SAMPLED-DATA SUPERVISORY CONTROL

By
YU WANG, B.ENG

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Applied Science
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE(2006)
(Software Engineering)

McMaster University
Hamilton, Ontario

TITLE: Sampled-data Supervisory Control

AUTHOR: Yu Wang, B.Eng(McMaster University)

SUPERVISOR: Dr. Ryan Leduc

NUMBER OF PAGES: i, 390

Abstract

This thesis focuses on issues related to implementing theoretical Discrete-Event Systems (DES) supervisors, and the concurrency and timing delay issues involved.

Sampled-data (SD) supervisory control deals with timed DES (TDES) systems where the supervisors will be implemented as SD controllers. An SD controller is driven by a periodic clock and sees the system as a series of inputs and outputs. On each clock edge (tick event), it samples its inputs, changes states, and updates its outputs.

In this thesis, we identify a set of existing TDES properties that will be useful to our work, but not sufficient. We extend the TDES controllability definition to a new definition, SD controllability, which captures several new properties that will be useful in dealing with concurrency issues, as well as make it easier to translate a TDES supervisor into an SD controller.

We then establish a formal representation of an SD controller as a Moore Finite State Machine (FSM), and describe how to translate a TDES supervisor to a FSM, as well as necessary properties to be able to do so. We discuss how to construct a single centralized controller, as well as a set of modular controllers and show that they will produce equivalent output.

Next, we capture the enablement and forcing action of a translated controller in the form of a TDES supervisory control map, and show that the closed-loop behavior of this map and the plant is the same as that of the plant and the original TDES supervisor. We also show that our method is robust with respect to nonblocking and certain variations in the actual behavior of our physical system.

We also introduce a set of predicate-based algorithms to verify the SD controllability property, as well as certain other conditions that we require. We have created a software tool for verifying these conditions and provide the source code in the appendix. We have implemented these algorithms using binary decision diagrams (BDD).

For illustrative purpose, we have produced a set of examples which fail the key conditions discussed in this thesis, as well as a successful application example based on a Flexible Manufacturing System. We also presented the corresponding FSM,

translated from the example's supervisors.

Acknowledgment

I will definitely first give my thanks to my supervisor, Dr. Ryan Leduc, who I have been working for since I was an undergraduate. I would have never been able to accomplish this task without the great great amount time, constant guidance, and support he has given me. His expertise in the area of discrete event control systems is the most valuable source of help for the whole period of this work.

I'd also like to thank Raoguang Song for his preceding work on the BDD based symbolic verification tool for HISC. His code base saved me a lot of effort in understanding and starting the software implementation.

At last my thanks go to my beloved father Xuhong Wang, and mother Baoxiang Yun for their unlimited support and Wen Xie for her great understanding and confidence in me. This thesis is dedicated to them.

Contents

List of Figures

Chapter 1

Introduction

In the area of Discrete-Event Systems (DES) [23], [29], [30], a lot of effort has been devoted to studying standard properties such as nonblocking (a form of deadlock detection) and controllability (a check on whether we can actually realize our desired control law) in a theoretical setting. However, limited effort has been made in investigating what an implementation of a DES supervisor would be like, how to do the conversion automatically, whether we can guarantee that it will retain the controllability and nonblocking properties of the theoretical supervisor, and how to handle timing delay and concurrency issues inherent in an implementation. This thesis will be attacking these problems, although issues with respect to timing delay will only be partially dealt with due to time limitations.

A logical implementation method for DES supervisors would be *sampled-data (SD) controllers*. An SD controller is driven by a periodic clock and sees the system as a series of inputs and outputs. On each clock edge, it samples its inputs, changes state, and updates its outputs. An example of an SD controller might be a programmable logic controller (PLC) [4] or a Moore synchronous finite state machine (FSM) [7]. In this thesis, we will focus on FSM SD controllers as they are a complete specification of an SD controller, yet still quite generic allowing an FSM to be implemented in digital logic, or as a computer program. For simplicity, we will assume inputs and outputs of an FSM can take the value of true or false.

When we are using an SD controller to manage a given system, we associate an input with each event, and output with each controllable event. We consider an event to have occurred when its corresponding input has gone true during a given clock period. We consider a controllable event to be enabled when its corresponding output has been set true by the controller, disabled otherwise.

