Symbolic Decentralized Supervisory Control

SYMBOLIC DECENTRALIZED SUPERVISORY CONTROL

BY

URVASHI AGARWAL, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

© Copyright by Urvashi Agarwal, September 2013

All Rights Reserved

Master of Science (2013)
(Computing & Software)

McMaster University
Hamilton, Ontario, Canada

TITLE: Symbolic Decentralized Supervisory Control

AUTHOR: Urvashi Agarwal
B.Eng., (Computer Science)

SUPERVISOR: Dr. Ryan J. Leduc
Dr. S. L. Ricker

NUMBER OF PAGES: viii, 117

Abstract

A decentralized discrete-event system (DES) consists of supervisors that are physically distributed. Co-observability is one of the necessary and sufficient conditions for the existence of a decentralized supervisors that correctly solve the control problem. In this thesis we present a state-based definition of co-observability and introduce algorithms for its verification. Existing algorithms for the verification of co-observability do not scale well, especially when the system is composed of many components. We show that the implementation of our state-based definition leads to more efficient algorithms.

We present a set of algorithms that use an existing structure for the verification of state-based co-observability (SB Co-observability). A computational complexity analysis of the algorithms show that the state-based implementation of algorithms result in quadratic complexity. Further improvements come from using a more compact way of representing finite-state machines namely Binary Decision Diagrams (BDD).

Acknowledgements

First, I would like to sincerely thank my supervisors Dr. Ryan Leduc and Dr. Laurie Ricker for their guidance, support and patience, without which it would have been impossible to accomplish my goals. I am very grateful to them.

I would also like to thank Dr. Robi Malik for suggesting the $\sim_{\Omega}$ relation and contributing to Non-BDD SB co-observability algorithm, particularly suggesting the observer approach. I also thank him for his timely review and feedback on certain portions of the thesis. I thank Raoguang Song and Yu Wang for their useful work on BDD based symbolic verification. I would like to thank Abubaker Shangab and Lana Kayyali for their interesting discussions on DESpot structure. I thank all my friends and relatives that supported me during the program.

At last, I would like to thank my beloved parents, Dr. Avinash Agarwal and Madhu Agarwal, for their unlimited support, great understanding and confidence in me. This thesis is dedicated to them.

Contents

Abstract **iii**

Acknowledgements **iv**

1 Introduction **1**

1.1 Thesis Outline 4

1.2 Related Work 5

2 Preliminaries **8**

2.1 Supervisory Control Theory 8

2.2 Decentralized Supervisory Control 11

2.2.1 Controller Synthesis 14

3 Introduction to State-Based Co-observability **17**

3.1 Co-observability Verification Structure 18

3.2 Towards State-based Co-observability 22

3.3 State-Based Definition of Co-observability 25

3.3.1 Basic Definitions 25

3.3.2 State-based Implies Language-based Definition 28

3.3.3 Does Language-Based Imply State-Based Definition 32

4 State-Based Algorithms 35

4.1 Verifying SB Co-observability 35

4.2 Algorithms 37

4.2.1 Is SB Co-observable Using Subset Construction 37

4.2.2 Enable Disable Set Find 40

4.2.3 Construct Synch Product and Hiding 43

4.2.4 Subset Construction 45

4.2.5 Is SB Co-observable Using Observer Construction 50

4.2.6 Observer Construction 53

4.2.7 Indistinguishability and Observers 58

4.3 Complexity Analysis 59

4.3.1 Algorithm Complexity 60

4.3.2 Comparing Complexity 62

4.4 Small Example 63

4.5 Counter Example 67

5 Symbolic Verification of Decentralized DES 71

5.1 Predicates And Predicate Transformers 72

5.1.1 Predicates 72

5.1.2 Predicate Transformers 74

5.2 Symbolic Representation 74

5.2.1 State Subsets 75

5.2.2 Transitions 75

5.3 Symbolic Computation 77
5.3.1 Transition and Inverse Transition 77
5.3.2 Computation of Reachability 78
5.4 Symbolic Verification 79
5.4.1 Computation Of P_{Dis} And P_{En} 81
5.5 Algorithms 82
5.5.1 Is SB Co-observable 83
5.5.2 Enable Disable Set Find 86
5.5.3 Construct Sync Product And Hiding 87
5.5.4 Observer Construction 89
5.6 BDD Complexity 94
5.7 Advantages of Using Predicates 94

6 Parallel Manufacturing Example 96
6.1 Introduction 96
6.2 Casting an HISC as a Decentralized Control Problem 102
6.3 Results 106

7 Conclusions and Future Work 110
7.1 Conclusions 110
7.2 Future Work 111

List of Figures

3.1 Plant and Specification Automaton 19

3.2 (a) Projection Automaton 1, (b) Projection Automaton 2 19

3.3 Verifier 22

3.4 State-based Verifier 24

3.5 Counter example 33

4.1 An Example DES 63

4.2 Synchronous Product of Plant and Specification Automaton 65

4.3 Counter Example for SB co-observability 68

4.4 Observer $\mathbf{H_1}$ 69

4.5 Observer $\mathbf{H_2}$ 70

6.1 Block Diagram of Parallel Plant [19] 97

6.2 Plant Models for Manufacturing unit j [19] 98

6.3 Low-Level system [19] 99

6.4 Complete Parallel Manufacturing System [19] 100

6.5 Flat Plant Model for Parallel Manufacturing System 104

6.6 Flat Supervisor Model for Parallel Manufacturing System 105

6.7 Screen Shot of the Decentralized editor 107

Chapter 1

Introduction

Discrete-Event Systems (DES) is an area of research that has been applied to a wide range of complex applications such as manufacturing systems [21], [30]; database management systems [18], [20]; communication protocols [3], [4], [35], [41]; and traffic systems [15]. We are modeling discrete-event systems using finite-state machines that generate regular languages [29]. A DES can model a deterministic or non-deterministic dynamic system with a finite state space and a state-transition structure. The control of DES requires the creation of supervisors that modify the behaviour of the DES through feedback control to achieve a given specification. This is accomplished by restricting the system, through issuing disable or enable control decisions, from performing a certain set of events that take the system out of the specified behaviour.

A decentralized DES control problem [9], [37], is one in which the supervisory control task is divided among many controllers, instead of just one controller. In this case, each controller does not have a complete view of the tasks being performed by the uncontrolled system, nor can it observe the control actions of the other controllers.

The controllers must co-ordinate to produce a final control decision which, ideally, keeps the system within the specified behaviour.

In a centralized DES, *controllability* is a property that must be satisfied to construct supervisors that will exhibit a satisfactory control over the uncontrolled system. A centralized supervisor is typically capable of observing all the events in a system but, a system that is physically distributed and requires the implementation of its supervisors on multiple sites, will have supervisors that can view and disable only local system events. To meet the requirements of the physical architecture of the system, we implement a decentralized control approach.

Consider an example of a computer network system with different nodes running on the network, each node managed by different controllers. Each controller can observe a set of local events. The controllers do not have access to other controller's control actions. Instead, the controllers take a local control decision on the basis of their local observations. Under such an architecture, it is not feasible for one controller to take the overall control decision, since its local observations may be insufficient to take the correct decision. In such a situation, when we have to implement a control strategy on controllers that can observe and disable only certain events of the system, there is no choice but to implement a decentralized control solution. An overall control decision is reached by combining the local control decisions: this is the decentralized DES approach.

A significant difference between a centralized and a decentralized DES control

problem is the computational complexity introduced, since the decentralized controllers have only a partial view of the overall system. Co-observability, in a decentralized DES, describes whether or not the decentralized controllers have a satisfactory view of the uncontrolled system to reach the correct control decision. For co-observability to be satisfied, there has to be at least one controller that knows when to disable a particular event, i.e., to prevent the system from performing a behaviour outside of the specification.

The decentralized DES control problem, as defined in [37], requires controllability and co-observability to be verified for the existence of decentralized controllers that correctly solve the control problem. The verification of co-observability, as introduced in [36], is based on a language-based definition of co-observability from [37]. The computational complexity of this approach is exponential in number of controllers. However, the approach does not scale well. As a result, we are interested in re-examining the definition of co-observability and exploring more computationally-efficient algorithms to verify this property.

In this thesis we introduce a state-based definition, as opposed to a language-based definition, of co-observability and use this definition to motivate a new algorithm for the verification of co-observability. We present a suite of algorithms that are more efficient from a computational perspective in time complexity. Further, the algorithm scales better as we increase the number of controllers. Additionally, from a data structure perspective, we introduce a more compact representation of finite-state machines, namely using Binary Decision Diagrams (BDD) [22]. This representation helps to reduce memory usage and provides a faster look-up of states.

The goal of this thesis is to show that in addition to controllability, state-based co-observability is an equivalent sufficient condition for the existence of decentralized controllers. Additionally, we will show that the structure and the algorithm used for the verification of co-observability makes the process of verification computationally more efficient when compared to the strategies that have been proposed in the past, particularly as the number of decentralized controllers increase.

1.1 Thesis Outline

The thesis has been organized as follows:

In Chapter 2, we provide the basic definitions and concepts related to decentralized DES.

In Chapter 3 we introduce a state-based definition of co-observability and a definition of indistinguishability of two states in a DES. We use these definitions to prove that the state-based definition of co-observability implies the language-based definition; however, the opposite is not true. We use the proof to confirm the existence of a solution to the decentralized problem in [37] using state-based co-observability.

In Chapter 4 we introduce automaton-based algorithms to verify co-observability using the *OP-Verifier* introduced in [26]. We perform a complexity analysis of the algorithms and give a small example that illustrates the working of the algorithm.

In Chapter 5 we introduce predicates and predicate transformers, and use them to introduce predicate-based algorithms. Additionally, we introduce a suite of algorithms to verify SB co-observability and analyze the advantages of using them over non-predicate based algorithms.

In Chapter 6 we illustrate our algorithms using an example taken from [19]. Because of time constraints the result has not been verified yet as our algorithms are in the process of implementation. It will be verified once the software has been completed.

We end the thesis with conclusions and give a brief idea about future work in Chapter 7.

1.2 Related Work

In the 1980s, Ramadge and Wonham initiated the framework of modeling and synthesis of supervisors for discrete-event systems [29]. The objective is to use a finite-state automaton to produce a control solution (i.e., a supervisor) that keeps the given plant's behaviour within a given specification's behaviour, assuming that the supervisor has full observation of the plant. Since then, DES control has been extended in a variety of ways.

The progression for these modifications is as follows: partial-observation control, modular supervisory control, decentralized supervisory control, hierarchical supervisory control and timed DES. We will focus on decentralized control of DES.

Decentralized DES control problems were first introduced in [9]. Later, additional work was done in [21] and [37] to determine the necessary and sufficient conditions, i.e., controllability and co-observability for the existence of a partial observation supervisor(s). An examination of state-based decentralized control appeared in [16]. All these research articles deal with a scenario where the default control decision is to enable all the events, i.e., when a controller is uncertain of the correct control decision to take, it chooses to enable. In this architecture, the rule to combine, or *fuse*, local

control decisions is conjunction. There are other decentralized architectures that use different rules. Decision fusion was discussed in [27], and since then, various fusion rules have been examined in [46], [44], [45]. The ambiguities related to the global control decision along with various models to handle multiple levels of inferencing were introduced in [17]. In [8], a so-called "parallel architecture" was introduced, but this approach requires a further partitioning of the state space to facilitate local decision making. The resulting computational complexity was the same as in [17]. For this thesis, we are focusing on the conjunctive decentralized architecture of Rudie and Wonham [37].

The verification of co-observability from a language-based perspective was first introduced in [36]. An exponential algorithm was formulated to test whether or not the specification language was co-observable w.r.t. the plant language. Next came [39] that presented variations of the problem mentioned in [36]. [39] focused on finding k-reliable decentralized supervisors that achieve a specification language under failure of $n - k$ decentralized supervisors. Additionally, the algorithm in [36] has been extended to verify co-observability under different decentralized architectures [13], [23], [24], [39], [46]. Algorithms for the DES control problem from a state-based perspective was presented in [13]. The algorithm is formulated for only two decentralized supervisors as the generalization of the approach for $n > 2$ supervisors is left as an open problem. The computational complexity for $n = 2$ supervisors is already daunting and it is unclear how to extend their approach in a computationally-feasible way.

A structure, based on subset construction [28], has been used for verifying properties like co-observability [3], [23], [24], [31] and [36]. The computational complexity of this construction from a DES perspective is presented in [42].

We will be using Binary Decision Diagrams (BDD) [6] for compact representation of finite-state machines. BDD have been used to represent DES and to verify discrete-event system properties before, although not for decentralized control. For an introduction to BDDs, we refer the reader to [14]. BDD implementations have been used by [1], [25], [38], [40].

Chapter 2

Preliminaries

This chapter presents the background of basic supervisory control theory, which was introduced by Ramdage and Wonham, followed by an introduction to decentralized supervisory control. In this thesis we assume all the automata to be reachable, for simplicity. For more details, please refer to [7], [43].

2.1 Supervisory Control Theory

The uncontrolled system, which we call a *plant*, is modeled by an automaton

$$\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m),$$

where Q is the set of *states*, Σ is a finite set of event *labels*, $q_0 \in Q$ is the *initial* state and $Q_m \subseteq Q$ is the set of *marked states*.

The transition function $\delta : Q \times \Sigma \to Q$, is a partial function. It can be extended to $\delta : Q \times \Sigma^* \to Q$ as follows:

Let $q \in Q$, $s \in \Sigma^*$ and $\sigma \in \Sigma$ such that,

$$\delta(q,\varepsilon) = q,$$
$$\delta(q,s\sigma) = \delta(\delta(q,s),\sigma).$$

as long as $q' := \delta(q,s)!$ and $\delta(q',\sigma)!$. The notation $\delta(q,\sigma)!$ indicates that δ is defined for σ at state q.

The set Σ^+ is the set of all sequences of symbols $\sigma_1\ \sigma_2\ \sigma_3...\ \sigma_k$, such that $\sigma_i \in \Sigma$ and $i \in \{1,2,....,k\}$, whereas Σ^* is the set of these finite strings over Σ including the empty string $\varepsilon \notin \Sigma$, i.e., $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

Various kinds of operations are used for regular languages:

▸ *Concatenation* of two or more strings is defined using the *cat* operator such that *cat*: $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$:

$$cat\ (\varepsilon, s) = cat\ (s, \varepsilon) = s,\ s \in \Sigma^*$$
$$cat\ (s, t) = st, \text{ where } st \in \Sigma^+$$

▸ A string $t \in \Sigma^*$ is a *prefix* of string $s \in \Sigma^*$, denoted by $t \leq s$, if $s = tu$, for some $u \in \Sigma^*$. The prefix-closure of any language $L \subseteq \Sigma^*$ is defined as:

$$\overline{L} := \{t \in \Sigma^* | t \leq s \text{ for some } s \in L\}$$

For plant $\mathbf{G}$, we define the following two languages:

▸ The *closed behaviour* of $\mathbf{G}$, denoted by $L(\mathbf{G})$, defines the set of all sequences that the plant may generate.

$$L(\mathbf{G}) := \{s \in \Sigma^* \mid \delta(q_0,s)!\}$$

▸ The *marked behaviour* of $\mathbf{G}$, denoted by $L_m(\mathbf{G})$, is the set of strings leading from the initial state to a marked state. It is defined as:

$$L_m(\mathbf{G}) := \{s \in \Sigma^* \mid \delta(q_0,s) \in Q_m\}$$

$L(\mathbf{G})$ is a prefix-closed language and $L_m(\mathbf{G}) \subseteq L(\mathbf{G})$.

We distinguish two state sets for plant $\mathbf{G}$ as follows:

▸ The set of *reachable* states, $Q_r \subseteq Q$, are all the states reachable from the initial state.

$$Q_r := \{q \in Q | (\exists s \in \Sigma^*)\ \delta(q_0, s)! \text{ and } \delta(q_0, s) = q\}.$$

▸ Similarly, the set of *co-reachable* states, $Q_{cr} \subseteq Q$, are all the states that have a path to a marked state:

$$Q_{cr} := \{q \in Q | (\exists s \in \Sigma^*)\ \delta(q, s)! \text{ and } \delta(q, s) \in Q_m\}.$$

In supervisory control problems under partial observation, the supervisor does not see all of the set of events; rather, they can view some subset of Σ, i.e., $\Sigma_o \subseteq \Sigma$. The natural projection, P: $\Sigma^* \to \Sigma_o^*$, defines the sequences of observable events viewed by the supervisor. Here, P erases all the unobservable events of the supervisor, i.e, $\Sigma \setminus \Sigma_o$.

$$P(\varepsilon) = \varepsilon$$

$$P(\sigma) = \begin{cases} \sigma, & \text{if } \sigma \in \Sigma_o; \\ \varepsilon, & \textit{otherwise.} \end{cases}$$

Natural projection can be extended to Σ^* via concatenation:

$$P(t\sigma) = P(t)P(\sigma),\ t \in \Sigma^*,\ \sigma \in \Sigma.$$

The inverse map of the natural projection for a set $L \subseteq \Sigma_o^*$ is:

$$P^{-1}(L) := \{t' \in \Sigma^* \mid P(t) = t' \text{ for some } t \in L\}$$

The synchronous product ($||$) of two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ can be defined using the natural projection P.

$$L_1||L_2 := P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$$

The synchronous product of $\mathbf{G_1} = (Q_1, \Sigma_1, \delta_1, q_{0,1}, Q_{m1})$ and $\mathbf{G_2} = (Q_2, \Sigma_2, \delta_2, q_{0,2}, Q_{m2})$ is the automaton

$$\mathbf{G_1} \parallel \mathbf{G_2} = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{0,1}, q_{0,2}), Q_{m1} \times Q_{m2}), \tag{1}$$

where

$$\delta((q_1,q_2),\sigma) := \begin{cases} (\delta_1(q_1,\sigma), \delta_2(q_2,\sigma)), & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \text{ and } \delta_1(q_1,\sigma)! \text{ and } \delta_2(q_2,\sigma)!; \\ (\delta_1(q_1,\sigma), q_2), & \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \text{ and } \delta_1(q_1,\sigma)!; \\ (q_1, \delta_2(q_2,\sigma)), & \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \text{ and } \delta_2(q_2,\sigma)!; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

In order to simplify our definition we will often assume that both automata are defined over Σ. In this special case $L(G_1)||L(G_2)$ would reduce to $L(G_1) \cap L(G_2)$. If either of G_1 or G_2 were actually defined over some subset of Σ, say $\Sigma' \subseteq \Sigma$, we would then add self loops of every event in $\Sigma \backslash \Sigma'$, to every state in the automaton to raise its alphabet to Σ. There is thus no loss of generality by our assumption.

2.2 Decentralized Supervisory Control

A supervisor is an agent that has the ability to control certain events based on its (partial) view on the plant's behaviour. To impose such supervision, we partition

Σ into disjoint sets. The set of controllable events, Σ_c, are the events that can be prevented from occurring in the system. The set of uncontrollable events, Σ_{uc}, are the events that cannot be prevented from occurring and are thus considered permanently enabled.

Similarly, there are certain events that are not visible to the *supervisor*, so we partition Σ into the sets of Σ_o and Σ_{uo}, where Σ_o is the set of events observable to the supervisor, and Σ_{uo} is the set of events that are not observable to the supervisor.

$$\Sigma = \Sigma_c \mathbin{\dot\cup} \Sigma_{uc}.$$

$$\Sigma = \Sigma_o \mathbin{\dot\cup} \Sigma_{uo}.$$

Here $\dot\cup$ represents the disjoint union of the two sets.

A *supervisor* S is a pair ($\mathbf{S}$,ψ) where $\mathbf{S} = (X, \Sigma, \xi, x_0, X_m)$ is an automaton that tracks the behaviour of $\mathbf{G}$. It keeps track of the current position of the plant.

The *feedback map*, denoted by $\psi : X \times \Sigma \rightarrow \{0,1\}$ is defined as follows:

$$\psi(x,\sigma) := \begin{cases} 1, & \text{if } \sigma \in \Sigma_{uc} \text{ and } x \in X; \\ \{0,1\}, & \text{if } \sigma \in \Sigma_c \text{ and } x \in X. \end{cases}$$

Here 1 corresponds to enable and 0 corresponds to disable.
The feedback map $\psi(x,\sigma)$ indicates whether σ should be enabled or disabled at a particular state in $\mathbf{G}$. The sequences of events generated while the plant $\mathbf{G}$ is under supervision $S = (\mathbf{S},\psi)$ is called the *closed-loop system*. The generated sequences are both in $\mathbf{G}$ and S, i.e., the plant automaton is under the control of the supervisor automaton.

The behaviour of $\mathbf{G}$ under the control of supervisor S is defined as $L(S/G)$ and is called the *supervised discrete event system* where,

$$S/G = (Q \times X, \Sigma, (\delta \times \xi)^{\psi}, (q_0, x_0), Q_m \times X_m).$$

This modified transition function $(\delta \times \xi)^{\psi}$, can be defined as a mapping $Q \times X \times \Sigma \to Q \times X$ where,

$$(\delta \times \xi)^{\psi}((q,x),\sigma) := \begin{cases} (\delta(q,\sigma), \xi(x,\sigma)), & \text{if } \delta(q,\sigma)! \text{ and } \xi(x,\sigma)! \text{ and } \psi(x,\sigma) = 1 \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

In decentralized supervisory control there is more than one controller. Without loss of generality, we will illustrate the concepts using $n = 2$ local supervisors, but this can easily be extended for arbitrary n. For the rest of this section we assume $i \in \{1,2\}$.

Let $S_i = (\mathbf{S}_i, \psi_i)$, for $i \in \{1,2\}$, be a decentralized supervisor controlling plant $\mathbf{G}$, with $\mathbf{S}_i = (X_i, \Sigma, \xi_i, x_{0,i}, X_{m,i})$ and the feedback map ψ_i. We denote the resulting specification, which is the conjunction of the decentralized supervisors S_i, by S_{dec}:

$$S_{dec} = S_1 \wedge S_2 = (\mathbf{S}_1 \times \mathbf{S}_2, \psi_1 * \psi_2),$$

where

$$\mathbf{S}_1 \times \mathbf{S}_2 := (X_1 \times X_2, \Sigma, \xi_1 \times \xi_2, (x_{0,1}, x_{0,2}), X_{m,1} \times X_{m,2}),$$

and, for $\sigma \in \Sigma, x_1 \in X_1, x_2 \in X_2$,

$$(\xi_1 \times \xi_2)((x_1,x_2),\sigma) = \begin{cases} (\xi_1(x_1,\sigma), \xi_2(x_2,\sigma)), & \text{if } \xi_1(x_1,\sigma)! \text{ and } \xi_2(x_2,\sigma)!; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$$(\psi_1 * \psi_2)((x_1, x_2), \sigma) = \begin{cases} 0, & \text{if } \psi_1(x_1, \sigma) = 0 \text{ or } \psi_2(x_2, \sigma) = 0; \\ 1, & \text{otherwise.} \end{cases}$$

Thus S_{dec} follows a decision-making architecture based on logical conjunction: an event is disabled by S_{dec} as long as at least one S_i issues a disable decision for that event; otherwise the event is enabled.

2.2.1 Controller Synthesis

We consider a problem in which we have a *plant* $\mathbf{G}$ defined over Σ with controllable events $\Sigma_c \subseteq \Sigma$ and observable events $\Sigma_o \subseteq \Sigma$. There is a *specification automaton* $\mathbf{S}$ with language $L(\mathbf{S}) \subseteq L(\mathbf{G})$. The problem is to construct the local controllers S_i, for $i \in \{1, 2\}$ such that

$$L(S_1 \wedge S_2 \backslash \mathbf{G}) = L(\mathbf{S}).$$

As stated above, the local controllers can only control and observe some subset of Σ_c and Σ_o such that $\Sigma_{c,i} \subseteq \Sigma_c$ and $\Sigma_{o,i} \subseteq \Sigma_o$. To find local controllers, we need to verify the properties: *controllability* [29], and *co-observability* [37]. For ease of explanation, we illustrate the definition for $n = 2$, but this can easily be extended for arbitrary n.

Controllability

For a plant $\mathbf{G}$ defined over Σ, where $\Sigma_c \subseteq \Sigma$ is the set of controllable events and $\Sigma_{uc} \subseteq \Sigma$, the set of uncontrollable events, any language $K \subseteq \Sigma^*$ is said to be *controllable* with respect to $\mathbf{G}$ if,

$$\overline{K}\Sigma_{uc} \cap L(G) \subseteq \overline{K},$$

where $\overline{K}\Sigma_{uc} := \{t\sigma \mid t \in \overline{K} \text{ and } \sigma \in \Sigma_{uc}\}$.

If we interpret $L(\mathbf{G})$ as the physically-possible behaviour and $\overline{K}$ as legal behaviour, then we need to make sure that the occurrence of any uncontrollable event does not take the system to an undesired state. That is, when an uncontrollable event is going to take place then either it should be prevented from happening (i.e., by disabling an earlier controllable event) or if it happens, then it should stay within the specification language. For this to occur, K should be controllable.

Co-observability

Co-observability is a necessary and sufficient condition for the synthesis of local controllers. In this section we consider $n \geq 2$ decentralized supervisors. Let $I = \{1,2,..,n\}$ be the index set for the set of controllers.

Given two regular languges L, $K \subseteq \Sigma^*$, where $L = \overline{L}$ and $K \subseteq L$. Let $\Sigma_{o,i} \subseteq \Sigma$ be the set of observable events and $\Sigma_{c,i} \subseteq \Sigma$ be the set of controllable events for the ith decentralized controller, where $i \in I$. Let $P_i : \Sigma^* \to \Sigma^*_{o,i}$ be a natural projection. For $\sigma \in \Sigma_c$, we define $I_c(\sigma) = \{i \in I | \sigma \in \Sigma_{c,i}\}$ to be the set of all controllers that are capable of disabling σ. We can now present the definition of co-observability, as originally defined in [37] and adapted by [3] and [31].

Definition 2.2.1. *A language $K \subseteq L$ is co-observable with respect to $L = \overline{L} \subseteq \Sigma^*$, P_i and $\Sigma_{c,i}$, where $i \in I$, if,*

$$(\forall s \in \overline{K})\ (\forall \sigma \in \Sigma_c)\ s\sigma \in (L \backslash \overline{K}) \Rightarrow (\exists i \in I_c(\sigma))\ P_i^{-1}(P_i(s))\sigma \cap \overline{K} = \varnothing$$

When using automata, we say that specification $\mathbf{S} = (X, \Sigma, \xi, x_0, X_m)$ is co-observable with respect to plant $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$, if, $L(\mathbf{G}||\mathbf{S}) = L(\mathbf{G}) \cap L(\mathbf{S})$

is co-observable with respect to $L(\mathbf{G})$. We cannot use $K = L(\mathbf{S})$ because, in general, $\mathbf{S}$ is not a sub-automaton of $\mathbf{G}$ and the co-observability definition requires that $K \subseteq L(\mathbf{G})$. By taking $K = L(\mathbf{G}||\mathbf{S}) \subseteq L(\mathbf{G})$, we restrict the language $L(\mathbf{S})$ to strings that belong to $L(\mathbf{G})$. This is analogous to how the language-based controllability definition is extended to apply to automata.

