5: **end if**
6: **return** $True$

B.2.4 S-Singular Prohibitable Behaviour with $||_{SD}$

The property of **S**-singular prohibitable behaviour with $||_{SD}$ is verified at **lines 12-16** of Algorithm B.5. Please note that we have redefined the meaning of variables Σ_{poss}, B_{conc}, and node b in our $||_{SD}$ setting, as discussed in Section 9.5.3. Please refer to Section B.3.2 for further details on Algorithm B.5.

B.3 Symbolic Verification of SD Controllability with $||_{SD}$

In this section, we present algorithms from Wang (2009) that can be used to verify Point ii and Point iii of our SD controllability with $||_{SD}$ property. Please recall that these two points correspond to Points iii and iv respectively of the SD controllability property (Definition 3.5.1) in the SD setting.

It is worth-mentioning that these algorithms can be used in our $||_{SD}$ setting without modifying their steps. However, the input parameters and the underlying definitions for some variables and functions used by these algorithms have changed in our $||_{SD}$ setting. Therefore, it is our implicit assumption that while verifying properties in our $||_{SD}$ setting, these algorithms use the variable and function definitions as specified in Chapter 9. In particular, from Chapter 9, we have used functions, $\hat{\delta}$ (Definition 9.3.2) and $\hat{\delta}_{\mathbf{G}}$ (Definition 9.4.2), as well as R (Algorithm 9.1) to calculate P_{reach}. Please refer to Section 9.5.3 for the definition of variables used in the following algorithms.

B.3.1 Point ii.1

In order to verify Point ii.1 of SD controllability with $||_{SD}$, Algorithm B.4 analyzes the concurrent behaviour of sampled state q_{ss} represented by predicate P_{ss}. Starting at q_{ss}, it builds a reachability tree until all nodes terminate at a *tick* event or one of the checks fails. As the tree is built, Point ii.1 of SD controllability with $||_{SD}$ property is tested at **line 22**.

This algorithm also calls Algorithms B.5 and B.6 that contribute in verifying Point ii.2 of SD controllability with $||_{SD}$.

Algorithm B.4 AnalyzeSampledState($\mathbf{G}, \mathbf{S}, P_{SF}, Z_{SP}, P_{reach}, P_{ss}, pNerFail$)

1: $B_{map} \leftarrow \{(0, P_{ss})\}$
2: $B_{conc} \leftarrow \emptyset$
3: $B_p \leftarrow \{0\}$
4: $nextLabel \leftarrow 1$
5: $\text{Occu}_B \leftarrow \{(0, \emptyset)\}$
6: **while** $(B_p \neq \emptyset)$ **do**
7: $b \leftarrow \text{Pop}(B_p)$
8: $P_q \leftarrow B_{map}(b)$
9: $\Sigma_{poss} \leftarrow \emptyset$
10: $\Sigma_{\mathbf{G}poss} \leftarrow \emptyset$
11: **for all** $(\sigma \in \Sigma)$ **do**
12: **if** $(\hat{\delta}(P_q, \sigma) \not\equiv false)$ **then**
13: $\Sigma_{poss} \leftarrow \Sigma_{poss} \cup \{\sigma\}$
14: **end if**
15: **if** $(\hat{\delta}_{\mathbf{G}}(P_q, \sigma) \not\equiv false)$ **then**
16: $\Sigma_{\mathbf{G}poss} \leftarrow \Sigma_{\mathbf{G}poss} \cup (\{\sigma\} \cap \Sigma_{hib})$
17: **end if**
18: **end for**
19: **if** $(P_q \equiv P_{ss})$ **then**
20: $\Sigma_{Elig} \leftarrow \Sigma_{poss} \cap \Sigma_{hib}$
21: **end if**
22: **if** $((\Sigma_{poss} \cup \text{Occu}_B(b)) \cap \Sigma_{hib} \neq \Sigma_{Elig})$ **then**
23: **return** $False$
24: **end if**
25: **if** $(\neg\, \text{NextState}(b, \Sigma_{poss}, \Sigma_{\mathbf{G}poss}, P_q, nextLabel, B_{map}, B_p, B_{conc}, P_{SF}, Z_{SP}, \text{Occu}_B(b)))$ **then**
26: **return** $False$
27: **end if**
28: **end while**
29: CheckNerodeCells($B_{conc}, \text{Occu}_B, pNerFail$)
30: **return** $True$

B.3.2 Point ii.2

The following algorithms can be used to verify Point ii.2 of the SD controllability with $||_{SD}$ property.

Next State

Algorithm B.5 is called by Algorithm B.4 to determine all the next states that need to be processed for checking Point ii.2 of SD controllability with $||_{SD}$. This algorithm

Algorithm B.5 NextState(...)

```
1: if (Σ_poss = ∅) then
2:     return True
3: end if
4: if (τ ∈ Σ_poss) then
5:     P_q' ← δ̂(P_q, tick)
6:     Push(B_conc, (b, P_q'))
7:     if (P_q' ∧ P_SF ≡ false) then
8:         P_SF ← P_SF ∨ P_q'
9:         Push(Z_SP, P_q')
10:    end if
11: end if
12: for all (σ ∈ Σ_Gposs) do
13:    if (Occu_B(b) ∩ {σ} ≠ ∅) then
14:        return False
15:    end if
16: end for
17: for all (σ ∈ Σ_poss − {τ}) do
18:    P_q' ← δ̂(P_q, σ)
19:    b' ← nextLabel
20:    nextLabel ← nextLabel + 1
21:    Push(B_map, (b', P_q'))
22:    Push(B_p, b')
23:    Push(Occu_B, (b', Occu_B(b) ∪ {σ}))
24: end for
25: return True
```

also checks the property of **S**-singular prohibitable behaviour with $||_{SD}$. This check is performed at **lines 12-16**.

Check Nerode Cells

Algorithm B.6 is called by Algorithm B.4 to determine if there are possible violations for Point ii.2 of SD controllability with $||_{SD}$. These potentially problematic states are recorded in $pNerFail$ to be analyzed later.

Recheck Nerode Cells

Algorithm B.7 is called by Algorithm 9.5 to recheck Point ii.2 with respect to the potentially problematic states stored in $pNerFail$. It makes use of Algorithm B.8 to conclude its result.

Algorithm B.6 CheckNerodeCells($B_{conc}, \text{Occu}_B, pNerFail$)

```
1: while (B_conc ≠ ∅) do
2:     (b, P_q) ← Pop(B_conc)
3:     Z_eqv ← ∅
4:     Push(Z_eqv, P_q)
5:     sameCell ← True
6:     for all ((b', P_q') ∈ B_conc) do
7:         if (Occu_B(b) = Occu_B(b')) then
8:             Push(Z_eqv, P_q')
9:             B_conc ← B_conc − {(b', P_q')}
10:            if (P_q ≢ P_q') then
11:                sameCell ← False
12:            end if
13:        end if
14:    end for
15:    if (¬ sameCell) then
16:        Push(pNerFail, Z_eqv)
17:    end if
18: end while
19: return
```

Algorithm B.7 RecheckNerodeCells($pNerFail$)

```
1: if (pNerFail = ∅) then
2:     return True
3: end if
4: Visited ← ∅
5: while (pNerFail ≠ ∅) do
6:     Z_eqv ← Pop(pNerFail)
7:     if (¬ RecheckNerodeCell(Z_eqv, Visited)) then
8:         return False
9:     end if
10: end while
11: return True
```

Recheck Nerode Cell

Algorithm B.8 is called by Algorithm B.7 to recheck Point ii.2 with respect to the potentially problematic states stored in $pNerFail$. It determines if the set of states that this algorithm is called with are λ-equivalent to each other. If they are not, the algorithm returns $False$ indicating the violation of Point ii.2 of the SD controllability with $||_{SD}$ property.

Algorithm B.8 RecheckNerodeCell($Z_{eqv}, Visited$)

1: $P_{q_1} \leftarrow \text{Pop}(Z_{eqv})$
2: $Pending \leftarrow \emptyset$
3: **while** $(Z_{eqv} \neq \emptyset)$ **do**
4: $\quad P_{q_2} \leftarrow \text{Pop}(Z_{eqv})$
5: $\quad \text{Push}(Pending, (P_{q_1}, P_{q_2}))$
6: **end while**
7: **while** $(Pending \neq \emptyset)$ **do**
8: $\quad (P_{q_1}, P_{q_2}) \leftarrow \text{Pop}(Pending)$
9: $\quad P \leftarrow P_{q_1} \vee P_{q_2}$
10: $\quad$ **if** $((P \wedge P_m \not\equiv false)$ & $(P \wedge P_m \not\equiv P))$ **then**
11: $\quad\quad$ **return** $False$
12: $\quad$ **end if**
13: $\quad$ **for all** $(\sigma \in \Sigma)$ **do**
14: $\quad\quad P' \leftarrow \hat{\delta}(P, \sigma)$
15: $\quad\quad P'_{q_1} \leftarrow \hat{\delta}(P_{q_1}, \sigma)$
16: $\quad\quad P'_{q_2} \leftarrow \hat{\delta}(P_{q_2}, \sigma)$
17: $\quad\quad$ **if** $(P' \not\equiv false)$ **then**
18: $\quad\quad\quad$ **if** $((P'_{q_1} \wedge P' \not\equiv false)$ & $(P'_{q_2} \wedge P' \not\equiv false))$ **then**
19: $\quad\quad\quad\quad$ **if** $((P'_{q_1} \not\equiv P'_{q_2})$ & $((P'_{q_1}, P'_{q_2}) \notin Visited))$ **then**
20: $\quad\quad\quad\quad\quad \text{Push}(Visited, (P'_{q_1}, P'_{q_2}))$
21: $\quad\quad\quad\quad\quad \text{Push}(Visited, (P'_{q_2}, P'_{q_1}))$
22: $\quad\quad\quad\quad\quad \text{Push}(Pending, (P'_{q_1}, P'_{q_2}))$
23: $\quad\quad\quad\quad$ **end if**
24: $\quad\quad\quad$ **else**
25: $\quad\quad\quad\quad$ **return** $False$
26: $\quad\quad\quad$ **end if**
27: $\quad\quad$ **end if**
28: $\quad$ **end for**
29: **end while**
30: **return** $True$

B.3.3 Point iii

Algorithm B.9 verifies Point iii of SD controllability with $||_{SD}$. It determines if there exists a marked state in $\mathbf{G}_{cl} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ with an incoming non-*tick* transition from a reachable state. If such a state exists, Point iii fails and the algorithm returns $False$. Please note that in our $||_{SD}$ setting, $P_{reach} = R(\mathbf{S} \,||_{SD}\, \mathbf{G}, true)$.

Algorithm B.9 CheckSDCPointiii(P_{reach})

1: $P \leftarrow \bigvee_{\sigma \in \Sigma - \{\tau\}} \hat{\delta}(P_{reach}, \sigma)$
2: **if** $(P \wedge P_m \not\equiv false)$ **then**
3: **return** $False$
4: **end if**
5: **return** $True$

Appendix C

TDES to Moore FSM Translation

While developing our FSM-TDES translation approach (described in Chapter 11), one of our major goals is to make our approach consistent and compatible with the existing TDES-FSM translation method, developed by Wang (2009); Wang and Leduc (2012), and implemented by Hamid (2014) in DESpot (2023). This provides an additional benefit by enabling the system designers to go back and forth between the two models using the two translation approaches, i.e. easily translate one model into the other, regardless of whether they started with a TDES supervisor or a Moore FSM.

It is obvious that Hamid's (2014) TDES-FSM implementation method was not developed taking into consideration the possibility of doing the reverse (FSM-TDES) translation. For this reason, we needed to make some changes in the existing TDES-FSM translation approach of DESpot in order to build compatibility between the two translation approaches. We describe these changes in this appendix.

We begin this appendix by presenting the modified XML (Ray, 2003) file format and structure for expressing the Moore FSM. Then, we describe the changes that we have made in the existing TDES-FSM translation method of DESpot. Specifically, we present our DESpot algorithms that are meant to replace Hamid's algorithms for doing the TDES-FSM translation and generating the output FSM files. Please note that these updated FSM files serve as an input to our FSM-TDES translation approach.

C.1 XML File Structure for Moore System

In this section, we describe the XML (Ray, 2003) file structure for our central and individual input FSM, along with the modifications that we have made in Hamid's output FSM XML file format. In Chapter 13, we apply our Moore FSM-TDES translation approach to an example of a 4-bit Combination Lock. We will use a small portion of this example to explain the XML file format.

The 4-bit Combination Lock is a digital lock system that uses a 4-bit passcode

to provide secured access to authentic users, and has three user buttons: **Enter**, **Change** and **Reset**. Users can unlock the Combination Lock, either to *open* the door or *change* the currently saved passcode, by entering the existing passcode and pressing the appropriate button (**Enter** or **Change**). However, if the user enters an incorrect passcode, the *alarm* goes off. *Alarm* can only be cancelled by pressing the **Reset** button, which will also reset the currently saved passcode to its default value. Please refer to Section 13.1 for a thorough explanation of the complete system specification and functionality.

C.1.1 Individual Moore FSM

Figure 11.1 (on page 181) shows the **OpenLock** Moore FSM from the Combination Lock example. We will use this FSM to describe how it can be expressed in the required XML file format. Our FSM-TDES translation method requires the designers to specify one such XML file corresponding to each individual Moore FSM that needs to be translated. Please refer to Section 13.2.1 to gain familiarity with the complete graphical notation of a Moore FSM.

The **OpenLock** FSM, as the name suggests, controls the functionality of unlocking the Combination Lock. In the lock state (state "1", where output of signal *open* = 0), if the user enters the correct passcode (indicated by the input signal, *equal*) and presses the **Enter** button (represented by the input signal, *enter*), the door opens (state "2", where *open* = 1). The door stays open until the user presses **Enter** again, after which the system returns back to its locked state.

Please note that in the boolean expressions of the Moore FSM, "$\cdot$" symbol represents the **AND** operator, "+" represents the **OR** operator, and "!" represents the **NOT** operator. However, in our figures and XML input files, we will write " . " instead of "$\cdot$", as " . " is easier to produce.

The XML Input File C.1 represents the **OpenLock** FSM in the XML format. At **line 1**, the XML file starts with a declaration that this is an XML document and states its *version number*, *encoding scheme* and *external dependencies*. **Line 2** defines the root element <FSM> that specifies the name of the individual FSM, which in our case is **OpenLock**. The reset/initial state of **OpenLock** is state "1". This is specified in the XML file at **line 3**, inside the <ResetState> element.