As mentioned above, an SD controller samples the value of its inputs on each clock edge, and uses this value to decide what its next internal state will be. This means the SD controller knows nothing about its inputs until the clock edge, and then all it learns is whether a given input is true or false, signifying that the corresponding event has occurred sometime in the clock period that just ended. This means that for the given clock period, all information about event ordering (which event occurred first etc) is lost, as well as how often a given event occurred if it has occurred more than once. The only ordering information that remains is which *sampling period* (clock period) a given event occurred in.

As an example, consider Figure 1.1. Here we have inputs Event 1 and 2, as well as our sampling clock. The diagram on the left shows when the inputs changed their value, in particular that Event 1 occurred first in the second sampling period. When

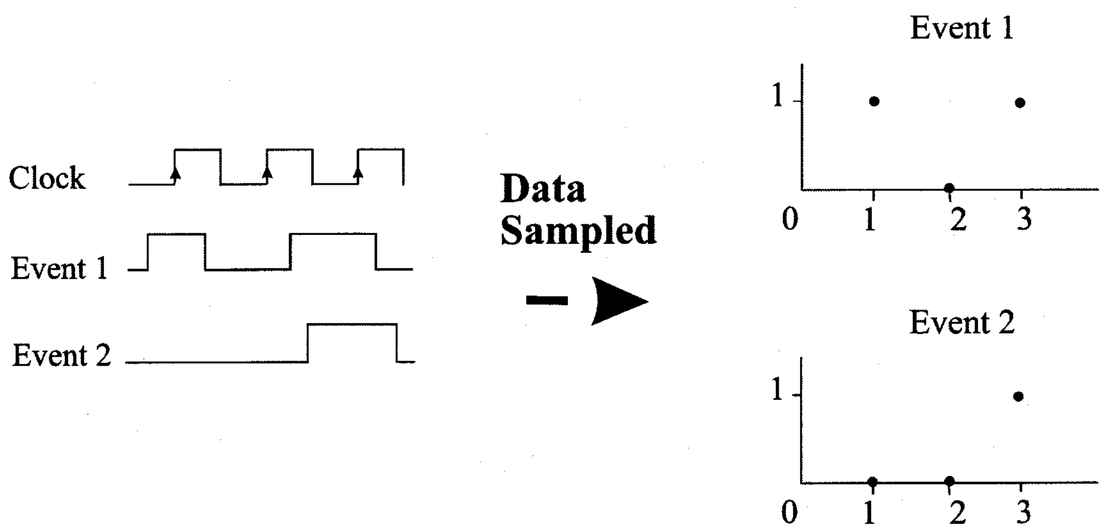


Figure 1.1: The Occurrences of Two Events

the SD controller samples its inputs, it simply gets a true or false value, based on the value of the input at the clock edge.[1] As we can see in the diagram on the right,

[1]In our example, we are sampling our inputs when the clock signal rises from low to high (the rising edge of the clock).

the controller simply knows that both Event 1 and 2 occurred in the last sampling period, nothing more.

Another important aspect of an SD controller is that it only changes state on a clock edge, and the value of its outputs are a function of its current state. That means its outputs can only change at a clock edge, and then must stay constant for the rest of the clock period.

For DES supervisors, we generally assume that a supervisor knows immediately when an event occurs, that it can change enablement information right away, and that events occur in an interleaving fashion so the supervisor can always determine the order events occurred in. Based on the above discussion, it is clear that an SD controller implementation violates these assumptions. First, the controller must wait until the next *sampling instance* (clock edge) before it will know if a given event has occurred. If the control law said something like "once event α occurs, controllable event β must not occur." However if β can occur in the same sampling period as α, β may have already occurred before we even know that α has occurred. Of course, even if we did know right away that alpha had occurred, we would not be able to update the enablement information for β until the next clock edge anyway, which could be too late. If we wanted to make sure β did not occur in this clock period, we would have to disable it at the start of the sampling period. This means that we cannot enforce a policy where an event is initially enabled (disabled) at the start of a clock period, and we then disable (enable) the event somewhere in the middle. Our supervisor must have a policy that is correct and constant for the entire sampling period.