As we will frequently refer to the synchronous product of $\mathbf{G}$ and $\mathbf{S}$, we introduce the following notation to refer to the resulting automaton for the remainder of the thesis: $\mathbf{G}||\ \mathbf{S} = (Y, \Sigma, \eta, y_0, Y_m)$, where, in accordance with the definition of synchronous product (1), we let $G_1 = \mathbf{G}$ and $G_2 = \mathbf{S}$.

Theorem 2.2.1 ([37])**.** *Consider a plant $\boldsymbol{G} = (Q, \Sigma, \delta, q_0, Q_m)$, where $\Sigma_{uc} \subseteq \Sigma$ is the set of uncontrollable events, $\Sigma_c = \Sigma \backslash \Sigma_{uc}$ is the set of controllable events, and $\Sigma_o \subseteq \Sigma$ is the set of observable events. For each site $i \in I$, consider the set of controllable events $\Sigma_{c,i}$ and the set of observable events $\Sigma_{o,i}$; overall, $\cup_{i=1}^{n} \Sigma_{c,i} = \Sigma_c$ and $\cup_{i=1}^{n} \Sigma_{o,i} = \Sigma_o$. Let $P_i : \Sigma^* \to \Sigma_{o,i}^*$ be the natural projection where $i \in I$. Consider also the language $K \subseteq L_m(\boldsymbol{G})$, where $K \neq \varnothing$. There exists a nonblocking decentralized supervisor S_{dec} for $\boldsymbol{G}$ such that $L_m(S_{dec}/\boldsymbol{G}) = K$ and $L(S_{dec}/\boldsymbol{G}) = \overline{K}$ if and only if the three following conditions hold:*

1. *K is controllable with respect to $L(\boldsymbol{G})$ and Σ_{uc};*

2. *K is co-observable with respect to $L(\boldsymbol{G})$, P_i and $\Sigma_{c,i}$, $i \in I$;*

3. *K is $L_m(\boldsymbol{G})$-closed.*

Chapter 3

Introduction to State-Based Co-observability

For the verification of a property, a structure with which the property can be verified needs to be built. Various deterministic and non-deterministic finite-state structures have been introduced in the past for the verification of co-observability. In this chapter we will briefly review the structure that used state estimates for the verification of classical definition of co-observability. Next we give our definition of state-based co-observability and, using this definition, prove the existence of decentralized controllers that synthesize a controllable and co-observable specification language.

3.1 Co-observability Verification Structure

To verify observational properties of DES with partial observation, the standard approach is to build an *observer*; however, as there have been several competing definitions of this word in the DES literature, including one we use in subsequent chapters, we will refer to the structure often called an observer (e.g., [7]) as a *projection automaton*. The algorithm used to build this structure, is based on a standard algorithm from automaton theory [12], that converts a non-deterministic automaton with ε-transitions to a deterministic automaton with no ε-transitions. For partial observation, ε-transitions correspond to replacing $\sigma \in \Sigma_{uo}$ by ε. The algorithm constructs a non-deterministic projection automaton where only observable events appear as transition labels and the states are power sets of the state space of the original automaton.

Given a plant $\mathbf{G} = (Q,\Sigma,\delta,q_0,Q_m)$ and a specification automaton $\mathbf{S} = (X, \Sigma, \xi, x_0, X_m)$. We build a projection automaton for each controller ($i \in I$), denoted by $\mathbf{R}_i$, from $\mathbf{G}||\mathbf{S} = (Y, \Sigma, \eta, y_0, Y_m)$. To calculate the states of this automaton, we use the algorithm for subset construction [28], which is based on the idea of ε-reach$_i$. The ε-reach$_i$ of a state $y \in Y$ is defined as follows:

$$\varepsilon - \text{reach}_i(y) := \{y' \in Y \mid \eta(y,\sigma) = y' \text{ and } \sigma \notin \Sigma_{o,i}\}.$$

This can be extended to a set of states: $\varepsilon\text{-reach}_i(Y') = \bigcup_{y' \in Y'} \varepsilon\text{-reach}_i(y')$. Formally, $\mathbf{R_i} = (R_i, \Sigma_{o,i}, \rho_i, r_{0,i}, R_{mi})$, where $R_i \subseteq Pwr(Y)$ and $r_{0,i} = \varepsilon\text{-reach}_i(y_0)$.

To describe the transition function ρ_i, we also need to calculate all states that can be reached in one step via a transition of event σ for a given set of states. Thus, for

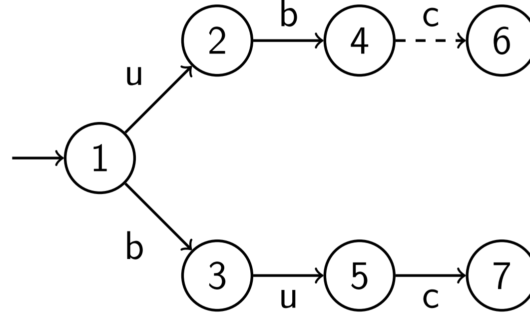


Figure 3.1: Plant and Specification Automaton

Figure 3.2: (a) Projection Automaton 1, (b) Projection Automaton 2

$Y' \in Pwr(Y)$:

$$step_\sigma(Y') = \{y' \in Y \mid (\exists y \in Y') \text{ such that } \eta(y,\sigma) = y'\}.$$

Then we can formally define the transition function, where $Y' \in Pwr(Y)$:

$$\rho_i(Y',\sigma) \;\; := \;\; \begin{cases} \varepsilon - \text{reach}_i(step_\sigma(Y')), & \text{if } \sigma \in \Sigma_{o,i}; \\ Y', & \text{otherwise.} \end{cases}$$

Example 3.1.1. *Figure 3.1, shows a plant automaton and the specification automaton.* $\boldsymbol{G}$= *(*Q,Σ,δ,q_0,Q_m*) is the plant automaton represented by the collection of all states and transitions, whereas* $\boldsymbol{S}$ = *(*X,Σ,ξ,x_0,X_m*) is the specification automaton with solid-line transitions and the states connected by them. We have two controllers, i = 1, 2, such that* $\Sigma_{o,1} = \Sigma_{c,1} = \{b, c\}$ *and* $\Sigma_{o,2} = \Sigma_{c,2} = \{u, c\}$.

The projection automata for the two controllers are given in Figure 3.2. According

to Example 3.1.1, controller 2 observes event u, hence it does not know if the plant is in state 1 or state 3. If no event has yet occurred then the plant is actually in state 1; however, if event b has occurred, then also controller 1 has no idea about the the current plant state because of its inability to observe event b. The occurrence of event u can provide a clear picture of the true plant state. Assume the plant is at state 1 and event u takes place, now the true plant state would be state 2. But, if the plant was at state 3 because of event b and now event u takes place then the true state of the plant will be state 5.

While there have been several state-based structures proposed to verify co-observability, (e.g., *Monitoring Automaton* [32], and *Estimator Structure* [3]), they are largely variations of each other. As a result we present a single summary structure that captures the core characteristics of the verifiers noted above. For the verification of co-observability the verifier we examine takes the synchronous product, $\mathbf{G}\|\ \mathbf{S}$, and the projection automaton R_1, R_2.

Given a plant $\mathbf{G}$= (Q, Σ, δ, q_0, Q_m) and a specification automaton $\mathbf{S}$ = (X, Σ, ξ, x_0, X_m). Let $\mathbf{G}\|\mathbf{S} = (Y, \Sigma, \eta, y_0, Y_m)$ be the synchronous product of the plant and the specification. Let $\Sigma_{o,i} \subseteq \Sigma$, $\Sigma_{c,i} \subseteq \Sigma$ be the set of observable and controllable events of the controllers. For a system with two controllers, where i = 1,2, let $\mathbf{R_1} = (R_1, \Sigma_{o,1}, \rho_1, r_{0,1}, R_{m1})$ and $\mathbf{R_2} = (R_2, \Sigma_{o,2}, \rho_2, r_{0,2}, R_{m2})$ be the respective projection automaton, as defined in Section 3.1, built as a result of the controller's view of the specification.

Definition 3.1.1. *A verifier, $\mathcal{M} = (M, \Sigma, m_0, \theta, M_m)$, for two controllers is defined such that $M \subseteq (Y \times R_1 \times R_2)$ is the set of states, m_0 is the initial state and M_m is the*

set of marked states.

The transition is defined, for $(y, r_1, r_2) \in M$ and for $\sigma \in \Sigma$, as follows:

$$\theta((y, r_1, r_2), \sigma) := \begin{cases} (\eta(y,\sigma), \rho_1(r_1,\sigma), \rho_2(r_2,\sigma)), & \text{if } \sigma \in \Sigma_{o,1} \cap \Sigma_{o,2} \text{ and} \\ & \quad \eta(y,\sigma)!, \rho_1(r_1,\sigma)!, \rho_2(r_2,\sigma))!; \\ (\eta(y,\sigma), \rho_1(r_1,\sigma), r_2), & \text{if } \sigma \in \Sigma_{o,1} \backslash \Sigma_{o,2} \text{ and} \\ & \quad \eta(y,\sigma)!, \rho_1(r_1,\sigma)!; \\ (\eta(y,\sigma), r_1, \rho_2(r_2,\sigma)), & \text{if } \sigma \in \Sigma_{o,2} \cap \Sigma_{o,1} \text{ and} \\ & \quad \eta(y,\sigma)!, \rho_2(r_2,\sigma))!; \\ (\eta(y,\sigma), r_1, r_2), & \text{if } \sigma \in \Sigma \backslash \Sigma_{o,1} \backslash \Sigma_{o,2} \text{ and} \\ & \quad \eta(y,\sigma)!; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

The verifier for our ongoing example is shown in Figure 3.3. In particular, note that a transition is defined from state $(3, \{3,4,5\}, \{1,3\})$ to state $(5, \{3,4,5\}, \{2,4,5\})$ via event u because there is a transition of u in $\mathbf{G}$ from state 3 to state 5, but since u is unobservable to controller 1, there is no change of state in R_1; however, since u is observable to controller 2, and there is such a transition defined from state $\{1,3\}$ in R_2, there is a corresponding state change to state $\{2,4,5\}$.

A violation of co-observability is encoded in $\mathcal{M}$ when it contains a state $(y, r_1, r_2) \in \mathcal{M}$ and some $\sigma \in \Sigma_c$ where $\theta((y, r_1, r_2), \sigma)!$ such that σ must be disabled at y w.r.t. $\mathbf{G}||\mathbf{S}$, and there exists some $y' \in r_1$ and $y'' \in r_2$ such that σ must be enabled at y' and

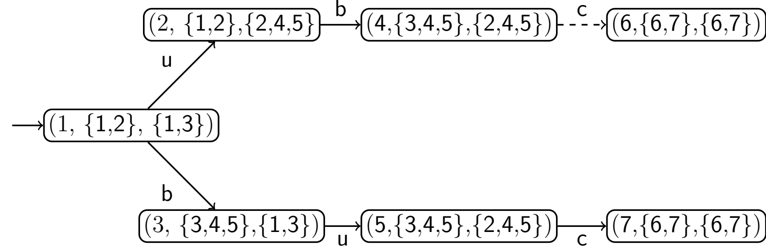


Figure 3.3: Verifier

y'' w.r.t. $\mathbf{G}||\mathbf{S}$. Thus, none of the controllers that have the ability to control σ can take a definitive control decision regarding σ based on the set of states it considers possible for the current plant state. In our example, a violation of co-observability occurs at state $(4, \{3, 4, 5\}, \{2, 4, 5\})$: c is allowed to occur in the plant, but not in the specification. Furthermore, controller 1 cannot distinguish states 3,4,5, and at state 5 c must be enabled: so it cannot take a disablement decision and issues an enablement decision. Similarly, controller 2 cannot distinguish between states 2,4,5 and has an identical uncertainty for event c, resulting in a local enablement decision. Therefore, the overall control decision will be to enable c, which violates the specification.

3.2 Towards State-based Co-observability

Recently, a state-based definition of co-observability was introduced [13]. Instead of tracking observable event labels, co-observability is now additionally defined in terms of transitions in the plant. For example, a controller's observations are now denoted by $\delta_{o,i} \subseteq \delta$ and a transition $(q, \sigma, q') \in \delta_{o,i}$ if $\sigma \in \Sigma_{o,i}$. To facilitate this new perspective on observations, a new natural projection $P_{\mathbf{G}} : Q \times \Sigma \to \Sigma \cup \{\varepsilon\}$ is defined. For

$\sigma \in \Sigma, q \in Q, i \in I$

$$P_{\mathbf{G},i}(q,\varepsilon) := \varepsilon,$$
$$P_{\mathbf{G},i}(q,\sigma) := \begin{cases} \sigma, & \text{if } (q,\sigma,\delta(\sigma,q) \in \delta_{o,i}); \\ \varepsilon, & \text{otherwise.} \end{cases}$$

This projection can be extended to strings. For $s \in \Sigma^*, \sigma \in \Sigma, q \in Q$

$$P_{\mathbf{G},i}(q,s\sigma) = P_{\mathbf{G},i}(q,s)P_{\mathbf{G},i}(\delta(q,s),\sigma).$$

Definition 3.2.1. *Given a plant $\boldsymbol{G}$ and a specification automaton $\boldsymbol{S}$, sets $\Sigma_{c,1}$, $\Sigma_{c,2} \subseteq \Sigma$, state-based projections $P_{\boldsymbol{G},1}$, $P_{\boldsymbol{G},2}$, the language $L(\boldsymbol{S})$ is state-based co-observable w.r.t. $\boldsymbol{G}$, $P_{\boldsymbol{G},1}$, $P_{\boldsymbol{G},2}$ if for all s, s', $s'' \in \Sigma^*$ such that $P_{\boldsymbol{G},1}(q_0,s) = P_{\boldsymbol{G},1}(q_0,s')$ and $P_{\boldsymbol{G},2}(q_0,s) = P_{\boldsymbol{G},1}(q_0,s'')$,*

$(\forall\sigma \in \Sigma_{c,1} \cap \Sigma_{c,2}) s \in L(\boldsymbol{S}) \wedge s\sigma \in L(\boldsymbol{G}) \wedge s'\sigma, s''\sigma \in L(\boldsymbol{S}) \Rightarrow s\sigma \in L(\boldsymbol{S})$ ***conjunct1;***

$(\forall\sigma \in \Sigma_{c,1} \backslash \Sigma_{c,2}) s \in L(\boldsymbol{S}) \wedge s\sigma \in L(\boldsymbol{G}) \wedge s'\sigma \in L(\boldsymbol{S}) \Rightarrow s\sigma \in L(\boldsymbol{S})$ ***conjunct 2;***

$(\forall\sigma \in \Sigma_{c,2} \backslash \Sigma_{c,1}) s \in L(\boldsymbol{S}) \wedge s\sigma \in L(\boldsymbol{G}) \wedge s''\sigma \in L(\boldsymbol{S}) \Rightarrow s\sigma \in L(\boldsymbol{S})$ ***conjunct 3.***

Note that this definition is only defined for two controllers and the authors suggest that the complexity of verifying this property for two controllers is the reason for this restriction. To that end, the authors propose the following non-deterministic structure to verify their state-based version of co-observability:

$$\mathcal{M} = \mathbf{S} \times \mathbf{S} \times (\mathbf{S} \times \mathbf{G}),$$

where the state set is $X \times X \times X \times Q \cup \{\mathsf{dump}\}$ and the other components are defined

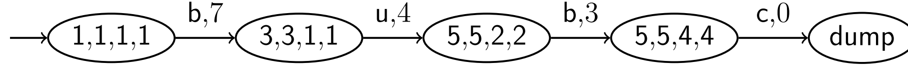


Figure 3.4: State-based Verifier

as usual with the exception of the classification of transitions. The idea here is to simultaneously track a sequence s' in the specification for controller 1, a sequence s'' in the specification for controller 2 and a sequence s in the plant and the specification, corresponding to the three conjuncts identified above. Because there is independence in the tracking of the three sequences, the authors then distinguish transitions in $\mathcal{M}$ depending on the synchronization of the transitions. For instance, if the transition in $\mathcal{M}$ causes only the first automaton component to change state, this is defined as a *type 1* transition. There are eight transition types in all, as summarized in Table 3.1. Of note is type zero, which captures a violation of state-based co-observability: a transition of σ is defined for controller 1 and for controller 2 within their tracking of sequences in the specification, but is outside the specification with respect to the plant. If there is a path from the initial state of $\mathcal{M}$ to the dump state, the co-observability does not hold. Such a situation for our running example is shown in Figure 3.4, where only one path in $\mathcal{M}$ is illustrated. Note that this path corresponds to s' = bu, s'' = bu and s = ub, and since c is not defined in the specification after ub, this set of sequences corresponds to a transition of type 0 for event c, and a violation of state-based co-observability.

As the number of controllers increases, one can see that the number of transition types becomes exponentially unmanageable, leaving this style of verification for state-based co-observability computationally infeasible.

Table 3.1: Transition types in $\mathcal{M}$ for $\sigma \in \Sigma$ (indicated transitions made only if defined)

Transition Type	Initial State Configuration	Final State Configuration
σ,1	(x_1, x_2, x_3, q_4)	$(\xi(x_1,\sigma), x_2, x_3, q_4)$
σ,2	(x_1, x_2, x_3, q_4)	$(x_1, \xi(x_2,\sigma), x_3, q_4)$
σ,3	(x_1, x_2, x_3, q_4)	$(x_1, x_2, \xi(x_3,\sigma), \delta(q_4,\sigma))$
σ,4	(x_1, x_2, x_3, q_4)	$(\xi(x_1,\sigma), \xi(x_2,\sigma), \xi(x_3,\sigma), \delta(q_4,\sigma))$
σ,5	(x_1, x_2, x_3, q_4)	$(x_1, \xi(x_2,\sigma), \xi(x_3,\sigma), \delta(q_4,\sigma))$
σ,6	(x_1, x_2, x_3, q_4)	$(\xi(x_1,\sigma), x_2, \xi(x_3,\sigma), \delta(q_4,\sigma))$
σ,7	(x_1, x_2, x_3, q_4)	$(\xi(x_1,\sigma), \xi(x_2,\sigma), x_3, q_4)$
σ,0	(x_1, x_2, x_3, q_4)	dump, if $\sigma \in \Sigma_{c,1} \cap \Sigma_{c,2}$ and only $\xi(x_1,\sigma)!$, $\xi(x_2,\sigma)!$,$\delta(q_4,\sigma)!$.

3.3 State-Based Definition of Co-observability

In this section, we present a new state-based definition of co-observability and prove that it implies the language-based definition. Additionally, we present a counter example to show that the language-based definition does not, in general, imply the state-based definition.

3.3.1 Basic Definitions

Before we introduce our definition of state-based co-observability, we first need to introduce a few maps and relations that we will need to define SB co-observability.

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ and $\mathbf{S} = (X, \Sigma, \xi, x_0, X_m)$ be an arbitrary plant and a specification automaton such that both are defined over Σ. Let $\mathbf{G} \parallel \mathbf{S}$ be the synchronous product of $\mathbf{G}$ and $\mathbf{S}$ such that $\mathbf{G} \parallel \mathbf{S} = (Y, \Sigma, \eta, y_0, Y_m)$. Let $I = \{1,2,..,n\}$ be the index set for the controllers. For each controller we consider the observable

set $\Sigma_{o,i} \subseteq \Sigma_o$ and the controllable set $\Sigma_{c,i} \subseteq \Sigma_c$ where $i \in I$. Also we define $I_c(\sigma)$ to be the set of all controllers that control event $\sigma \in \Sigma_c$, i.e., $I_c(\sigma) = \{i \in I | \sigma \in \Sigma_{c,i}\}$. Let Y_r be the set of reachable states for **G**||**S**.

We define the map $Y_{disable} : \Sigma_c \to Pwr(Y)$, for any $\sigma \in \Sigma_c$ where $Y_{disable}(\sigma)$ maps to the set of reachable states in **G**||**S** with an outgoing σ transition in **G**, but no σ transition in **S**.

Definition 3.3.1. *For a plant* $\boldsymbol{G}$ *and a specification automaton* $\boldsymbol{S}$*, we define the map* $Y_{disable} : \Sigma_c \to Pwr(Y)$ *for* $\sigma \in \Sigma_c$ *as follows:*

$$Y_{disable}(\sigma) := \{y = (q,x) \in Y_r |\ \delta(q,\sigma)! \wedge \neg\xi(x,\sigma)!\}$$

The set $Y_{disable}(\sigma)$ thus identifies the states in **G** || **S** where $\sigma \in \Sigma_c$ is possible in the plant but disabled with respect to the specification.

We now define the map $Y_{enable} : \Sigma_c \to Pwr(Y)$, for any $\sigma \in \Sigma_c$ where $Y_{enable}(\sigma)$ maps σ to the set of reachable states in **G**||**S** with an outgoing transition in both **G** and **S**.

Definition 3.3.2. *For a plant* $\boldsymbol{G}$ *and a specification automaton* $\boldsymbol{S}$*, we define the map* $Y_{enable} : \Sigma_c \to Pwr(Y)$ *for* $\sigma \in \Sigma_c$ *as follows:*

$$Y_{enable}(\sigma) := \{y = (q,x) \in Y_r |\ \delta(q,\sigma)! \wedge \xi(x,\sigma)!\}$$

The set $Y_{enable}(\sigma)$ thus identifies the states in **G** || **S** where $\sigma \in \Sigma_c$ is possible in the plant and is enabled by the specification.

We now introduce a binary relation on the state-space Y of **G**||**S**, with respect to a subset $\Omega \subseteq \Sigma$. The relation says that two states are indistinguishable if each

state can be reached from the initial state by strings that are equal under the natural projection $P_\Omega : \Sigma^* \to \Omega^*$.

Definition 3.3.3. *Let $\Omega \subseteq \Sigma$ and let $P_\Omega : \Sigma^* \to \Omega^*$ be a natural projection. For $\boldsymbol{G} \|$ $\boldsymbol{S}$, we define the Ω indistinguishability relation to be:*

$$(\forall y_1, y_2 \in Y),$$
$$y_1 \sim_\Omega y_2 \Leftrightarrow (\exists s_1, s_2 \in \Sigma^*)(P_\Omega(s_1) = P_\Omega(s_2)) \wedge (\eta(y_0, s_1) = y_1) \wedge (\eta(y_0, s_2) = y_2).$$

For states y_1, $y_2 \in Y$ if $y_1 \sim_\Omega y_2$, we say that y_1 is indistinguishable from y_2 w.r.t Ω.

This relation was designed to match Lemmas 4.2.1 and 4.2.2 from [26], which we will discuss in Section 4.2.7. The lemmas allow us to relate $\sim_\Omega$ to the observers that we use in our algorithms in Chapter 4. It is easy to see that $\sim_\Omega$ is symmetric. If **G** $\|$ **S** is also reachable, then $\sim_\Omega$ is reflexive.

We need an easy means to refer to the subset of states that are indistinguishable from some state $y \in Y$ with respect to $\Omega \subseteq \Sigma$.

Definition 3.3.4. *Let $\Omega \subseteq \Sigma$ and let $P_\Omega : \Sigma^* \to \Omega^*$ be the natural projection. For $\boldsymbol{G} \| \boldsymbol{S}$ and $y \in Y$, we define the subset $[y]_{\sim_\Omega} \subseteq Y$ to be:*

$$[y]_{\sim_\Omega} := \{y' \in Y \mid y \sim_\Omega y'\}$$

We say the subset $[y]_{\sim_\Omega}$ is the coset of y with respect to $\sim_\Omega$.

Having introduced Y_{enable}, $Y_{disable}$ and $[y]_{\sim_{\Sigma_{o,i}}}$, we can now proceed to our definition of state-based co-observability.

Definition 3.3.5. *Given maps $Y_{enable} : \Sigma_c \to Pwr(Y)$ and $Y_{disable} : \Sigma_c \to Pwr(Y)$ defined for a plant **G** and a specification **S**. Let $\Sigma_c \subseteq \Sigma$, $\Sigma_{o,i} \subseteq \Sigma$ and $\Sigma_{c,i} \subseteq \Sigma$, where $i = 1,2,....,n$. Specification **S** is SB co-observable w.r.t to **G**, P_i and $\Sigma_{c,i}$, if*

$$(\forall y \in Y)(\forall \sigma \in \Sigma_c)\ y \in Y_{disable}(\sigma) \Rightarrow (\exists i \in I_c(\sigma))\ [y]_{\sim \Sigma_{o,i}} \cap Y_{enable}(\sigma) = \varnothing$$

Essentially, the SB co-observable definition states that if a state $y = (q,x) \in Y$ of $\mathbf{G}||\mathbf{S}$ is a state where $\sigma \in \Sigma_c$ is allowed to occur in **G** but is not part of the behaviour of **S**, then there must be at least one controller $i \in I$ that can disable σ and can distinguish, w.r.t $\Sigma_{o,i}$, y from all states $y' = (q', x') \in Y$ where σ is possible in both **G** and **S**. We note that even if state y is unreachable, this will not affect the result as this would imply that $[y]_{\sim \Omega} = \varnothing$, which easily follows from Definition 3.3.3.

3.3.2 State-based Implies Language-based Definition

We present a theorem that states that the SB co-observability definition implies the language based definition. If this is true, then we can synthesize decentralized controllers that correctly solve the decentralized control problem.

Theorem 3.3.1. *Consider plant $\boldsymbol{G} = (Q, \Sigma, \delta, q_0, Q_m)$ and a specification automaton $\boldsymbol{S} = (X, \Sigma, \xi, x_0, X_m)$, where $\boldsymbol{G}||\boldsymbol{S} = (Y, \Sigma, \eta, y_0, Y_m)$ is their synchronous product. For $i = \{1,2,...,n\}$, sets Σ_c, $\Sigma_{c,i} \subseteq \Sigma$ are the sets of controllable events and Σ_o, $\Sigma_{o,i} \subseteq \Sigma$ is the set of observable events. Let $P_i : \Sigma^* \to \Sigma_{o,i}^*$ be a natural projection for $i = 1, 2, ..., n$. Consider maps, $Y_{disable} : \Sigma_c \to Pwr(Y)$ and $Y_{enable} : \Sigma_c \to Pwr(Y)$ defined over $\boldsymbol{G}||\boldsymbol{S}$. For all $y \in Y$ let $[y]_{\sim \Sigma_{o,i}}$ be the coset of y with respect to $\sim_{\Sigma_{o,i}}$. If the*

*specification **S** is SB co-observable w.r.t **G**, P_i and $\Sigma_{c,i}$, then $L(\mathbf{G}\|\mathbf{S})$ is co-observable w.r.t $L(\mathbf{G})$, P_i and $\Sigma_{c,i}$:*

Proof. We prove by contradiction.