The **OpenLock** FSM has three signals: one input-output (IO) signal (*open*), and two input signals (*enter* and *equal*). The <Signals> element is defined in the XML file to contain a list of all the FSM signals (**lines 4-8**). Inside <Signals>, each <Signal> element specifies the name, order and type of one FSM signal (**lines 5-7**). The input-output signals of the FSM have type "IO" (**line 5**), whereas input signals are of type "I" (**lines 6-7**).

At each state of the FSM, we specify the output information for the IO signals, i.e. whether the output of the IO signal is $True$ (1) or $False$ (0) at this state. The **OpenLock** FSM has two states, "1" and "2". The output of the *open* signal is set to

XML Input File C.1: Moore FSM **OpenLock**

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <FSM Name="OpenLock">
3    <ResetState Name="1"> </ResetState>
4    <Signals>
5      <Signal Name="open" order="0" type="IO"/>
6      <Signal Name="enter" order="1" type="I"/>
7      <Signal Name="equal" order="2" type="I"/>
8    </Signals>
9    <States>
10     <State Name="1" outputvector=""/>
11     <State Name="2" outputvector="open"/>
12   </States>
13   <Transitions>
14     <StartState Name="1">
15       <Transition inputvector="enter.equal" endstate="2"/>
16       <Transition inputvector="!enter" endstate="1"/>
17       <Transition inputvector="!equal" endstate="1"/>
18     </State>
19     <StartState Name="2">
20       <Transition inputvector="enter" endstate="1"/>
21       <Transition inputvector="!enter" endstate="2"/>
22     </State>
23   </Transitions>
24 </FSM>
```

False at state "1", and *True* at state "2". This information about the FSM states and their corresponding output information is enclosed in the <States> element in the XML file (**lines 9-12**). We write one <State> element corresponding to each state of the FSM (**lines 10-11**). Inside this element, the *outputvector* contains a comma separated list of all the FSM IO signals whose outputs are set to *True* at the state identified by *Name*. At any given state, if all outputs are set to *False* (e.g. at state "1" of **OpenLock**), then *outputvector* will represent an empty string (**line 10**).

The <Transitions> element of the XML file contains all the next state logics (NSL) specified in the FSM (**lines 13-23**). We write one <StartState> element

corresponding to each state of the FSM (**lines 14-18, 19-22**). This element encloses all the NSL defined at the FSM state identified by *Name* (source state). In each <Transition> element, the *inputvector* represents the next state conditions that take the FSM from *StartState* to the *endstate* (destination state) (**lines 15-17, 20-21**).

Our FSM-TDES translation method requires that the next state conditions of the *inputvector* must be specified either: 1) as *boolean expressions*, or 2) using one of our three reserved *keywords*. As we have already discussed about boolean expressions in Section 11.1.1, below we only discuss our reserved keywords with reference to the XML file. We have introduced these keywords as shorthand notations in the XML file in order to enable the designers to conveniently specify some generic next state conditions in their individual FSM.

i) Tick: <TICK>

One of the possible next state conditions that needs to be specified at every state of an FSM is that all FSM input signals have the value of '0', i.e. none of the FSM inputs occurred in the previous clock period. For example, if an FSM has three inputs, $open, enter$ and $equal$, the boolean expression to represent this next state condition is "$!open \cdot !enter \cdot !equal$". It is obvious that as the signals of an FSM increase in number, this boolean expression becomes lengthier.

As this next state condition is generic for all individual FSM, i.e. '0' for all FSM inputs regardless of how many signals an FSM have, we introduce the keyword of <**TICK**> to represent it. We are using **TICK** as our keyword because this next state condition of the FSM corresponds to a concurrent string of the TDES supervisor that does not contain any activity events, i.e. a concurrent string that only contains a *tick* event.

As an example, consider a sample Moore FSM named **TestFSM** given in XML Input File C.2. This FSM goes from state x1 to state x2, when all the FSM signals are '0'. Instead of writing a lengthy boolean expression that **ANDs** together all **TestFSM** signals in their complemented form, we have simply written our <**TICK**> keyword to specify this next state condition (**line 13**).

As we are introducing this keyword in the input XML file for our FSM-TDES translation approach, we need to modify DESpot's existing TDES-FSM translation algorithm and add the support for this keyword to make the two translation approaches compatible with each other. We will discuss our modified algorithm later in Section C.2.

ii) Global Don't Care: <GDC>

While defining the next state conditions of an FSM, the designers may wish to specify an input combination at an FSM state where every signal of the FSM is a DC. We refer to this next state condition as the "*Global Don't Care (GDC)*" condition. The reason is that the FSM does not care what the value of each signal is. Rather, it always goes to the same next state, which could even be the current state of the FSM, depending upon its design logic. This means that at a given

XML Input File C.2: Sample Moore FSM **TestFSM**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<FSM Name="TestFSM">
  ...
  <Signals>
    ...
  </Signals>
  <States>
    ...
  </States>
  <Transitions>
    ...
    <StartState Name="x1">
      <Transition inputvector="<TICK>" endstate="x2"/>
      ...
    </State>
    ...
    <StartState Name="x3">
      <Transition inputvector="<GDC>" endstate="x4"/>
    </State>
    ...
    <StartState Name="x4">
      <Transition inputvector="<DEF>" endstate="x4"/>
    </State>
    ...
  </Transitions>
</FSM>
```

FSM state, all the specified input combinations have the very same destination state.

As discussed in Section 11.1.1, if we want to specify an FSM signal as a DC, we can simply exclude this signal from our boolean expression to make it compact. However, expressing GDC by following this strategy becomes complicated, as every signal of the FSM is a DC.

One possible way to express a GDC condition is to specify it using non-minimal

boolean expressions. For example, if an FSM has two signals, *alpha* and *beta*, then we can express our GDC by writing four boolean expressions, "$alpha \cdot beta$", "$alpha \cdot !beta$", "$!alpha \cdot beta$", and "$!alpha \cdot !beta$". It is easy to see that as the FSM signals increase in number, writing these non-minimal boolean expressions to cover all possible input combinations becomes a laborious task. For instance, for an FSM that has four signals, the designers may need to write $2^4 = 16$ boolean expressions to specify GDC.

In order to make it easier for the designers to manually specify a GDC condition, we introduce the <**GDC**> keyword for the input XML files of our FSM-TDES translation approach. In the **TestFSM** given in XML Input File C.2, we specify a **GDC** condition at state x3 (**lines 17-19**). As per the given design logic, if **TestFSM** is at state x3, then regardless of what input combination it gets, it always goes to state x4.

It is noteworthy that **GDC** covers all possible next state conditions that could be defined at an FSM state. This means that if we specify **GDC** at any given state of the FSM, then it must be the *only* transition defined at this state and no other next state conditions could be specified at this FSM state. Otherwise, our FSM might become non-deterministic.

In our TDES-FSM translation algorithm developed for DESpot, discussed in Section C.2, we also add support for the <**GDC**> keyword. This will allow designers to use the DESpot's output FSM file directly as an input to our FSM-TDES translation method, without any modification.

iii) Default: <DEF>

In the TDES-FSM translation approach, Wang (2009) introduces the shorthand notation of a *default transition*, abbreviated as **DEF**. This is because the transition function of a TDES supervisor is a partial function, whereas the next state function of a Moore FSM must be a total function. Wang added a selfloop of **DEF** transition at every state of the translated FSM in order to cover the next state conditions (input combinations) that are not explicitly specified in the TDES supervisor. In other words, Wang used a selfloop of **DEF** in the translated FSM to represent *invalid* transitions of the TDES supervisor. By this, we mean transitions that the supervisor asserts cannot occur.

At any given state of the FSM, **DEF** is equivalent to taking the logical **OR** of all the next state conditions explicitly defined at this state, and then negating the result. It is worth clarifying that at any FSM state, if the explicitly specified next state conditions cover all possible input combinations, then **DEF** will be empty at this state.

While implementing Wang's TDES-FSM translation approach in DESpot (2023), Hamid (2014) interpreted the **DEF** transition in a completely different and contradictory way. In his translated FSM, Hamid uses **DEF** to represent all the input combinations that do not cause a state change, i.e. **DEF** represents all the next state conditions that are selflooped. Hamid did not take into account whether

these next state conditions were explicitly defined in the TDES supervisor or not, and treated both *valid* and *invalid* selflooped next state conditions in the same way.

In our FSM-TDES translation approach, we will use **DEF** transition with its original and correct meaning, as defined by Wang (2009). This means that we need to modify Hamid's TDES-FSM translation algorithm so that DESpot generates the output FSM XML file with the correct meaning of **DEF** transition. We will discuss this further in Section C.2.

As mentioned earlier, **DEF** represents the invalid next state conditions that cannot occur in the physical system. Keeping this in view, it seems reasonable and logical that our FSM-TDES translation method does **not** generate any transitions in the translated TDES supervisor corresponding to the **DEF** transition of the FSM. Also, since the TDES transition function is a partial function, there is no need to make our translated supervisor complicated and cluttered by generating invalid transitions corresponding to **DEF**, when these transitions cannot even happen in the closed-loop system.

In our input FSM format, we allow the inclusion of **DEF**, primarily to keep our input structure consistent with the DESpot's TDES-FSM translation output. However, if designers are manually designing their FSM, then we ***strongly discourage*** the use of **DEF** transition. Rather, we encourage the designers to explicitly specify all the next state conditions of the FSM and to not rely on **DEF** to make the FSM's next state function a total function. This is exactly what we have done while manually designing the FSM for our Combination Lock example (Chapter 13) and writing their corresponding XML files. That is why, there is no next state condition expressed as **DEF** in our **OpenLock** FSM and its corresponding XML file.

We discourage the use of **DEF** while manually designing FSM because it is really easy to misinterpret and misuse the **DEF** transition and introduce design errors. For instance, since **DEF** is always defined as a selfloop, it looks very tempting just to define one **DEF** transition at every state of the FSM to cover all the next state conditions that do not cause an explicit state change (as Hamid (2014) did for his translated FSM). However, by doing so, designers make the *valid* selflooped next state conditions of their FSM *invalid* by merging them together and using **DEF** to represent all of them.

In this case, since the manually designed FSM does not specify the correct next state conditions that designers actually wanted to specify, it is obvious that the corresponding TDES supervisor generated by our FSM-TDES translation method will be incorrect. Precisely, valid transitions will be missing from the translated supervisor because valid selflooped next state conditions were merged with **DEF** in the input FSM, and our translation method does not generate anything corresponding to **DEF**. Since the translated supervisor is logically incorrect, this in turn makes it more likely that it will fail the desired TDES and $||_{SD}$ properties

(Chapter 4).

It is worth-mentioning here that the only situation when designers must write a **DEF** transition in the input FSM is if no valid next state conditions exist at an FSM state. In this case, designers must specify a selfloop of **DEF** transition in order to fulfill the requirement of making the FSM's next state function a total function. For example, if no valid next state conditions exist at state x4 of our **TestFSM**, then we can specify **DEF** at this state, as shown in XML Input File C.2 (**lines 21-23**).

C.1.2 Central FSM

Our FSM-TDES translation approach requires the designers to specify one central FSM file corresponding to the control system whose one or more individual FSM need to be translated into TDES supervisor(s). We use the same XML file format for specifying our central FSM that Hamid (2014) has defined while implementing his TDES-FSM translation method in DESpot (2023). As a result, the corresponding DESpot algorithm to generate the central FSM XML file remains unmodified. Please refer to Hamid (2014) to see this algorithm.

XML Input File C.3 shows the central FSM XML file for our Combination Lock example. After the XML declaration at **line 1**, **line 2** specifies the root element <FSMMain> that defines the name of our central FSM, **CombinationLock**. We require that the name of the central FSM must be the same as the name of the DESpot project, within which our FSM-TDES translation process is taking place. In other words, this should be the DESpot project that contains the plant TDES for the Combination Lock system. Please refer to Section 11.2 for further explanation.

At **lines 3-11**, the <Signals> element specifies a global list of input and output signals of all the individual FSM. Inside <Signals>, each <Signal> element specifies the name, type and order of one global signal (**lines 4-10**). The global output signals have type "O" (**lines 4-7**), whereas global input signals are of type "I" (**lines 8-10**).

XML Input File C.3: Central FSM **CombinationLock**

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <FSMMain Name="CombinationLock">
3   <Signals>
4     <Signal Name ="alarm" type="O" order="0"/>
5     <Signal Name ="new" type="O" order="1"/>
6     <Signal Name ="open" type="O" order="2"/>
7     <Signal Name ="save_code" type="O" order="3"/>
8     <Signal Name= "change" type="I" order="4"/>
9     <Signal Name= "enter" type="I" order="5"/>
```