Another important issue is event ordering. If we could get either string '$\alpha\beta$' or '$\beta\alpha$' in the same clock period, our SD controller would only know that at least one α and at least one β had occurred. It would not know which of the two had actually occurred. If our DES supervisor enabled event γ when string '$\alpha\beta$' occurs, but disables γ when string '$\beta\alpha$' occurs, we could not implement this using an SD controller as it would not be able to determine which of the two strings had occurred. This means that a supervisor must always do the same thing for two concurrent strings containing the same individual events, both immediately after the strings have occurred and in the future. Of course, this raises the question of how to determine if two strings are concurrent.

1.1 Objective

Clearly, untimed DES does not provide a rich enough modeling method to allow us to work with an SD controller, and its inherent timing information. Therefore, we will base our work on the timed DES (TDES) theory developed by Brandin et al. [5] [6]. TDES extends untimed DES theory by adding a new tick event, corresponding to the tick of a global clock. The event set of a TDES contains the tick event as well as other non-tick events called *activity* events. The occurrence of a tick event provides us with a concept of time passing, allowing us to model upper and lower time bounds for the occurrence of activity events. It also allows us to introduce a new type of events called *forcible events*, which we can guarantee to occur and preempt the next clock tick. This means that now we cannot only prevent some events (referred to as *prohibitable events* in TDES terminology) from occurring by disabling them, but we can also choose to have certain events occur before the next clock tick.

To make the TDES theory work with SD controllers, we identify a tick event occurring with the clock edge that the SD controller uses for sampling and state change. That means that once a tick event occurs, any two strings that are now possible in the system and only contain a single tick at the end of the string, are considered concurrent. We will refer to such strings as *concurrent strings*. If one of these strings contains at least one different event from the other string, we can distinguish between them. Otherwise, we must treat them the same.

Now that we can force an event to occur in a specific clock period, we have a new concern with respect to nonblocking. The plant model might say that we can do either an '$\alpha\beta\tau$' concurrent string, or a '$\beta\alpha\tau$' string, where $\tau = tick$. Both might be safe to do, but depending on our implementation, only one of the two might ever occur. Some reasons this could occur are due to time delay, or our implementation might be a sequential program that must choose one version or the other to perform. It might be the case that for some implementations, when two or more concurrent strings are possible and they contain the same events but in a different order or numbers, not all variations might ever actually occur. The problem is that one of the variations that does not occur might have been the only path in the TDES back to a marked state. Basically, if an SD controller cannot tell the difference between concurrent strings, they should have the same marked future. This also means that marked strings can only be the empty string (represents the initial state of the system which is always observable), or strings ending in a tick as these are the points in the system's behavior that are observable to an SD controller. We refer to such strings as *sampled strings.*

The next problem we intend to address is the issue of when a forced event should occur. As noted by Balemi in [2] for untimed systems, controllable events tend to be events fully under the control of our controller implementation.[2] They may be a software function we call, an output we set to true, or a message we send. That means that we can make these events occur whenever we want. It is not unusual that a plant might be modeled such that these events are suppose to only occur under certain situations. This might be for flexibility (some implementations have these restrictions, for example) or to make the system easier to model or understand. However, the reality for some controller implementations is that these events could occur even when the plant said they cannot. This also applies to forcible events. When we are forcing an event to occur in a given clock period, we have no information on when it will actually occur. Depending on our implementation, it could occur right away, or in the middle or end of the clock period. We need to make sure that when it finally does occur, it does not contradict the plant model so that our implementation will correspond to the theoretical model in this respect.

The last issue we intend to address is the issue of when a forcible event should actually occur. We want our supervisor specified in such a way that it is straightforward to convert it into an SD controller. Normally for DES systems, we are interested in maximally permissive behavior. We enable all controllable events except for when they must be disabled to enforce our control law, and to ensure the system is nonblocking. However, controller implementations are usually much more procedural. We would disable all controllable events until we want them to occur, and then disable the event again once it has occurred. In our setup, we will be assuming that the set of prohibitable events and forcible events are the same[3] and that we disable the event until we wish to force it, and then disable it once it has occurred. This

[2]This is generally a matter of how a system is modeled. We can always model the sending of our enable/disable signal as the controllable event, and the occurrence of the actual action as the uncontrollable event. Of course, the occurrence of the enablement event would toggle the eligibility of the uncontrollable event.