Assume that **S** is SB co-observable w.r.t **G**, P_i and $\Sigma_{c,i}$:

$$(\forall y \in Y)(\forall \sigma \in \Sigma_c) y \in Y_{disable}(\sigma) \Rightarrow (\exists i \in I_c(\sigma))[y]_{\sim \Sigma_{o,i}} \cap Y_{enable}(\sigma) = \varnothing \quad (1)$$

and that $L(\mathbf{G}\|\mathbf{S})$ is not co-observable w.r.t $L(\mathbf{G})$, P_i and $\Sigma_{c,i}$,

$$(\exists s \in L(\mathbf{G}\|\mathbf{S}))(\exists \sigma \in \Sigma_c) s\sigma \in (L(\mathbf{G}) \backslash L(\mathbf{G}\|\mathbf{S})) \wedge (\forall i \in I_c(\sigma))$$
$$P_i^{-1}(P_i(s))\sigma \cap L(\mathbf{G}\|\mathbf{S}) \neq \varnothing \quad (2)$$

We will now show that (2) produces a contradiction in (1).

To do this, we need to show that:

$$(\exists y \in Y)(\exists \sigma \in \Sigma_c) y \in Y_{disable}(\sigma) \wedge (\forall i \in I_c(\sigma))[y]_{\sim \Sigma_{o,i}} \cap Y_{enable}(\sigma) \neq \varnothing \quad (3)$$

On examining equation (2) we conclude that there is some string $s \in L(\mathbf{G}\|\mathbf{S})$,

and an event $\sigma \in \Sigma_c$ such that: (4)

$$s\sigma \in L(\mathbf{G}) \backslash L(\mathbf{G}\|\mathbf{S}) \text{and} (\forall i \in I_c(\sigma)) P_i^{-1}(P_i(s))\sigma \cap L(\mathbf{G}\|\mathbf{S}) \neq \varnothing \quad (5)$$

Since $s \in L(\mathbf{G}\|\mathbf{S})$ by (4), we thus have $\eta(y_0, s)!$.

Let $y = \eta(y_0, s)$ and thus $y \in Y$. (6)

As $s\sigma \in (L(\mathbf{G}) \backslash L(\mathbf{G}\|\mathbf{S}))$, we have $s\sigma \in L(\mathbf{G})$ and $s\sigma \notin L(\mathbf{G}\|\mathbf{S})$.

Since **S** and **G** are both over Σ, we have $L(\mathbf{G}\|\mathbf{S}) = L(\mathbf{G}) \cap L(\mathbf{S})$.

As $s\sigma \in L(\mathbf{G})$ and $s\sigma \notin L(\mathbf{G}\|\mathbf{S}) = L(\mathbf{G}) \cap L(\mathbf{S})$, it follows that $s\sigma \notin L(\mathbf{S})$.

$\Rightarrow y \in Y_{disable}(\sigma)$ (7)

From (3) we now need to show:

$$(\forall i \in I_c(\sigma))[y]_{\sim \Sigma_{o,i}} \cap Y_{enable}(\sigma) \neq \varnothing \quad (8)$$

On examining (5) we see that $(\forall i \in I_c(\sigma))\ P_i^{-1}(P_i(s))\sigma \cap L(\mathbf{G}||\mathbf{S}) \neq \varnothing$.

Since this property is true for all $i \in I_c(\sigma)$, we can choose an arbitrary $i \in I_c(\sigma)$, and prove this implies (8), without loss of generality.

Let i be an arbitrary element of $I_c(\sigma)$. From (5) we can conclude that:

$$(\exists s' \in \Sigma^*)(s' \in P_i^{-1}(P_i(s))) \wedge (s'\sigma \in L(\mathbf{G}||\mathbf{S})) \quad (9)$$

This implies that $P_i(s') = P_i(s)$ and $s' \in L(\mathbf{G}||\mathbf{S})$. This is because of the definition of P_i and the fact that $L(\mathbf{G}||\mathbf{S})$ is prefix-closed. (10)

We also note that $s'\sigma \in L(\mathbf{G}||\mathbf{S}) = L(\mathbf{G}) \cap L(\mathbf{S})$, which implies that $s'\sigma \in L(\mathbf{S})$, and $s'\sigma \in L(\mathbf{G})$. (11)

We thus have $\eta(y_0, s')!$.

Let $y' = \eta(y_0, s')$ and thus $y' \in Y$.

By (11), we have $y' \in Y_{enable}(\sigma)$ (12)

As $P_i(s') = P_i(s)$ by (10), we have: $y \sim_{\Sigma_{o,i}} y'$

$\Rightarrow y' \in [y]_{\sim \Sigma_{o,i}}$

$\Rightarrow [y]_{\sim \Sigma_{o,i}} \wedge Y_{enable}(\sigma) \neq \varnothing$, by (12).

As i is an arbitrary element of $I_c(\sigma)$, we have:

$$(\forall i \in I_c(\sigma))\ [y]_{\sim_{\Sigma_{o,i}}} \cap Y_{enable}(\sigma) \neq \emptyset$$

This contradicts (1).

Thus, we conclude that (2) is false, i.e., L($\mathbf{G}\|\mathbf{S}$) is co-observable w.r.t $L(\mathbf{G})$ as required.

□

Corollary 3.3.1 ([37]). *Consider a plant* $\boldsymbol{G} = (Q, \Sigma, \delta, q_0, Q_m)$*, a specification* $\boldsymbol{S} = (X, \Sigma, \xi, x_0, X_m)$ *and their nonblocking synchronous product* $\boldsymbol{G}\|\boldsymbol{S}$*, where* $\Sigma_{uc} \subseteq \Sigma$ *is the set of uncontrollable events,* $\Sigma_c = \Sigma \backslash \Sigma_{uc}$ *is the set of controllable events, and* $\Sigma_o \subseteq \Sigma$ *is the set of observable events. For each site* $i \in I$*, consider the set of controllable events* $\Sigma_{c,i}$ *and the set of observable events* $\Sigma_{o,i}$*; overall,* $\cup_{i=1}^{n} \Sigma_{c,i} = \Sigma_c$ *and* $\cup_{i=1}^{n} \Sigma_{o,i} = \Sigma_o$*. Let* P_i *be the natural projection from* Σ^* *to* $\Sigma_{o,i}^*$*,* $i \in I$*. Consider also the language* $L_m(\boldsymbol{G}\|\boldsymbol{S}) \subseteq L_m(\boldsymbol{G})$*, where* $L_m(\boldsymbol{G}\|\boldsymbol{S}) \neq \emptyset$*. There exists a nonblocking decentralized supervisor* S_{dec} *for* $\boldsymbol{G}$ *such that*

$$L_m(S_{dec}/\boldsymbol{G}) = L_m(\boldsymbol{G}\|\boldsymbol{S}) \text{ and } L(S_{dec}/\boldsymbol{G}) = L(\boldsymbol{G}\|\boldsymbol{S})$$

if the following conditions hold:

1. $L_m(\boldsymbol{G}\|\boldsymbol{S})$ *is controllable w.r.t.* $L(\boldsymbol{G})$ *and* Σ_{uc}*;*
2. $\boldsymbol{S}$ *is SB co-observable w.r.t.* $\boldsymbol{G}$*,* P_i *and* $\Sigma_{c,i}, i = 1, \ldots n$*;*
3. $L_m(\boldsymbol{G}\|\boldsymbol{S})$ *is* $L_m(\boldsymbol{G})$*-closed.*

Proof Follows easily from Theorems 2.2.1 and 3.3.1.

3.3.3 Does Language-Based Imply State-Based Definition

An obvious question that arises from Theorem 3.3.1 is, does the language-based definition of co-observability imply the state-based definition?

We will show that the language-based definition of co-observability does not always imply the state-based definition, i.e., the state-based definition of co-observability is more restrictive than the language-based definition. This means that if a system is not SB co-observable, it is not always the case that the system is not language-based co-observable. We will prove this using a counter example.

Counter example: Let $\mathbf{G} = (Q,\Sigma,\delta,q_0,Q_m)$ be a plant automaton and $\mathbf{S} = (X, \Sigma, \xi, x_0, X_m)$ be a specification automaton, such that both are defined over Σ. The plant and the specification together is given in Figure 3.5. The dashed line shows that state 4 is reachable in the plant but is unreachable in the specification automaton. Let there be two controllers, $i \in \{1, 2\}$ such that $\Sigma_{o,1} = \{a1, a2\}$, $\Sigma_{o,2} = \{b1, b2\}$. Let $\Sigma_{c,1} = \Sigma_{c,2} = \{c\}$ be the set of controllable events. In this example, the only control decision to be implemented by specification is to disable c in state 2 while enabling it in state 1.

On applying the language-based definition of co-observability defined in [33] to the counter example, we observe that together, the two supervisors can make the right control decision. If $a2$ occurs then controller 1 knows to disable c, and if $b2$ occurs then controller 2 knows to disable c. As long as neither $a2$ or $b2$ occurs, it is safe to leave c enabled. Although states 1 and 2 are indistinguishable to both controllers, in cases where the two states are indistinguishable to controller 1, controller 2 can distinguish them, and vice versa.

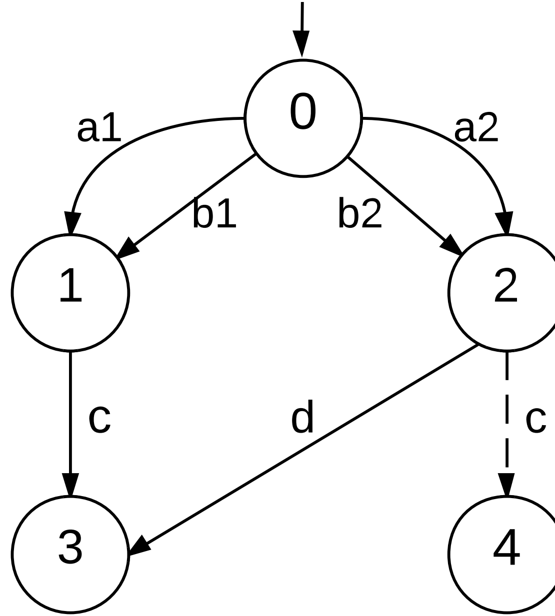


Figure 3.5: Counter example

According to the definition of co-observability given in Section 2.2.1, $K = \{a1c, b1c, a2, b2, a2d, b2d\}$, $L = \{\varepsilon, a1c, b1c, a2c, b2c, a1, b1, a2, b2, a2d, b2d\}$ and $\overline{K} = \{\varepsilon, a1, b1, a1c, b1c, a2, b2, a2d, b2d\}$. Let $s = a2$ and $\sigma = c$, then according to the definition, $a2c \in (L \backslash \overline{K})$. This implies that there is a controller $i = 1$ that can control c, such that the intersection of the prefix-closure of the legal language and inverse projection of the projection of string $a2$ is empty. Thus, $L(\mathbf{G}||\mathbf{S})$ is co-observable with respect to $L(\mathbf{G})$. Likewise, $b2$ is handled by controller $i = 2$.

Using Definition 3.3.3, we note that states 1 and 2 are indistinguishable w.r.t $\Sigma_{o,1}$ because events $b1$ and $b2$ are un-observable to controller 1. Similarly, events $a1$ and $a2$ are unobservable to controller 2, which makes states 1 and 2 indistinguishable w.r.t $\Sigma_{o,2}$. Additionally, using Definition 3.3.1 and Definition 3.3.2, we note that state 2

belongs to $Y_{disable}(c)$ and state 1 belongs to $Y_{enable}(c)$. On applying the state-based definition of co-observability, given in Definition 3.3.5,one notes that indistinguishable states 1 and 2, that belong to different $Y_{disable}(c)$ and $Y_{enable}(c)$ sets are grouped together, failing the definition of SB co-observability.

In the next chapter, after we have introduced our algorithm for SB co-observability and, given an example, we also evaluate our counter example and prove that it is not SB co-observable. For more information refer to Section 4.5.

Chapter 4

State-Based Algorithms

In this chapter we present two algorithms to verify SB co-observability.

4.1 Verifying SB Co-observability

For the verification of SB co-observability, we define two methods:

- *SubsetConstruction* [28];
- *Observer Construction* [42].

Subset Construction: The algorithm makes use of a classical method that we adopt for the verification of SB co-observability. It has also been used for the verification of the observability property. We use subset construction for the verification of SB co-observability so that it can be more easily compared to existing methods that verify co-observability. We will also perform a computational complexity analysis of this method.

Observer Construction: Here we introduce a new method for the verification of SB co-observability that is more efficient than the subset construction approach. We will perform a complexity analysis for this algorithm and compare the results to algorithms for language-based co-observability verification. We will also present an example using the observer-based algorithm.

We assume that we are given a plant $\mathbf{G} = (Q,\Sigma,\delta,q_0,Q_m)$ and a specification $\mathbf{S} = (X,\Sigma,\xi,x_0,X_m)$, for both the *subset construction* and *observer-based* algorithms.

These methods share two sub-algorithms. The first shared algorithm, called the *EnableDisableSetFind* and is defined in Section 4.2.2 and constructs the maps $Y_{disable}$ and Y_{enable} for $\mathbf{G}$ and $\mathbf{S}$. The definitions for these maps are given in Section 3.3.1.

The second shared algorithm is called the *ConstructSynchProductAndHiding* and defined in Section 4.2.3. For a given controller $i \in I = \{1,2,..,n\}$, we construct an automaton $\mathbf{H} = (Y,\Sigma,\eta,y_0,Y_m)$, which is equivalent to $\mathbf{G}||\mathbf{S}$ with the events $\Sigma \backslash \Sigma_{o,i}$ (i.e., the events that are unobservable by controller i) hidden. Essentially, we construct $\mathbf{H} = \mathbf{G}||\mathbf{S}$, and then we replace every $\sigma \in \Sigma \backslash \Sigma_{o,i}$ transition in $\mathbf{G}||\mathbf{S}$ with a new event $\tau \notin \Sigma$. Typically, this makes $\mathbf{H}$ non-deterministic.

Our algorithms require that $\mathbf{G}$ and $\mathbf{S}$ be deterministic. An automaton is deterministic if it has a single initial state and, at each state, there is at most one transition defined, leaving that state for a given $\sigma \in \Sigma$. Since $\mathbf{G}$ and $\mathbf{S}$ are deterministic, it follows that $\mathbf{G}||\mathbf{S}$ is also deterministic.

Typically, the hiding process makes $\mathbf{H}$ non-deterministic as we could end up with multiple τ transitions leaving a given state. It is easy to see that for $\sigma \in \Sigma$, we will

never have more than one σ transition at a given state.

The main difference between **G**||**S** and **H** is that, in **H**, η becomes the next state relation defined as $\eta \subseteq Y \times \Sigma_\tau \times Y$, where for $\Sigma' \subseteq \Sigma$, we use the notation $\Sigma_\tau{}'$ to mean $\Sigma_\tau{}' = \Sigma' \dot{\cup} \{\tau\}$. To keep our algorithms simple, we use the notation $\eta(y, \sigma) = y'$, for $\sigma \in \Sigma$ and $y \in Y$, as we normally would for deterministic automaton. We will assume, though, that $\eta(y, \tau) \subseteq Y$, i.e., returns a subset of state set Y, possibly the empty set. We also use $\eta(y, \tau)!$ to mean $\eta(y, \tau) \neq \emptyset$.

4.2 Algorithms

In this section we present our subset construction-based SB co-observability algorithm and our observer-based SB co-observability algorithm.

To make things easier to understand and to avoid duplication, we have broken each algorithm into several sub-algorithms. The main subset construction algorithm is presented in Algorithm 1 and it calls Algorithm 2 - 4. The main observer-based algorithm is presented in Algorithm 5 and it calls Algorithms 2, 3, and 6.

4.2.1 Is SB Co-observable Using Subset Construction

The algorithm determines whether specification **S** is SB Co-observable with respect to plant **G**, P_i and $\Sigma_{c,i}$, $i = 1, 2, ..., n$.

The subroutines *EnableDisableSetFind*, *ConstructSynchProductAndHiding* and *SubsetConstruction* have been used as a part of *IsSBCo-observableUsingSubsetConstruction* algorithm.

The algorithm makes use of the following variables:

I: The set $I := \{1,2,..,n\}$ of controllers where n is the total number of controllers.

Σ_c : Set of controllable events such that $\Sigma_c \subseteq \Sigma$.

$\Sigma_{o,i}$: The set of observable events $\Sigma_{o,i} \subseteq \Sigma$, for each controller $i \in I$.

$\mathbf{H}$: The automaton based on $\mathbf{G}||\mathbf{S}$, with events $\Sigma \backslash \Sigma_{o,i}$ hidden.

$Y_{disable}$: As defined in Definition 3.3.1 in Section 3.3.1, is a map such that $Y_{disable}$: $\Sigma_c \rightarrow Pwr(Y)$.

Y_{enable} : As defined in Definition 3.3.2 in Section 3.3.1, is a map such that Y_{enable}: $\Sigma_c \rightarrow Pwr(Y)$.

$Y_{tempDisable}$: The map $Y_{tempDisable} : \Sigma_c \rightarrow Pwr(Y)$ is a map such that for each $\sigma \in \Sigma_c$, $Y_{tempDisable}(\sigma)$ reports the unhandled $Y_{disable}(\sigma)$ states that are returned from the procedure *SubsetConstruction*, so that the subsequent controllers have to handle only the remaining problem states.

pass, skipIt : Boolean variables used to test loop termination.

Algorithm 1:

1: **procedure** IsSBCo-observableUsingSubsetConstruction($\mathbf{G}, \mathbf{S}$)
2: *EnableDisableSetFind*(Y_{enable}, $Y_{disable}$, $\mathbf{G}$, $\mathbf{S}$)
3: pass $\longleftarrow$ True
4: **for** $\sigma \in \Sigma_c$ **do**
5: **if** ($Y_{disable}(\sigma) \neq \emptyset$) **then**
6: pass $\longleftarrow$ False
7: **end if**

```
8:      end for
9:      if (pass) then
10:         return True
11:     end if
12:     for i ∈ I do
13:         skipIt ⟵ True
14:         for σ ∈ Σ_c do
15:             if ((i ∈ I_c(σ)) ∧ (Y_disable(σ) ≠ ∅)) then
16:                 skipIt ⟵ False
17:             end if
18:         end for
19:         if (!skipIt) then
20:             H ⟵ ConstructSynchProductAndHiding(G, S, Σ_{o,i})
21:             Y_tempDisable ⟵ SubsetConstruction(H, Y_enable, Y_disable, Σ_{o,i}, i)
22:             pass ⟵ True
23:             for σ ∈ Σ_c do
24:                 if (Y_tempDisable(σ) ≠ ∅) then
25:                     pass ⟵ False
26:                 end if
27:                 Y_disable(σ) ⟵ Y_tempDisable(σ)
28:             end for
29:             if (pass) then
30:                 return True
31:             end if
```

32: **end if**

33: **end for**

34: **return** False

35: **end procedure**

Lines 4-11 check for the presence of states in $Y_{disable}(\sigma)$, for every $\sigma \in \Sigma_c$. Absence of any state means that there are no problem states that are need to be handled by the controllers. Lines 12-33 process each controller $i \in I$ in turn. Lines 14-18, make sure that controller i can control at least one $\sigma \in \Sigma_c$ so that a disable state can be taken care of. If the above mentioned conditions are met, then we construct the **H** automaton and perform the subset construction test. Otherwise, we skip the current controller and check the same condition for the remaining controllers.

For a particular $\sigma \in \Sigma_c$, $Y_{tempDisable}(\sigma)$ is checked for states after every time a controller goes through the procedure *SubsetConstruction* (Line 24). If $Y_{tempDisable}(\sigma) = \varnothing$ for every $\sigma \in \Sigma_c$, we exit the loop and return *True*. However, if $Y_{tempDisable}(\sigma) \neq \varnothing$ for at least one $\sigma \in \Sigma_c$, it implies that procedure *SubsetConstruction* has returned some subset of $Y_{disable}(\sigma)$ states that could not be handled by the active controller. These remaining states are then assigned to $Y_{disable}(\sigma)$ (Line 27), which are then handled by subsequent controllers.

4.2.2 Enable Disable Set Find

This algorithm constructs the maps $Y_{disable}$ and $Y_{disable}$ (see Section 3.3.1) for **G** and **S**.

The algorithm makes use of the following variables.

q_0 : Initial state of plant **G**.

x_0 : Initial state of specification **S**.

y_0 : Initial state of **G**||**S**.

Y_{fnd} : The set $Y_{fnd} \subseteq Y$ holds the states that have been found already by the algorithm.

Y_{pend} : The set $Y_{pend} \subseteq Y$ holds the states that are waiting to be processed by the algorithm.

δ : Next state function for plant **G**

ξ : Next state function for specification automaton **S**

Algorithm 2:

1: **procedure** EnableDisableSetFind(Y_{enable}, $Y_{disable}$, **G**, **S**)
2: **for** $\sigma \in \Sigma_c$ **do**
3: $Y_{disable}(\sigma) \longleftarrow \varnothing$
4: $Y_{enable}(\sigma) \longleftarrow \varnothing$
5: **end for**
6: $y_0 \longleftarrow (q_0, x_0)$
7: $Y_{fnd} \longleftarrow \{y_0\}$
8: $Y_{pend} \longleftarrow \{y_0\}$
9: **while** $(Y_{pend} \neq \varnothing)$ **do**
10: Select $y = (q, x) \in Y_{pend}$
11: $Y_{pend} \longleftarrow Y_{pend} \setminus \{y\}$
12: **for** $(\sigma \in \Sigma)$ **do**
13: **if** $(\delta(q,\sigma)!)$ **then**
14: **if** $(\neg(\xi(x,\sigma)!))$ **then**

15: **if** $(\sigma \in \Sigma_c)$ **then**
16: $Y_{disable}(\sigma) \longleftarrow Y_{disable}(\sigma) \cup \{y\}$
17: **end if**
18: **else**
19: **if** $(\sigma \in \Sigma_c)$ **then**
20: $Y_{enable}(\sigma) \longleftarrow Y_{enable}(\sigma) \cup \{y\}$
21: **end if**
22: $y' \longleftarrow (\delta(q,\sigma), \xi(x,\sigma))$
23: **if** $(y' \notin Y_{fnd})$ **then**
24: $Y_{fnd} \longleftarrow Y_{fnd} \cup \{y'\}$
25: $Y_{pend} \longleftarrow Y_{pend} \cup \{y'\}$
26: **end if**
27: **end if**
28: **end if**
29: **end for**
30: **end while**
31: **end procedure**

In Lines 12-29, a reachability search is performed to find all the set of states that can be reached by the events that are possible in $\mathbf{G}||\mathbf{S}$. As the reachability search is performed, we focus on finding the states that belong to sets, $Y_{enable}(\sigma)$ and $Y_{disable}(\sigma)$, for each $\sigma \in \Sigma_c$. The checks in Lines 13-15 place the states in $Y_{disable}(\sigma)$. The checks at Line 13 and 19 place the states in $Y_{enable}(\sigma)$.

4.2.3 Construct Synch Product and Hiding

This algorithm constructs the synchronous product $\mathbf{G}||\mathbf{S}$, of the plant $\mathbf{G}$ and specification automaton $\mathbf{S}$, both defined over Σ. As $\mathbf{G}||\mathbf{S}$ is being constructed, we also replace all unobservable events ($\sigma \in \Sigma \backslash \Sigma_{o,i}$) with the special event $\tau \notin \Sigma$. We label the resulting automaton with $\mathbf{H}$ which may be non-deterministic (see Section 2.1 for more information). The process of replacing the unobservable events with τ is called "hiding". This algorithm is called every time a new controller is evaluated.

The algorithm makes use of the following variables:

Y : State set of $\mathbf{H}$.

Y_m : The set $\mathrm{Y}_m \subseteq Y$ is the set of marked states of $\mathbf{H}$.

Y_{rch} : The set $Y_{rch} \subseteq Y$ contains the states that have already been encountered by the algorithm. We maintain this set so that the states that have been processed once do not get processed again.

Y_{pend} : The set $Y_{pend} \subseteq Y$ contains the states that have been encountered but not yet processed.

η : Next-state relation for $\mathbf{H}$ such that $\eta \subseteq \mathrm{Y} \times \Sigma_\tau \times \mathrm{Y}$, where we use the notation $\Sigma_\tau{}' = \Sigma' \dot{\cup} \{\tau\}$, for set $\Sigma' \subseteq \Sigma$.

q_0 : Initial state of plant $\mathbf{G}$.

x_0 : Initial state of specification $\mathbf{S}$.

Q_m : The set $\mathrm{Q}_m \subseteq Q$ is the set of marked states of $\mathbf{G}$.

X_m : The set $\mathrm{X}_m \subseteq X$ is the set of marked states of $\mathbf{S}$.

Algorithm 3:

```
1: procedure ConstructSynchProductAndHiding(G, S, H, Σ_{o,i})
2:     Y_m ⟵ ∅
3:     η ⟵∅
4:     y_0⟵(q_0,x_0)
5:     Y_rch ⟵ {y_0}
6:     Y_pend ⟵ {y_0}
7:     while (Y_pend ≠ ∅) do
8:         Select y = (q, x) ∈ Y_pend
9:         Y_pend ⟵ Y_pend \ {y}
10:        if ((q∈Q_m) ∧ (x∈X_m)) then
11:            Y_m ⟵ Y_m ∪ {y}
12:        end if
13:        for σ∈Σ do
14:            if (δ(q,σ)! ∧ ξ(x,σ)!) then
15:                y′ ⟵ (δ(q,σ), ξ(x,σ))
16:                if (σ∈Σ_{o,i}) then
17:                    η ⟵ η ∪ {(y,σ,y′)}
18:                else
19:                    η ⟵ η ∪ {(y,τ,y′)}
20:                end if
21:                if (y′ ∉ Y_rch) then
22:                    Y_rch ⟵ Y_rch ∪ {y′}
23:                    Y_pend ⟵ Y_pend ∪ {y′}
```

24: **end if**

25: **end if**

26: **end for**

27: **end while**

28: $Y \longleftarrow Y_{rch}$

29: **return** $(Y, \Sigma, \eta, y_0, Y_m)$

30: **end procedure**

Lines 14-15 determine our next state y', reachable by some $\sigma \in \Sigma$. For Lines 16-20, if the event σ does not belong to the observable event set of the controller $(\Sigma \backslash \Sigma_{o,i})$, then it is replaced by a silent event τ, else it remains the same. On Line 29, we return the automaton **H**.