```
10      <Signal Name= "equal" type="I" order="6"/>
11    </Signals>
12    <FSMS>
13      <FSM Name="ActivateAlarm">
14        <Signals>
15          <Signal Name="open" type="IO"/>
16          <Signal Name="new" type="IO"/>
17          <Signal Name="alarm" type="IO"/>
18          <Signal Name="enter" type="I"/>
19          <Signal Name="change" type="I"/>
20          <Signal Name="equal" type="I"/>
21        </Signals>
22      </FSM>
23      <FSM Name="ChangeCode">
24        <Signals>
25          <Signal Name="new" type="IO"/>
26          <Signal Name="save_code" type="IO"/>
27          <Signal Name="change" type="I"/>
28          <Signal Name="equal" type="I"/>
29        </Signals>
30      </FSM>
31      <FSM Name="OpenLock">
32        <Signals>
33          <Signal Name="open" type="IO"/>
34          <Signal Name="enter" type="I"/>
35          <Signal Name="equal" type="I"/>
36        </Signals>
37      </FSM>
38    </FSMS>
39  </FSMMain>
```

Please note that while implementing the TDES-FSM translation method in DESpot, Hamid (2014) allows an event to belong to the project that may or may not belong to the event set of any TDES supervisor. This event is not under the control of any supervisor and is assumed to be always allowed by the supervisor model. Consequently,

in the translated FSM, its corresponding signal appears in the global list of signals in the central FSM, but does not belong to any individual FSM. If this is an output signal, this means its value will always be set to $True$ (1).

For our FSM-TDES translation approach, this assumption does not cause any issues with respect to input signals (uncontrollable events). However, it is not suitable for our translation approach with respect to output signals (prohibitable events). The reason being, this allowance might cause our translated TDES supervisors to fail the desired $||_{SD}$ properties, especially the property of plant completeness with $||_{SD}$ (Definition 4.4.1) and **S**-singular prohibitable behaviour with $||_{SD}$ (Definition 4.4.2).

In order to make it more likely that our translated supervisors satisfy the desired $||_{SD}$ properties, we require that every output signal included in the global list of signals in the central FSM must belong to at least one individual FSM. Section C.2 discusses how we incorporate this constraint in our DESpot TDES-FSM translation algorithm.

The <FSMS> element contains a list of all the individual Moore FSM that designers wish to translate into the TDES supervisor(s) (**lines 12-38**). Each <FSM> element encloses the name and signal information about one individual FSM (**lines 13-22, 23-30, 31-37**). For each individual FSM, the <Signals> element contains a list of all the FSM signals (**lines 14-21, 24-29, 32-36**). Each <Signal> element specifies one IO or input signal of the FSM (**lines 15-20, 25-28, 33-35**). This information about each individual FSM stated in the central FSM must be the same as specified in the individual FSM XML files.

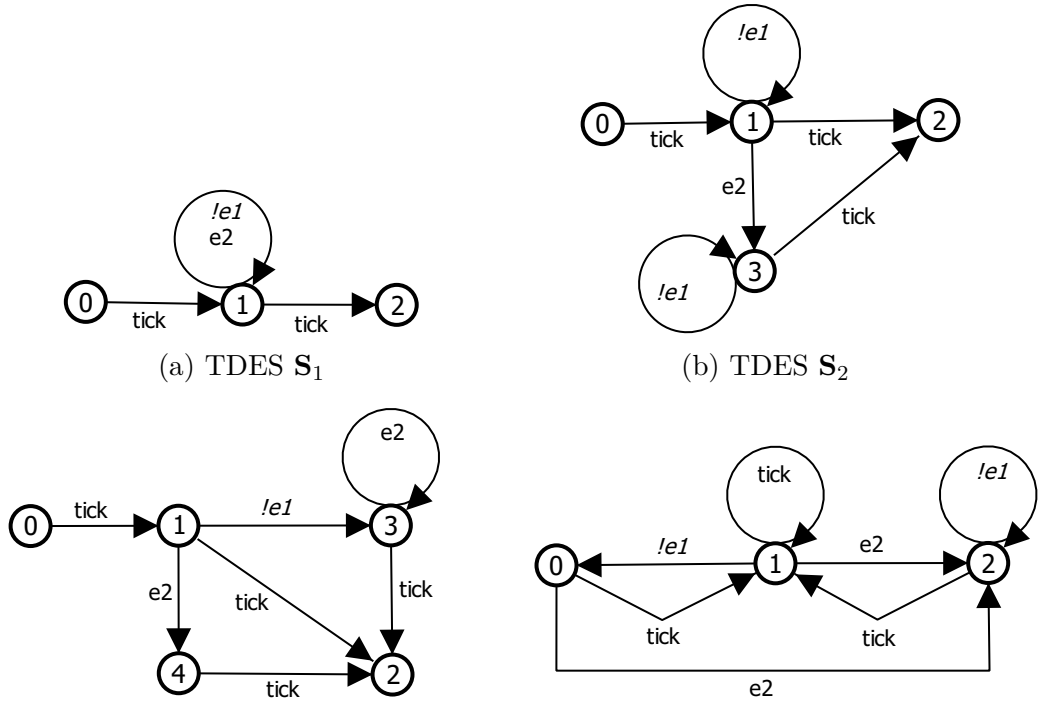
C.2 Generating Individual Moore FSM with DESpot

In order to implement Wang's (2009) TDES-FSM translation approach in DESpot (2023), Hamid (2014) developed several algorithms, including one translation algorithm to generate output files for the individual Moore FSM. As our individual FSM XML file structure (described in Section C.1.1) is different than Hamid's output format, we develop two new algorithms to replace Hamid's DESpot algorithm. Our DESpot algorithms are designed to perform the TDES-FSM translation process and write the output individual FSM of this translation in our XML file format.

Our algorithms should enable the designers to go back and forth between the two translation approaches using DESpot, by allowing them to use the TDES-FSM translation output files directly as an input to our FSM-TDES translation approach, and vice versa. Please note that the implementation of our two TDES-FSM translation algorithms in DESpot has been left as future work due to time constraints.

Precisely, our algorithms add the following refinements to DESpot's existing TDES-FSM translation method:

1. Algorithm C.1 ensures that all prohibitable events of the DESpot project must belong to at least one TDES supervisor. If TDES designers violate this constraint,



(a) TDES $\mathbf{S}_1$ (b) TDES $\mathbf{S}_2$

(c) TDES $\mathbf{S}_3$ (d) TDES $\mathbf{S}_4$

Figure C.1: Some Examples for the Global Don't Care Transition, "<**GDC**>"

DESpot gives an error message and halts the TDES-FSM translation process.

2. Algorithm C.2 enables DESpot to interpret and use the **DEF** transition with its correct meaning, as defined by Wang (2009), while performing the TDES-FSM translation process. In other words, we do not merge *valid* concurrent string selfloops that are explicitly defined in the supervisor with the *invalid* next state conditions that are represented by **DEF** in the translated FSM.

3. Algorithm C.2 also adds support for identifying the **GDC** and **TICK** transitions that could be defined in the TDES supervisors. While doing the TDES-FSM translation, our algorithm specifies these transitions in the individual FSM XML files using our reserved keywords instead of writing the lengthy non-minimal boolean expressions currently generated by Hamid (2014).

In Figure C.1, we present some examples to illustrate what the modelling of a **GDC** next state condition could look like in a TDES supervisor. For our supervisors $\mathbf{S}_1 - \mathbf{S}_4$, let $\Sigma = \{\text{e1}, \text{e2}, \textit{tick}\}$, where $\Sigma_u = \{\text{e1}\}$ and $\Sigma_{hib} = \{\text{e2}\}$. Please note that these are not complete supervisor models, and present only the relevant states just to show the **GDC** concurrent string transition.

In all four supervisors $\mathbf{S}_1 - \mathbf{S}_4$, the **GDC** condition is defined at sampled state "1". In $\mathbf{S}_1 - \mathbf{S}_3$ (Figures C.1a-C.1c), **GDC** transition takes the TDES from sampled state "1" to sampled state "2". In other words, once the TDES is at state "1", it always goes to state "2", regardless of which events have occurred in the concurrent string.

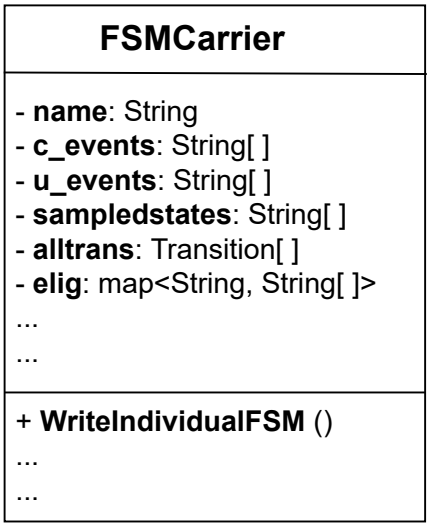


Figure C.2: UML Class: **FSMCarrier**

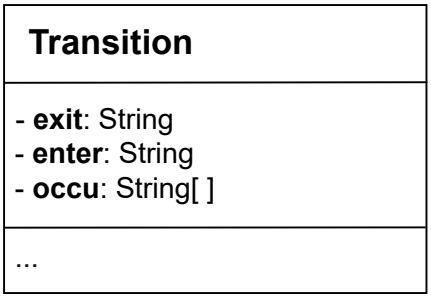


Figure C.3: UML Struct: **Transition**

TDES $\mathbf{S}_4$ (Figure C.1d) shows the **GDC** transition as a concurrent string selfloop, i.e. $\mathbf{S}_4$ stays at sampled state "1", no matter which events the given concurrent string is made up of.

In order to convert a TDES supervisor into a Moore FSM, Hamid (2014) defines an **FSMCarrier** class and uses an array of **FSMCarrier** objects to store the required information about each supervisor for translation. First, we introduce the member variables and functions of the **FSMCarrier** class, shown in Figure C.2, that are relevant to our TDES-FSM translation algorithms.

- *name*: A string variable that contains the name of the supervisor. This name will be used as the name of the corresponding translated FSM.
- *c_events*: An array of strings containing the prohibitable events of the supervisor. They will become IO signals in the translated FSM.
- *u_events*: An array of strings containing the uncontrollable events of the supervisor. They will become input signals in the translated FSM.
- *sampledstates*: An array of strings containing the sampled states of the supervisor. The first element of this array (*sampledstates*[0]) is always the initial state of the supervisor. Sampled states will become the states of the translated FSM.
- *alltrans*: An array to store the concurrent string transitions (NSL) that are defined in the supervisor. These transitions are stored as a **Transition** structure shown in Figure C.3. **Transition** consists of the following three member variables:
  - *exit*: The source sampled state of a concurrent string transition defined in the supervisor.
  - *enter*: The destination sampled state of a concurrent string transition defined in the supervisor.

  - *occu*: An array of strings containing the occurrence image of the concurrent string that starts at the *exit* state and ends at the *enter* state in the supervisor. Please note that *tick* event is not stored as part of the occurrence image, as it is obvious that a concurrent string always ends with a *tick*.
- *elig*: It maps the sampled state of a supervisor to an array of strings which represents the prohibitable events that are enabled at this sampled state. This mapping is used to determine the output information at each state of the translated FSM.
- $WriteIndividualFSM()$: This method corresponds to our Algorithm C.2 that writes the translated FSM specifications to an output XML file as per our required format.

Now we will discuss our Algorithms C.1 and C.2. We assume that the array of **FSMCarrier** objects has been populated before calling our Algorithm C.1.

C.2.1 Algorithm C.1

In Algorithm C.1, we use the following variables:

- $FSMCarrierObjs[]$: This is an array of **FSMCarrier** objects that is populated by Hamid's (2014) TDES-FSM translation algorithms. This array will be accessed like a set.
- Σ_{free}: This is a set of prohibitable events of the DESpot project that do not belong to the event set of any TDES supervisor. Hamid (2014) populates this set while populating the **FSMCarrier** objects.
- $print()$: We use this implementation-independent method to indicate that we wish to print a message to DESpot users in a dialog box. The message that we want to display will be enclosed in quotation marks, i.e. print("*message*"). If we pass a variable to this method, we assume that the content of the variable will be displayed to the users.

Algorithm C.1 starts by checking whether Σ_{free} is empty or not (**line 2**). If Σ_{free} is not empty, this means there exists one or more prohibitable events in the DESpot project that do not belong to the event set of any supervisor. In this case, our algorithm prints an error message along with the list of these prohibitable events, and terminates the TDES-FSM translation process (**lines 3-7**).

However, if Σ_{free} is empty, then for every supervisor that needs to be translated, Algorithm C.1 calls Algorithm C.2 to do the TDES-FSM translation and generate the output XML file for each translated FSM (**lines 9-11**).

C.2.2 Algorithm C.2

Algorithm C.2 converts a TDES supervisor into a Moore FSM, and writes the translated individual FSM in our desired XML file format. Our algorithm is capable of

Algorithm C.1 TranslateTDEStoFSM_Main(Σ_{free})

```
1:  /* After calling Hamid's (2014) TDES-FSM algorithms for populating the
       FSMCarrier objects */
2:  if (Σ_free ≠ ∅) then
3:      print("Error! The following prohibitable event(s) of the DESpot project do
        not belong to the event set of any TDES supervisor:")
4:      for all (e ∈ Σ_free) do
5:          print(e)
6:      end for
7:      return False
8:  end if
9:  for all (FC ∈ FSMCarrierObjs[]) do
10:     FC.WriteIndividualFSM()
11: end for
12: return True
```

identifying the **GDC** and **TICK** transitions defined in the TDES supervisor, and makes use of our three reserved keywords, <**GDC**>, <**TICK**> and <**DEF**>, while writing the FSM's NSL in the output XML file.

We use the following variables in Algorithm C.2:

- FC: An object of the **FSMCarrier** class.
- idx: This integer variable serves two purposes: 1) It is used to specify the order of signals while writing the list of signals in the FSM file. 2) After listing all the signals, idx contains the total number of signals that belong to the current FSM. We then use this signal count to determine the existence of a **GDC** condition in the supervisor, so that we can write this transition with <**GDC**> keyword in the translated FSM.
- $tickTrans$: This boolean variable is set to $True$, if there exists a **TICK** transition at the current sampled state of the supervisor, i.e. a concurrent string transition whose occurrence image does not contain any activity events. In other words, it is a "*tick* only" transition. Otherwise, it retains its default value of $False$.
- $destState$: If $tickTrans$ boolean variable is set to $True$, then we use $destState$ to store the destination sampled state of the **TICK** transition.
- $sameDest$: This boolean variable is set to $False$, if there exists a non-**TICK** concurrent string transition at the current sampled state that goes to a different destination sampled state than the **TICK** transition. Otherwise, it retains its default value of $True$.
- $transCount$: This integer variable keeps count of the total number of concurrent string transitions that are defined at the current sampled state of the supervisor.

- *ineligCount*: This integer variable stores the number of prohibitable events that are disabled at the current sampled state of the supervisor.
- $|t.occu|$: This size function returns the length of *occu* (the occurrence image of a concurrent string). $|t.occu| = 0$ means the occurrence image is empty, i.e. the corresponding concurrent string does not contain any activity events and represents a **TICK** transition. If a concurrent string contains at least one activity event, then $|t.occu| > 0$. Please recall that *tick* event is not stored in *occu* as part of the occurrence image.
- $inputVector$: A string variable for storing the content of one transition input vector that needs to be written to the output XML file.
- $TrimLast$: This string processing function trims off the last character from a given string.
- $write()$: We use this method to write to the output XML file. If we pass a variable to this method, the content of the variable will be written to the file. Usually, while calling this method, we assume that it creates a line in the output XML file with a 'new line character' at the end. However, in some cases, we will be writing a line in chunks, i.e. we will pass and write the first part of the line to the file, generate the rest of the line algorithmically, and then write it to the file by calling the *write* method again. To do so, we will write the first part of the line by passing only the opening brackets "<" to this method and skip the closing brackets ">", to signal that it should not generate a 'new line character' at the end (e.g. at **line 55**). For writing the rest of the line, we call the *write* method again with ">" in order to create a 'new line character' at the end of the line (e.g. at **line 69**).

Algorithm C.2 starts by writing the XML declaration at **line 1** to the output XML file. At **line 2**, we begin the <FSM> root element to write the FSM name, which is same as the name of the supervisor that we are currently translating. At **line 3**, we specify the initial state of the supervisor as the reset state of the translated FSM.

At **lines 4-14**, we write the list of FSM signals, delimited by the <Signals> element, to the XML file. At **lines 6-9**, we loop through all prohibitable events of the supervisor ($FC.c\_events$) and specify them as IO signals of the FSM. Likewise, at **lines 10-13**, we loop through all uncontrollable events of the supervisor ($FC.u\_events$) and specify them as input signals of the FSM. While writing the list of FSM signals, we use the *idx* integer variable to specify the order of these signals (**lines 7, 11**). After writing each IO and input signal, we increment the *idx* variable by 1 (**lines 8, 12**). Once all the IO and input signals are written to the XML file, *idx* now contains the total number of FSM signals.

At **lines 15-19**, we enclose the list of FSM states and their corresponding output information within the <States> element and write it to the XML file. To do so, at **lines 16-18**, we loop through all sampled states of the supervisor ($FC.sampledstates$) and write them as the FSM states. At every sampled state q, we use the list of enabled

Algorithm C.2 WriteIndividualFSM() Part I