[3]Again, this is a matter of modeling. We can always model our forcing signal as the controllable event, and then model the event corresponding to the actual action as an uncontrollable event that must occur before the next clock tick, once the forcing event has occurred.

requires our supervisor to specify exactly which clock period the event should occur in and this makes it very straight forward to translate to a controller. Currently, a supervisor could say something like controllable event α is now enabled, and will stay enabled for the next three clock cycles, but must occur before the fourth. You could potentially force it sooner, but that might cause blocking. Such an ambiguous supervisor will be a lot harder to translate to an SD controller.

In this thesis, we will develop a new property for TDES systems that will address the above issues, as well as make our TDES supervisor more consistent with SD controllers, making them easy to translate. First, we will provide the preliminaries of untimed and timed DES in Chapter 2, which is required to understand the following chapters.

Then in Chapter 3 we will introduce the sampled-data setting based on timed DES. The sampled-data setting will be formally defined, and we will develop a new property called SD controllability to address the issues we identified above.

In Chapter 4, we will provide the definition of Moore FSM [17] and a method to translate a CS deterministic supervisor (defined in Chapter 3) into a Moore FSM controller. We will present both a centralized translation method and a modular method. We will then show that they will both produce equivalent output information.

Then in Chapter 5 we capture the enablement and forcing action of a translated controller in the form of a TDES supervisory control map, and show that the closed loop behavior of this map and the plant is the same as that of the plant and the original TDES supervisor. We also show that our method is robust with respect to nonblocking and certain variations in the actual behavior of our physical system.

In Chapter 6 we will introduce logic predicates and predicate transformers, as well as symbolic representation and computation based on [26]. Then we will introduce a set of algorithms to verify SD controllability and other properties of interest to us.

Then in Chapter 7 we will present examples which fail the key conditions in this thesis, to help understand the definitions. We will then present a successful application example inspired by the untimed Flexible Manufacturing System from [11], including the Moore FSM controllers translated from the supervisors developed in the example.

We will close the thesis with our conclusions and a brief discussion of future work.

Also, in the appendix we will present the input files used for the FSM example given in Chapter 7, as well as the source code for our software tool that we have developed that implements the algorithms presented in Chapter 6. The software tool makes use of binary decision diagrams (BDD) [8].

1.2 Related Work

Supervisory control of DES with timing information, known as timed DES (TDES), was firstly introduced in [5], [6], based on the timed transition model from [19], [20], and [21]. The theory added timing information to supervisory control allowing one to specify lower and upper time bounds for events. It also introduced a forcing technology to ensure certain events occur when we desired. We will use this as the basis of our SD supervisory control theory.

Balemi [2] pointed out that typically, controllable events are part of the supervisor implementation, and often can occur whenever we want them to. For simplicity, the plant may be modeled such that these events are assumed to only occur at certain times. Balemi's plant completeness condition helps ensures that the implementation of the supervisor will be consistent with the plant model so that controllable events do not occur when the plant model says that they cannot.

In the sampled-data setting, if the same event occurs once or multiple times in the same sampling period, an SD controller will not be able to detect a difference. In [3], the authors require that the system has the property that an event cannot be generated more than once during a sampling period. The paper also discussed the loss of ordering information when events occur in the same sampling period. To handle these timing related issues, the author adds a dispatcher to the existing supervisor to solve the problems that could occur when event ordering cannot be ignored. The model is implemented based on Petri Nets [16, 33] and an algorithm to translate the Petri Net implementation into computer language is provided.

Translating abstract model into a computer understandable form is an interesting topic for researchers. In [12], Leduc discusses the modeling and implementation of real-life DES problems as well. Theorems for model reduction were created and applied to the DES designed for a programmable logic controller (PLC) based manufacturing testbed. The author investigated implementing DES as Moore finite state machines (FSM) and created an implementation by hand for the testbed. As mentioned earlier, FSM can be converted to other forms of state based logic sequences, such as a relay ladder logic program for the testbed. The idea of implementing SD controllers as FSM is motivated by this thesis.

Similarly, [18] also discusses translating DES into PLC programs. The difference is that they first convert automata into the Grafcet language, which describes the specification of logic controllers. They then translate the Grafcet language into a PLC program. Both [12] and [18] uses automated manufacturing testbeds as examples.