4.2.4 Subset Construction

The algorithm operates on the non-deterministic automaton **H**, which is described in Section 2.1. This algorithm evaluates the state-space and next-state relation of **H** to do subset construction. The algorithm groups states into subsets such that for any two states, they can be reached from the initial state by strings $s_1, s_2 \in \Sigma_\tau^*$ respectively, and $P_\tau(s_1) = P_\tau(s_2)$, where $P_\tau : \Sigma_\tau^* \rightarrow \Sigma^*$ is a natural projection.

Once the subsets are constructed, the algorithm checks that for each $\sigma \in \Sigma_c$, a state from $Y_{disable}(\sigma)$ is never part of a subset that also contains a state from $Y_{enable}(\sigma)$. If this occurs, it means controller $i \in I$ can not distinguish between the states using the observable events, $\Sigma_{o,i}$. These $Y_{disable}(\sigma)$ states that fail are returned via $Y_{tmpDisable}(\sigma)$ so that another controller can be tested to see if it can handle these states.

The algorithm uses the following variables:

$Pwr(Y)$: It is the set of all subsets of Y.

$Y_{incSubsFound}$: The set of subsets having states that have been encountered before but the algorithms have not yet done a τ-closure on them. We have $Y_{incSubsFound} \subseteq Pwr(Y)$

$Y_{pending}$: The set of subsets that have been found but have not yet been processed. We have $Y_{pending} \subseteq Pwr(Y)$

$Y_{compSubsFound}$: It is the set of subsets that have been encountered and the algorithm has already done a τ-closure on them. We have $Y_{compSubsFound} \subseteq Pwr(Y)$.

$Y_{tmpDisable}$: The map $Y_{tmpDisable}$: $\Sigma_c \rightarrow Pwr(Y)$ takes each $\sigma \in \Sigma_c$, and reports the remaining $Y_{disable}(\sigma)$ states which, at some point, got paired with $Y_{enable}(\sigma)$ states in a subset. These are the states that are returned so that the subsequent controllers can be tested to see if they can handle these states.

$Y_{loopPend}$: Set $Y_{loopPend} \subseteq Y$ contains the states waiting to be processed by the τ-closure operation of a specific subsystem.

Y_{tmp1}: Set $Y_{tmp1} \subseteq Y$ is used as temporary storage.

$Y_{loopFound}$: Set $Y_{loopFound} \subseteq Y$ stores states encountered during the τ-closure operation of a specific subset.

η: Next-state relation for **H** such that $\eta \subseteq Y \times \Sigma_\tau \times Y$, where $\Sigma'_\tau := \Sigma' \cup \{\tau\}$ for set $\Sigma' \subseteq \Sigma$, and $\tau \notin \Sigma$.

NOTE: For next-state relation η of **H**, when $\sigma' \in \Sigma$, η will return a single state but,

for $\tau \notin \Sigma$, η will always return a subset $Y' \subseteq Y$, possibly the empty subset. Also, for τ only, we consider $\eta(y,\tau)!$ to mean that $\eta(y,\tau) \neq \varnothing$, otherwise we use the standard interpretation for a partial function. This is done to keep the notation simple in the algorithm.

Algorithm 4:

1: **procedure** SUBSETCONSTRUCTION($\mathbf{H}$, Y_{enable}, $Y_{disable}$, $\Sigma_{o,i}$, i)
2: $\quad Y_{compSubsFound} \longleftarrow \varnothing$
3: $\quad Y_{tmp1} \longleftarrow \varnothing$
4: $\quad Y_{incSubsFound} \longleftarrow \{\{y_o\}\}$
5: $\quad Y_{pending} \longleftarrow \{\{y_o\}\}$
6: $\quad$ **for** $\sigma \in \Sigma_c$ **do**
7: $\quad\quad$ **if** $(i \in I_c(\sigma))$ **then**
8: $\quad\quad\quad Y_{tmpDisable}(\sigma) \longleftarrow \varnothing$
9: $\quad\quad$ **else**
10: $\quad\quad\quad Y_{tmpDisable}(\sigma) \longleftarrow Y_{disable}(\sigma)$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad$ **while** $(Y_{pending} \neq \varnothing)$ **do**
14: $\quad\quad$ Select $Y' \in Y_{pending}$
15: $\quad\quad Y_{pending} \longleftarrow Y_{pending} \setminus \{Y'\}$
16: $\quad\quad Y_{loopPend} \longleftarrow Y'$
17: $\quad\quad Y_{loopFound} \longleftarrow Y'$
18: $\quad\quad$ **while** $(Y_{loopPend} \neq \varnothing)$ **do**

19: Select $y \in Y_{loopPend}$
20: $Y_{loopPend} \longleftarrow Y_{loopPend} \setminus \{y\}$
21: **if** $(\eta(y, \tau)!)$ **then**
22: **for** $y' \in \eta(y, \tau)$ **do**
23: **if** $(y' \notin Y_{loopFound})$ **then**
24: $Y' \longleftarrow Y' \cup \{y'\}$
25: $Y_{loopFound} \longleftarrow Y_{loopFound} \cup \{y'\}$
26: $Y_{loopPend} \longleftarrow Y_{loopPend} \cup \{y'\}$
27: **end if**
28: **end for**
29: **end if**
30: **end while**
31: **if** $(Y' \notin Y_{compSubsFound})$ **then**
32: **for** $\sigma \in \Sigma_c$ **do**
33: **if** $((Y' \cap Y_{disable}(\sigma) \neq \varnothing) \wedge (Y' \cap Y_{enable}(\sigma) \neq \varnothing) \wedge (i \in I_c(\sigma)))$ **then**
34: $Y_{tmpDisable}(\sigma) \longleftarrow Y_{tmpDisable}(\sigma) \cup (Y' \cap Y_{disable}(\sigma))$
35: **end if**
36: **end for**
37: $Y_{compSubsFound} \longleftarrow Y_{compSubsFound} \cup \{Y'\}$
38: **for** $\sigma \in \Sigma_{o,i}$ **do**
39: $Y_{tmp1} \longleftarrow \varnothing$
40: **for** $y \in Y'$ **do**
41: **if** $(\eta(y,\sigma)!)$ **then**
42: $Y_{tmp1} \longleftarrow Y_{tmp1} \cup \{\eta(y,\sigma)\}$

43: **end if**

44: **end for**

45: **if** $(Y_{tmp1} \notin Y_{incSubsFound})$ **then**

46: $Y_{incSubsFound} \longleftarrow Y_{incSubsFound} \cup \{Y_{tmp1}\}$

47: $Y_{pending} \longleftarrow Y_{pending} \cup \{Y_{tmp1}\}$

48: **end if**

49: **end for**

50: **end if**

51: **end while**

52: **return** $Y_{tmpDisable}$

53: **end procedure**

For a particular $\sigma \in \Sigma_c$, the contents of $Y_{tmpDisable}(\sigma)$ remain the same as $Y_{disable}(\sigma)$ if the controller cannot handle that particular σ (Lines 6-12). Lines 13-51 form the loop where the subset construction and the testing occurs.

Starting with subset Y' (Line 14), Lines 18-30 do a τ-closure. A τ closure operation adds to Y' every state reachable from a state $y \in Y$ by a string $t \in \{\tau\}^+$.

If Y' (after τ-closure) has not yet been processed (Line 31), we process it on Lines 32-49. On Lines 32-36, we check for each $\sigma \in \Sigma_c$ that controller $i \in I$ can control whether a state from $Y_{disable}(\sigma)$ and a state from $Y_{enable}(\sigma)$ are both in Y', and if so, we add the $Y_{disable}(\sigma)$ state to $Y_{tmpDisable}(\sigma)$.

In Lines 39-49, we determine new partial subsets (they lack the τ-closure operation), Y_{tmp1}, which consists of states that can be reached, for a given $\sigma \in \Sigma$, from a state $y \in Y'$. If Y_{tmp1} has not yet been encountered, (Line 45), it is marked as encountered and added to pending subset list (Lines 46-47).

The procedure returns the map $Y_{tmpDisable}$ which are all the $Y_{disable}$ states that the current controller could not handle.

4.2.5 Is SB Co-observable Using Observer Construction

The procedure is almost the same as procedure *IsSBCo-observableUsingSubsetConstruction*. The subroutines *EnableDisableSetFind* and *ConstructSynchProductAndHiding* will be reused in this algorithm. The difference is that this algorithm will call the *ObserverConstruction* algorithm, which will do the main SB co-observability check.

The algorithm makes use of the following variables:

I: The set $I := \{1,2,..,n\}$ of controllers where n is the total number of controllers.

Σ_c : Set of controllable events such that $\Sigma_c \subseteq \Sigma$.

$\Sigma_{o,i}$: The set of observable events $\Sigma_{o,i} \subseteq \Sigma$, for each controller $i \in I$.

H : The automaton based on $\mathbf{G}||\mathbf{S}$, with events $\Sigma \backslash \Sigma_{o,i}$ hidden.

$Y_{disable}$: See 3.3.1; is a map such that $Y_{disable}$: $\Sigma_c \rightarrow Pwr(Y)$.

Y_{enable} : See in Definition 3.3.2; is a map such that Y_{enable}: $\Sigma_c \rightarrow Pwr(Y)$.

$Y_{tempDisable}$: The map $Y_{tempDisable}$: $\Sigma_c \rightarrow Pwr(Y)$ is a map such that for each $\sigma \in \Sigma_c$, $Y_{tempDisable}(\sigma)$ reports the un-handled $Y_{disable}(\sigma)$ states that are returned from the procedure *ObserverConstruction*, so that the subsequent controllers have to handle only the remaining problem states.

pass, skipIt : Boolean variables used to test loop termination.

Algorithm 5:

1: **procedure** IsSBCo-observableUsingObserverConstruction($\mathbf{G}, \mathbf{S}$)
2: $\quad$ *EnableDisableSetFind*(Y_{enable}, $Y_{disable}$, $\mathbf{G}$, $\mathbf{S}$)
3: $\quad$ pass $\longleftarrow$ True
4: $\quad$ **for** $\sigma \in \Sigma_c$ **do**
5: $\quad\quad$ **if** ($Y_{disable}(\sigma) \neq \varnothing$) **then**
6: $\quad\quad\quad$ pass $\longleftarrow$ False
7: $\quad\quad$ **end if**
8: $\quad$ **end for**
9: $\quad$ **if** (pass) **then**
10: $\quad\quad$ **return** True
11: $\quad$ **end if**
12: $\quad$ **for** $i \in I$ **do**
13: $\quad\quad$ skipIt $\longleftarrow$ True
14: $\quad\quad$ **for** $\sigma \in \Sigma_c$ **do**
15: $\quad\quad\quad$ **if** $((i \in I_c(\sigma)) \wedge (Y_{disable}(\sigma) \neq \varnothing))$ **then**
16: $\quad\quad\quad\quad$ skipIt $\longleftarrow$ False
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end for**
19: $\quad\quad$ **if** (!skipIt) **then**
20: $\quad\quad\quad$ $\mathbf{H} \longleftarrow$ *ConstructSynchProductAndHiding*($\mathbf{G}$, $\mathbf{S}$, $\Sigma_{o,i}$)
21: $\quad\quad\quad$ $Y_{tempDisable} \longleftarrow$ *ObserverConstruction*($\mathbf{H}$, Y_{enable}, $Y_{disable}$, $\Sigma_{o,i}$, i)
22: $\quad\quad\quad$ pass $\longleftarrow$ True
23: $\quad\quad\quad$ **for** $\sigma \in \Sigma_c$ **do**

24: **if** ($Y_{tempDisable}(\sigma) \neq \varnothing$) **then**

25: pass $\longleftarrow$ False

26: **end if**

27: $Y_{disable}(\sigma) \longleftarrow Y_{tempDisable}(\sigma)$

28: **end for**

29: **if** (pass) **then**

30: **return** True

31: **end if**

32: **end if**

33: **end for**

34: **return** False

35: **end procedure**

Lines 4-11 check for the presence of states in $Y_{disable}(\sigma)$, for every $\sigma \in \Sigma_c$. Absence of any state proves that there are no problem states that are needed to be handled by the controllers. Lines 12-33 process each controller $i \in I$ in turn. Lines 14-18 make sure that controller i can control at least one $\sigma \in \Sigma_c$ so that a disable state can be taken care of. If the above mentioned conditions are met, then we construct the **H** automaton and perform the observer construction test. Otherwise, we skip the current controller and check the same condition for the remaining controllers.

For a particular $\sigma \in \Sigma_c$, $Y_{tempDisable}(\sigma)$, is checked for states after every time a controller goes through the procedure *ObserverConstruction* (Line 24). If $Y_{tempDisable}(\sigma)$ $= \varnothing$ for every $\sigma \in \Sigma_c$, we exit the loop and return *True*. However, if $Y_{tempDisable}(\sigma) \neq \varnothing$ for at least one $\sigma \in \Sigma_c$, it implies that procedure *ObserverConstruction* has returned some subset of $Y_{disable}(\sigma)$ states that could not be handled by the active controller.

These remaining states are then assigned to $Y_{disable}(\sigma)$ (Line 27), which are then handled by the subsequent controllers.

4.2.6 Observer Construction

The purpose of this algorithm is to construct all state pairs $(y_1, y_2) \in Y \times Y$ that are indistinguishable with respect to $\Sigma_{o,i}$ (see definition in section 3.3.3). This means y_1 is reachable from y_0 (the initial state), via a string $s_1 \in \Sigma_\tau^*$ ($\Sigma_\tau = \Sigma \cup \{\tau\}$), and y_2 is reachable from y_0 via a string $s_2 \in \Sigma_\tau^*$, and $P_\tau(s_1) = P_\tau(s_2)$, where $P_\tau : \Sigma_\tau^* \rightarrow \Sigma^*$ is a natural projection. Please note that this is done relative to the automaton $\mathbf{H} = (Y, \Sigma, \eta, y_0, Y_m)$ (described in Section 4.1).

For a given $\sigma \in \Sigma_c$ such that $i \in I_c(\sigma)$, the system fails the SB co-observability test if the algorithm pairs a state $y \in Y_{enable}(\sigma)$ and a state $y' \in Y_{disable}(\sigma)$. This means that the two states are indistinguishable with respect to $\Sigma_{o,i}$. We return this y' as part of $Y_{tmpDisable}(\sigma)$ so that we can test if another controller can handle this state.

The algorithm uses the following variables:

Y_{found}: Set of tuples that have been encountered by the algorithm. We have $\mathrm{Y}_{found} \subseteq Y \times Y$.

$\mathrm{Y}_{pending}$: Set of tuples that have been encountered and are waiting to get processed. We have $\mathrm{Y}_{pending} \subseteq Y \times Y$.

$Y_{tmpDisable}$: The map $Y_{tmpDisable}$: $\Sigma_c \rightarrow Pwr(Y)$ takes each $\sigma \in \Sigma_c$, and reports the remaining $Y_{disable}(\sigma)$ states which, at some point, got paired with $Y_{enable}(\sigma)$ states in a subset. These are the states that are returned so that the subsequent controllers can be tested to see if they can handle these states.

y_0 : State y_0 is the initial state of automaton **H**.

η: Next-state relation for **H**, such that $\eta \subseteq Y \times \Sigma_\tau \times Y$ where $\Sigma'_\tau := \Sigma' \cup \{\tau\}$ for set $\Sigma' \subseteq \Sigma$, and $\tau \notin \Sigma$.

NOTE: For next-state relation η of **H**, when $\sigma' \in \Sigma$, η will return a single state but, for $\tau \notin \Sigma$, η will always return a subset $Y' \subseteq Y$, possibly the empty subset. Also, for τ only, we consider $\eta(y,\tau)!$ to mean that $\eta(y,\tau) \neq \varnothing$, otherwise we use the standard interpretation of "!" for a partial function. This is done to keep the notation simple in the algorithm.

Algorithm 6:

1: **procedure** ObserverConstruction($\mathbf{H}$,Y_{enable},$Y_{disable}$, $\Sigma_{o,i}$, i)
2:     **for** $\sigma \in \Sigma_c$ **do**
3:         **if** $(i \in I_c(\sigma))$ **then**
4:             $Y_{tmpDisable}(\sigma) \longleftarrow \varnothing$
5:         **else**
6:             $Y_{tmpDisable}(\sigma) \longleftarrow Y_{disable}(\sigma)$
7:         **end if**
8:     **end for**
9:     $Y_{found} \longleftarrow \{(y_0,y_0)\}$
10:     $Y_{pending} \longleftarrow \{(y_0,y_0)\}$
11:     **while** $(Y_{pending} \neq \varnothing)$ **do**
12:         Select $(y_1,y_2) \in Y_{pending}$

13: $Y_{pending} \longleftarrow Y_{pending} \setminus \{(y_1,y_2)\}$

14: **for** $\sigma \in \Sigma_{o,i}$ **do**

15: **if** ($\eta(y_1,\sigma)! \wedge \eta(y_2,\sigma)!$) **then**

16: $(y_1{}',y_2{}') \longleftarrow (\eta(y_1,\sigma),\eta(y_2,\sigma))$

17: **if** $((y_1{}',y_2{}') \notin Y_{found})$ **then**

18: **for** $\sigma' \in \Sigma_c$ **do**

19: **if** $((((y_1{}' \in Y_{disable}\ (\sigma')) \wedge (y_2{}' \in Y_{enable}\ (\sigma'))) \vee ((y_1{}' \in Y_{enable}$ $(\sigma')) \wedge (y_2{}' \in Y_{disable}\ (\sigma')))) \wedge (i \in I_c(\sigma')))$ **then**

20: **if** $(y_1 \in Y_{disable}(\sigma'))$ **then**

21: $Y_{tmpDisable}(\sigma') \longleftarrow Y_{tmpDisable}(\sigma') \cup \{y_1\}$

22: **else**

23: $Y_{tmpDisable}(\sigma') \longleftarrow Y_{tmpDisable}(\sigma') \cup \{y_2\}$

24: **end if**

25: **end if**

26: **end for**

27: $Y_{found} \longleftarrow Y_{found} \cup \{(y_1{}',y_2{}')\}$

28: $Y_{found} \longleftarrow Y_{found} \cup \{(y_2{}',y_1{}')\}$

29: $Y_{pending} \longleftarrow Y_{pending} \cup \{(y_1{}',y_2{}')\}$

30: **end if**

31: **end if**

32: **end for**

33: $Y' \longleftarrow \eta(y_1,\tau)$

34: **for** $y \in Y'$ **do**

35: **if** $((y,\ y_2) \notin Y_{found})$ **then**

36: **for** $\sigma \in \Sigma_c$ **do**

37: **if** $((((y \in Y_{disable}\ (\sigma)) \wedge (y_2 \in Y_{enable}\ (\sigma))) \vee ((y \in Y_{enable}\ (\sigma)) \wedge (y_2 \in Y_{disable}\ (\sigma)))) \wedge (i \in I_c(\sigma)))$ **then**

38: **if** $(y \in Y_{disable}(\sigma))$ **then**

39: $Y_{tmpDisable}(\sigma) \longleftarrow Y_{tmpDisable}(\sigma) \cup \{y\}$

40: **else**

41: $Y_{tmpDisable}(\sigma) \longleftarrow Y_{tmpDisable}(\sigma) \cup \{y_2\}$

42: **end if**

43: **end if**

44: **end for**

45: $Y_{found} \longleftarrow Y_{found} \cup \{(y, y_2)\}$

46: $Y_{found} \longleftarrow Y_{found} \cup \{(y_2, y)\}$

47: $Y_{pending} \longleftarrow Y_{pending} \cup \{(y, y_2)\}$

48: **end if**

49: **end for**

50: $Y' \longleftarrow \eta(y_2, \tau)$

51: **for** $y \in Y'$ **do**

52: **if** $((y_1, y) \notin Y_{found})$ **then**

53: **for** $\sigma \in \Sigma_c$ **do**

54: **if** $((((y_1 \in Y_{disable}\ (\sigma)) \wedge (y \in Y_{enable}\ (\sigma))) \vee ((y_1 \in Y_{enable}\ (\sigma)) \wedge (y \in Y_{disable}\ (\sigma)))) \wedge (i \in I_c(\sigma)))$ **then**

55: **if** $(y_1 \in Y_{disable}(\sigma))$ **then**

56: $Y_{tmpDisable}(\sigma) \longleftarrow Y_{tmpDisable}(\sigma) \cup \{y_1\}$

57: **else**

58: $Y_{tmpDisable}(\sigma) \longleftarrow Y_{tmpDisable}(\sigma) \cup \{y\}$

59: **end if**

60: **end if**

61: **end for**

62: $Y_{found} \longleftarrow Y_{found} \cup \{(y_1,y)\}$

63: $Y_{found} \longleftarrow Y_{found} \cup \{(y,y_1)\}$

64: $Y_{pending} \longleftarrow Y_{pending} \cup \{(y_1,y)\}$

65: **end if**

66: **end for**

67: **end while**

68: **return** $Y_{tmpDisable}$

69: **end procedure**

For a particular $\sigma \in \Sigma_c$, the contents of $Y_{tmpDisable}(\sigma)$ remain same as $Y_{disable}(\sigma)$, if the controller cannot handle that particular σ, (Lines 2-8). Lines 11-67 form the main loop where the observer analysis occurs. Of course, we are only interested in determining the state-pairs. We are not interested in determining all the details such that we could build the observer.

On Line 12, we select the next tuple (y_1, y_2) from the pending set. In Lines 14-32, we determine all of the tuples (y'_1, y'_2) such that these are the next states of (y_1, y_2) after a $\sigma \in \Sigma_{o,i}$ transition. If this tuple has not been encountered before (Line 17), we process it (Lines 18-30). On Lines 18-26, we check to see if the algorithm groups together any enable and disable states for a given $\sigma' \in \Sigma_c$. If so, we add the disable state to $Y_{tmpDisable}$ so that we return the state and check to see if another controller can handle it.

Next, we check for states reachable via a τ transition from state y_1 (Lines 33-49). If $y \in \eta(y_1, \tau)$, we reach the tuple (y, y_2) and process it in a similar manner to tuple (y_1', y_2'). This process is then repeated for y_2 (Lines 50-66).

The procedure returns the map $Y_{tmpDisable}$, which contains all the $Y_{disable}$ states that the current controller could not handle.

4.2.7 Indistinguishability and Observers

In order to verify SB co-observability for a plant **G** and a specification **S**, we need to construct the maps $Y_{enable} : \Sigma_c \rightarrow Pwr(Y)$, where Y is the state set of $\mathbf{G}||\mathbf{S}$ and $Y_{disable} : \Sigma_c \rightarrow Pwr(Y)$, as described in Section 3.3.1. We then need to check, for a given $\sigma \in \Sigma_c$ and controller $i \in I_c(\sigma)$, that no state in $Y_{disable}(\sigma)$ is indistinguishable with respect to $\Sigma_{o,i}$ (see Definition 3.3.1) to a state in $Y_{disable}(\sigma)$.

For Algorithms (1-4), the subset construction approach is a standard method to attack a problem like this, and it is easy to see that it will produce the correct result. It is not clear that the observer-based approach, used in Algorithm 6 produces the correct result.

For the observer-based approach, to produce the correct result, the algorithm must group two states together if and only if the two states are indistinguishable with respect to $\Sigma_{o,i}$, for the controller $i \in I = \{1,2,..,n\}$. As the observer construction in Algorithm 6 is based on the OP-Verifier algorithm from [26], we can use the same results from [26] to show that our algorithms will indeed group states together correctly.

To show the above, we will introduce some lemmas from [26]. We will rewrite

the orginal lemmas to match the notation used in this thesis to make the connection clear. We will further express the lemmas in terms of the notation of Definition 3.3.3 for event set $\Sigma_{o,i} \subseteq \Sigma$.

Let $\Sigma_{o,i} \subseteq \Sigma$. Let $P_{\Sigma_{o,i}} : \Sigma^* \to \Sigma_{o,i}^*$ be a natural projection. Let $\mathbf{H} = (Y, \Sigma, \eta, y_0, Y_m)$ $= \mathbf{G}||\mathbf{S}$. We can now state the lemmas used to govern the observer construction in [26].

Lemma 4.2.1. *[26] Let $s_1, s_2 \in \Sigma^*$ and $y_1, y_2 \in Y$ such that $P_{\Sigma_{o,i}}(s_1) = P_{\Sigma_{o,i}}(s_2)$ and $\eta(y_0, s_1) = y_1$ and $\eta(y_0, s_2) = y_2$. Then the observer will group states y_1 and y_2.*

Lemma 4.2.2. *[26] Let the observer group states y_1, $y_2 \in Y$. Then there exists $s_1, s_2 \in \Sigma^*$ such that $P_{\Sigma_{o,i}}(s_1) = P_{\Sigma_{o,i}}(s_2)$, $\eta(y_0, s_1) = y_1$ and $\eta(y_0, s_2) = y_2$.*

If we compare the above lemmas to the definition of indistinguishability 3.3.3 with respect to $\Omega = \Sigma_{o,i}$, (i.e., $\sim_{\Sigma_{o,i}}$), it is easy to see that the lemmas imply that the observer will only group state $y_1, y_2 \in Y$ if and only if $y_1 \sim_{\Sigma_{o,i}} y_2$.

4.3 Complexity Analysis

In this section, we will perform a complexity analysis on the preceding algorithms. We denote the number of states by $|Y|$, the number of events by $|\Sigma|$, and the number of controllers by n. We perform a per algorithm complexity analysis and present the results for each algorithm.

4.3.1 Algorithm Complexity

EnableDisableSetFind (Algorithm 2)

The *while* loop starting at Line 9 will be executed $O\big(|Y|\big)$ times. The *for* loop for $\sigma \in \Sigma_c$, at Line 12 takes $O\big(|\Sigma|\big)$ time. The time complexity for the assignment operation, the membership look-up on Line 23 and the conditional statements are constant time as given in [10]. The complexity for *EnableDisableSetFind* is thus $O\big(|\Sigma||Y|\big)$.