```
1: write(<?xml version=1.0 encoding=UTF-8 standalone=yes?>)
2: write(<FSM Name=FC.name>)
3: write(<ResetState Name=FC.sampledstates[0]> </ResetState>)
4: write(<Signals>)
5: idx ← 0
6: for all (e ∈ FC.c_events) do
7:     write(<Signal Name=e order=idx type=IO/>)
8:     idx ← idx + 1
9: end for
10: for all (e ∈ FC.u_events) do
11:     write(<Signal Name=e order=idx type=I/>)
12:     idx ← idx + 1
13: end for
14: write(</Signals>)
15: write(<States>)
16: for all (q ∈ FC.sampledstates) do
17:     write(<State Name=q outputvector=FC.elig[q]/>)
18: end for
19: write(</States>)
20: write(<Transitions>)
21: for all (q ∈ FC.sampledstates) do
22:     write(<StartState Name=q>)
23:     tickTrans ← False
24:     sameDest ← True
25:     destState ← ∅
26:     transCount, ineligCount ← 0
27:     for all (t ∈ FC.alltrans) do
28:         if (t.exit = q ∧ |t.occu| = 0) then
29:             tickTrans ← True
30:             destState ← t.enter
31:         end if
32:     end for
33:     if (tickTrans) then
34:         for all (t ∈ FC.alltrans) do
35:             if (t.exit = q) then
36:                 transCount ← transCount + 1
37:                 if (|t.occu| > 0 ∧ t.enter ≠ destState) then
38:                     sameDest ← False
39:                 end if
40:             end if
41:         end for
```

Algorithm C.2 WriteIndividualFSM() Part II

```
42:     end if
43:     if (tickTrans ∧ sameDest) then
44:         for all (e ∈ FC.c_events) do
45:             if (e ∉ FC.elig[q]) then
46:                 ineligCount ← ineligCount + 1
47:             end if
48:         end for
49:     end if
50:     if (tickTrans ∧ sameDest ∧ transCount = 2^(idx − ineligCount)) then
51:         write(<Transition inputvector=<GDC> endstate=destState/>)
52:     else
53:         for all (t ∈ FC.alltrans) do
54:             if (t.exit = q) then
55:                 write(<Transition inputvector=)
56:                 inputVector ←""
57:                 if (|t.occu| = 0) then
58:                     inputVector ← "<TICK>"
59:                 else
60:                     for all (e ∈ FC.c_events ∪ FC.u_events) do
61:                         if (e ∈ t.occu) then
62:                             inputVector ← inputVector + e + "."
63:                         else
64:                             inputVector ← inputVector + "!" + e + "."
65:                         end if
66:                     end for
67:                     TrimLast(inputVector)
68:                 end if
69:                 write(inputVector endstate=t.enter/>)
70:             end if
71:         end for
72:     end if
73:     write(<Transition inputvector=<DEF> endstate=q/>)
74:     write(</State>)
75: end for
76: write(</Transitions>)
77: write(</FSM>)
```

prohibitable events ($FC.elig[q]$) to generate the output information, and write it as an output vector at the corresponding FSM state (**line 17**).

At **lines 20-76**, we examine the concurrent string transitions defined in the supervisor and use them to write the corresponding NSL for the translated FSM, delimited by the <Transitions> element. For this purpose, **lines 21-75** loop through all sampled states of the supervisor and generate NSL at each FSM state.

At **lines 21-22**, we take one sampled state of the supervisor, say q, and specify it as the *start state* for the subsequent NSL that we are about to write in the FSM XML file. Please recall that we also need to identify the **GDC** and **TICK** transitions in the supervisor in order to specify them with our reserved keywords in the XML file, instead of the regular boolean expressions.

At **lines 27-49**, we specify the logic for determining whether the concurrent string transitions defined at state q of the supervisor collectively represent a **GDC** transition or not. In order to identify a **GDC** transition, we define three conditions that must be satisfied at q. Below, we describe these conditions and discuss how Algorithm C.2 checks for them.

1) The first and foremost condition for the existence of a **GDC** condition is that a **TICK** transition must be defined at q. This is because, by definition, a **GDC** must cover all possible input combinations. The absence of a **TICK** transition means that one input combination is missing at q, hence it cannot be a **GDC**.

We evaluate this condition at **lines 27-32** of Algorithm C.2. Of all the concurrent string transitions defined in the supervisor ($FC.alltrans$ at **line 27**), we consider only those transitions that start at q ($t.exit = q$), and then check to see if they include a "*tick* only" transition, i.e. $|t.occu| = 0$ (**line 28**). If so, we set the $tickTrans$ boolean variable to $True$, and store the destination sampled state of this **TICK** transition in the *destState* variable (**lines 29-30**).

However, if **TICK** transition is not defined at q, then $tickTrans$ will retain its default value of $False$, and we will not evaluate the next two **GDC** conditions discussed below.

2) The second condition for the existence of a **GDC** transition is that the concurrent strings defined at q must go to the same destination sampled state. This is because the **GDC** condition means that no matter which input combination occurs, we always go to the same destination state. If any concurrent string defined at q goes to a different sampled state than the rest of the concurrent string transitions, this condition fails. Please note that the destination sampled state could be the same as the current state (in case of selfloops) or could be a different one (an explicit state change).

If the first **GDC** condition is satisfied, i.e. $tickTrans = True$ (**line 33**), then Algorithm C.2 evaluates the second **GDC** condition at **lines 34-41**. Of all the concurrent string transitions defined at q (**lines 34-35**), we determine if the destination sampled state of a non-**TICK** transition (a concurrent string that has at least one activity event, hence $|t.occu| > 0$) is not same as the **TICK** transition (**line 37**). If any such concurrent string transition exists, we set the *sameDest* variable to $False$, indicating that the second condition has failed and **GDC** is

not defined at q (**line 38**).

While looping through all the transitions (**lines 34-41**), we also count the number of concurrent string transitions defined at q and store this count in the *transCount* variable (**line 36**). We will use this count in evaluating our next **GDC** condition.

3) Our third and last condition for the existence of **GDC** in a supervisor is that the number of all concurrent strings defined at q with a unique occurrence image must be equal to 2^n, where n is the total number of all the uncontrollable events of the supervisor and those prohibitable events that are enabled at q. It is notable that we are excluding the disabled prohibitable events from this count. Since these prohibitable events are disabled, it is obvious that they will not occur as part of any valid concurrent string defined at q.

Hypothetically, we could have taken n to be the total number of uncontrollable and prohibitable events of the supervisor. However, this implies that all prohibitable events must be enabled and their transitions must be defined at q, in order to satisfy the existence of a **GDC** transition. This looks like a stringent requirement that seems to limit the usage of our <**GDC**> keyword, as practically it seems quite uncommon that all prohibitable events will be enabled at many different states of the supervisor.

Therefore, in order to enhance the applicability of our **GDC** transition, we have restricted our requirement to uncontrollable and enabled prohibitable events only. This should also prove to be beneficial in our FSM-TDES translation, as it will allow the designers to use our <**GDC**> keyword in the input FSM, even if the outputs of some/all IO signals are set to $False$ (0) at state q of the FSM.

If the previous two conditions for the existence of a **GDC** transition are satisfied (**line 43**), then **lines 44-48** of Algorithm C.2 count the number of prohibitable events that are disabled at q. We store this count in the *eligCount* variable.

After evaluating the three required conditions for the existence of a **GDC** transition individually, we are now ready to actually write to the XML file all the NSL corresponding to state q of the FSM. For this purpose, we first make a final check for the existence of a **GDC** transition at q by ANDing the individual result of the three required **GDC** conditions (discussed above) at **line 50**. By our algorithm logic, we know that *tickTrans* and *sameDest* variables will be $True$, if **GDC** conditions 1 and 2 are satisfied respectively.

In order to evaluate the third **GDC** condition, we exclude the number of prohibitable events that are disabled at q (*eligCount*) from the total number of activity events of the supervisor (*idx*). We then determine if all possible input combinations for the uncontrollable and enabled prohibitable events are defined at q. If so, the last condition specified at **line 50** evaluates to $True$. If all the three **GDC** conditions are satisfied, we write the **GDC** transition to the XML file by assigning <**GDC**> keyword to the transition input vector and specifying its corresponding destination state using our *destState* variable (**line 51**).

However, if any of the three **GDC** conditions evaluates to $False$ and **GDC** transition does not exist at q, then we loop through all the concurrent string transitions defined at q and generate their corresponding NSL one by one (**lines 53-71**). At **line 55**, we begin the <Transition> element and write it to the XML file. After that, we algorithmically generate the input vector corresponding to the concurrent string transition that we are currently processing and store it in the $inputVector$ string variable.

At **line 57**, we check to determine if the currently processed concurrent string transition is a "*tick* only" transition. If so, we assign the <**TICK**> keyword to the $inputVector$ variable (**line 58**). Otherwise, we loop through all prohibitable and uncontrollable events of the supervisor to generate a boolean expression that represents the current concurrent string transition and store it in the $inputVector$ (**lines 60-66**).

In order to generate a boolean expression, we use the occurrence image of the concurrent string. If an event is present in the occurrence image, we write the event in the uncomplemented form (without '!') in the boolean expression (**lines 61-62**). Otherwise, we write the event in the complemented form (with '!') (**lines 63-64**). Please note that we are generating boolean expressions in the non-minimal SOP form.

After generating the complete boolean expression, we trim off the extra **AND** operator ('$\cdot$') from the end (**line 67**). At **line 69**, we write the boolean expression (stored in the $inputVector$ variable) and the destination state of this next state condition to the XML file and close the <Transition> element. As stated earlier, this process of generating the boolean expression and writing the NSL to the XML file will be repeated for every concurrent string defined in the supervisor at state q, as we are writing one <Transition> element for each defined concurrent string.

After writing all *valid* NSL at state q of the FSM to the XML file, **line 73** writes the **DEF** transition as a selfloop. Please recall that **DEF** transition is included to make the translated FSM's next state function a total function, and it covers all the *invalid* transitions (NSL) that are not defined in the supervisor at q. At **line 74**, we close the <State> element corresponding to the start state q in the XML file, and then proceed to the remaining sampled states of the supervisor to process their concurrent string transitions and write their NSL to the XML file.

Once this process is complete, we close the <Transitions> element at **line 76**. Finally, we end the XML file at **line 77** by writing the closing <FSM> element that indicates the end of the translated FSM specification.

Appendix D

Supporting Algorithms for Moore FSM to TDES Translation

This appendix contains three algorithms that support the main FSM-TDES translation algorithms presented in Chapter 12. These supporting algorithms perform some consistency and design checks (listed in Section 11.2) and do the required setup for the FSM-TDES translation process that begins with Algorithm 12.1. Please see Section 12.2 for the definition of several shared variables that the algorithms in this appendix make use of.

D.1 Verify Central FSM

Algorithm D.1 uses the information of the central FSM and the current DESpot project to verify some consistency and design prerequisites of our FSM-TDES translation method. If any of these requirements is not satisfied, Algorithm D.1 displays an appropriate error message to the users and immediately returns control to Algorithm 12.1 by returning $False$.

In addition to the variables introduced in Section 12.2, Algorithm D.1 defines the following two local variables:

- Σ_{tmpIO}: This temporary set is created to store the unique IO signals that are listed in all the individual Moore FSM specified in the central FSM.
- Σ_{tmpIn}: This temporary set is created to store the unique input signals that are listed in all the individual Moore FSM specified in the central FSM.

We begin Algorithm D.1 by verifying the design requirement **DR-1** at **lines 1-8**. Specifically, we check that all individual Moore FSM listed in the central FSM must have a unique name. We do this by comparing the names of two Moore FSM at a time inside a nested **for** loop.

At **lines 9-14**, we check **DR-2** which requires that all IO and input signals of the

Algorithm D.1 VerifyCentralFSM($mainFSMS, \Sigma_{gout}, \Sigma_{gin}, \Sigma_{hib}, \Sigma_u$) Part I

```
1: for (i ← 0 to |mainFSMS| − 1) do
2:     for (j ← i + 1 to |mainFSMS| − 1) do
3:         if (mainFSMS_i.fsmName = mainFSMS_j.fsmName) then
4:             print("Error! All FSM listed in the central FSM must have a unique
               name.")
5:             return False
6:         end if
7:     end for
8: end for
9: for all (σ ∈ Σ_gout) do
10:    if (σ ∈ Σ_gin) then
11:        print("Error! The names of the global input and output signals listed in
           the central FSM must be unique.")
12:        return False
13:    end if
14: end for
15: Σ_tmpIO, Σ_tmpIn ← ∅
16: for all (fsm ∈ mainFSMS) do
17:    for all (σ ∈ fsm.Σ_locIO) do
18:        if (σ ∉ Σ_tmpIO) then
19:            Σ_tmpIO ← Σ_tmpIO ∪ {σ}
20:        end if
21:    end for
22:    for all (σ ∈ fsm.Σ_locIn) do
23:        if (σ ∉ Σ_tmpIn) then
24:            Σ_tmpIn ← Σ_tmpIn ∪ {σ}
25:        end if
26:    end for
27: end for
28: if (|Σ_gout| ≠ |Σ_hib| ∨ |Σ_gout| ≠ |Σ_tmpIO|) then
29:    print("Error! The number of global output signals must be equal to the number
       of prohibitable events in the project. It must also be equal to the number of
       unique local IO signals of all the FSM listed in the central FSM.")
30:    return False
31: end if
32: for all (σ ∈ Σ_gout) do
33:    if (σ ∉ Σ_hib ∨ σ ∉ Σ_tmpIO) then
34:        print("Error! Every global output signal must belong to the project and
           at least one FSM listed in the central FSM, and vice versa.")
35:        return False
36:    end if
```

Algorithm D.1 VerifyCentralFSM($mainFSMS, \Sigma_{gout}, \Sigma_{gin}, \Sigma_{hib}, \Sigma_u$) Part II