In [9], DES theory is used as a tool to assist programming in the system control area. The authors describe an approach to generate Java code for concurrency control automatically. The approach formalizes each individual code portion without concurrency control into specifications, builds the DES model, and then generates the code with verifications.

A real world application of DES supervisory control is given in [10], where Petri Nets are used to model railway networks and ensure controllability and liveness.

An important tool to allow supervisory control methods to be applied to larger systems, is the use of binary decision diagrams (BDD)[8]. BDD methods have been applied to standard DES [32], [27], state tree structures [14], Hierarchical Interface-based Supervisory Control [26], and state based control of TDES [24].

When synthesizing controllers there is often a need to consider other components in the system, which lower the flexibility and increase the cost of synthesis in changing environments. With the I/O based hierarchical structure from [22], each controller can be designed independently, and controllability and nonblocking is retained when the controllers are combined.

However, even if the DES supervisor is nonblocking for the DES plant does not mean that the controller implementation is nonblocking as well. To ensure a controller is nonblocking, [15] studied several different systems for implementing controllers. The author suggested conditions to be satisfied for the implemented controllers to be nonblocking.

Another practical issue in implementing controllers based on DES is communication. In [25], the authors study the communication between modular and decentralized supervisors on switch networks. A communication model is then introduced for a large distributed controller network where communication delay and collisions are a concern. In [31], the authors resolve communication issues by introducing an asynchronous implementation. The work formalizes the delay between the controller and the plant, and defines bounded-delay implementability, in addition to the standard controllability and nonblocking properties.

Chapter 2

Discrete-Event Systems Preliminaries

Supervisory control theory provides a framework for the control of discrete-event systems (DES), systems that are discrete in space and time. For a detailed exposition of DES, see [29]. Below, we present a summary of the terminology that we use in this thesis.

2.1 Algebraic Preliminaries

2.1.1 Strings

An *alphabet* Σ is defined to be a finite set of distinct symbols. A *string* over Σ is a finite sequence of symbols $\sigma_1\sigma_2..\sigma_k$, where $\sigma_i \in \Sigma$ for $i = 1, 2, .., k$. Given a string $s = \sigma_1\sigma_2..\sigma_k$, $|s| = k$ is the *length* of the string. The string ϵ is called the *empty string* with $|\epsilon| = 0$. Let Σ^* be the set of all finite symbol sequences and define Σ^+ be

$$\Sigma^+ := \Sigma^* \ - \ \{\epsilon\}$$

Definition 2.1.1. Let $s_1, s_2 \in \Sigma^*$, where $s_1 = \sigma_1\sigma_2..\sigma_m$ and $s_2 = \tau_1\tau_2..\tau_n$. The *catenation* of s_1 and s_2 is define to be $cat : \Sigma^* \times \Sigma^* \to \Sigma^*$ such that

$$cat(s_1, \epsilon) = cat(\epsilon, s_1) = s_1 = \sigma_1\sigma_2..\sigma_m$$
$$cat(s_1, s_2) = s_1s_2 = \sigma_1\sigma_2..\sigma_m\tau_1\tau_2..\tau_n$$

As $|s_1| = m$ and $|s_2| = n$, the length of concatenated string is $|s_1 s_2| = |s_1| + |s_2| = m + n$.

Definition 2.1.2. Let $s, t \in \Sigma^*$. We say s is a *prefix* of t, denoted as $s \leq t$, if

$$(\exists u \in \Sigma^*) su = t$$

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

2.1.2 Languages

Definition 2.1.3. Let $L \subseteq \Sigma^*$. The *prefix closure* of L, denoted as $\overline{L}$, is defined as

$$\overline{L} = \{s \in \Sigma^* | (\exists t \in L) s \leq t\}$$

By definition, we can see that a language L is a subset of the prefix closure of itself, i.e. $L \subseteq \overline{L}$. We say a language $L \subseteq \Sigma^*$ is *prefix closed* if $L = \overline{L}$. Let $K \subseteq L$. We say K is *L-closed* if $K = \overline{K} \cap L$.