ConstructSynchProductAndHiding (Algorithm 3)

The *while* loop from Lines 7 to 27 take $O\big(|Y|\big)$ time. The nested *for* loop at Line 13, has a time complexity of $O\big(|\Sigma|\big)$. Since the steps occurring inside the loop are constant time as defined in [10], the time complexity for *ConstructSynchProductAndHiding* is $O\big(|\Sigma||Y|\big)$.

SubsetConstruction (Algorithm 4)

As defined in [42], while building the subsets, the entire state space is of size $O\big(2^{|Y|}\big)$. The *while* loop, from Lines 13-51, will execute $O\big(2^{|Y|}\big)$ times, which is the total number of possible subsets. The inner *while* loop from Lines 18-30 is bounded by $O\big(|Y||\Sigma|\big)$, as the while loop will execute $O\big(|Y|\big)$ while the nested *for* loop at Lines 22-28 will have $O\big(|\Sigma|\big)$, assuming the original automaton is deterministic. The *for* loops from Lines 32-36 and 39-45, bounded by $O\big(|\Sigma|\big)$, are additive. The computational complexity of the subset construction algorithm is $O\big(2^{|Y|}|\Sigma||Y|\big)$.

IsSBCo-observableUsingSubsetConstruction (Algorithm 1)

The complexity of *EnableDisableSetFind* defined at Line 2 is $O\big(|\Sigma||Y|\big)$. The *for*

loops from Lines 4-8, 14-18, 23-28 are bounded by $O(|\Sigma|)$ and they are all additive. The complexity of *ConstructSynchProductAndHiding* at Line 20 is $O(|\Sigma||Y|)$ and that of *SubsetConstruction* at Line 21 is $O(2^{|Y|}|\Sigma||Y|)$. As the *for* loop for Lines 12-37 is $O(n)$, the overall complexity for *IsSBCo-observableUsingSubsetConstruction* is $O(n2^{|Y|}|\Sigma||Y|)$.

ObserverConstruction (Algorithm 6)

We first note that the maximum number of tuples is $|Y \times Y|$, which is $O(|Y|^2)$. This means the main *while* loop from Lines 11-67 executes $O(|Y|^2)$ times. We next note that this *while* loop consists three sequential *for* loops at Lines 14-32, 34-49 and 51-66, and that each *for* loop will be activated once per tuple. We note that the *for* loop at Lines 14-32 will execute $O(|\Sigma|)$ times. The *for* loop at Lines 34-49 and Lines 51-66 will also execute $O(|\Sigma|)$ times, assuming that the initial automaton was deterministic. We next note that each *for* loop, has an inner *for* loop such as Lines 18-26 is an inner *for* loop for Lines 14-32. At first glance, it would appear that this loop will be multiplicative, but Lines 17, 35 and 52 ensure that the inner loops execute only once per tuple. This means that the overall complexity of each outer *for* loop remains at $O(|\Sigma|)$. When we combine the *while* loop with the three inner *for* loops, we find the complexity of the algorithm to be $O(|\Sigma||Y|^2)$.

IsSBCo-observableUsingObserverConstruction (Algorithm 5)

The complexity of *EnableDisableSetFind* defined at Line 2 is $O(|\Sigma||Y|)$. The *for* loops from Lines 4-8, 14-18, 23-28 are bounded by $O(|\Sigma|)$ transitions and they are all additive.The complexity of *ConstructSynchProductAndHiding* at Line 20 is $O(|\Sigma||Y|)$

and that of *ObserverConstruction* at Line 21 is $O\big(|\Sigma||Y|^2\big)$. As the *for* loop in Lines 12-32 is $O\big(n\big)$, the overall complexity for *IsSBCo-observableUsingObserverConstruction* is $O\big(n|\Sigma||Y|^2\big)$.

We noted earlier that the construction complexity of *IsSBCo-observableUsing SubsetConstruction* algorithm is $O\big(n2^{|Y|}|\Sigma||Y|\big)$. This implies that the observer-based approach should perform much better for systems with a large state-space. However, the above is a worst case analysis. We still need to apply both algorithms to real-life examples to ensure that we see this savings in practice.

4.3.2 Comparing Complexity

To verify co-observability for large systems, we need algorithms that are efficient in time and space. The algorithms to verify language-based co-observability are exponential in the number of decentralized controllers. The computational complexity to verify co-observability using the approach in [36] is $O\big(|\Sigma|^{n+2}|Y|^{n+2}\big)$ and the complexity to verify co-observability using the approach in [33] is $O\big(|\Sigma|^{n+1}|Y|^{n+1}\big)$. Additionally, the computational complexity to verify SB co-observability using observers is $O\big(n|\Sigma||Y|^2\big)$, which is quadratic. This implies that as the number of controllers increases, verifying SB co-observability should be significantly more efficient, allowing systems with larger state spaces to be verified. However, the above is a worst case analysis. We still need to apply both algorithms to real-life examples to ensure that we see this savings in practice.

The trade off is that SB co-observability is only a sufficient condition to verify co-observability.

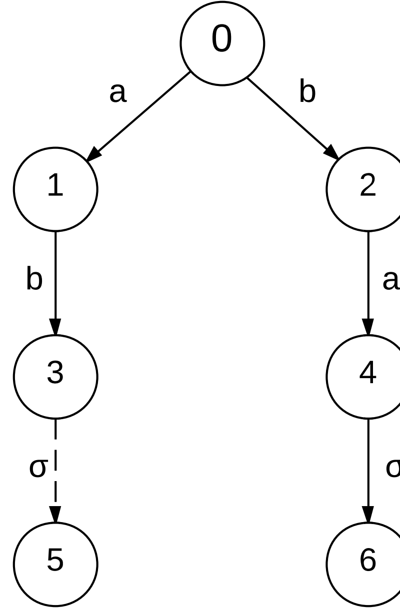


Figure 4.1: An Example DES

4.4 Small Example

We now present a small example to illustrate the algorithm defined in Section 4.2.5. Figure 4.1 shows a plant **G** and specification automaton **S**. The plant includes all transitions but the specification does not include the transition from state 3 to state 5. This means that $\mathbf{G} \| \mathbf{S} = \mathbf{S}$.

Let $I = \{1{,}2\}$, $\Sigma = \{a{,}b{,}\sigma\}$ $\Sigma_{o,1} = \{a{,}\sigma\}$ and $\Sigma_{o,2} = \{b{,}\sigma\}$. We thus have two controllers. Further, $\Sigma_{c,1} = \{\sigma\}$ and $\Sigma_{c,2} = \varnothing$. As $\Sigma_c = \{\sigma\}$, we thus have $I_c(\sigma) = \{1\}$. This means that we can verify SB co-observability using only one controller.

Algorithm 2: *EnableDisableSetFind*

Input: G and **S**.

According to Figure 4.1, starting from the initial state and comparing the *plant* and

the *specification automaton*, we observe that σ, which is the only controllable event, takes the system from state 4 to state 6, both in **G** and **S**, therefore, state 4 is added to $Y_{enable}(\sigma)$. However, a σ transition is possible at state 3 in the plant but not in the specification automaton. Therefore, we add state 3 to the $Y_{disable}(\sigma)$.

Output: $Y_{disable}(\sigma) = \{3\}$, $Y_{enable}(\sigma) = \{4\}$.

Algorithm 3: ***ConstructSynchProductAndHiding***

Next step is to evaluate the synchronous product $\mathbf{G}||\mathbf{S}$ and to hide the set of unobservable events for each controller. It has been noted earlier that $\mathbf{G}||\mathbf{S} = \mathbf{S}$.

Input: $i = 1$, Controller 1, $\Sigma_{o,i} = \{a,\sigma\}$.

Traversing from the initial state, each b transition is replaced by a τ transition since b is unobservable to controller 1. The automaton obtained is shown in Figure 4.2. This is the automaton **H**. Transitions for all events in $\Sigma \setminus \Sigma_{o,1}$ have been replaced by τ.

Algorithm 4: ***SubsetConstruction***

According to the *SubsetConstruction* algorithm, we group states into subsets such that for any two states, they can be reached by the initial state by strings $s_1, s_2 \in \Sigma_\tau^*$ respectively, and $P_\tau(s_1) = P_\tau(s_2)$, where $P_\tau : \Sigma_\tau^* \rightarrow \Sigma^*$ is a natural projection.

Input: {0}

Starting at the initial state 0, the system does not know if it is at state 0 or in a state reachable from state 0 via an unobservable τ event. This means the system could be in state 0 or state 2. Since the system cannot determine the current position, the two states, state 0 and state 2 are indistinguishable. Hence, they belong to the same subset i.e.

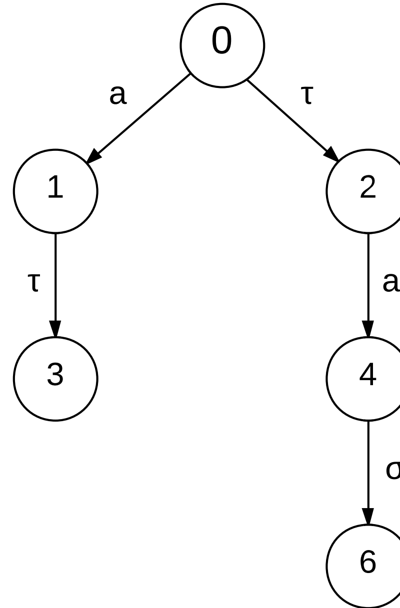


Figure 4.2: Synchronous Product of Plant and Specification Automaton

$\mathbf{\{0\}} \xrightarrow{\tau} \mathbf{\{0,2\}}$.

Since there are no more τ transitions possible from the initial state, we then identify all subsets that can be reached from {0,2} by an observable event. According to Figure 4.2, observable event a takes the system from state 0 to state 1, and state 2 to state 4. Therefore, observable event a takes the system from subset {0,2}, to subset {1,4}. We thus have:

$\mathbf{\{0,2\}} \xrightarrow{a} \mathbf{\{1,4\}}$.

We now need to expand subset {1,4} to include states reachable by a sequence of τ transitions. From state 1, there is an unobservable τ transition possible that takes state 1 to state 3. Because of the unobservable event that takes state 1 to state 3, these states again become indistinguishable, which places state 3 in the same set as state 1. We thus have:

$\mathbf{\{0,2\}} \xrightarrow{a} \mathbf{\{1,4,3\}}$.

As we can see from the subsets, state 3 is in $Y_{disable}(\sigma)$ and it is grouped with state 4 which is in $Y_{enable}(\sigma)$. This means that the test has failed for this particular controller. We set $Y_{tmpDisable}(\sigma) = \{3\}$ and return this value.

Recall that, $I_c(\sigma) = \{1\}$ so the other controller cannot disable σ so it too fails the test. We thus fail to verify SB co-observability.

Output: $Y_{tmpDisable}(\sigma) = \mathbf{\{3\}}$

Algorithm 5: *ObserverConstruction*

This algorithm groups together states that are indistinguishable with respect to $\Sigma_{o,i}$. As noted previously, for this example, we need only to consider $i = 1$. We start with the tuple (0,0), as state 0 is the initial state of **H**.

Input: (0,0).

We then look for any observable events (i.e., in $\Sigma_{o,1}$) that are possible at both states of the tuple. This gives:

$\mathbf{(0,0)} \xrightarrow{a} \mathbf{(1,1)}$

Now there are no more observable events possible from (0,0) so, we determine state pairs reachable from (0,0) via an unobservable τ transition. This gives:

$\mathbf{(0,0)} \xrightarrow{\tau} \mathbf{(2,0)}$, and

$\mathbf{(0,0)} \xrightarrow{\tau} \mathbf{(0,2)}$

Now we keep determining the state pairs formed from the new state pairs obtained. This gives:

$\mathbf{(1,1)} \xrightarrow{\tau} \mathbf{(3,1)}$

$\mathbf{(1,1)} \xrightarrow{\tau} \mathbf{(1,3)}$

$\mathbf{(2,0)} \xrightarrow{a} \mathbf{(4,1)}$

$\mathbf{(2,0)} \xrightarrow{a} \mathbf{(1,4)}$

$\mathbf{(2,0)} \xrightarrow{\tau} \mathbf{(2,2)}$

$\mathbf{(3,1)} \xrightarrow{\tau} \mathbf{(3,3)}$

$\mathbf{(1,3)} \xrightarrow{\tau} \mathbf{(3,3)}$

$\mathbf{(4,1)} \xrightarrow{\tau} \mathbf{(4,3)}$

The state pair (4,3) has a $Y_{enable}(\sigma)$ and a $Y_{disable}(\sigma)$ state grouped together which means that the test fails for $Y_{disable}(\sigma)$ at state 3. We thus set $Y_{tmpDisable}(\sigma) = \{3\}$.

The algorithm continues until all state-pairs encountered are processed, but as $Y_{disable}(\sigma) = \{3\}$, this will not affect the outcome of the algorithm. We will thus not describe these steps.

Output: $Y_{tmpDisable}(\sigma) = \mathbf{\{3\}}$

As $I_c(\sigma) = \{1\}$, the second controller can not disable σ so it can not handle state 3 either. This means that the algorithm *IsSBCo-observableUsingObserverConstruction* returns *False*, which means that the verification of SB co-observability has failed.

4.5 Counter Example

Since we are now aware of the working of the Algorithm 5, *IsSBCo-observableUsing ObserverConstruction*, we quickly discuss the counter example, given in Figure 4.3, which was introduced in Section 3.3.3. We note that Figure 4.3 shows both, the plant **G** and the specification automaton **S**. The plant contains all transitions, while the specification does not contain the c transition at state 2. We have $n = 2$ such that $\Sigma_{o,1} = \{a1, a2\}$, $\Sigma_{o,2} = \{b1, b2\}$ and $\Sigma_c = \Sigma_{c,1} = \Sigma_{c,2} = \{c\}$ are the set of observable and controllable events.

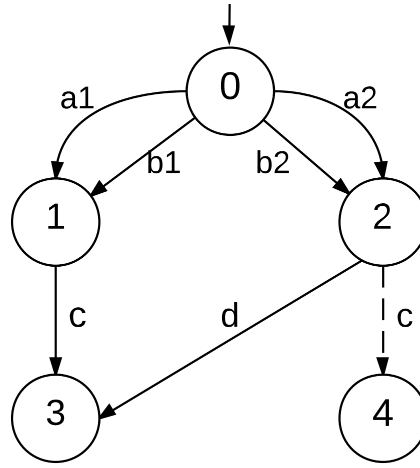


Figure 4.3: Counter Example for SB co-observability

By using Algorithm 2, *EnableDisableSetFind*, we have $Y_{enable}(c) = \{1\}$ and $Y_{disable}(c) = \{2\}$.

We construct the synchronous product with hiding for $i = 1$ using Algorithm 3, *ConstructSynchProductAndHiding*, where $\Sigma_{o,1} = \{a1,a2\}$. This is shown in Figure 4.4. The following are the sequences of tuples generated when we start with the initial state tuple:

Input: (0,0).

(0,0) $\xrightarrow{a1}$ **(1,1)**

(0,0) $\xrightarrow{\tau}$ **(0,1)**

(0,0) $\xrightarrow{\tau}$ **(0,2)**

(0,0) $\xrightarrow{\tau}$ **(1,0)**

(0,0) $\xrightarrow{\tau}$ **(2,0)**

(0,0) $\xrightarrow{a2}$ **(2,2)**

We then examine tuple **(0,1)**.

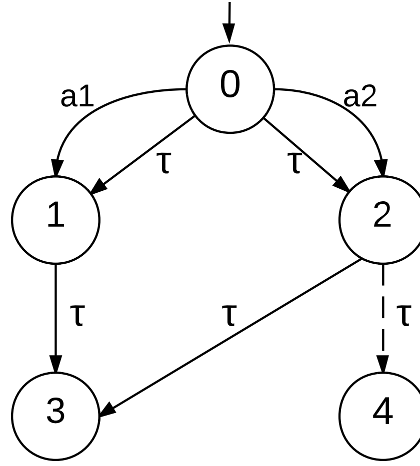


Figure 4.4: Observer $\mathbf{H_1}$

(0,1) $\xrightarrow{\tau}$ **(1,1)**

(0,1) $\xrightarrow{\tau}$ **(2,1)**

The state pair (2,1) has a $Y_{enable}(c)$ and a $Y_{disable}(c)$ state grouped together, which means that the test fails for $Y_{disable}(c)$ at state 2.

Set $Y_{tmpDisable}(c) = \{2\}$.

The algorithm continues until all state pairs encountered are processed. $Y_{tmpDisable}(c) = \{2\}$ in Algorithm 5, *IsSBCo-observableUsingObserverConstruction*. Therefore, $Y_{disable}(c) = \{2\}$ for the next controller i.e., $i = 2$. where $\Sigma_{o,2} = \{b1, b2\}$.

We construct the synchronous product with hiding for $i = 2$ using Algorithm 3, *ConstructSynchProductAndHiding*, where $\Sigma_{o,2} = \{b1, b2\}$. This is shown in Figure 4.5. The following are the sequences of tuples generated when we start with the initial state tuple:

Input: (0,0).

(0,0) $\xrightarrow{b1}$ **(1,1)**

(0,0) $\xrightarrow{\tau}$ **(1,0)**

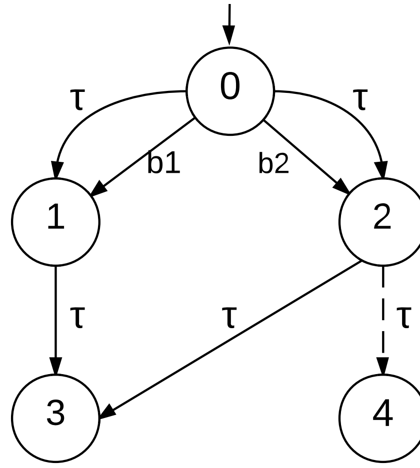


Figure 4.5: Observer $\mathbf{H_2}$

$\mathbf{(0,0)} \xrightarrow{\tau} \mathbf{(2,0)}$

$\mathbf{(0,0)} \xrightarrow{\tau} \mathbf{(0,1)}$

$\mathbf{(0,0)} \xrightarrow{\tau} \mathbf{(0,2)}$

$\mathbf{(0,0)} \xrightarrow{b2} \mathbf{(2,2)}$

We then examine tuple **(1,0)**.

$\mathbf{(1,0)} \xrightarrow{\tau} \mathbf{(3,0)}$

$\mathbf{(1,0)} \xrightarrow{\tau} \mathbf{(1,1)}$

$\mathbf{(1,0)} \xrightarrow{\tau} \mathbf{(1,2)}$

Again the state pair (1,2) contains both, a state in $Y_{enable}(c)$ and $Y_{disable}(c)$, which means that the test fails for $Y_{disable}(c)$ at state 2 which is why $Y_{tmpDisable}(c) = \{2\}$. The algorithm continues until all state pairs encountered are processed.

Since the second controller also fails to handle state 2. This means that the verification of SB co-observability fails, even though language-based co-observability holds.

Chapter 5

Symbolic Verification of Decentralized DES

The automaton-based algorithms presented in the previous chapter work well for small systems, but for a large system, use of these automaton-based algorithms may not be computationally efficient since there may be millions of transitions and states. For large systems with more than 10^{10} states, it is almost impossible to explicitly define each transition in computer memory. A better way to represent such a large system is to use Binary Decision Diagram(BDD) [5] based algorithms.

Transition predicates and state predicates to store the transition information and state sets are used to represent large systems efficiently, allowing the algorithms to scale well. Additionally, it helps to save memory which leads to faster computation. Properties like controllability and nonblocking are easily verified for systems that have a large number of states.

We make use of predicate-based algorithms built on the work of [38] and [40].

In this chapter we present predicate-based algorithms for the verification of SB co-observability. We first present a short introduction to predicates and predicate transformers followed by the symbolic representation of state subsets and transitions and how to use them to compute predicate transformers.

We have used a similar structure of writing predicate-based algorithms and implementing the transition predicates was used in [40]. The implementation of these algorithms will be done by making use of Reduced Ordered Binary Decision Diagram, which we will refer to simply as BDDs. The *BuDDy* library [22] is a *C++* Library which is used for implementing various BDD operations, and will be used in the software implementation of these algorithms.

5.1 Predicates And Predicate Transformers

Before defining predicates, we wish to specify that from here onwards '≡' will be used for logical equivalence between the state predicates. 'T' will correspond to logical *True*, similarly 'F' will correspond to logical *False*.

5.1.1 Predicates

Let $\mathbf{G} = (Q,\Sigma,\delta,q_0,Q_m)$ be a DES. A *predicate* P defined on state set Q, is a function

$$P : Q \to \{T, F\},$$

and is identified by the corresponding state subset

$$Q_P := \{q \in Q | P(q) = T\} \subseteq Q.$$

We write $q \models P$ if $q \in Q_P$ and say q *satisfies* P or P *includes* q. We write $Pred(Q)$ for the set of all predicates defined on Q, hence $Pred(Q)$ is identified by $Pwr(Q)$.

We identify state predicate *true* by Q, state predicate *false* by $\varnothing$, and state predicate P_m by Q_m. For $P \in Pred(Q)$, we write $st(P)$ for the corresponding state subset $Q_P \subseteq Q$ which identifies P. We write $pr(Q')$ to represent the predicate that is identified by $Q' \subseteq Q$.

If P, P_1, $P_2 \in Pred(Q)$ and $q \in Q$, we build the following boolean expressions using predicate operations.

$$(\neg P)(q) = T \iff P(q) = F$$

$$(P_1 \wedge P_2)(q) = T \iff P_1(q) = T \text{ and } P_2(q) = T;$$

$$(P_1 \vee P_2)(q) = T \iff P_1(q) = T \text{ or } P_2(q) = T;$$

$$(P_1 \text{ - } P_2)(q) = T \iff P_1(q) = T \text{ and } P_2(q) = F.$$

Similarly, P_1 - $P_2 = P_1 \wedge \neg P_2$.

The relation $\preceq$ over $Pred(Q)$ is defined as

$$(\forall\ P_1, P_2 \in Pred(Q))\ P_1 \preceq P_2 \iff (P_1 \wedge P_2) \equiv P_1.$$

Clearly, $Q_{P_1} \subseteq Q_{P_2} \iff P_1 \preceq P_2$. Hence,

$$(\forall\ q \in Q)\ q \models P_1 \implies q \models P_2$$

We say P_1 is a sub-predicate of P_2 if $P_1 \preceq P_2$ and we say P_1 is *stronger* than P_2 or P_2 is *weaker* than P_1. We define $Sub(P)$ to be the set of all the sub-predicates of $P \in Pred(Q)$ such that $Sub(P)$ is identified by $Pwr(Q_P)$.

5.1.2 Predicate Transformers

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a DES and let $P \in Pred(Q)$. We define a *predicate transformer* to be a function $f : Pred(Q) \rightarrow Pred(Q)$.

There are four major predicate transformers defined in [38]. Here we will only define the predicate transformer used in this thesis.

- $R(\mathbf{G}, P)$

The *reachability predicate* $R(\mathbf{G}, P)$, is true for those states that can be reached in $\mathbf{G}$ from q_0, via states satisfying P.

1. $q_0 \models P \implies q_0 \models R(\mathbf{G}, P)$
2. $q_0 \models R(\mathbf{G}, P) \wedge \sigma \in \Sigma \wedge \delta(q, \sigma)! \wedge \delta(q, \sigma) \models P \implies \delta(q, \sigma) \models R(\mathbf{G}, P)$
3. No other states satisfy $R(\mathbf{G}, P)$.

In other words, a state $q \models R(\mathbf{G}, P)$ if and only if there exists a path in $\mathbf{G}$ from q_0 to q such that each state in that path satisfies P. To represent the set of all reachable states in Q, we use $R(\mathbf{G}, \text{true})$.

5.2 Symbolic Representation

To make use of predicates in algorithms, we need to symbolically represent states and transitions. We use the representation that was used in [38] and [40].

Predicates that are defined over the state set of a DES are called *state predicates*. Similarly, the predicates defined on the transitions of a DES are called *transition predicates*.

5.2.1 State Subsets

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m) = \mathbf{G_1} \parallel \mathbf{G_2} \parallel \mathbf{G_3} \parallel \ldots \parallel \mathbf{G}_n$ where $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{0,i}, Q_{m,i})$ such that $i = 1, 2, ..., n$. Let $q \in Q$ such that $q = (q_1, q_2,.., q_n)$ where $q_1 \in Q_1, ..., q_n \in Q_n$

Definition 5.2.1. *For $\boldsymbol{G} = \boldsymbol{G_1} \parallel \boldsymbol{G_2} \parallel \boldsymbol{G_3} \parallel \ldots \parallel \boldsymbol{G}_n$, let i = 1,2,...,n and $q_i \in Q$. Let v_i be the* state variable *with domain Q_i for component $\boldsymbol{G}_i$. If v_i is set to q_i then the equation $v_i = q_i$ returns T otherwise, F is returned.*

Definition 5.2.2. *A state variable vector, denoted by $\mathbf{v}$, is a vector of state variables [v_1,v_2,...,v_n], such that for a state subset $A \subseteq Q$, the predicate P_A is:*

$$P_A(\mathbf{v}) \coloneqq \bigvee_{q \in A}(v_1 = q_1 \wedge v_2 = q_2 \wedge \ldots \wedge v_n = q_n)$$

For convenience $P_A(\mathbf{v})$ is written as P_A, if $\mathbf{v}$ is understood.

5.2.2 Transitions

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m) = \mathbf{G_1} \parallel \mathbf{G_2} \parallel \mathbf{G_3} \parallel \ldots \parallel \mathbf{G}_n$ where $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{0,i}, Q_{m,i})$ such that i = 1,2,...,n.

Definition 5.2.3. *For $\sigma \in \Sigma$, a transition predicate, denoted by $N_\sigma : Q \times Q \to \{T, F\}$, identifies all the transitions for σ in $\boldsymbol{G} = \mathbf{G}_1 \| \mathbf{G}_2 \| \ldots \| \mathbf{G}_n$. It is defined as follows:*

$$(\forall q, q' \in Q) N_\sigma(q, q') \coloneqq \begin{cases} T, & \textit{if } \delta(q, \sigma)! \ \wedge \delta(q, \sigma) = q'; \\ F, & \textit{otherwise.} \end{cases}$$

To differentiate between source and destination states, we need to define different sets of state variables.

Definition 5.2.4. *For some $\boldsymbol{G} = \mathbf{G}_1 \| \mathbf{G}_2 \| \ldots \| \mathbf{G}_n$, let i = 1,2,...,n. We define v_i to be the normal state variable (source state) and v_i' to be the prime state variable*

(destination state), both for DES $\boldsymbol{G}_i = (Q_i, \Sigma_i, \delta_i, q_{0,i}, Q_{m,i})$ and with domain Q_i. We have the normal state variable vector $\mathbf{v} = [v_1, v_2, .., v_n]$ and the prime state variable vector $\mathbf{v}' = [v_1', v_2', \ldots, v_n']$, both for $\boldsymbol{G}$.