```
37: end for
38: if (|Σ_gin| ≠ |Σ_u|) then
39:     print("Error! The number of global input signals must be equal to the number
        of uncontrollable events in the project.")
40:     return False
41: end if
42: for all (σ ∈ Σ_gin) do
43:     if (σ ∉ Σ_u) then
44:         print("Error! Every global input signal must belong to the project, and
            vice versa.")
45:         return False
46:     end if
47: end for
48: for all (σ ∈ Σ_tmpIn) do
49:     if (σ ∉ Σ_gin) then
50:         print("Error! In the central FSM, every input signal of an individual FSM
            must belong to the list of global input signals.")
51:         return False
52:     end if
53: end for
54: return True
```

individual Moore FSM must have a unique name. Please note that all IO and input signals of the individual FSM must be included in the respective global list of output and input signals specified in the central FSM (**CR-4** and **CR-5**). Therefore, instead of checking **DR-2** for each individual FSM, we use our global list of output and input signals to perform this check. Specifically, we verify that the two sets of Σ_{gout} and Σ_{gin} are disjoint. Please note that we cannot have duplicate elements within one set because of our assumption listed in Section 12.2. Therefore, we do not need to do a uniqueness check on Σ_{gout} or Σ_{gin} individually.

Line 15 declares the two local variables, Σ_{tmpIO} and Σ_{tmpIn}. At **lines 16-27**, we loop through all individual Moore FSM specified in the central FSM to populate these two sets. At **lines 17-21**, we traverse the set of IO signals of each individual FSM to create Σ_{tmpIO}. Likewise, at **lines 22-26**, we loop through the set of input signals of each individual FSM to populate Σ_{tmpIn}. Now we will use Σ_{tmpIO} and Σ_{tmpIn} to perform our next consistency checks.

We verify the consistency requirements **CR-2** and **CR-4** at **lines 28-37**. First, at **lines 28-31**, we compare the number of global output signals listed in the central FSM (Σ_{gout}) with the number of prohibitable events in the DESpot project (Σ_{hib}), and the number of unique IO signals of all the individual FSM specified in the central

FSM (Σ_{tmpIO}). If the number of signals/events in these three sets is not equal, this means requirements **CR-2** and/or **CR-4** cannot be satisfied, and we return $False$ to the caller algorithm after displaying the error message.

However, if these three sets have the same number of signals/events, then we determine whether or not the names of all the signals/events match at **lines 32-37**. We evaluate this by making sure that every global output signal listed in the central FSM must belong to the DESpot project as a prohibitable event. Also, a global output signal must be present as an IO signal in at least one individual FSM specified in the central FSM, i.e. it must be in Σ_{tmpIO}.

At **lines 38-47**, we perform the consistency check **CR-3**. At **lines 38-41**, we compare the number of global input signals listed in the central FSM (Σ_{gin}) with the number of uncontrollable events of the DESpot project (Σ_u). If the size of the two sets is the same, then at **lines 42-47**, we examine if the names of the global input signals and the project's uncontrollable events match. If not, the **CR-3** check fails and we return $False$ to the caller algorithm after printing an error message.

At **lines 48-53**, we evaluate **CR-5** by checking whether or not an input signal of an individual Moore FSM (Σ_{tmpIn}) belongs to the set of global input signals of the central FSM (Σ_{gin}). If this is not the case, then we terminate the FSM-TDES translation process by generating an error.

If all the above-mentioned consistency and design requirements are satisfied, then we return $True$ at **line 54**. This indicates to Algorithm 12.1 that all consistency and design checks of Algorithm D.1 have been successful.

D.2 Verify Individual Moore FSM

In Algorithm D.2, we verify the consistency requirements **CR-6−8** by examining the information of all the individual Moore FSM specified in the central and individual FSM XML files. If we find any inconsistency, we generate an appropriate error message and terminate the FSM-TDES translation process by returning control to Algorithm 12.1.

Besides the variables defined in Section 12.2, Algorithm D.2 makes use of the following two local variables:

- $fsmFound$: This flag is set to $True$ if the individual Moore FSM, that we wish to check for consistency, is found in the list of individual FSM specified in the central FSM.
- $mfsm$: This variable temporarily stores the instance of the individual Moore FSM specified in the central FSM that we are currently processing.

Algorithm D.2 begins by verifying part of the consistency requirement **CR-6** at **lines 1-4**. Specifically, we compare the number of individual Moore FSM that are listed in the central FSM ($mainFSMS$) with the number of Moore FSM that are specified individually ($FSMS$). If this number is the same, then at **lines 5-38**, we

Algorithm D.2 VerifyIndividualFSM($mainFSMS, FSMS$) Part I

```
1: if (|FSMS| ≠ |mainFSMS|) then
2:     print("Error! The number of FSM listed in the central FSM must be equal
       to the number of FSM specified individually.")
3:     return False
4: end if
5: for all (fsm ∈ FSMS) do
6:     fsmFound ← False
7:     mfsm ← ∅
8:     for all (f ∈ mainFSMS) do
9:         if (fsm.name = f.fsmName) then
10:            mfsm ← f
11:            fsmFound ← True
12:        end if
13:    end for
14:    if (¬fsmFound) then
15:        print("Error! Every FSM specified individually must be listed in the
           central FSM with the same name.")
16:        return False
17:    end if
18:    if (|fsm.Σ_IO| ≠ |mfsm.Σ_locIO|) then
19:        print("Error! For every FSM, the number of IO signals listed in the
           central and individual FSM must be equal.")
20:        return False
21:    end if
22:    for all (σ ∈ fsm.Σ_IO) do
23:        if (σ ∉ mfsm.Σ_locIO) then
24:            print("Error! For every FSM, the list of IO signals specified in the
               central and individual FSM must be same.")
25:            return False
26:        end if
27:    end for
28:    if (|fsm.Σ_In| ≠ |mfsm.Σ_locIn|) then
29:        print("Error! For every FSM, the number of input signals listed in the
           central and individual FSM must be equal.")
30:        return False
31:    end if
32:    for all (σ ∈ fsm.Σ_In) do
33:        if (σ ∉ mfsm.Σ_locIn) then
34:            print("Error! For every FSM, the list of input signals specified in the
               central and individual FSM must be same.")
35:            return False
```

Algorithm D.2 VerifyIndividualFSM($mainFSMS, FSMS$) Part II

```
36:         end if
37:     end for
38: end for
39: return True
```

loop through all Moore FSM that are specified individually to perform the required consistency checks.

At **line 5**, we take one individually specified Moore FSM (fsm), and look it up in the list of individual FSM specified in the central FSM at **lines 8-13**. We use the FSM name to perform the search (**CR-6**). If we find a Moore FSM in the central FSM with the matching name, then we store the instance of this Moore FSM specified in the central FSM in the $mfsm$ variable, and set the $fsmFound$ variable to $True$. On the other hand, if the current Moore FSM specified individually is not present in the list of individual FSM specified in the central FSM, then we display an error message and return to the caller algorithm at **lines 14-17**.

At **lines 18-27**, we verify **CR-7** by making sure that the list of IO signals of the current Moore FSM is the same in the central FSM ($mfsm.\Sigma_{locIO}$) and individual FSM specification ($fsm.\Sigma_{IO}$). In order to do this, first we compare the number of IO signals of the current Moore FSM at **lines 18-21**. If the number of IO signals listed in the central and individual FSM specification is the same, then we evaluate whether or not all IO signals have matching names at **lines 22-27**.

At **lines 28-37**, we repeat the above-mentioned process with respect to the input signals of the Moore FSM that we are currently processing. In other words, we check **CR-8** to verify that the number and names of input signals of the current Moore FSM are the same in the central FSM ($mfsm.\Sigma_{locIn}$) and individual FSM specification ($fsm.\Sigma_{In}$).

If all Moore FSM satisfy **CR-6−8**, then Algorithm D.2 returns $True$ at **line 39** to indicate to Algorithm 12.1 that all consistency checks have been successfully completed.

D.3 Generate Enablement Information

Algorithm D.3 uses the output information of an individual Moore FSM ($fsm.QZ$) to populate Q_{Elig} that represents enablement information for the corresponding TDES supervisor. This algorithm also verifies design requirements **DR-3−4** for the given Moore FSM. If any of these requirements is not satisfied, we display an error message and return $False$ to Algorithm 12.1.

Algorithm D.3 defines the following two local variables:

- Σ_{Elig}: This set is used to store IO signals that are listed in the output vector of the

Algorithm D.3 GenerateEnablementInfo(fsm, Q_{Elig})

```
1: if (fsm.QZ = ∅) then
2:     print("Error! In the FSM, the list of states along with the corresponding
       output information of the IO signals must be specified.")
3:     return False
4: end if
5: for all (z ∈ fsm.QZ) do
6:     Σ_Elig ← ∅
7:     IOSignal ← ""
8:     if (z.outputVector ≠ "") then
9:         for (i ← 0 to |z.outputVector| − 1) do
10:            if (z.outputVector[i] = ",") then
11:                Σ_Elig ← Σ_Elig ∪ {IOSignal}
12:                IOSignal ← ""
13:            else
14:                IOSignal ← IOSignal + z.outputVector[i]
15:            end if
16:        end for
17:        Σ_Elig ← Σ_Elig ∪ {IOSignal}
18:    end if
19:    for all (σ ∈ Σ_Elig) do
20:        if (σ ∉ fsm.Σ_IO) then
21:            print("Error! The signals specified in the output vectors must be
               listed as IO signals in the FSM's list of signals.")
22:            return False
23:        end if
24:    end for
25:    Q_Elig ← Q_Elig ∪ {(z.q, Σ_Elig)}
26: end for
27: return True
```

Moore FSM which is currently being processed. In terms of a TDES supervisor, this set contains prohibitable events that are enabled at the given state of the supervisor that we are currently generating.

- *IOSignal*: This string variable temporarily holds the characters, as we read the name of an IO signal (one character at a time) from the output vector that we are currently processing.

We begin Algorithm D.3 by verifying design requirement **DR-3** at **lines 1-4**. Specifically, we check QZ to make sure that there exists at least one state along with its output information in the Moore FSM that we are currently translating.

At **lines 5-26**, we loop through the set QZ, process its elements (tuples) one by

one to generate the corresponding enablement information, and use this information to populate Q_{Elig}. We take one tuple (z) from QZ at **line 5**. Please recall from Section 12.2 that the first element of this tuple is an FSM state ($z.q$), and the second element is an output vector string ($z.outputVector$).

At **line 8**, we examine the output vector of z's tuple to make sure that it is not empty. Please recall from our FSM-TDES translation method that, at any given state of the FSM, the output vector will be empty if outputs for all the IO signals of the FSM are set to $False$. In this case, we do not have any information in the output vector to process, and Σ_{Elig} will be empty. Therefore, the algorithm jumps to **line 25** and adds an empty set to Q_{Elig} corresponding to the current FSM state. This signifies that all prohibitable events are disabled at the corresponding sampled state in the supervisor.

However, if the output vector string is not empty, then we start reading the characters of the output vector one by one at **lines 9-16**. Every time we read a character from the output vector that is not a comma (","), we append this character to the end of the $IOSignal$ string (**line 14**). If we come across a comma (**line 10)**, this means we have read one complete signal name from the output vector. Therefore, we add this signal name to Σ_{Elig}, and empty out $IOSignal$ to prepare it to store the name of the next IO signal (**lines 11-12**). Once we reach the end of the output vector, we add the last read signal name to Σ_{Elig} at **line 17**.

We verify design requirement **DR-4** at **lines 19-24**. In simple words, we check to make sure that every IO signal included in the output vector (Σ_{Elig}) must belong to the list of IO signals of the current FSM ($fsm.\Sigma_{IO}$). If **DR-4** is satisfied, we add the current FSM state (that corresponds to a sampled state in the translated supervisor) and its corresponding enablement information to Q_{Elig} at **line 25**.

Appendix E

Supplementary Material for Combination Lock Example

This appendix contains supplementary material for the Moore FSM-TDES translation example of a 4-bit Combination Lock system discussed in Chapter 13. Specifically, we present XML files for the input Moore system and figures for the translated non-minimal TDES supervisors for the Combination Lock example. We also demonstrate the correctness of our FSM-TDES translation approach for the Combination Lock system by performing the FSM-TDES-FSM translation cycle for this example.

E.1 XML Files for Input FSM

This section contains XML files for the 4-bit Combination Lock example expressed as a Moore system.

E.1.1 Individual Moore FSM

In this section, we present three individual Moore FSM XML files for the Combination Lock system. In particular, XML Input File E.1 represents the Moore FSM **OpenLock** (Figure 11.1 on page 181), XML Input File E.2 represents the FSM **ChangeCode** (Figure 13.3 on page 255), and XML Input File E.3 represents the FSM **ActivateAlarm** (Figure 13.4 on page 256). Please see Section C.1.1 for details of our XML file format.

E.1.2 Central FSM

XML Input File E.4 expresses the central FSM for the 4-bit Combination Lock system in our XML file format. Please see the details of our XML file structure in Section C.1.2.

XML Input File E.1: Moore FSM **OpenLock**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<FSM Name="OpenLock">
  <ResetState Name="1"> </ResetState>
  <Signals>
    <Signal Name="open" order="0" type="IO"/>
    <Signal Name="enter" order="1" type="I"/>
    <Signal Name="equal" order="2" type="I"/>
  </Signals>
  <States>
    <State Name="1" outputvector=""/>
    <State Name="2" outputvector="open"/>
  </States>
  <Transitions>
    <StartState Name="1">
      <Transition inputvector="enter.equal" endstate="2"/>
      <Transition inputvector="!enter" endstate="1"/>
      <Transition inputvector="!equal" endstate="1"/>
    </State>
    <StartState Name="2">
      <Transition inputvector="enter" endstate="1"/>
      <Transition inputvector="!enter" endstate="2"/>
    </State>
  </Transitions>
</FSM>
```

XML Input File E.2: Moore FSM **ChangeCode**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<FSM Name="ChangeCode">
  <ResetState Name="1"> </ResetState>
  <Signals>
    <Signal Name="new" order="0" type="IO"/>
    <Signal Name="save_code" order="1" type="IO"/>
    <Signal Name="change" order="2" type="I"/>
```