Definition 2.1.4. Let $L \subseteq \Sigma^*$. The *eligibility operator*, $\mathrm{Elig}_L : \Sigma^* \rightarrow \mathrm{Pwr}(\Sigma)$, is defined for $s \in \Sigma^*$ as,

$$\mathrm{Elig}_L(s) := \{\sigma \in \Sigma \,|\, s\sigma \in L\}$$

2.1.3 Nerode Equivalence Relation

Definition 2.1.5. Let X be a nonempty set. Let $E \subseteq X \times X$ be a binary relation on X. The relation E is an *equivalence relation* on X if

1. $(\forall x \in X) xEx$ (reflexivity)
2. $(\forall x, x' \in X) xEx' \implies x'Ex$ (symmetry)
3. $(\forall x, x', x'' \in X) xEx' \;\&\; x'Ex'' \implies xEx''$ (transitivity) [1]

[1]We use '&' to stand for logical AND here to avoid confusion with later definitions in this section.

Here we are using standard infix notation, where we use xEx' to represent the ordered pair $(x, x') \in E$. For xEx', we may also write $x \equiv x'(\text{mod}E)$.

For $x \in X$, let $[x]_E \subseteq X$ represent the subset of elements that are equivalent mod E to x. That is

$$[x]_E := \{x' \in X | x'Ex\}$$

If relation E is understood by the context, we will just write $[x]$. We will also refer to $[x]$ as the coset or the equivalence class of x with respect to E.

Let $s, t \in \Sigma^*$, and $L \subseteq \Sigma^*$. We say s and t are *Nerode equivalent* with respect to language L, if and only if they can be extended by any string $u \in \Sigma^*$ such that the two extended strings are either both in L or neither in L. In this case, we write $s \equiv t$ (mod L) or $s \equiv_L t$. The formal definition is given below.

Definition 2.1.6. Let $L \subseteq \Sigma^*$. Let $s, t \in \Sigma^*$.

$$s \equiv_L t \ \text{ or } \ s \equiv t \,(\text{mod}\, L)$$

iff

$$(\forall u \in \Sigma^*) su \in L \iff tu \in L$$

Essentially, if strings s and t are equivalent mod L, then they can both be extended in the same way by right concatenation.

Example 2.1. *Let* $\Sigma = \{\alpha, \beta, \gamma\}$, $L = \{\epsilon, \alpha, \beta, \alpha\gamma^*, \beta\gamma^*\}$, *then* $\alpha \equiv_L \beta$.

2.2 Discrete Event Systems

2.2.1 Generator

We model DES formally as a generator $\mathbf{G}$, which is a five tuple

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

where

Q is the *state set*.

Σ is the finite set of distinct symbols representing event labels. We partition Σ into two parts

$$\Sigma = \Sigma_c \,\dot{\cup}\, \Sigma_u$$

where

Σ_c is the set of *controllable* events, which can be enabled or disabled by an external agent. A controllable event can only occur when it is enabled.

Σ_u is the set of *uncontrollable* events, which cannot be disabled by any external agent. Once the DES has reached a state where an uncontrollable event can occur, the event cannot be prevented.

$\delta : Q \times \Sigma \to Q$ is the (partial) transition function where each transition is a tuple (q, σ, q'), where $\delta(q, \sigma) = q'$. We refer to q as the *exit (source) state*, and q' as the *entrance (destination) state.* We write $\delta(q, \sigma)!$ if $\delta(q, \sigma)$ is defined.

We can extend the transition function to $\delta : Q \times \Sigma^* \to Q$ as

$\delta(q, \epsilon) = q \quad \text{for } q \in Q.$

$\delta(q, s\sigma) = \delta(\delta(q, s), \sigma) \quad \text{for } s \in \Sigma^*,\ \sigma \in \Sigma, \text{ and } q \in Q.$

as long as $q' = \delta(q, s)!$ and $\delta(q', \sigma)!$.

$q_0 \in Q$ is the *initial state.*

$Q_m \subseteq Q$ is the subset of *marked states.*

We can extend the transition function to $\delta : Q \times \Sigma^* \to Q$ as

$\delta(q, \epsilon) = q \quad \text{for } q \in Q.$

$\delta(q, s\sigma) = \delta(\delta(q, s), \sigma) \quad \text{for } s \in \Sigma^*,\ \sigma \in \Sigma, \text{ and } q \in Q.$

as long as $q' = \delta(q, s)!$ and $\delta(q', \sigma)!$.