For event $\sigma \in \Sigma$, in $\boldsymbol{G}$, the transition predicate N_σ is defined as :

$$N_\sigma(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \le i \le n\}} \left(\bigvee_{\{q_i, q_i' \in Q_i | \delta_i(q_i, \sigma) = q_i'\}} (v_i = q_i) \wedge (v_i' = q_i') \right)$$

such that when we set $\mathbf{v} = q$ and $\mathbf{v}' = q'$ with $\delta(q, \sigma) = q'$, then $N_\sigma(\mathbf{v}, \mathbf{v}')$ returns T.

The above expression assumes each DES component is defined over the event set Σ. However, when multiple DES components are used, they might be defined over different event sets such that when we perform a synchronous product, each component DES will be self looped with the events that does not belong to its event set. To solve this problem we make use of *transition tuples*. This is defined below.

Definition 5.2.5. *A* Transition tuple *is denoted as $(\mathbf{v}_\sigma, \mathbf{v}_\sigma', N_\sigma)$ for each $\sigma \in \Sigma$ where $\mathbf{v}_\sigma = \{v_i \in \mathbf{v} | \sigma \in \Sigma_i\}$, $\mathbf{v}_\sigma' = \{v_i' \in \mathbf{v}' | \sigma \in \Sigma_i\}$ where*

$$N_\sigma(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \le i \le n | \sigma \in \Sigma_i\}} \left(\bigvee_{\{q_i, q_i' \in Q_i | \delta_i(q_i, \sigma) = q_i'\}} (v_i = q_i) \wedge (v_i' = q_i') \right)$$

Here the self-loop transitions are not explicitly defined, but state variables that are not defined in $\mathbf{v}_\sigma$ for a component DES will be implicitly self-looped with event σ.

We use the *existential quantifier elimination* method for finite domain [2] to either eliminate the prime or the normal variable, since BDD [22] does not support first order logic. The results obtained from this is still a propositional formula which can be represented as a BDD.

Definition 5.2.6. *For a given system $\boldsymbol{G} = \mathbf{G}_1 || \mathbf{G}_2 || \ldots || \mathbf{G}_n$, let $\sigma \in \Sigma$. The transition tuple for σ in $\boldsymbol{G}$ is $(\mathbf{v}_\sigma, \mathbf{v}_\sigma', N_\sigma)$. If $v_i \in \mathbf{v}_\sigma$ and $v_i' \in \mathbf{v}_\sigma'$, where $i = 1, 2, .., n$, we define*

$$\exists v_i N_\sigma := \bigvee_{q_i \in Q_i} N_\sigma[q_i/v_i] \ \textit{and}\ \exists v_i' N_\sigma := \bigvee_{q_i \in Q_i} N_\sigma[q_i/v_i'],$$

where $N_\sigma[q_i/v_i]$ *is the predicate* N_σ *where each* v_i *is substituted by* q_i*, and* $N_\sigma[q_i/v_i']$ *is where* v_i' *is substituted for* q_i*.*

Let $\mathbf{v}_\sigma = \{v_1, v_2, \ldots, v_m\}$, with $m>0$. Computation of state predicates require them to be consistent, i.e., either the resulting predicates should contain only prime variables or should contain only normal variables. To maintain the consistency of the resultant predicates, we denote $\exists \mathbf{v}_\sigma N_\sigma$, which is $\exists v_1(\exists v_2..(\exists v_n N_\sigma))$ as $\exists \mathbf{v}_\sigma N_\sigma[\mathbf{v}_\sigma' \rightarrow \mathbf{v}_\sigma]$ where the prime variables are substituted by normal variables. This represents the set of states that have an entering σ transition.

Let $\mathbf{v'}_\sigma = \{v_1', v_2', \ldots, v_m'\}$. Similarly, we use $\exists \mathbf{v'}_\sigma N_\sigma$ to represent $\exists v_1'$ $(\exists v_2'$ ...$(\exists v_n'$ $N_\sigma))$. This predicate represents the set of states with an exiting σ transition.

5.3 Symbolic Computation

On the basis of our symbolic representation, we define symbolic computation. This work is based on [25], [38] and [40].

5.3.1 Transition and Inverse Transition

For $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$, we want to calculate $\delta(q, \sigma)$, where $q \in Q$ and $\sigma \in \Sigma$. Using our symbolic representation for state subset $Q_P \subseteq Q$, where $P \in Pred(Q)$, state subset $Q_P' = \bigcup_{q \in Q_p}\{\delta(q, \sigma)\}$ needs to be calculated to find all the states that are reachable by a σ transition from any state in Q_P. This is quite time consuming so we use a method to calculate the predicates of next states using the predicates of the current

states. We use the function $\hat{\delta} : Pred(Q) \times \Sigma \to Pred(Q)$, with $P \in Pred(Q)$, and $\sigma \in \Sigma$ defined as:

$$\hat{\delta}(P, \sigma) := pr(\{q' \in Q | (\exists q \models P)\delta(q, \sigma) = q'\}).$$

If $\sigma \in \Sigma$ and $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ is the transition tuple for σ in $\mathbf{G}$, then for some $P \in Pred(Q)$

$$\hat{\delta}(P, \sigma) := (\exists \mathbf{v}_\sigma (N_\sigma \wedge P))[\mathbf{v}'_\sigma \to \mathbf{v}_\sigma].$$

This returns a predicate representing all states reachable by a σ transition from a state that satisfies P.

Similary we define $\hat{\delta}^{-1} : Pred(Q) \times \Sigma \to Pred(Q)$, with $P \in Pred(Q)$ and $\sigma \in \Sigma$, as:

$$\hat{\delta}^{-1}(P, \sigma) := pr(\{q \in Q | \delta(q, \sigma) \models P\}).$$

If $\sigma \in \Sigma$ and $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ is the transition tuple for σ in $\mathbf{G}$, then for some $P \in Pred(Q)$

$$\hat{\delta}^{-1}(P, \sigma) := \exists \mathbf{v}'_\sigma (N_\sigma \wedge (P[\mathbf{v}_\sigma \to \mathbf{v}'_\sigma])).$$

This returns a predicate representing all states that can reach a state that satisfies P via a σ transition.

5.3.2 Computation of Reachability

In this section we will look at how to compute the reachability predicate R. We use the algorithm from [40].

Let there be a *plant* $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ where $\mathbf{G} = \mathbf{G}_1 \parallel \mathbf{G}_2 \parallel \ldots \parallel \mathbf{G}_n$. $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{0,i}, Q_{m,i})$ such that $i = 1, 2, ..., n$ and $P \in Pred(Q)$.

Reachability check

1: **procedure** $R(\mathbf{G}, P)$
2: $P_1 \leftarrow P \wedge pr(\{q_o\})$
3: **repeat**
4: $P_2 \leftarrow P_1$
5: **for** $i \leftarrow 1$ to n **do**
6: **repeat**
7: $P_3 \leftarrow P_1$
8: $P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma_i} (\hat{\delta}(P_1, \sigma) \wedge P) \right)$
9: **until** $P_1 \equiv P_3$
10: **end for**
11: **until** $P_1 \equiv P_2$
12: **return** P_1
13: **end procedure**

$R(\mathbf{G}, P)$ takes $\mathbf{G}$ and P as an input and returns a predicate which consists of states in $\mathbf{G}$ that can be reached from q_o via states satisfying P. Normally we would set $P = true$.

5.4 Symbolic Verification

Let $G_i = (Q_i, \Sigma, \delta_i, q_{0,i}, Q_{m,i})$ and $S_j = (X_j, \Sigma, \xi_j, x_{0,j}, X_{m,j})$, for $i \in \{1,2...,n\}$ and $j \in \{1,2...,m\}$ such that:

$\mathbf{G} = \mathbf{G}_1 \| \mathbf{G}_2 \| \ldots \| \mathbf{G}_n := (Q, \Sigma, \delta, q_o, Q_m)$ is the plant automaton, and

$\mathbf{S} = \mathbf{S}_1 \| \mathbf{S}_2 \| \ldots \| \mathbf{S}_m := (X, \Sigma, \xi, x_o, X_m)$ is the specification automaton.

Here both **G** and **S** are defined over Σ. If either of **G** or **S** were defined over some subset of Σ, say $\Sigma' \subseteq \Sigma$, we would then add self loops of every event in $\Sigma \backslash \Sigma'$, to every state in the automaton, so that there is no loss of generality to assume that they are both defined over Σ.

The synchronous product of the plant and the specification automaton is $\mathbf{H} = \mathbf{G} \| \mathbf{S}$ where $\mathbf{H} = (Y, \Sigma, \eta, y_o, Y_m)$ such that $Y = Q \times X = Q_1 \times Q_2 \times \ldots \times Q_n \times X_1 \times X_2 \times \ldots \times X_m$, $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$, $\eta = \delta \times \xi$, $y_o = (q_o, x_o)$ and $Y_m = Q_m \times X_m$ as given in 2.1.

Definition 5.4.1. *Let* $\boldsymbol{H} = \boldsymbol{G} \parallel \boldsymbol{S} := (Y, \Sigma, \eta, y_o, Y_m)$ *where* $\boldsymbol{G} = \boldsymbol{G}_1 \| \boldsymbol{G}_2 \| \ldots \| \boldsymbol{G}_n = (Q, \Sigma, \delta, q_o, Q_m)$ *and* $\boldsymbol{S} = \boldsymbol{S}_1 \| \boldsymbol{S}_2 \| \ldots \| \boldsymbol{S}_m = (X, \Sigma, \xi, x_o, X_m)$. *For a given event* $\sigma \in \Sigma$, *the* σ *plant transition predicate* $N_{\mathbf{G},\sigma} : Y \times Y \rightarrow \{T, F\}$, *can be written as*

$$N_{\mathbf{G},\sigma}(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \leq i \leq n\}} \left(\bigvee_{\{q_i, q_i' \in Q_i | \delta_i(q_i, \sigma) = q_i'\}} (v_i = q_i) \wedge (v_i' = q_i') \right)$$

and the σ *supervisor transition predicate* $N_{\mathbf{S},\sigma} : Y \times Y \rightarrow \{T, F\}$ *can be written as*

$$N_{\mathbf{S},\sigma}(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \leq i \leq m\}} \left(\bigvee_{\{x_i, x_i' \in X_i | \xi_i(x_i, \sigma) = x_i'\}} (v_{i+n} = x_i) \wedge (v_{i+n}' = x_i') \right)$$

$N_{\mathbf{G},\sigma}$ and $N_{\mathbf{S},\sigma}$ are state predicates defined on $Y \times Y$ and use the $\mathbf{v}$ and $\mathbf{v}'$ variables. We use $N_{\mathbf{G},\sigma}$ to determine if there is a σ defined at the plant portion of the indicated states. Similarly, we use $N_{\mathbf{S},\sigma}$ to determine if there is a σ defined at the supervisor portion of the indicated states.

Definition 5.4.2. *Let* $\sigma \in \Sigma$ *and* $N_{\boldsymbol{G},\sigma}$ *be the* σ *transition predicate for plant* $\boldsymbol{G} =$

$(Q, \Sigma, \delta, q_o, Q_m)$. *We define* $\hat{\delta}_{\boldsymbol{G}} : Pred(Y) \times \Sigma \to Pred(Y)$, *for* $P \in Pred(Y)$, *to be*

$$\hat{\delta}_{\boldsymbol{G}}(P, \sigma) := (\exists \mathbf{v}(N_{\boldsymbol{G},\sigma} \wedge P))[\mathbf{v}' \to \mathbf{v}]$$

and we also define $\hat{\delta}^{-1}_{\boldsymbol{G}} : Pred(Y) \times \Sigma \to Pred(Y)$ *to be*

$$\hat{\delta}^{-1}_{\boldsymbol{G}}(P, \sigma) := \exists \mathbf{v}'(N_{\boldsymbol{G},\sigma} \wedge (P[\mathbf{v} \to \mathbf{v}']))$$

Definition 5.4.3. *Let* $\sigma \in \Sigma$ *and* $N_{\boldsymbol{S},\sigma}$ *be the* σ *transition predicate for specification* $\boldsymbol{S}$= $(X, \Sigma, \xi, x_o, X_m)$. *We define* $\hat{\xi} : Pred(Y) \times \Sigma \to Pred(Y)$, *for* $P \in Pred(Y)$, *to be*

$$\hat{\xi}(P, \sigma) := (\exists \mathbf{v}(N_{\boldsymbol{S},\sigma} \wedge P))[\mathbf{v}' \to \mathbf{v}]$$

and we also define $\hat{\xi}^{-1} : Pred(Y) \times \Sigma \to Pred(Y)$ *to be*

$$\hat{\xi}^{-1}(P, \sigma) := \exists \mathbf{v}'(N_{\boldsymbol{S},\sigma} \wedge (P[\mathbf{v} \to \mathbf{v}']))$$

In Section 5.3.1, we defined maps $\hat{\delta}$ and $\hat{\delta}^{-1}$. In an analogous way we can define, for $\mathbf{H} = \mathbf{G} \parallel \mathbf{S}$, the maps $\hat{\eta} : Pred(Y) \times \Sigma \to Pred(Y)$ and $\hat{\eta}^{-1} : Pred(Y) \times \Sigma \to Pred(Y)$. These would use a corresponding N_σ defined to use variable v_1 to v_{n+m}.

5.4.1 Computation Of P_{Dis} And P_{En}

From the definition of $Y_{disable}(\sigma)$, in Definition 3.3.1, we know that $Y_{disable}(\sigma)$ maps a $\sigma \in \Sigma_c$ to the set of reachable states in $\mathbf{H} = \mathbf{G} || \mathbf{S}$ with an outgoing σ transition in $\mathbf{G}$ but no σ transition in $\mathbf{S}$.

We can express all the states that would qualify for $Y_{disable}(\sigma)$ (if they are reachable) as follows:

$$Y_{dis}(\sigma) := \{y = (q, x) \in Y |\ \delta(q, \sigma)! \wedge \neg\xi(x, \sigma)!\}$$

The corresponding predicate $P_{Dis}(\sigma) = pr(Y_{dis}(\sigma))$ is defined as:

$$P_{Dis}(\sigma) = \left(\hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma) \wedge \neg\hat{\xi}^{-1}(true, \sigma)\right),$$

Of course, not every state in $P_{Dis}(\sigma)$ is reachable. We thus have to combine $P_{Dis}(\sigma)$ with $P_{reach} = R(\mathbf{G}\|\mathbf{S}, true)$.

Similarly, from the definition of $Y_{enable}(\sigma)$, in Definition 3.3.2, we know that $Y_{enable}(\sigma)$ maps $\sigma \in \Sigma_c$ to the set of reachable states in $\mathbf{G}\|\mathbf{S}$ with an outgoing σ transition in both $\mathbf{G}$ and $\mathbf{S}$.

We can express all the states that would qualify for $Y_{enable}(\sigma)$ (if they are reachable) as follows:

$$Y_{en}(\sigma) := \{y = (q, x) \in Y |\ \delta(q, \sigma)! \wedge \xi(x, \sigma)!\}$$

The corresponding predicate $P_{En}(\sigma) := pr(Y_{en}(\sigma))$ is defined as:

$$P_{En}(\sigma) = \left(\hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma) \wedge \hat{\xi}^{-1}(true, \sigma)\right),$$

Of course, not every state in $P_{En}(\sigma)$ is reachable. Therefore, we have to combine $P_{En}(\sigma)$ with $P_{reach} = R(\mathbf{G}\|\mathbf{S}, true)$.

5.5 Algorithms

In this section we present the observer-based SB co-observability algorithm using predicates. To make the algorithms easy to understand we have broken the algorithm

into sub-algorithms. The main observer-based algorithm is Algorithm 1, which calls algorithms 2, 3 and 4.

Let $G_i = (Q_i, \Sigma, \delta_i, q_{0,i}, Q_{m,i})$ and $S_j = (X_j, \Sigma, \xi_j, x_{0,j}, X_{m,j})$, for $i \in \{1,2...,n\}$ and $j \in \{1,2...,m\}$ such that $\mathbf{G} = \mathbf{G}_1||\mathbf{G}_2||\ldots||\mathbf{G}_n := (Q, \Sigma, \delta, q_o, Q_m)$ is the plant automaton and $\mathbf{S} = \mathbf{S}_1||\mathbf{S}_2||\ldots||\mathbf{S}_m := (X, \Sigma, \xi, x_o, X_m)$ is the specification automaton, both defined over Σ. We construct automaton $\mathbf{H} = (Y,\ \Sigma,\ \eta,\ y_0,\ Y_m)$ based on the synchronous product of $\mathbf{G}$ and $\mathbf{S}$. For a description on $\mathbf{H}$, refer to Section 4.2.3. The purpose of the algorithm is to determine if the specification automaton $\mathbf{S}$ is SB co-observable w.r.t plant $\mathbf{G}$, P_i and $\Sigma_{c,i}$.

5.5.1 Is SB Co-observable

The main algorithm consists of subroutines *EnableDisableSetFind*, *ConstructSyncProductAndHiding* and *ObserverConstruction*.

The algorithm makes use of certain variables:

I: The set $I := \{1,2,..,n\}$ of controllers where n is the total number of controllers.

Σ_c: Set of controllable events such that $\Sigma_c \subseteq \Sigma$.

$\Sigma_{o,i}$: The set of observable events for each controller i.

H : The automaton based on $\mathbf{G}||\mathbf{S}$, with events $\Sigma \backslash \Sigma_{o,i}$ hidden and replaced with event $\tau \in \Sigma$. As it will be described in Algorithm 9 of Section 5.5.3, $\mathbf{H}$ is represented as a set of state and transition predicates. From the transition predicates, the maps $\hat{\eta}$ and $\hat{\eta}^{-1}$ are defined. For $\mathbf{H}$, these maps are equivalent to the $\hat{\delta}$ and $\hat{\delta}^{-1}$ maps defined

in section 5.3.1

P_{reach} : A predicate that represents the set of reachable states for $\mathbf{G}||\mathbf{S}$.

$P_{Disable}$: It is a map such that $P_{Disable} : \Sigma_c \rightarrow Pred(Y)$. For $\sigma \in \Sigma_c$, $P_{Disable}$ maps σ to a state predicate for $\mathbf{G}||\mathbf{S}$ that represents the states in $Y_{disable}(\sigma)$, as defined in Definition 3.3.1.

P_{Enable} : It is a map such that $P_{Enable} : \Sigma_c \rightarrow Pred(Y)$. For $\sigma \in \Sigma_c$, P_{Enable} maps σ to a state predicate for $\mathbf{G}||\mathbf{S}$ that represents the states in $Y_{enable}(\sigma)$, as defined in Definition 3.3.2

$P_{tempDisable}$: The map $P_{tempDisable} : \Sigma_c \rightarrow Pred(Y)$ is such that, for each $\sigma \in \Sigma_c$, $P_{tempDisable}(\sigma)$ is a predicate that represents all the un-handled $P_{Disable}(\sigma)$ states that are returned from the procedure *SubsetConstruction*, so that the subsequent controllers have to handle only the remaining problem states.

pass, skipIt : Boolean variables used for loop termination.

Algorithm 7:

1: **procedure** IsSBCo-observable($\mathbf{G}, \mathbf{S}$)
2: $\quad P_{reach} \leftarrow R(\mathbf{G}||\mathbf{S}, true)$
3: $\quad$ *EnableDisableSetFind*(P_{Enable}, $P_{Disable}$, $\mathbf{G}$, $\mathbf{S}$, P_{reach})
4: $\quad$ pass $\longleftarrow$ True
5: $\quad$ **for** $\sigma \in \Sigma_c$ **do**
6: $\quad\quad$ **if** ($P_{Disable}(\sigma) \not\equiv$ false) **then**
7: $\quad\quad\quad$ pass $\longleftarrow$ False
8: $\quad\quad$ **end if**
9: $\quad$ **end for**

10: **if** (pass) **then**
11: **return** True
12: **end if**
13: **for** $i \in I$ **do**
14: skipIt $\longleftarrow$ True
15: **for** $\sigma \in \Sigma_c$ **do**
16: **if** $((i \in I_c(\sigma)) \wedge (P_{Disable}(\sigma) \not\equiv \text{false}))$ **then**
17: skipIt $\longleftarrow$ False
18: **end if**
19: **end for**
20: **if** (!skipIt) **then**
21: $\mathbf{H} \longleftarrow \mathit{ConstructSyncProductAndHiding}(\mathbf{G}, \mathbf{S}, \Sigma_{o,i}, P_{reach})$
22: $P_{tempDisable} \longleftarrow \mathit{ObserverConstruction}(\mathbf{H}, P_{Enable}, P_{Disable}, \Sigma_{o,i}, i)$
23: pass $\longleftarrow$ True
24: **for** $\sigma \in \Sigma_c$ **do**
25: **if** $(P_{tempDisable}(\sigma) \not\equiv \text{false})$ **then**
26: pass $\longleftarrow$ False
27: **end if**
28: $P_{Disable}(\sigma) \longleftarrow P_{tempDisable}(\sigma)$
29: **end for**
30: **if** (pass) **then**
31: **return** True
32: **end if**
33: **end if**

34: **end for**

35: **return** False

36: **end procedure**

The algorithm functions in the same way as Algorithm 5 in Chapter 4. We calculate P_{reach} in this algorithm and pass the value to Algorithms 8 and 9 so that we do not have to calculate them over and over. As computing P_{reach} is expensive, this saves us a lot of time and helps to improve efficiency. Lines 5-12 check to see if any states satisfy $P_{disable}(\sigma)$, for every $\sigma \in \Sigma_c$. Absence of any state proves that there are no problem states that are needed to be handled by the controllers. Lines 13-34 process each controller $i \in I$ in turn. Lines 15-19, make sure that controller i can control at least one $\sigma \in \Sigma_c$ so that a disable state can be taken care of. If the above mentioned conditions are met, then we construct the **H** automaton and perform the observer construction test. Otherwise we skip the current controller and check the same condition for subsequent controllers.

For lines 24 - 29, $P_{tempDisable}(\sigma)$ is checked to see if there are any disable states, still to process. If $P_{tempDisable}(\sigma) \equiv false$, for at least one $\sigma \in \Sigma_c$, it implies that procedure *ObserverConstruction* has returned some subset of $P_{Disable}(\sigma)$ states that could not be handled by the active controller. These remaining states are then assigned to $P_{Disable}(\sigma)$ on Line 28.

5.5.2 Enable Disable Set Find

EnableDisableSetFind calculates the predicate maps $P_{Disable}(\sigma)$, $P_{Enable}(\sigma)$ using $P_{Dis}(\sigma)$ and $P_{En}(\sigma)$ defined in section 5.4.1. P_{reach} is used to ensure the states are reachable.

The algorithm uses the following variables:

$\hat{\delta}_{\mathbf{G}}^{-1}$ is the inverse transition predicate function for plant **G**, as defined in Section 5.4.

$\hat{\xi}^{-1}$ is the inverse transition predicate function for specification **S** as defined in Section 5.4.

Algorithm 8:

1: **procedure** EnableDisableSetFind(P_{Enable}, $P_{Disable}$, **G**, **S**, P_{reach})

2: **for** $\sigma \in \Sigma_c$ **do**

3: $P_{Disable}(\sigma) \leftarrow \hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma) \wedge \neg\hat{\xi}^{-1}(true, \sigma) \wedge P_{reach}$

4: $P_{Enable}(\sigma) \leftarrow \hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma) \wedge \hat{\xi}^{-1}(true, \sigma) \wedge P_{reach}$

5: **end for**

6: **end procedure**

5.5.3 Construct Sync Product And Hiding

The algorithm calculates the synchronous product, $\mathbf{G}||\mathbf{S}$, of the plant **G** and the specification automaton **S**, both defined over Σ. As the synchronous product is constructed, we also replace all unobservable events ($\sigma \in \Sigma \backslash \Sigma_{o,i}$) with the special event $\tau \notin \Sigma$. We label the resulting automaton **H**, which may be non-deterministic.

The process of replacing the unobservable events with τ is called hiding. The hiding process could make **H** non-deterministic as we might end up with multiple τ transitions leaving a given state. To handle large systems, we use transition predicates as defined in Section 5.2.2.

The algorithm makes use of following set of variables:.

P_{y_m}: A predicate that represents the reachable, marked states of $\mathbf{G}||\mathbf{S}$.

$pr(\{y_0\})$: It takes the state y_0 and converts it to a predicate. This predicate is true only for state y_0, the initial state of $\mathbf{G}||\mathbf{S}$.

N_ω: Transition predicate, $N_\omega = Y \times Y \to \{T, F\}$, is for $\mathbf{G}||\mathbf{S}$ and event $\omega \in \Sigma$.

N_σ: Transition predicate, $N_\sigma = Y \times Y \to \{\text{T, F}\}$, is for $\mathbf{G}||\mathbf{S}$ and event $\sigma \in \Sigma \backslash \Sigma_{o,i}$.

N_τ: Transition predicate, $N_\tau = Y \times Y \to \{\text{T, F}\}$, is for $\mathbf{H}$ and event $\tau \notin \Sigma$. This represents the $\sigma \in \Sigma \backslash \Sigma_{o,i}$ events that were hidden.

P_m: A predicate which represents the marked states of $\mathbf{G}||\mathbf{S}$. These states may not all be reachable.

P_Y: This predicate represents the reachable states of $\mathbf{G}||\mathbf{S}$.

P_{y_0}: The predicate that represents $\{y_0\}$, the initial state of $\mathbf{G}||\mathbf{S}$.

N: This is the set of transition predicates for $\mathbf{H}$. It includes the predicate for τ and $\sigma \in \Sigma_{o,i}$.

Algorithm 9:

1: **procedure** CONSTRUCTSYNCPRODUCTANDHIDING($\mathbf{G}$, $\mathbf{S}$, $\Sigma_{o,i}$, P_{reach})

2: $P_{y_0} \leftarrow pr(\{y_0\})$

3: For each $\omega \in \Sigma$, construct N_ω as described in Definition 5.2.3.

4: $N_\tau \longleftarrow \bigvee_{\sigma \in \Sigma \backslash \Sigma_{o,i}} N_\sigma$

5: $P_{Y_m} \longleftarrow P_{reach} \wedge P_m$

6: $N \longleftarrow \{N_\tau\} \cup \left(\bigcup_{\omega \in \Sigma_{o,i}} \{N_\omega\} \right)$

7: $P_Y \leftarrow P_{reach}$

8: return $(P_Y, N, P_{y_0}, P_{Y_m})$

9: **end procedure**

On Line 3, we construct transition predicates N_ω, for each $\omega \in \Sigma$. On Line 4, we construct the transition preidcate N_τ so that it represents the transitions for all $\sigma \in \Sigma \backslash \Sigma_{o,i}$, the unobservable events. We then discard the transition predicate N_σ, for each $\sigma \in \Sigma \backslash \Sigma_{o,i}$, as we no longer need them.