```
8     <Signal Name="equal" order="3" type="I"/>
9   </Signals>
10  <States>
11    <State Name="1" outputvector=""/>
12    <State Name="2" outputvector="new"/>
13    <State Name="3" outputvector="new,save_code"/>
14  </States>
15  <Transitions>
16    <StartState Name="1">
17      <Transition inputvector="change.equal" endstate="2"/
    >
18      <Transition inputvector="!change" endstate="1"/>
19      <Transition inputvector="!equal" endstate="1"/>
20    </State>
21    <StartState Name="2">
22      <Transition inputvector="change.new" endstate="3"/>
23      <Transition inputvector="!change" endstate="2"/>
24      <Transition inputvector="!new" endstate="2"/>
25    </State>
26    <StartState Name="3">
27      <Transition inputvector="save_code" endstate="1"/>
28      <Transition inputvector="!save_code" endstate="3"/>
29    </State>
30  </Transitions>
31 </FSM>
```

XML Input File E.3: Moore FSM **ActivateAlarm**

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <FSM Name="ActivateAlarm">
3   <ResetState Name="1"> </ResetState>
4   <Signals>
5     <Signal Name="open" order="0" type="IO"/>
6     <Signal Name="new" order="1" type="IO"/>
7     <Signal Name="alarm" order="2" type="IO"/>
```

```
8     <Signal Name="enter" order="3" type="I"/>
9     <Signal Name="change" order="4" type="I"/>
10    <Signal Name="equal" order="5" type="I"/>
11  </Signals>
12  <States>
13    <State Name="1" outputvector="open,new"/>
14    <State Name="2" outputvector="alarm"/>
15  </States>
16  <Transitions>
17    <StartState Name="1">
18      <Transition inputvector="enter.!equal.!open"
    endstate="2"/>
19      <Transition inputvector="change.!equal.!new"
    endstate="2"/>
20      <Transition inputvector="!enter.!change" endstate="1
    "/>
21      <Transition inputvector="!enter.new" endstate="1"/>
22      <Transition inputvector="equal" endstate="1"/>
23      <Transition inputvector="open.!change" endstate="1"/
    >
24      <Transition inputvector="open.new" endstate="1"/>
25    </State>
26    <StartState Name="2">
27      <Transition inputvector="<GDC>" endstate="2"/>
28    </State>
29  </Transitions>
30 </FSM>
```

XML Input File E.4: Central FSM **CombinationLock**

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <FSMMain Name="CombinationLock">
3   <Signals>
4     <Signal Name ="alarm" type="O" order="0"/>
5     <Signal Name ="new" type="O" order="1"/>
```