Example 2.2. *Let* $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ *be the DES shown in Figure 2.1. By convention, a controllable event is graphically represented by a slash across its transition* *arrow. Marked states are represented by a black dot. The state pointed at by an arrow with no exit state, is the initial state. For the DES shown we have:*

$Q = \{I, W, D\}$;

$\Sigma = \Sigma_c \dot{\cup} \Sigma_u$, *where* $\Sigma_c = \{\alpha, \mu\}$ *and* $\Sigma_u = \{\beta, \lambda\}$;

$\delta = \{(I, \alpha, W), (W, \beta, I), (W, \lambda, D), (D, \mu, I)\}$;

$q_0 = I$; $Q_m = \{I\}$

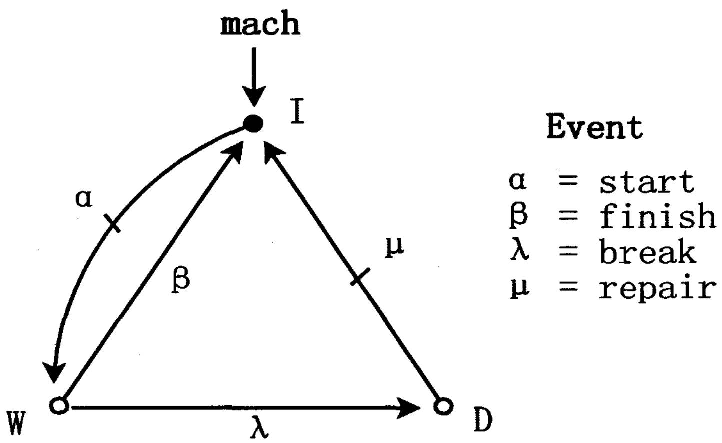


Figure 2.1: An Example DES

Given DES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$, we have the following definitions.

Definition 2.2.1. A state $q \in Q$ is *reachable* if

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

Definition 2.2.2. A state $q \in Q$ is *coreachable* if

$$(\exists s \in \Sigma^*)\delta(q, s)! \text{ and } \delta(q, s) \in Q_m$$

To simplify the following discussions, we will always assume a given DES is reachable unless explicitly stated otherwise.

Definition 2.2.3. The *closed behavior* of DES $\mathbf{G}$ is

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

Definition 2.2.4. The *marked behavior* of DES $\mathbf{G}$ is

$$L_m(\mathbf{G}) = \{s \in \Sigma^* | \delta(q_0, s)! \;\&\; \delta(q_0, s) \in Q_m\}$$

Clearly, $L_m(\mathbf{G}) \subseteq L(\mathbf{G})$.

Definition 2.2.5. The *control action* for some $q \in Q$ for DES $\mathbf{G}$ is defined to be a mapping $\zeta : Q \to \mathrm{Pwr}(\Sigma_c)$ that takes q and returns a set of controllable events enabled at q.

Definition 2.2.6. DES $\mathbf{G}$ is said to be *nonblocking* if every reachable state is also coreachable. This can be expressed as

$$L(\mathbf{G}) = \overline{L_m(\mathbf{G})}$$

Definition 2.2.7. Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ and let λ be an equivalence relation on Q such that for $q, q' \in Q$, $q \equiv q' \ mod\ \lambda$ if and only if

1. $(\forall s \in \Sigma^*)\delta(q, s)! \iff \delta(q', s)!$

2. $(\forall s \in \Sigma^*)[\delta(q, s)! \;\&\; \delta(q, s) \in Q_m] \iff [\delta(q', s)! \;\&\; \delta(q', s) \in Q_m]$

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

Definition 2.2.8. DES $\mathbf{G}$ is said to be *minimal*, if

$$(\forall q, q' \in Q) q \equiv q' \ (\mathrm{mod}\ \lambda) \iff q = q'$$

It says that for all states $q, q' \in Q$, if q is equivalent to q' mod λ, then q and q' are the same state. DES $\mathbf{G}$ is minimal if it does not have two distinct states in Q that are λ equivalent.