For the construction of **H**, we keep the transition predicates that correspond to $\omega \in \Sigma_{o,i}$. **H** consists of P_{y_0}, P_{Y_m}, P_Y, and N.

5.5.4 Observer Construction

The algorithm performs the SB co-observability verification by constructing all state predicate pairs (P_{y1}, P_{y2}) where $((y_1, y_2) \in Y \times Y)$ such that y_1, y_2 are indistinguishable with respect to $\Sigma_{o,i}$ (see definition of indistinguishability, Definition 3.3.3). For a given $\sigma \in \Sigma_c$ where $i \in I_c(\sigma)$, the system fails the SB co-observability verification test if state $y \vDash P_{Disable}(\sigma)$ is paired with state $y' \vDash P_{Enable}(\sigma)$, which means that the two state pairs are indistinguishable w.r.t $\Sigma_{o,i}$. State y is added to $P_{tmpDisable}(\sigma)$ so that we can test if the rest of the controllers can handle these states.

The use of state predicates and transition predicates will decrease computation time and memory usage. This will help the algorithm to scale better.

The algorithm makes use of the following variables:

P_{y_0} : Predicate that represents the initial state of **H**.

$\hat{\eta}$: Transition predicate function for **H**, as defined in section 5.3.1.

Found: Set of state pairs that represents the state tuples that have already been encountered but not necessarily processed. We have $Found \subseteq Pred(Y) \times Pred(Y)$.

Pending: Set of state pairs that have been encountered and are waiting to be processed. We have $Pending \subseteq Pred(Y) \times Pred(Y)$.

$P_{tmpDisable}$: The map $P_{tmpDisable} : \Sigma_c \rightarrow Pred(Y)$ is such that for each $\sigma \in \Sigma_c$, $P_{tmpDisable}(\sigma)$ is a predicate which represents all of the un-handled $P_{Disable}(\sigma)$ states which, at some point, got paired with $P_{Enable}(\sigma)$ states, that the active controller could not handle. These are the states that are returned to procedure *IsSBCo-observable*, so that the subsequent controllers have to handle only these problem states.

NOTE: Before defining the algorithm, we would like to specify that *Found* has been defined in the algorithm as a set with elements being added and removed in the form of state tuples. However, it will be implemented as a transition predicate, which take two states as input and returns T or F. This will make the algorithm scale better.

Algorithm 10:

1: **procedure** OBSERVERCONSTRUCTION($\mathbf{H}$, P_{Enable}, $P_{Disable}$, $\Sigma_{o,i}$, i)

2:     **for** $\sigma \in \Sigma_c$ **do**

3:         **if** $(i \in I_c(\sigma))$ **then**

4:             $P_{tmpDisable}(\sigma) \longleftarrow$ False

5:         **else**

6:             $P_{tmpDisable}(\sigma) \longleftarrow P_{Disable}(\sigma)$

7: **end if**

8: **end for**

9: $Found \longleftarrow \{(pr\{y_0\}, pr\{y_0\})\}$

10: $Pending \longleftarrow \{(pr\{y_0\}, pr\{y_0\})\}$

11: **while** $(Pending \neq \varnothing)$ **do**

12: Select $(P_{y1}, P_{y2}) \in Pending$

13: $Pending \longleftarrow Pending \setminus \{(P_{y1}, P_{y2})\}$

14: **for** $\sigma \in \Sigma_{o,i}$ **do**

15: **if** $(\hat{\eta}(P_{y_1}, \sigma) \not\equiv false) \wedge (\hat{\eta}(P_{y_2}, \sigma) \not\equiv false)$ **then**

16: $P_{y1}' \longleftarrow \hat{\eta}(P_{y_1}, \sigma)$

17: $P_{y2}' \longleftarrow \hat{\eta}(P_{y_2}, \sigma)$

18: **if** $((P_{y1}', P_{y2}') \notin Found)$ **then**

19: **for** $\sigma' \in \Sigma_c$ **do**

20: **if** $((((P_{y1}' \wedge P_{Disable}(\sigma') \not\equiv \text{false}) \wedge (P_{y2}' \wedge P_{Enable}(\sigma') \not\equiv \text{false})) \vee ((P_{y1}' \wedge P_{Enable}(\sigma') \not\equiv \text{false}) \wedge (P_{y2}' \wedge P_{Disable}(\sigma') \not\equiv \text{false}))) \wedge (i \in I_c(\sigma')))$ **then**

21: **if** $(P_{y1}' \wedge P_{Disable}(\sigma') \not\equiv \text{false})$ **then**

22: $P_{tmpDisable}(\sigma') \longleftarrow P_{tmpDisable}(\sigma') \vee P_{y1}'$

23: **else**

24: $P_{tmpDisable}(\sigma') \longleftarrow P_{tmpDisable}(\sigma') \vee P_{y2}'$

25: **end if**

26: **end if**

27: **end for**

28: $Found \longleftarrow Found \cup \{(P_{y1}', P_{y2}')\}$

29: $\quad\quad\quad\quad\quad\quad Found \longleftarrow Found \cup \{(P_{y2}', P_{y1}')\}$

30: $\quad\quad\quad\quad\quad\quad Pending \longleftarrow Pending \cup \{(P_{y1}', P_{y2}')\}$

31: **end if**

32: **end if**

33: **end for**

34: $P_{Y'} \longleftarrow \hat{\eta}(P_{y_1}, \tau)$

35: **for** $(y \vDash P_{Y'})$ **do**

36: $P_y \longleftarrow pr(\{y\})$

37: **if** $((P_y, P_{y2}) \notin Found)$ **then**

38: **for** $\sigma \in \Sigma_c$ **do**

39: **if** $((((P_y \wedge P_{Disable}(\sigma) \not\equiv \text{false}) \wedge (P_{y2} \wedge P_{Enable}(\sigma) \not\equiv \text{false})) \vee ((P_y \wedge P_{Enable}(\sigma) \not\equiv \text{false}) \wedge (P_{y2} \wedge P_{Disable}(\sigma) \not\equiv \text{false}))) \wedge (i \in I_c(\sigma)))$ **then**

40: **if** $(P_y \wedge P_{Disable}(\sigma) \not\equiv \text{false})$ **then**

41: $P_{tmpDisable}(\sigma) \longleftarrow P_{tmpDisable}(\sigma) \vee P_y$

42: **else**

43: $P_{tmpDisable}(\sigma) \longleftarrow P_{tmpDisable}(\sigma) \vee P_{y2}$

44: **end if**

45: **end if**

46: **end for**

47: $Found \longleftarrow Found \cup \{(P_y, P_{y2})\}$

48: $Found \longleftarrow Found \cup \{(P_{y2}, P_y)\}$

49: $Pending \longleftarrow Pending \cup \{(P_y, P_{y2})\}$

50: **end if**

51: **end for**

52: $P_{Y'} \longleftarrow \hat{\eta}(P_{y_2}, \tau)$

53: **for** $(y \models P_{Y'})$ **do**

54: $P_y \longleftarrow pr(\{y\})$

55: **if** $((P_{y1}, P_y) \notin Found)$ **then**

56: **for** $\sigma \in \Sigma_c$ **do**

57: **if** $((((P_{y1} \wedge P_{Disable}(\sigma) \not\equiv \text{false}) \wedge (P_y \wedge P_{Enable}(\sigma) \not\equiv \text{false})) \vee ((P_{y1} \wedge P_{Enable}(\sigma) \not\equiv \text{false}) \wedge (P_y \wedge P_{Disable}(\sigma) \not\equiv \text{false}))) \wedge (i \in I_c(\sigma)))$ **then**

58: **if** $(P_{y1} \wedge P_{Disable}(\sigma) \not\equiv \text{false})$ **then**

59: $P_{tmpDisable}(\sigma) \longleftarrow P_{tmpDisable}(\sigma) \vee P_{y1}$

60: **else**

61: $P_{tmpDisable}(\sigma) \longleftarrow P_{tmpDisable}(\sigma) \vee P_y$

62: **end if**

63: **end if**

64: **end for**

65: $Found \longleftarrow Found \cup \{(P_{y1}, P_y)\}$

66: $Found \longleftarrow Found \cup \{(P_y, P_{y1})\}$

67: $Pending \longleftarrow Pending \cup \{(P_{y1}, P_y)\}$

68: **end if**

69: **end for**

70: **end while**

71: **return** $P_{tmpDisable}$

72: **end procedure**

The algorithm works in the same way as the Algorithm 6 described in Section 4.2.6. Please see the description there. The structure here is very similar to the one

described in Chapter 4 but we use predicates to improve scalability.

We note here that on lines 16-17, $P_{y1'}$ and $P_{y2'}$, each represent a predicate for a single state because the system is initially deterministic. It is only the τ transitions that make the system non-deterministic.

5.6 BDD Complexity

The representation of logic formulas that we have discussed so far can be performed by using Binary Decision Diagrams. The *BuDDy* package will be used for implementation. Huth *et al.* [14] is a good source of information on BDDs and their applications.

Ordering of state variables in BDD can change the number of BDD nodes. The ordering of the BDD nodes play a major role in improving the efficiency of the BDD operations. Algorithms defined in [34] are used for optimized ordering of variables and thus reduce computation.

5.7 Advantages of Using Predicates

Using predicates for a small example might take the same amount of time as using non-predicate-based algorithms. However, predicates are generally used for systems that have a large number of states. Predicate-based algorithms have better scalability as compared to non-BDD based algorithms, where we explicitly determine the states and the transitions. Here we present a few advantages that predicate-based algorithms provide as compared to non-predicate-based algorithms. We refer only to advantages that relate to our algorithms.

▸ The evaluation of $P_{Disable}(\sigma)$ and $P_{Enable}(\sigma)$ (for $\sigma \in \Sigma_c$) in Algorithm 8, *EnableDisableSetFindUsingPredicates* in Section 5.5.2, prevents the system from going through every state to find out the states that belong to enable and disable sets. Predicates allow us to evaluate the set information in a direct fashion. If a state belongs to $Y_{disable}(\sigma)$ then the $P_{Disable}(\sigma)$ returns T for that state. In other words, a predicate does not explicitly lists states, it only contains the ability to determine membership.

▸ Not only does the predicate help to store the information in a compact form but it is also a very efficient way to calculate the set.

▸ Since Algorithm 10, *ObserverConstruction*, defined in Section 5.5.4, requires us to know the transitions of $\mathbf{G}||\mathbf{S}$, we represent the synchronous product as transition predicates which helps to represent the behaviour of a large system efficiently in terms of memory as well as look-up time.

▸ In Algorithm 10, *ObserverConstruction*, the tuples are stored as transition predicates. Thus, instead of finding a transition between two states, the predicate returns T for those two states if they are indistinguishable. This also helps to handle a large system.

▸ At lines 34 and 52 of procedure *ObserverConstruction*, a large number of τ transitions can be evaluated at once while constructing $P_{Y'}$. Since we combine all the τ transitions together, there will be single look-up operation instead of large number of look-ups.

Chapter 6

Parallel Manufacturing Example

We demonstrate our approach of verifying co-observability by applying it to a Hierarchical Interface-based Supervisory Control (HISC) manufacturing example from [19]. We are using the HISC example because it already contains a partitioning of enablement of events and observability of events that seemed like they could be cast as a decentralized control problem. In this chapter we convert the HISC modeled system to a flat (non-hierarchical) project and then convert this to a decentralized control application.

A few figures from [19] have been used with the authors permission, and will be cited when used.

6.1 Introduction

The manufacturing system shown in Figure 6.1 consists of three manufacturing units operating in parallel, followed by a testing unit and a packaging unit. The system also contains four 4-slot buffers located as shown in Figure 6.1. Each manufacturing

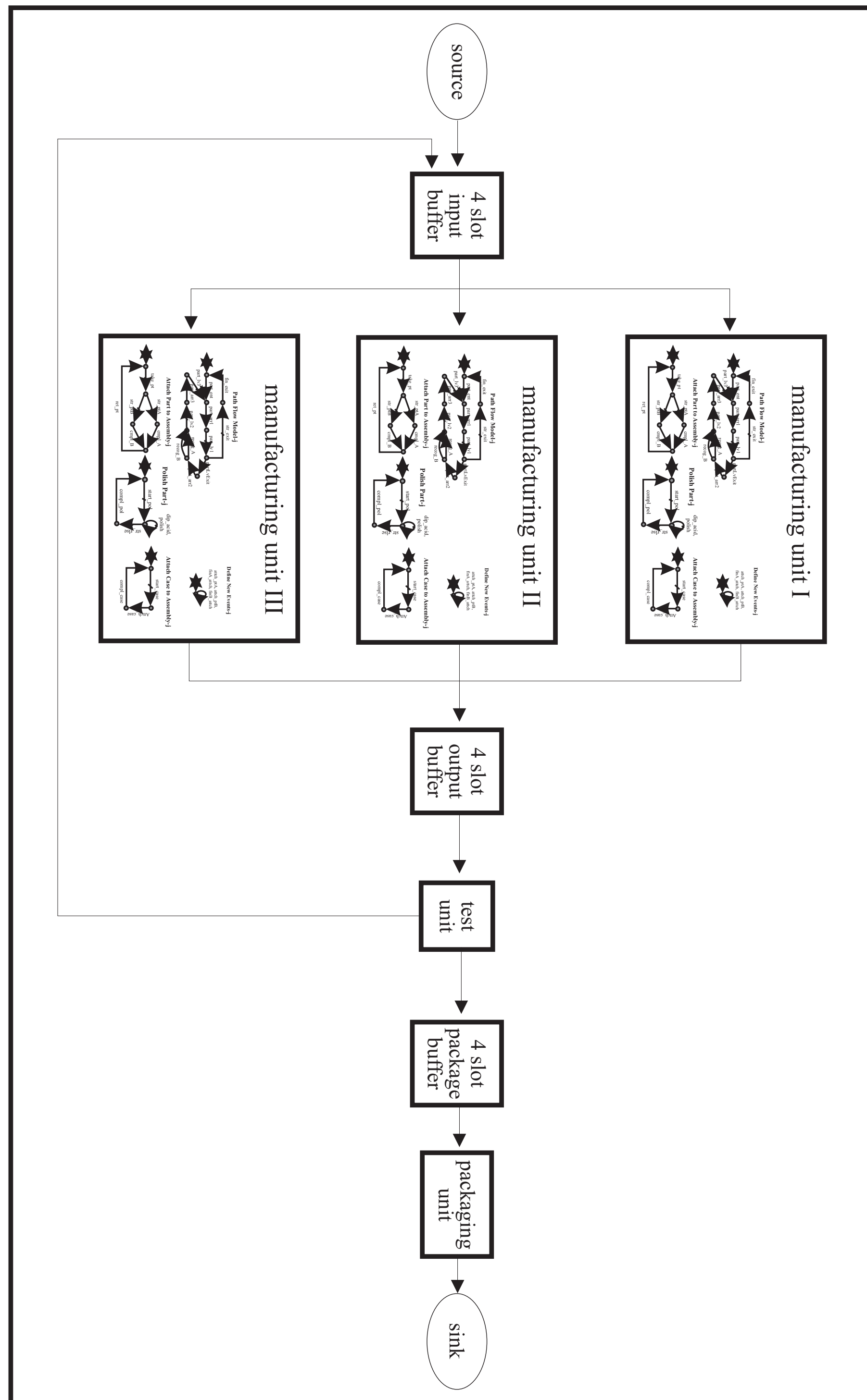


Figure 6.1: Block Diagram of Parallel Plant [19]

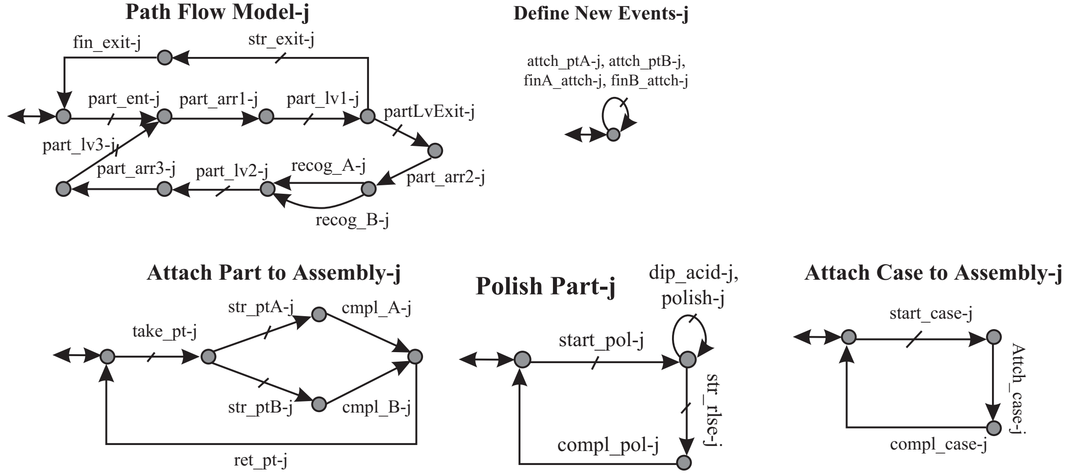


Figure 6.2: Plant Models for Manufacturing unit j [19]

unit consists of three cells connected by a conveyor belt. Cell one polishes the part, cell two attaches a part of type A or B, and cell three attaches a case to the assembly. For more information on the physical set up of the manufacturing unit refer to [19]. Figure 6.2 shows the plant models for the jth manufacturing unit, $j =$ I, II, III.

In [19], Leduc developed an HISC model of the system, which consists of a high-level, and three low-levels. Figure 6.4, shows the full HISC model of the system. Figure 6.3 shows low-level for $j =$ I in more detail. The other low-levels are identical upto relabeling. For an explanation of the plant and the components, please see [19].

An HISC system partitions the system event set into high-level events (Σ_H), low-level events (Σ_{L_j}), request events (Σ_{R_j}) and answer events (Σ_{A_j}), $j =$ I, II, III. We thus have the system partition:

$\Sigma := [\dot\cup_{k \in \{\text{I,II,III}\}} (\Sigma_{L_k} \dot\cup \Sigma_{R_k} \dot\cup \Sigma_{A_k})] \;\dot\cup\; \Sigma_H$

Low Level Subsystem $_{I}$

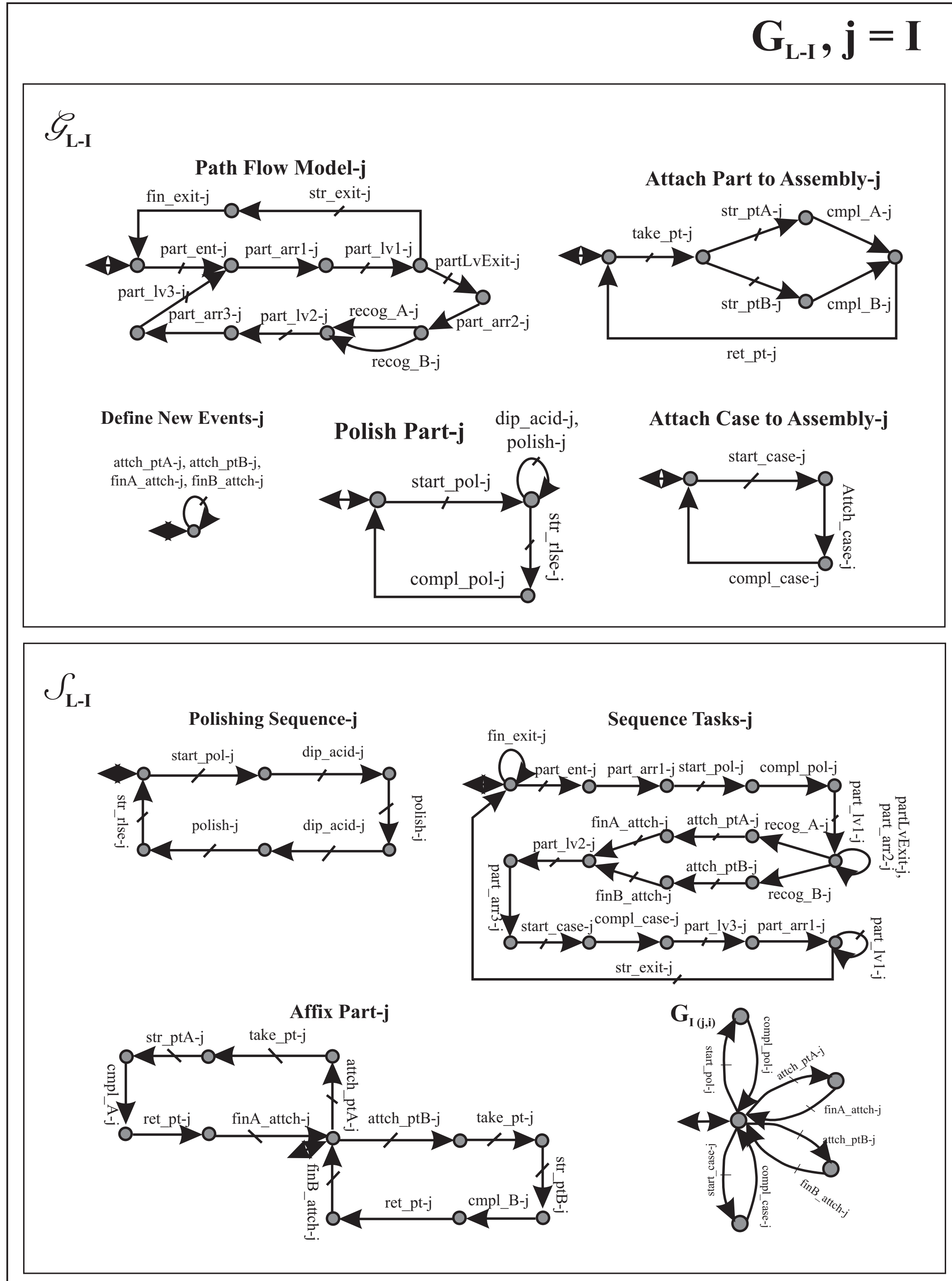
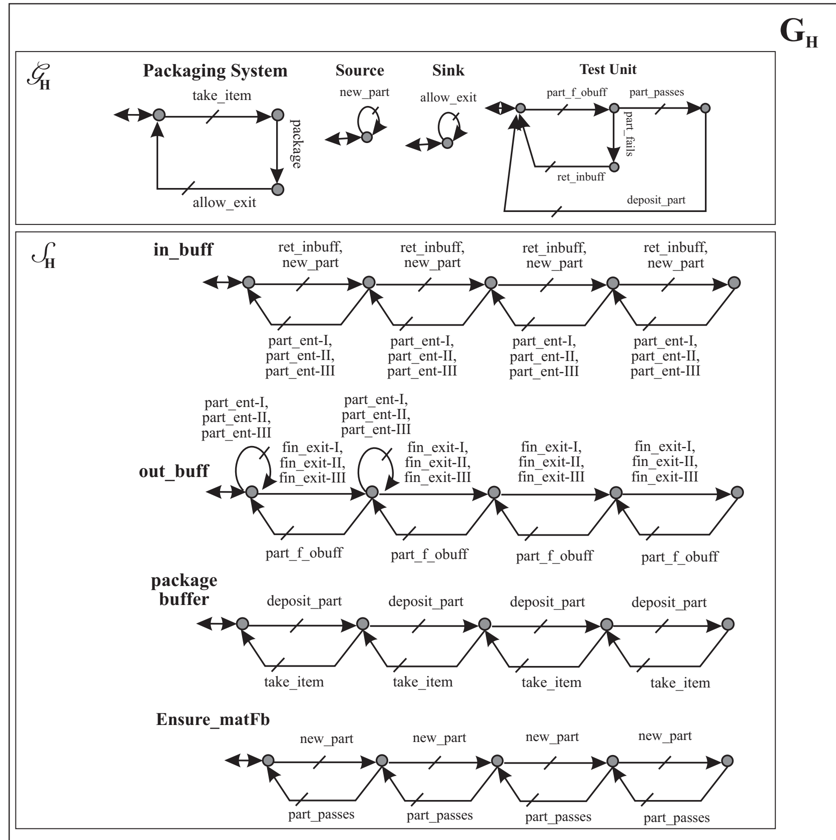


Figure 6.3: Low-Level system [19]

High level Subsystem



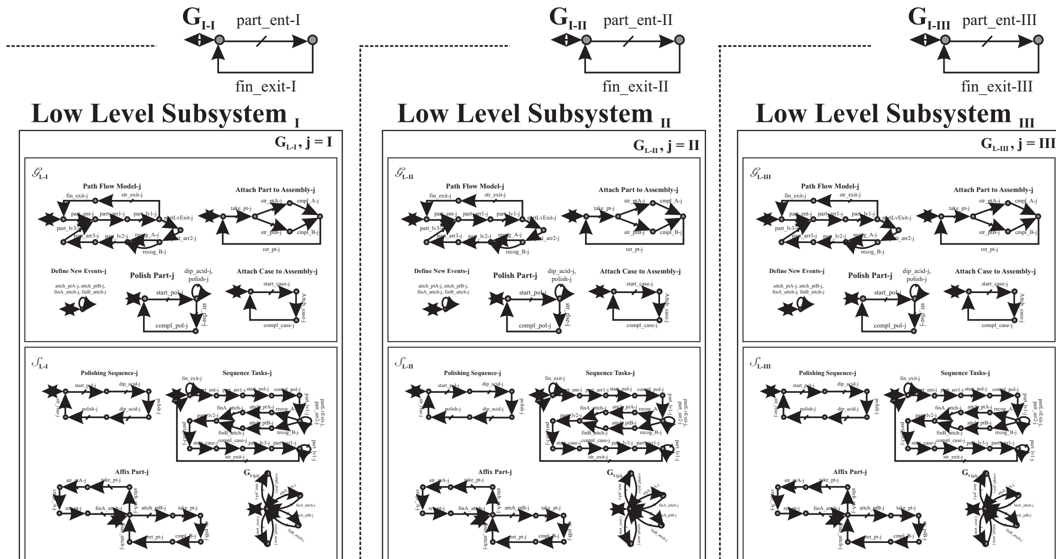


Figure 6.4: Complete Parallel Manufacturing System [19]

We assume the standard partition $\Sigma = \Sigma_c \dot\cup \Sigma_{uc}$ for controllable and uncontrollable events. For this example our HISC event sets are defined to be:

$$
\begin{aligned}
\Sigma_H &= \{take_item,\ package,\ allow_exit,\ new_part,\ part_f_obuff,\ part_passes,\ part_fails,\\
&\quad ret_inbuff,\ deposit_part\}\\
\Sigma_{R_j} &= \{part_ent\text{-}j\}\\
\Sigma_{A_j} &= \{fin_exit\text{-}j\}\\
\Sigma_{L_j} &= \{start_pol\text{-}j,\ attch_ptA\text{-}j,\ attch_ptB\text{-}j,\ start_case\text{-}j,\ comp_pol\text{-}j,\ finA_attch\text{-}j,\ finB_attch\text{-}j,\\
&\quad compl_case\text{-}j,\ part_arr1\text{-}j,\ part_lv1\text{-}j,\ partLvExit\text{-}j,\ str_exit\text{-}j,\ part_arr2\text{-}j,\ recog_A\text{-}j,\ recog_B\text{-}j,\\
&\quad part_lv2\text{-}j,\ part_arr3\text{-}j,\ part_lv3\text{-}j,\ take_pt\text{-}j,\ str_ptA\text{-}j,\ str_ptB\text{-}j,\ compl_A\text{-}j,\ compl_B\text{-}j,\\
&\quad ret_pt\text{-}j,\ dip_acid\text{-}j,\ polish\text{-}j,\ str_rlse\text{-}j,\ attch_case\text{-}j\}
\end{aligned}
$$

We first note that the HISC method breaks the system into four groups of DES, one for each level. Further, the DES at each level are only allowed to contain certain types of events. The high-level DES contains only the high-level, request and answer events. The jth low-level DES contains only the jth low-level events, and the jth request and answer events. This suggests that we can cast this example into a decentralized problem with four controllers (one per component), and match the observable and controllable events for the controller to match the events that the DES at a given level is defined over.