```
6     <Signal Name ="open" type="O" order="2"/>
7     <Signal Name ="save_code" type="O" order="3"/>
8     <Signal Name= "change" type="I" order="4"/>
9     <Signal Name= "enter" type="I" order="5"/>
10    <Signal Name= "equal" type="I" order="6"/>
11  </Signals>
12  <FSMS>
13    <FSM Name="ActivateAlarm">
14      <Signals>
15        <Signal Name="open" type="IO"/>
16        <Signal Name="new" type="IO"/>
17        <Signal Name="alarm" type="IO"/>
18        <Signal Name="enter" type="I"/>
19        <Signal Name="change" type="I"/>
20        <Signal Name="equal" type="I"/>
21      </Signals>
22    </FSM>
23    <FSM Name="ChangeCode">
24      <Signals>
25        <Signal Name="new" type="IO"/>
26        <Signal Name="save_code" type="IO"/>
27        <Signal Name="change" type="I"/>
28        <Signal Name="equal" type="I"/>
29      </Signals>
30    </FSM>
31    <FSM Name="OpenLock">
32      <Signals>
33        <Signal Name="open" type="IO"/>
34        <Signal Name="enter" type="I"/>
35        <Signal Name="equal" type="I"/>
36      </Signals>
37    </FSM>
38  </FSMS>
39 </FSMMain>
```

E.2 Deriving a Simplified Boolean Expression

Below, we present the step by step simplification process for the complemented boolean expression specified in the **ActivateAlarm** Moore FSM (Figure 13.4 discussed in Section 13.2) at state 1. Please note that there exists a variety of online tools and websites that automatically do these simplifications for us for free.

$$!((enter \cdot !equal \cdot !open) + (change \cdot !equal \cdot !new))$$

$$= (!enter + !(!equal) + !(!open)) \cdot (!change + !(!equal) + !(!new)) \quad \text{(DeMorgan's Theorem)}$$

$$= (!enter + equal + open) \cdot (!change + equal + new) \quad \text{(Involution Law)}$$

$$= !enter \cdot (!change + equal + new) + equal \cdot (!change + equal + new) + open \cdot (!change + equal + new) \quad \text{(Distributive Property)}$$

$$= !enter \cdot !change + !enter \cdot equal + !enter \cdot new + equal \cdot !change + equal \cdot equal + equal \cdot new + open \cdot !change + open \cdot equal + open \cdot new \quad \text{(Distributive Property)}$$

$$= !enter \cdot !change + !enter \cdot equal + !enter \cdot new + equal \cdot !change + equal + equal \cdot new + open \cdot !change + open \cdot equal + open \cdot new \quad \text{(Idempotent Law)}$$

$$= !enter \cdot !change + !enter \cdot new + equal \cdot !change + equal + equal \cdot new + open \cdot !change + open \cdot equal + open \cdot new \quad \text{(Absorption Law)}$$

$$= !enter \cdot !change + !enter \cdot new + equal + equal \cdot new + open \cdot !change + open \cdot equal + open \cdot new \quad \text{(Absorption Law)}$$

$$= !enter \cdot !change + !enter \cdot new + equal + open \cdot !change + open \cdot equal + open \cdot new \quad \text{(Absorption Law)}$$

$$= !enter \cdot !change + !enter \cdot new + equal + open \cdot !change + open \cdot new \quad \text{(Absorption Law)}$$

E.3 Translated Non-Minimal TDES Supervisors

This section contains the translated non-minimal TDES supervisors for the 4-bit Combination Lock example discussed in Chapter 13. These supervisors are the direct output of our FSM-TDES translation algorithms presented in Chapter 12, before performing the state space minimization process.

For the **ChangeCode** FSM (Figure 13.3 on page 255), the translated non-minimal TDES supervisor is shown in Figure E.1. For the **ActivateAlarm** FSM (Figure 13.4 on page 256), the translated non-minimal supervisor is shown in Figure E.2. Please note that for the **OpenLock** FSM (Figure 11.1 on page 181), the translated non-minimal supervisor is shown in Figure 11.2 (on page 192 in Chapter 11).

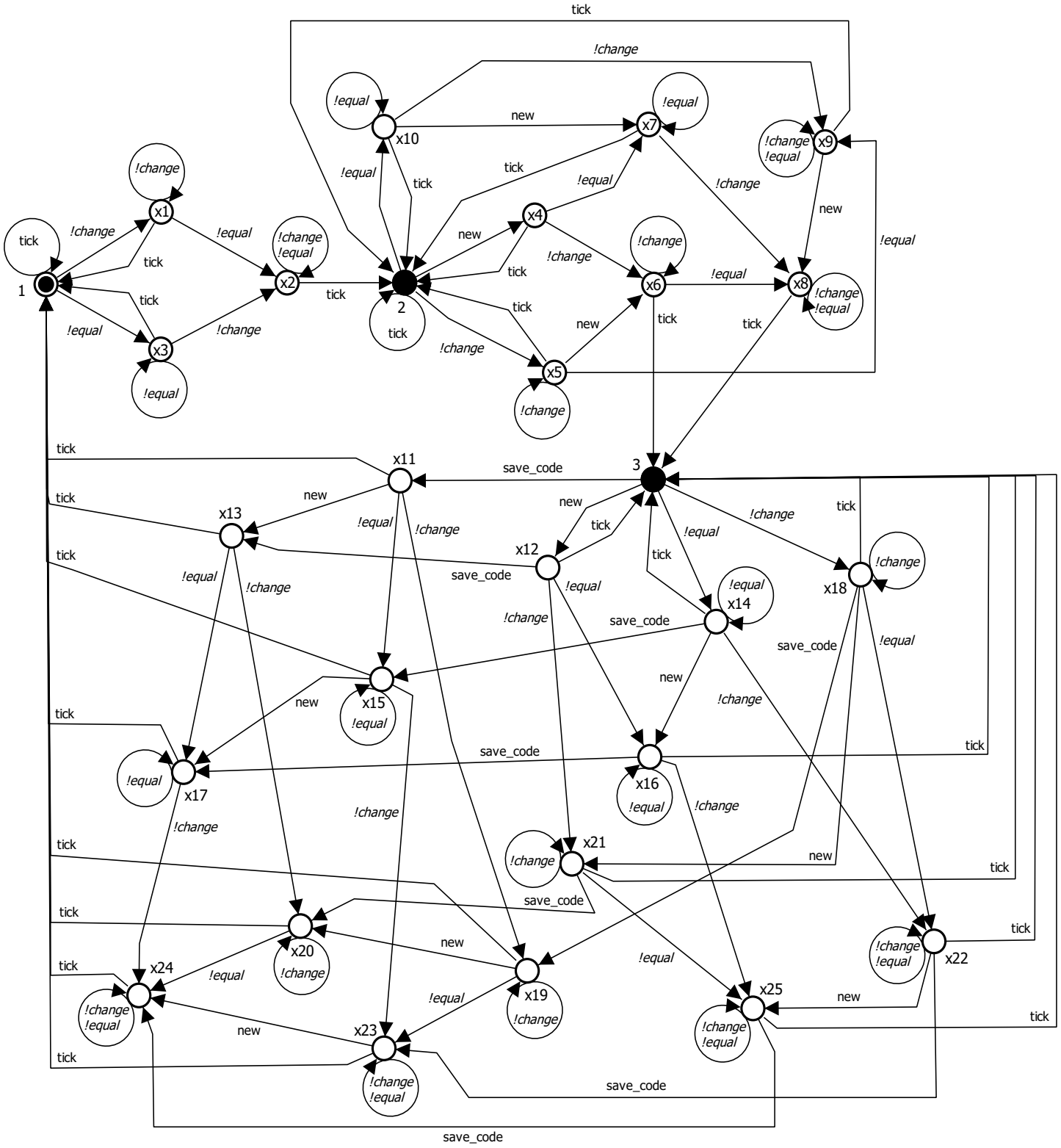


Figure E.1: Translated TDES Supervisor **ChangeCode**

E.4 Correctness of FSM-TDES Translation Approach

As mentioned in Section 11.3.6, our primary goal while devising the FSM-TDES translation approach is to formulate an automatic translation method that should be capable of generating a "correct" TDES supervisor from the Moore FSM without violating the given control specifications. By looking at the supervisors (discussed in Section 13.3) that our FSM-TDES translation approach has generated for the 4-bit Combination Lock, it is evident that the translated TDES supervisors fulfill the desired system specifications and abide by all the given control laws.

However, in order to rigorously verify the correctness of our FSM-TDES translation approach for the Combination Lock system, we decided to complete the cycle of FSM-TDES-FSM translation for this system. Specifically, we started our Combination Lock example by manually designing its controllers expressed as Moore FSM (Section 13.2). Then, we converted these Moore FSM into TDES supervisors by applying our FSM-TDES translation approach (Section 13.3). Now, in this section, we will convert our translated TDES supervisors back to Moore FSM using the TDES-FSM translation method defined by Wang (2009) and our modified TDES-FSM translation algorithms (presented in Section C.2).

Once we have the translated Moore FSM for the Combination Lock system, we will compare them with our manually designed Moore FSM that we started with. We intend to do this comparison in order to determine the exact similarities and/or differences between the two Moore FSM. In simple words, we will try to investigate if the translated Moore FSM are isomorphic to our manually designed Moore FSM up to relabelling of states. If not, we will explore how the two versions of the Moore FSM differ, and what this difference signifies with respect to the correctness of our FSM-TDES translation approach. This FSM-TDES-FSM translation cycle also helps us in verifying consistency and compatibility between the the two translation approaches that we aim to achieve.

It is worth-mentioning that for the same specifications of the 4-bit Combination Lock, we can design TDES supervisors in a variety of different ways. Therefore, we cannot really compare our translated TDES supervisors with manually designed supervisors to verify the correctness of our FSM-TDES translation approach. However, we hypothesize that the translated Moore FSM should be *equivalent* to our manually designed Moore FSM with respect to valid input combinations (defined later), if they model the same set of specifications.

For the Combination Lock system, we provided the three translated modular TDES supervisors (given in Section 13.3) as an input to the TDES-FSM translation method in DESpot (2023). As an output, we get four XML files: one file for the central FSM and three files for individual Moore FSM.

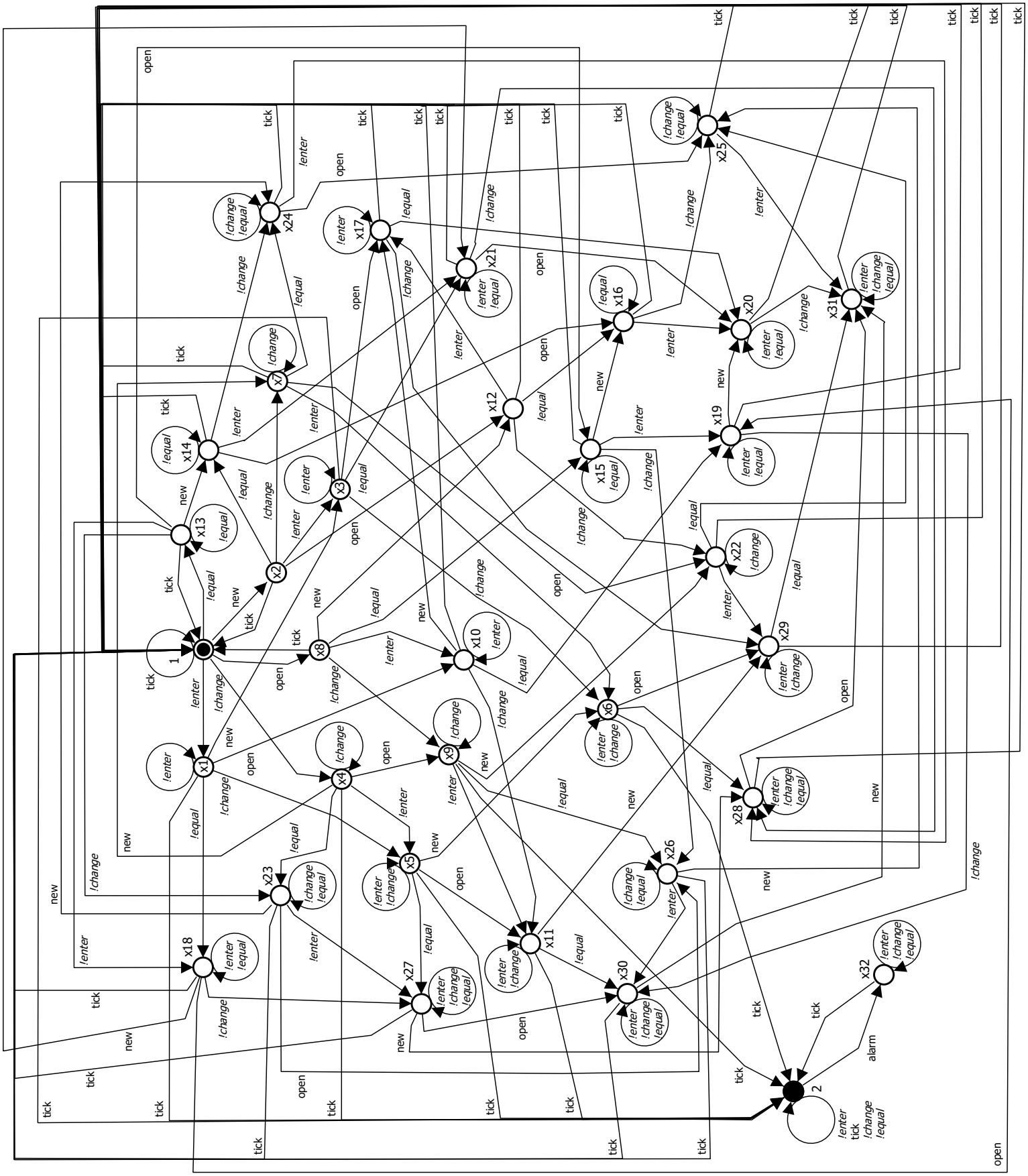


Figure E.2: Translated TDES Supervisor **ActivateAlarm**

E.4.1 Central FSM

By comparing the translated central FSM XML file with our manually designed central FSM file, we note that the two central FSM for the 4-bit Combination Lock are identical, i.e. we get the exact same translated central FSM as the central FSM that we started with while manually specifying our Moore system. Our central FSM for the Combination Lock system is given in XML Input File E.4 (Section E.1.2).

E.4.2 Individual Moore FSM

Now we will compare our manually designed individual Moore FSM for the 4-bit Combination Lock with the Moore FSM that we obtained by applying the TDES-FSM translation algorithms (Section C.2) on our translated TDES supervisors. In the following discussion, we will refer to our manually designed Moore FSM that we started with (Section 13.2.1) as the "designed" FSM, and the output FSM of the TDES-FSM translation method as the "translated" FSM.

Moore FSM-1: OpenLock

By applying the TDES-FSM translation method on our translated TDES supervisor **OpenLock** (Figure 11.2), we get the translated Moore FSM shown in Figure E.3. After comparing this translated FSM with our designed FSM (Figure 11.1), we note that the two FSM have the same number of states and signals. Also, they produce the same output information for the IO signal *open* at both states of the FSM.

The designed and translated FSM **OpenLock** express their next state conditions (input combinations) as boolean expressions. Please note that currently, the TDES-FSM translation method generates boolean expressions in a non-minimal form. We have simplified these non-minimal boolean expressions by applying boolean algebra

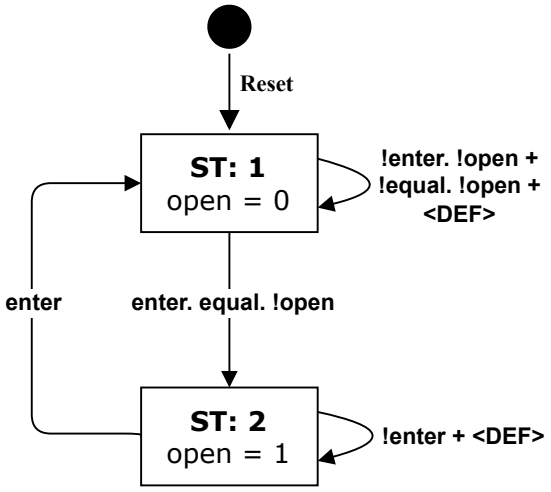


Figure E.3: Translated Moore FSM **OpenLock**

properties (Brown and Vranesic, 2013), so that they could fit the graphical representation of our translated FSM without making it cluttered, as shown in Figure E.3.

It is apparent that at state 1, the boolean expressions for the designed and translated **OpenLock** FSM are not identical. A closer examination reveals that the corresponding boolean expressions of the two FSM are not even equivalent. This implies that the NSL of the two FSM at state 1 are different, hence confirming that the two FSM are not isomorphic up to state relabelling. For example, in the designed FSM, the input combination of $enter \cdot equal \cdot open$ goes from state 1 to state 2. On the other hand, this input combination does not appear explicitly at state 1 in the translated FSM. This means that $enter \cdot equal \cdot open$ is covered by the **DEF** transition that is generated by DESpot at state 1 of the translated FSM. Not to mention, that DESpot always specifies **DEF** as a selfloop transition.

However, it is notable that at state 1, the output of *open* is set to '0' ($False$) by the designed as well as the translated **OpenLock** FSM. As discussed at **Step 4** of Section 11.3.5, if the output of an IO signal is set to '0' by a modular controller, this guarantees that the global output of this IO signal will be '0'. Since the global output is fed back as an input to all individual FSM, this means the input of this IO signal cannot be '1'. This in turn implies that all the input combinations in which this IO signal appears as '1' cannot be $True$. In other words, these input combinations will be considered as *invalid*, as they cannot occur in the physical system at this point in time.

Please note that for the designed **OpenLock** FSM, since $enter \cdot equal \cdot open$ is invalid and cannot evaluate to $True$ at state 1, it does not really matter what destination state we specify for this invalid input combination in our theoretical model. Keeping this in view, we combined the valid next state condition of $enter \cdot equal \cdot !open$ with the invalid input combination $enter \cdot equal \cdot open$ in order to simplify the NSL and obtain a compact boolean expression while designing the **OpenLock** FSM by hand.

The above discussion makes it clear that although the designed and translated FSM specify different destination states for the input combination of $enter \cdot equal \cdot open$, it does not really represent a "conflict" that makes the two FSM dissimilar. Also, since this input combination is invalid, it does not make any difference whether it is explicitly defined at state 1 or is covered by **DEF**. In fact, it is not surprising that an invalid next state condition that is explicitly specified in the manually designed FSM becomes part of the **DEF** transition in the translated FSM, since this was the primary purpose of introducing **DEF** in the TDES-FSM translation (Wang, 2009), i.e. to represent all invalid transitions of the TDES supervisor. Consequently, this difference between the NSL of the two FSM with respect to the invalid input combination of $enter \cdot equal \cdot open$ looks negligible, as it does not seem to affect the system specifications or violate any control laws.

Keeping this in view, instead of comparing NSL of the designed and translated FSM for all input combinations, we will now restrict our comparison to *valid* input

combinations only, i.e. at any given state, whether or not the two FSM have the same destination states for valid input combinations. Therefore, we will now focus on determining *equivalence* of the designed and translated FSM with respect to valid next state conditions that could occur in the system.

In order to clearly compare the NSL of the designed and translated **OpenLock** FSM, it looks suitable to present this information in the form of a table. Table E.1 shows all NSL specified by the designed and translated FSM. After writing **Row No.** in the first column of the table, the next three columns represent three signals of the FSM, *open*, *enter* and *equal*. The column of **Start State: 1** gives all NSL for state 1 of the designed and translated FSM. Specifically, its two subcolumns list down the end (destination) states that are specified in the designed and translated FSM for all input combinations. Likewise, the last column of **Start State: 2** shows the complete NSL for state 2 of the two FSM. Starting at an FSM state, if the end states of the designed and translated FSM are different for the given input combination (valid or invalid), we highlight this difference by using grey background for the respective end state cells of the two FSM.

First, we compare the designed and translated FSM with respect to the NSL specified at state 1. By looking at Table E.1, it is easy to see that the two FSM specify same end states for the input combinations of $!open \cdot !enter \cdot !equal$ (*R-1*), $!open \cdot !enter \cdot equal$ (*R-2*), $!open \cdot enter \cdot !equal$ (*R-3*) and $!open \cdot enter \cdot equal$ (*R-4*). Since $open = 0$ at state 1, all these input combinations are valid and could occur in the system. This means that the two FSM have the same NSL for these four valid input combinations that are defined at state 1.

The row *R-5* represents the input combination of $open \cdot !enter \cdot !equal$. The table shows that the destination state for this input combination in the designed FSM is 1. However, this input combination is covered by the **DEF** transition in the translated FSM. Since $open = 0$ at state 1, this input combination of *R-5* is invalid and cannot occur in the system while **OpenLock** is at state 1. Hence, we will ignore this difference between the NSL of the two FSM, as this row does not contribute towards determining the equivalence of the two FSM.

We note that for rows *R-6*, *R-7* and *R-8*, the cells for the end states of the designed and translated FSM are highlighted with grey background. This indicates that for the input combinations represented by these three rows, the end states of the two FSM are different. However, we can ignore this difference between the NSL of the two FSM, as these three rows represent invalid input combinations at state 1. In summary, it is clear that there are four valid input combinations at state 1 (*R-1*−*R-4*), and the designed and translated FSM have the same destination states for all these valid input combinations.

Now we compare the NSL of the two FSM at state 2. By looking at the two subcolumns of **Start State: 2**, we quickly notice that none of the end state cells of these subcolumns is highlighted in grey. This indicates that the designed and translated FSM specify same end states for all possible input combinations. Since

Table E.1: Boolean Next State Logic for Designed vs. Translated Moore FSM **OpenLock**

Row No.	***open***	***enter***	***equal***	**Start State: 1**		**Start State: 2**	
				End State *(Designed)*	**End State** *(Translated)*	**End State** *(Designed)*	**End State** *(Translated)*
R-1	*0*	*0*	*0*	**1**	**1**	**2**	**2**
R-2	*0*	*0*	*1*	**1**	**1**	**2**	**2**
R-3	*0*	*1*	*0*	**1**	**1**	**1**	**1**
R-4	*0*	*1*	*1*	**2**	**2**	**1**	**1**
R-5	*1*	*0*	*0*	**1**	**DEF**	**2**	**2**
R-6	*1*	*0*	*1*	**1**	**DEF**	**2**	**2**
R-7	*1*	*1*	*0*	**1**	**DEF**	**1**	**1**
R-8	*1*	*1*	*1*	**2**	**DEF**	**1**	**1**

open = 1 at state 2, all input combinations (*R-1−R-8*) will be considered as valid. This implies that at state 2, the two FSM specify the same NSL for all valid input combinations.

Based on the aforementioned analysis, it is evident that at any given state, the designed and translated **OpenLock** FSM have the same NSL for all valid input combinations. Hence, we conclude that the two FSM are equivalent with respect to valid next state conditions that could occur in the physical system. This verifies the correctness of our FSM-TDES translation approach for the **OpenLock** FSM and shows that our devised translation method is capable of preserving the defined system specifications and control laws for this FSM.

Finally, please recall that as mentioned in Chapter 11, one of our major goals while developing the FSM-TDES translation approach was to establish compatibility and consistency between our approach and the existing TDES-FSM translation method. By performing this complete cycle of FSM-TDES-FSM translation for **OpenLock** and going back and forth between the two models, we have also demonstrated the desired compatibility and consistency between the two translation approaches. This also verifies the correctness of the changes that we made in the TDES-FSM translation method, specifically our TDES-FSM translation algorithms presented in Section C.2.

Moore FSM-2: ChangeCode

Figure E.4 shows the translated Moore FSM that is generated from our translated TDES supervisor (Figure E.1) using the TDES-FSM translation method. By comparing the translated FSM **ChangeCode** with our designed FSM (Figure 13.3), it is obvious that the two FSM have the same number of states and signals. Also, they produce the same output information for the two IO signals, *new* and *save_code*, at all FSM states.

Table E.2 gives NSL at all states of the designed and translated FSM for all input combinations. First, by looking at the graphical representation of the designed (Figure 13.3) and translated (Figure E.4) FSM, we note that at state 1, the outputs for both IO signals, *new* and *save_code*, are set to '0'. This means that at state 1, we are only interested in comparing the destination states of those input combinations in which *new* and *save_code* appear as '0'.

In Table E.2, the valid input combinations at state 1 are represented by rows *R-1−R-4*. By comparing the end states of the designed and translated FSM for *R-1−R-4*, it is evident that both FSM specify the same end states for these four valid input combinations. Please note that for the rest of the input combinations at state 1 (*R-5−R-16*), the end states cells of the two FSM are coloured in grey. This indicates that the designed and translated FSM specify different end states for all input combinations that are invalid at state 1. However, we will ignore this difference because of the reasons discussed in the previous section.

At state 2, the designed and translated FSM set the output value of *new* to '1'

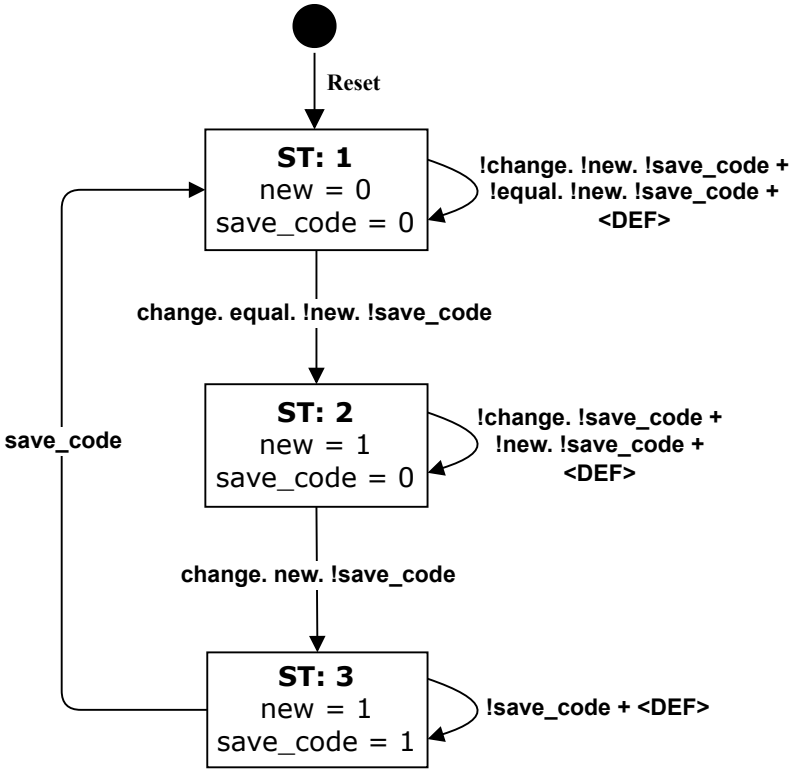


Figure E.4: Translated Moore FSM **ChangeCode**

and *save_code* to '0'. Hence, the set of input combinations that are valid at state 2 is represented by rows *R-1−R-4* and *R-9−R-12*. Since, for these rows, none of the end state cells of the two FSM are coloured in grey, this shows that the two FSM specify the same end states for all valid input combinations. Also, it is obvious that all the coloured end state cells for **Start State: 2** correspond to the input combinations that have *save_code* as '1' (*R-5−R-8*, *R-13−R-16*), i.e. they represent invalid next state conditions at state 2 of the FSM that cannot occur in the physical system.

At state 3, the outputs for both IO signals, *new* and *save_code* are set to '1' in the designed and translated FSM. This implies that all input combinations represented by rows *R-1−R-16* could occur in the system and are considered as valid. By looking at the column of **Start State: 3**, we note that the two FSM specify the same end states for all input combinations, i.e. they have the same NSL for all valid input combinations at state 3.

From the above-mentioned comparison, it is clear that the designed and translated **ChangeCode** FSM specify the same NSL for all input combinations that are valid at any given FSM state. Hence, we conclude that the two FSM are equivalent with respect to valid next state conditions that could occur in the physical system.

Table E.2: Boolean Next State Logic for Designed vs. Translated Moore FSM **ChangeCode**

Row No.	***new***	***save_code***	***change***	***equal***	**Start State: 1**		**Start State: 2**		**Start State: 3**	
					End State *(Designed)*	**End State** *(Translated)*	**End State** *(Designed)*	**End State** *(Translated)*	**End State** *(Designed)*	**End State** *(Translated)*
R-1	*0*	*0*	*0*	*0*	**1**	**1**	**2**	**2**	**3**	**3**
R-2	*0*	*0*	*0*	*1*	**1**	**1**	**2**	**2**	**3**	**3**
R-3	*0*	*0*	*1*	*0*	**1**	**1**	**2**	**2**	**3**	**3**
R-4	*0*	*0*	*1*	*1*	**2**	**2**	**2**	**2**	**3**	**3**
R-5	*0*	*1*	*0*	*0*	**1**	**DEF**	**2**	**DEF**	**1**	**1**
R-6	*0*	*1*	*0*	*1*	**1**	**DEF**	**2**	**DEF**	**1**	**1**
R-7	*0*	*1*	*1*	*0*	**1**	**DEF**	**2**	**DEF**	**1**	**1**
R-8	*0*	*1*	*1*	*1*	**2**	**DEF**	**2**	**DEF**	**1**	**1**
R-9	*1*	*0*	*0*	*0*	**1**	**DEF**	**2**	**2**	**3**	**3**
R-10	*1*	*0*	*0*	*1*	**1**	**DEF**	**2**	**2**	**3**	**3**
R-11	*1*	*0*	*1*	*0*	**1**	**DEF**	**3**	**3**	**3**	**3**
R-12	*1*	*0*	*1*	*1*	**2**	**DEF**	**3**	**3**	**3**	**3**
R-13	*1*	*1*	*0*	*0*	**1**	**DEF**	**2**	**DEF**	**1**	**1**
R-14	*1*	*1*	*0*	*1*	**1**	**DEF**	**2**	**DEF**	**1**	**1**
R-15	*1*	*1*	*1*	*0*	**1**	**DEF**	**3**	**DEF**	**1**	**1**
R-16	*1*	*1*	*1*	*1*	**2**	**DEF**	**3**	**DEF**	**1**	**1**

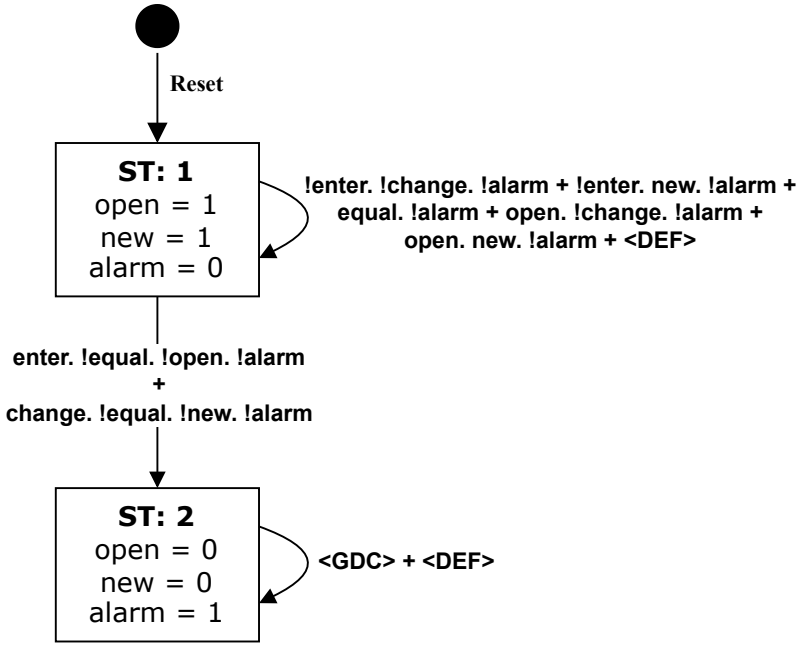


Figure E.5: Translated Moore FSM **ActivateAlarm**

Moore FSM-3: ActivateAlarm

The Moore FSM that we obtain from our translated TDES supervisor **Activate Alarm** (Figure E.2) using the TDES-FSM translation method is shown in Figure E.5. By looking at our designed (Figure 13.4) and translated FSM of **ActivateAlarm**, it is evident that the two FSM have the same number of states and signals. Also, they specify the same output information for the IO signals, *open*, *new* and *alarm*, at both states of the FSM.

Table E.3 shows NSL at both states of the designed and translated **ActivateAlarm** FSM for all input combinations. The graphical representation of the designed (Figure 13.4) and translated (Figure E.5) FSM shows that at initial state 1, both FSM set the output values of *open* and *new* to '1', and *alarm* to '0'. Therefore, for both FSM, the valid input combinations at state 1 are represented by rows *R-1*$-$*R-8*, *R-17*$-$*R-24*, *R-33*$-$*R-40* and *R-49*$-$*R-56* of Table E.3.

By examining the end states of the input combinations that are valid at state 1 of the designed and translated FSM, we observe that the two FSM specify the same NSL for all valid input combinations. Also, it is apparent that all the end state cells that are coloured in grey for **Start State: 1** correspond to the input combinations in which *alarm* appears as '1'. In other words, they represent invalid next state conditions that cannot be $True$ in the physical system while **ActivateAlarm** FSM is at state 1.

At state 2, the designed and translated FSM set the output values of *alarm* to '1', and *open* and *new* to '0'. Therefore, for both FSM, *R-1*$-$*R-16* represent the set of

valid input combinations at state 2. Table E.3 shows that none of the end state cells for **Start State: 2** are coloured in grey for *R-1*$-$*R-16*. This indicates that the two FSM specify the same NSL for all input combinations that are valid at state 2.

It is noteworthy that at state 2, both FSM have the same end states even for the invalid input combinations. This is because of the **GDC** transition that is detected and generated by our TDES-FSM translation Algorithm C.2 at state 2.

The above-stated comparison makes it obvious that at any given state, the designed and translated **ActivateAlarm** FSM specify the same NSL for all valid input combinations. Hence, we conclude that the two FSM are equivalent with respect to valid next state conditions that could occur in the physical system.

Table E.3: Boolean Next State Logic for Designed vs. Translated Moore FSM **ActivateAlarm**

Row No.	***open***	***new***	***alarm***	***enter***	***change***	***equal***	**Start State: 1**		**Start State: 2**	
							End State *(Designed)*	**End State** *(Translated)*	**End State** *(Designed)*	**End State** *(Translated)*
R-1	*0*	*0*	*0*	*0*	*0*	*0*	**1**	**1**	**2**	**2**
R-2	*0*	*0*	*0*	*0*	*0*	*1*	**1**	**1**	**2**	**2**
R-3	*0*	*0*	*0*	*0*	*1*	*0*	**2**	**2**	**2**	**2**
R-4	*0*	*0*	*0*	*0*	*1*	*1*	**1**	**1**	**2**	**2**
R-5	*0*	*0*	*0*	*1*	*0*	*0*	**2**	**2**	**2**	**2**
R-6	*0*	*0*	*0*	*1*	*0*	*1*	**1**	**1**	**2**	**2**
R-7	*0*	*0*	*0*	*1*	*1*	*0*	**2**	**2**	**2**	**2**
R-8	*0*	*0*	*0*	*1*	*1*	*1*	**1**	**1**	**2**	**2**
R-9	*0*	*0*	*1*	*0*	*0*	*0*	**1**	**DEF**	**2**	**2**
R-10	*0*	*0*	*1*	*0*	*0*	*1*	**1**	**DEF**	**2**	**2**
R-11	*0*	*0*	*1*	*0*	*1*	*0*	**2**	**DEF**	**2**	**2**
R-12	*0*	*0*	*1*	*0*	*1*	*1*	**1**	**DEF**	**2**	**2**
R-13	*0*	*0*	*1*	*1*	*0*	*0*	**2**	**DEF**	**2**	**2**
R-14	*0*	*0*	*1*	*1*	*0*	*1*	**1**	**DEF**	**2**	**2**
R-15	*0*	*0*	*1*	*1*	*1*	*0*	**2**	**DEF**	**2**	**2**
R-16	*0*	*0*	*1*	*1*	*1*	*1*	**1**	**DEF**	**2**	**2**
R-17	*0*	*1*	*0*	*0*	*0*	*0*	**1**	**1**	**2**	**2**
R-18	*0*	*1*	*0*	*0*	*0*	*1*	**1**	**1**	**2**	**2**
R-19	*0*	*1*	*0*	*0*	*1*	*0*	**1**	**1**	**2**	**2**
R-20	*0*	*1*	*0*	*0*	*1*	*1*	**1**	**1**	**2**	**2**

Continued on the next page

Continued from previous page

Row No.	*open*	*new*	*alarm*	*enter*	*change*	*equal*	Start State: 1		Start State: 2	
							End State *(Designed)*	**End State** *(Translated)*	**End State** *(Designed)*	**End State** *(Translated)*
R-21	*0*	*1*	*0*	*1*	*0*	*0*	**2**	**2**	**2**	**2**
R-22	*0*	*1*	*0*	*1*	*0*	*1*	**1**	**1**	**2**	**2**
R-23	*0*	*1*	*0*	*1*	*1*	*0*	**2**	**2**	**2**	**2**
R-24	*0*	*1*	*0*	*1*	*1*	*1*	**1**	**1**	**2**	**2**
R-25	*0*	*1*	*1*	*0*	*0*	*0*	**1**	**DEF**	**2**	**2**
R-26	*0*	*1*	*1*	*0*	*0*	*1*	**1**	**DEF**	**2**	**2**
R-27	*0*	*1*	*1*	*0*	*1*	*0*	**1**	**DEF**	**2**	**2**
R-28	*0*	*1*	*1*	*0*	*1*	*1*	**1**	**DEF**	**2**	**2**
R-29	*0*	*1*	*1*	*1*	*0*	*0*	**2**	**DEF**	**2**	**2**
R-30	*0*	*1*	*1*	*1*	*0*	*1*	**1**	**DEF**	**2**	**2**
R-31	*0*	*1*	*1*	*1*	*1*	*0*	**2**	**DEF**	**2**	**2**
R-32	*0*	*1*	*1*	*1*	*1*	*1*	**1**	**DEF**	**2**	**2**
R-33	*1*	*0*	*0*	*0*	*0*	*0*	**1**	**1**	**2**	**2**
R-34	*1*	*0*	*0*	*0*	*0*	*1*	**1**	**1**	**2**	**2**
R-35	*1*	*0*	*0*	*0*	*1*	*0*	**2**	**2**	**2**	**2**
R-36	*1*	*0*	*0*	*0*	*1*	*1*	**1**	**1**	**2**	**2**
R-37	*1*	*0*	*0*	*1*	*0*	*0*	**1**	**1**	**2**	**2**
R-38	*1*	*0*	*0*	*1*	*0*	*1*	**1**	**1**	**2**	**2**
R-39	*1*	*0*	*0*	*1*	*1*	*0*	**2**	**2**	**2**	**2**
R-40	*1*	*0*	*0*	*1*	*1*	*1*	**1**	**1**	**2**	**2**
R-41	*1*	*0*	*1*	*0*	*0*	*0*	**1**	**DEF**	**2**	**2**
R-42	*1*	*0*	*1*	*0*	*0*	*1*	**1**	**DEF**	**2**	**2**

Continued on the next page

Continued from previous page

Row No.	*open*	*new*	*alarm*	*enter*	*change*	*equal*	Start State: 1		Start State: 2	
							End State *(Designed)*	**End State** *(Translated)*	**End State** *(Designed)*	**End State** *(Translated)*
R-43	*1*	*0*	*1*	*0*	*1*	*0*	**2**	**DEF**	**2**	**2**
R-44	*1*	*0*	*1*	*0*	*1*	*1*	**1**	**DEF**	**2**	**2**
R-45	*1*	*0*	*1*	*1*	*0*	*0*	**1**	**DEF**	**2**	**2**
R-46	*1*	*0*	*1*	*1*	*0*	*1*	**1**	**DEF**	**2**	**2**
R-47	*1*	*0*	*1*	*1*	*1*	*0*	**2**	**DEF**	**2**	**2**
R-48	*1*	*0*	*1*	*1*	*1*	*1*	**1**	**DEF**	**2**	**2**
R-49	*1*	*1*	*0*	*0*	*0*	*0*	**1**	**1**	**2**	**2**
R-50	*1*	*1*	*0*	*0*	*0*	*1*	**1**	**1**	**2**	**2**
R-51	*1*	*1*	*0*	*0*	*1*	*0*	**1**	**1**	**2**	**2**
R-52	*1*	*1*	*0*	*0*	*1*	*1*	**1**	**1**	**2**	**2**
R-53	*1*	*1*	*0*	*1*	*0*	*0*	**1**	**1**	**2**	**2**
R-54	*1*	*1*	*0*	*1*	*0*	*1*	**1**	**1**	**2**	**2**
R-55	*1*	*1*	*0*	*1*	*1*	*0*	**1**	**1**	**2**	**2**
R-56	*1*	*1*	*0*	*1*	*1*	*1*	**1**	**1**	**2**	**2**
R-57	*1*	*1*	*1*	*0*	*0*	*0*	**1**	**DEF**	**2**	**2**
R-58	*1*	*1*	*1*	*0*	*0*	*1*	**1**	**DEF**	**2**	**2**
R-59	*1*	*1*	*1*	*0*	*1*	*0*	**1**	**DEF**	**2**	**2**
R-60	*1*	*1*	*1*	*0*	*1*	*1*	**1**	**DEF**	**2**	**2**
R-61	*1*	*1*	*1*	*1*	*0*	*0*	**1**	**DEF**	**2**	**2**
R-62	*1*	*1*	*1*	*1*	*0*	*1*	**1**	**DEF**	**2**	**2**
R-63	*1*	*1*	*1*	*1*	*1*	*0*	**1**	**DEF**	**2**	**2**
R-64	*1*	*1*	*1*	*1*	*1*	*1*	**1**	**DEF**	**2**	**2**