6.2 Casting an HISC as a Decentralized Control Problem

To verify if the given example is co-observable, we need to convert the *HISC* project into a *flat* project.

We now present the logic that we have used to transform the HISC modeled manufacturing system into a flat project:

▸ We need to implement decentralized controllers instead of dividing the entire project into various levels. Since we had three low-level subsystems and one high-level subsystem, we will create four controllers.

▸ Instead of events being categorized as high-level, low-level, answer and request events, we instead label them as being observable/controllable or not for a given decentralized controller.

▸ The observable events for controller 1 will be all high-level events along with all the interface events (i.e., all the request and answer events).

▸ The controllable events for controller 1 will be all the controllable events that observer 1 is able to observe, i.e., $\Sigma_{c,1} = \Sigma_{o,1} \cap \Sigma_c$.

▸ The observable events for controllers 2,3,4, will be all the jth low-level, request and answer events, for $j =$ I, II, III, respectively. For example, $\Sigma_{o,2} = \Sigma_{L_I} \cup \Sigma_{R_I} \cup \Sigma_{A_I}$.

▸ The controllable events for controllers 2,3,4, will be the controllable events that they can observe. For example, $\Sigma_{c,2} = \Sigma_{o,2} \cap \Sigma_c$.

▸ We note that in *HISC*, interface DES are DES supervisors.

▸ Our *flat* plant model is the synchronous product of the high-level and all of the low-level plant components. These are shown in Figure 6.5.

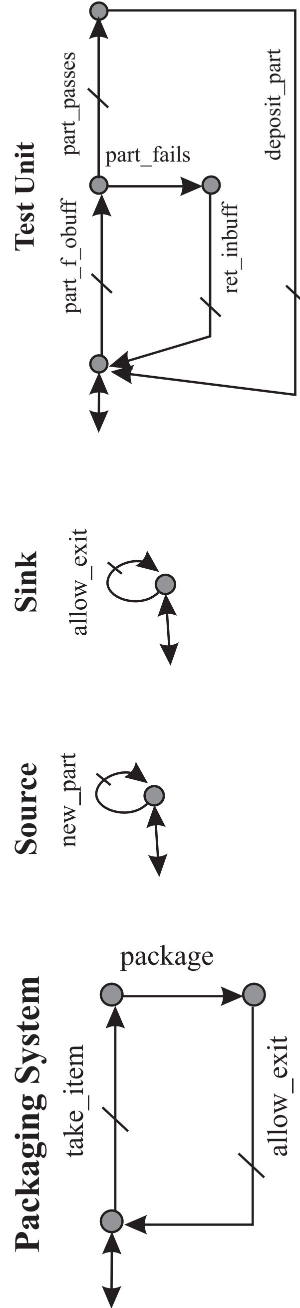
▸ Our *flat* specification is the synchronous product of the high-level supervisor, all of the low-level supervisors, and all of the interface DES. These are shown in Figure 6.6.

There are four controllers. Here are the sets of controllable and observable events for each one.

Controller 1:

$$\begin{aligned}\Sigma_{o,1} &= \{\textit{take_item}, \textit{package}, \textit{allow_exit}, \textit{new_part}, \textit{part_f_obuff}, \textit{part_passes}, \textit{part_fails},\\ &\quad \textit{ret_inbuff}, \textit{deposit_part}, \textit{part_ent-I}, \textit{fin_exit-I}, \textit{part_ent-II}, \textit{fin_exit-II},\\ &\quad \textit{part_ent-III}, \textit{fin_exit-III}\}\\ \Sigma_{c,1} &= \{\textit{take_item}, \textit{allow_exit}, \textit{new_part}, \textit{part_f_obuff}, \textit{part_passes},\\ &\quad \textit{ret_inbuff}, \textit{deposit_part}, \textit{part_ent-I}, \textit{part_ent-II}, \textit{part_ent-III}\}\end{aligned}$$

The rest of the three controllers will have event sets identical up-to relabeling. For controller i = 2,3,4, we will use j = I, II, III, respectively in the definition of their observable and controllable event sets. For example, we will use j = I with controller $i = 2$.



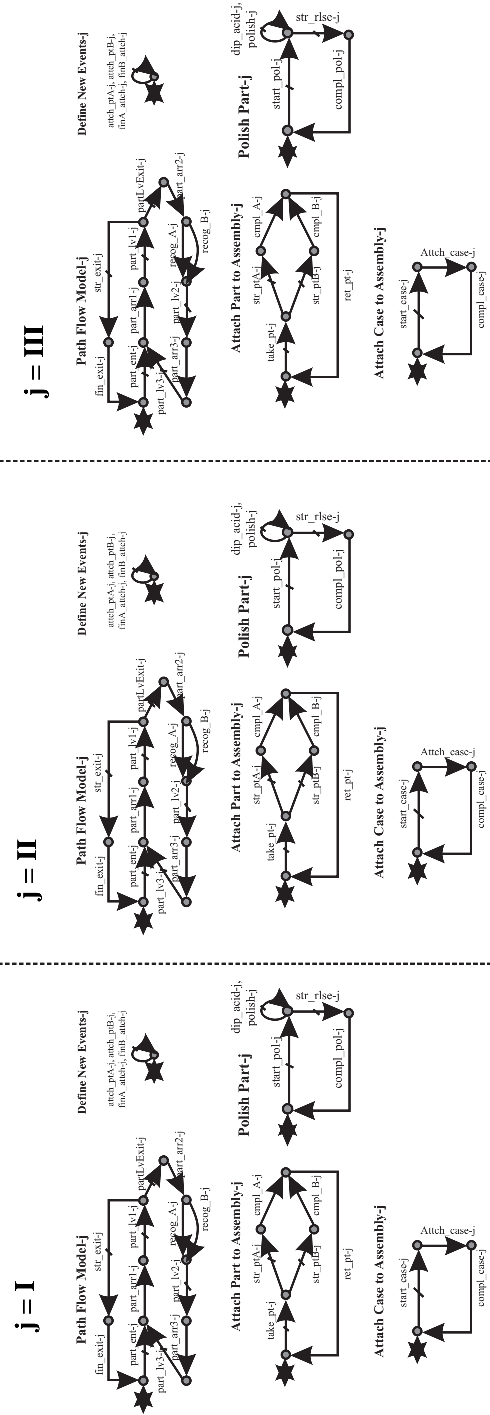


Figure 6.5: Flat Plant Model for Parallel Manufacturing System

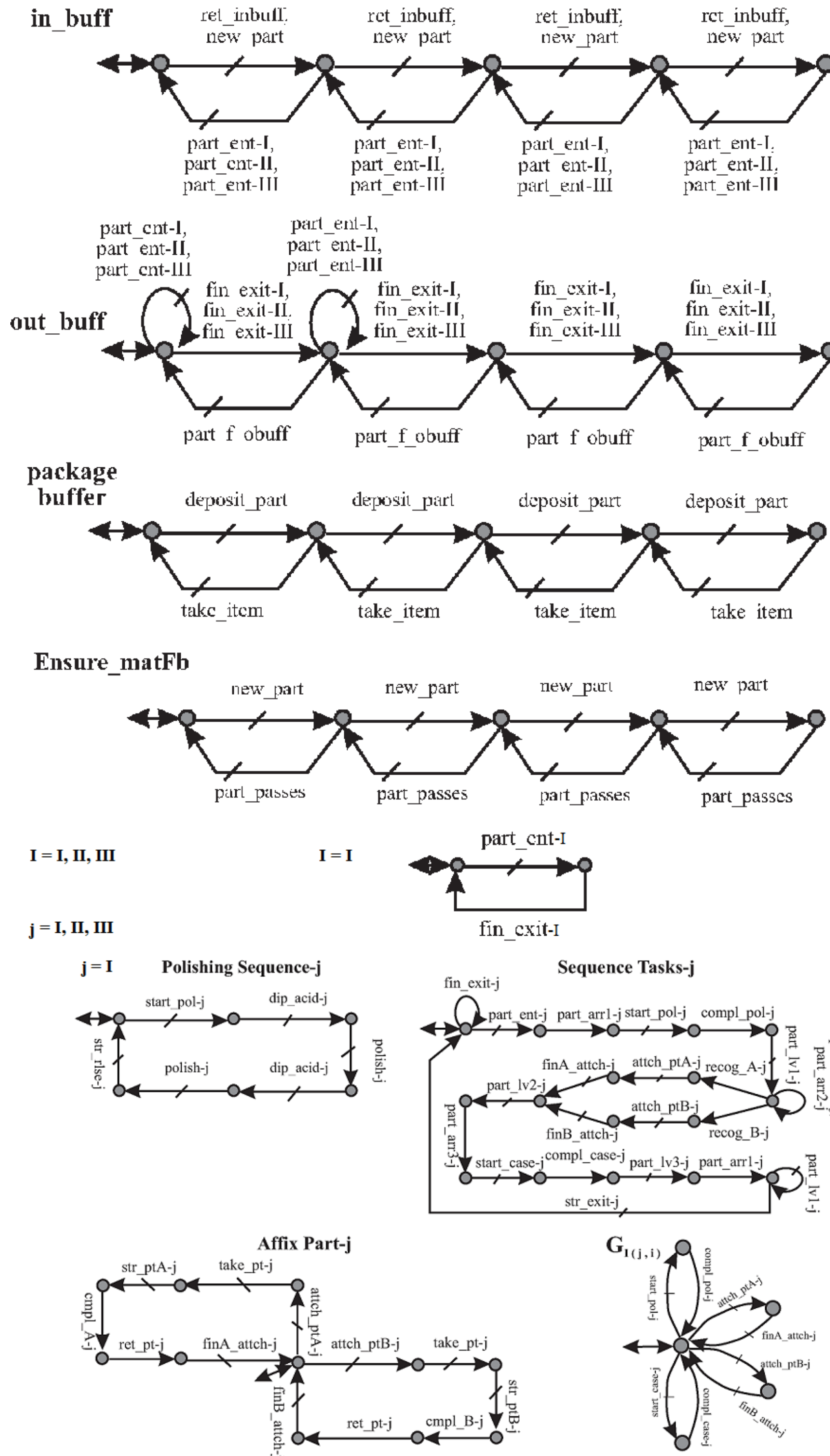


Figure 6.6: Flat Supervisor Model for Parallel Manufacturing System

Controller i :

$\Sigma_{o,i}$ = {*start_pol-i*, *attch_ptA-i*, *attch_ptB-i*, *start_case-i*, *comp_pol-i*, *finA_attch-i*, *finB_attch-i*, *compl_case-i*, *part_arr1-i*, *part_lv1-i*, *partLvExit-i*, *str_exit-i*, *part_arr2-i*, *recog_A-i*, *recog_B-i*, *part_lv2-i*, *part_arr3-i*, *part_lv3-i*, *take_pt-i*, *str_ptA-i*, *str_ptB-i*, *compl_A-i*, *compl_B-i*, *ret_pt-i*, *dip_acid-i*, *polish-i*, *str_rlse-i*, *attch_case-i*, *fin_exit-i*, *part_ent-i*}

$\Sigma_{c,i}$ = {*start_pol-i*, *attch_ptA-i*, *attch_ptB-i*, *start_case-i*, *finA_attch-i*, *finB_attch-i* *part_lv1-i*, *partLvExit-i*, *str_exit-i*, *part_lv2-i*, *part_lv3-i*, *take_pt-i*, *str_ptA-i*, *str_ptB-i* *dip_acid-i*, *polish-i*, *str_rlse-i*, *part_ent-i*}

6.3 Results

We have extended the graphical DES software research, *DESpot* [11], to allow the user to specify the needed information to define a decentralized control problem. *DESpot* now allows the user to add the desired number of decentralized controllers to a flat project and then specify, for each controller, its set of observable and controllable events. *DESpot* can also read and save this information to the project file.

We have begun implementing the BDD algorithms from Chapter 5, for verifying that a given system is SB co-observable, but due to time constraints, this is not yet complete. We hope to complete this in the near future.

Below is a screen shot of the decentralized editor in DESpot [11] and an example project of the new file format to store the observer information. This is an extension of the DESpot file format.

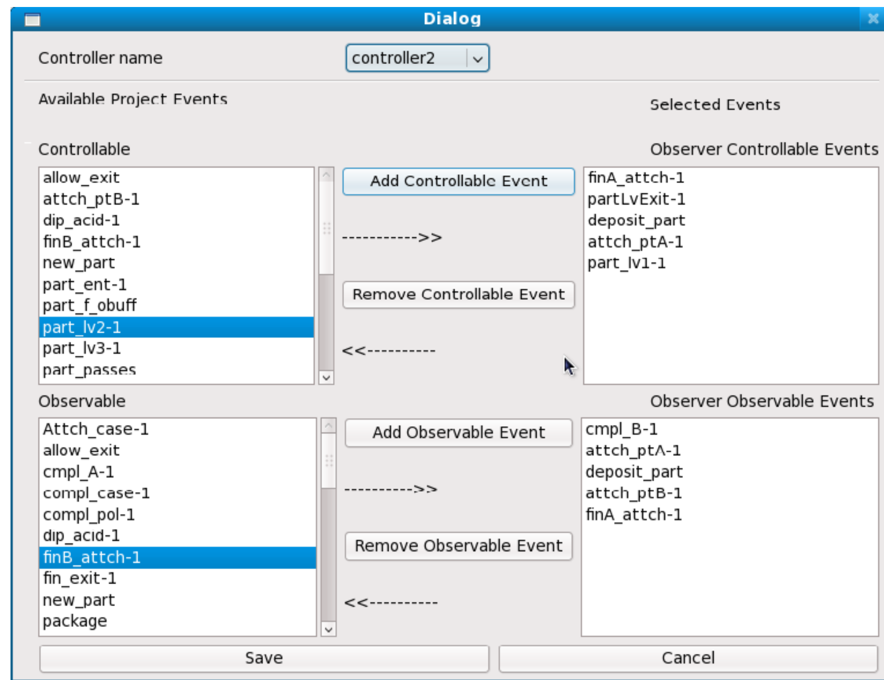


Figure 6.7: Screen Shot of the Decentralized editor

File Format

1: <?xml version="1.0" encoding="UTF-8"?>

2: <Project projVer="1.0.0">

3: <Header name="pmeflatlwlvl23" type="flat">

4: <Integrity status="not-verified" date-stamp=" "/>

5: <Nonblocking status="not-verified" date-stamp=" "/>

6: <Controllable status="not-verified" date-stamp=" "/>

7: </Header>

8: <Subsystem name="pmeflatlwlvl23">

```
9:      <Supervisor>
10:         <Des name=“Ensure” location=“Ensure.des”/>
11:         <Des name=“InBuf” location=“InBuf.des”/>
12:         <Des name=“Intf-1” location=“Intf-1.des”/>
13:         <Des name=“LowIntf-1” location=“LowIntf-1.des”/>
14:         <Des name=“OutBuf” location=“OutBuf.des”/>
15:         <Des name=“PackBuf” location=“PackBuf.des”/>
16:         <Des name=“PolishSeq-1” location=“PolishSeq-1.des”/>
17:     </Supervisor>
18:     <Plant>
19:         <Des name=“AttachCase-1” location=“AttachCase-1.des”/>
20:         <Des name=“AttachPart-1” location=“AttachPart-1.des”/>
21:         <Des name=“NewEvents-1” location=“NewEvents-1.des”/>
22:         <Des name=“PackSys” location=“PackSys.des”/>
23:         <Des name=“PathFlow-1” location=“PathFlow-1.des”/>
24:         <Des name=“PolishPart-1” location=“PolishPart-1.des”/>
25:         <Des name=“Sink” location=“Sink.des”/>
26:         <Des name=“Source” location=“Source.des”/>
27:         <Des name=“TestUnit” location=“TestUnit.des”/>
28:     </Plant>
29:     <Observer name=“controller2”>
30:         <ObservableList>
31:             <Event name=“attch_ptA-1”/>
32:             <Event name=“attch_ptB-1”/>
```

```
33:                 <Event name="cmpl_B-1"/>
34:                 <Event name="deposit_part"/>
35:                 <Event name="finA_attch-1"/>
36:             </ObservableList>
37:             <ControllableList>
38:                 <Event name="attch_ptA-1"/>
39:                 <Event name="deposit_part"/>
40:                 <Event name="finA_attch-1"/>
41:                 <Event name="partLvExit-1"/>
42:                 <Event name="part_lv1-1"/>
43:             </ControllableList>
44:         </Observer>
45:     </Subsystem>
46: </Project>
```

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis we have presented a state-based definition of co-observability, called SB co-observability, in contrast to the classical language-based definition. We have also shown that the state-based definition implies the language-based definition but not the reverse.

We have presented an algorithm to verify SB co-observability using the concept of observers. We have shown that the computational complexity of SB co-observability algorithm is $O\big(n|\Sigma||Y|^2\big)$, where n is the number of controllers, $|\Sigma|$ is the number of events, and $|Y|$ is the number of states. The SB co-observability algorithm is linear in the number of controllers and quadratic in the system's statespace. This compares very favorably to existing language-based algorithms that are exponential in the number of controllers. However, the above is a worst case analysis. We still need to apply both algorithms to real-life examples to ensure that we see this savings in practice.

To improve scalability, we presented BDD-based algorithms for SB co-observability. We expect this to allow us to verify much larger systems. We have added the ability to specify decentralized controllers to the software tool *DESpot*, but have not finished implementing the BDD algorithms due to time constraints.

Finally, we have discussed a large decentralized control example of a manufacturing system which has been transformed into a flat system from an HISC system. The events have been re-organized to specify the observable and controllable event set for each controller. Unfortunately, we have not been able to verify the example yet as the BDD algorithms are not yet completed.

7.2 Future Work

Once our BDD-based SB co-observability algorithm has been implemented, it would be useful to have a non-BDD-based algorithm for comparision. We would also like to develop more large, decentralized examples for comparison.

It would also be useful to develop a BDD-based algorithm to verify language-based co-observability, as SB co-observability is only a sufficient condition. It would also be useful to compare the scalability of the two algorithms.

Bibliography

[1] Martin Fabian Arash Vahidi, Bengt Lennartson. Efficient analysis of large discrete-event systems with binary decision diagrams. *44th IEEE Conference on Decision and Control*, pages 2751– 2756, December 2005.

[2] Dennis S. Arnon. Bibliography of quantifier elimination for real closed fields. *J. Symb. Comput.*, 5(1-2):267–274, 1988.

[3] George Barrett and Stéphane Lafortune. Decentralized supervisory control with communicating controllers. *IEEE Transactions on Automatic Control*, 45(9):1620 – 1638, September 2000.

[4] Milind Borkar, Volkan Cevher, and James H. McClellan. Decentralized state initialization with delay compensation for multi-modal sensor networks. *Journal of Vlsi Signal Processing Systems for Signal Image and Video Technology*, 48(1-2):109 – 125, August 2007.

[5] Al E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992.

[6] R. E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677 – 691, August 1986.

[7] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd edition, 2007.

[8] H. Chakib and A. Khoumsi. Multi-decision supervisory control: Parallel decentralized architectures cooperating for controlling discrete event systems. *IEEE Trans. Autom. Control*, 56(11):2608 – 2622, November 2011.

[9] Randy Cieslak, C. Desclaux, Ayman S Fawaz, and Pravin Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249 – 260, March 1988.

[10] Pengcheng Dai. Synthesis method for hierarchical interface-based supervisory control. Master's thesis, Department of Computing and Software McMaster University, April 2006.

[11] DESpot. `www.cas.mcmaster.ca/~leduc/DESpot.html`. The official website for the DESpot project, 2013.

[12] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory , Languages and Computation*. Addison-Wesley, 1979.

[13] Ying Huang, Karen Rudie, and Feng Lin. Decentralized control of discrete-event systems when supervisors observe particular event occurrences. *IEEE Transactions on Automatic Control*, 53(1):384–388, June 2008.

[14] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, August 2004.

[15] S. Kapralov and V. Dyankavo. Modeling a system with discrete events. *Computer Modeling and Simulation (EMS)*, pages 167 – 172, November 2012.

[16] P. Kozák and W.Wonham. Fully decentralized solutions of supervisory control problems. *IEEE Transactions on Automatic Control*, 40(12):2094–2097, December 1995.

[17] R. Kumar and S. Takai. Inference-based ambiguity management in decentralized decision-making: Decentralized control of discrete event systems. *IEEE Trans. Autom. Control*, 52(10):479 – 491, May 2007.

[18] Ratnesh Kumar, Hok M. Cheung, and Steven I. Marcus. Extension based limited lookahead supervision of discrete event systems. *Automatica*, 34(11):1327 – 1344, November 1998.

[19] Ryan James Leduc. *Hierarchical Interface-based Supervisory Control*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 2002.

[20] M. L. Lenard. A prototype implementation of a model management system for discrete-event simulation models. pages 560 – 568, December 1993.

[21] Feng Lin and W.M Wonham. Decentralized control and co-ordination of discrete event systems with partial observation. *IEEE Transactions on Automatic Control*, 35(12):1330–1337, December 1990.

[22] Jorn Lind-Nielsen. *BuDDy: Binary Decision Diagram Package*. IT-University of Copenhagen, November 2002.

[23] Fuchun Liu and Hain Lin. Reliable supervisory control for general architecture of decentralized discrete-event systems. *Automatica*, 46(9):1510–1516, September 2010.

[24] Fuchun Liu and Hain Lin. Reliable decentralized supervisors for discrete-event systems under communication delays: Existence and verification. *Asian Journal of Control*, 15(5):1346 –1355, September 2013.

[25] Chuan Ma. *NonBlocking Supervisory Control of State Tree Structure*. PhD thesis, Department of Electrical and Computer Engineering University of Toronto, 2004.

[26] Patricia N. Pena, Hugo J. Bravo, Antonio E. C. da Cunha, Robi Malik, and Jose E. R. Cury. Verification of the observer property in discrete event systems. *Proc. 11th International Workshop on Discrete Event Systems (WODES, 2012)*, pages 337 – 342, October 2012.

[27] J. Prosser, M. Kam, and H. Kwatny. Decision fusion and supervisor synthesis in decentralized discrete-event systems. *Proc. Ameri. Contr. Conf.*, pages 2251–2255, June 1997.

[28] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114 – 125, April 1959.

[29] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, January 1987.

[30] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. 77:81 – 98, January 1989.

[31] Laurie Ricker and Benoit Caillaud. Mind the gap: Expanding communication options in decentralized discrete-event control. *Automatica*, 47:2364 – 2372, September 2011.

[32] S. L. Ricker. *Knowledge and Communication in Decentralized Discrete-Event Control*. PhD thesis, Department of Computing and Information Science Queen's University, December 1999.

[33] S. L. Ricker. An overview of synchronous communication for control of decentralized discrete-event systems. In M. Silva. J.H. van Schuppen C. Seatzu, editor, *Control of Discrete-Event Systesm*, chapter 7, pages 127 –146. Springer, 2013.

[34] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. *IEEE/ACM International Conference on CAD*, pages 42–47, November 1993.

[35] K. Rudie, S. Lafortune, and F. Lin. Minimal communication in a distributed discrete-event control system. 48(6):957 – 975, June 2003.

[36] Karen Rudie and Jan C. Willems. The computational complexity of decentralized discrete-event control problems. *IEEE Transactions on Automatic Control*, 40(7):1313–1319, July 1995.

[37] Karen Rudie and W.M.Wonham. Think globally act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, November 1992.

[38] Raoguang Song. Symbolic synthesis and verification of hierarchical interface-based supervisory control. Master's thesis, Dept. of Computing and Software, McMaster University, March 2006.

[39] Shigemasa Takai and Toshimitsu Ushio. Reliable decentralized supervisory control of discrete event systems. 30(5):661 – 667, October 2000.

[40] Yu Wang. Sampled-data supervisory control. Master's thesis, Dept. of Computing and Software, McMaster University, January 2009.

[41] K. C. Wong and J. H. van Schuppen. Decentralized supervisory control of discrete-event systems with communication. *Proc. Int. Workshop on Discrete Event Systems*, pages 284–289, 1996.

[42] Kai C. Wong and W. Murray Wonham. On the computation of observers in discrete-event systems. discrete event dynamic systems. *Discrete Event Dynamic Systems*, 14(1):55–107, January 2003.

[43] W. M. Wonham. *Supervisory Control of Discrete-Event Systems.* Department of Electrical Engineering, University of Toronto, 2005.

[44] Tae-Sic Yoo and Stéphane Lafortune. New results on decentralized supervisory control of discrete-event systems. volume 1, pages 1–6. Decision and Control, 2000. Proceedings of the 39th IEEE Conference, December 2000.

[45] Tae-Sic Yoo and Stéphane Lafortune. Decentralized supervisory control: a new architecture with a dynamic decision fusion rule. pages 11–17, 2002.

[46] Tae-Sic Yoo and Stéphane Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. 12(3):335 – 377, July 2002.