2.2.2 Synchronization and Product DES

In real world, it is often easier to model a system as several smaller components. For a DES plant, we use the *synchronous product* operator to combine the individual DES components instead of modeling the whole system at once. We first need to define the *natural projection* operator and its inverse.

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a DES. Take $\Sigma_o \subseteq \Sigma$ to be the set of *observable events* through some filtering channel of the events generated by $\mathbf{G}$.

Definition 2.2.9. The *natural projection* $P : \Sigma^* \to \Sigma_o^*$ is defined as follows. For $s \in \Sigma^*$, $\sigma \in \Sigma$,

$$\begin{aligned} P(\epsilon) &= \epsilon \\ P(\sigma) &= \begin{cases} \epsilon & \text{if } \sigma \notin \Sigma_o \\ \sigma & \text{if } \sigma \in \Sigma_o \end{cases} \\ P(s\sigma) &= P(s)P(\sigma) \end{aligned}$$

Example 2.3. *For* $\Sigma = \{\alpha, \beta, \gamma\}$, $\Sigma_o = \{\alpha, \beta\}$ *and* $s = \alpha\beta\alpha\gamma\beta\alpha$,

$$P(s) = P(\alpha)P(\beta)P(\alpha)P(\gamma)P(\beta)P(\alpha) = \alpha\beta\alpha\beta\alpha$$

Let $L \subseteq \Sigma^*$. We define $P(L) \subseteq \Sigma_o^*$ as an extension of the natural projection as

$$P(L) := \{P(s) | s \in L\}$$

We also define its inverse image $P^{-1} : \mathrm{Pwr}(\Sigma_o^*) \to \mathrm{Pwr}(\Sigma^*)$ such that, for $H \subseteq \Sigma_o^*$

$$P^{-1}(H) := \{s \in \Sigma^* | P(s) \in H\}$$

Example 2.4. *For* $\Sigma = \{\alpha, \beta, \gamma, \mu\}$, $\Sigma_o = \{\alpha, \beta\}$ *and* $s_o = \alpha\beta\alpha\beta\alpha$, *the inverse projection is*

$$P^{-1}(\{s_o\}) := \{\gamma, \mu\}^*\alpha\{\gamma, \mu\}^*\beta\{\gamma, \mu\}^*\alpha\{\gamma, \mu\}^*\beta\{\gamma, \mu\}^*\alpha\{\gamma, \mu\}^*$$

Definition 2.2.10. For $i = 1, 2$, let $L_i \subseteq \Sigma_i^*$, $\Sigma = \Sigma_1 \cup \Sigma_2$ and $P_i : \Sigma^* \to \Sigma_i^*$ be natural projections. The *synchronous product* of L_1 and L_2 is defined to be

$$\begin{aligned} L_1 || L_2 &= P_1^{-1}(L_1) \cap P_2^{-1}(L_2) \\ &= \{s \in \Sigma^* | P_1(s) \in L_1 \& P_2(s) \in L_2\} \end{aligned}$$

Definition 2.2.11. Let $\mathbf{G}_1 = (Q_1, \Sigma, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma, \delta_2, q_{o,2}, Q_{m,2})$ be two DES defined over the same event set Σ. The *product* of two DES is defined as

$$\mathbf{G}_1 \times \mathbf{G}_2 = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$$

where $\delta_1 \times \delta_2 : Q_1 \times Q_2 \times \Sigma \to Q_1 \times Q_2$ is defined as

$$(\delta_1 \times \delta_2)((q_1, q_2), \sigma) := (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$$

whenever $\delta_1(q_1, \sigma)!$ and $\delta_2(q_2, \sigma)!$.

By Definition 2.2.11, we have $L(\mathbf{G}_1 \times \mathbf{G}_2) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2)$ and $L_m(\mathbf{G}_1 \times \mathbf{G}_2) = L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2)$

Definition 2.2.12. The *meet* of $\mathbf{G}_1$ and $\mathbf{G}_2$, or $\mathbf{meet}(\mathbf{G}_1, \mathbf{G}_2)$, is defined to be the reachable subautomaton of the product DES $\mathbf{G}_1 \times \mathbf{G}_2$.

Definition 2.2.13. The *synchronous product of DES* $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o_i}, Q_{m_i})$ $(i = 1, 2)$, denoted $\mathbf{G}_1 || \mathbf{G}_2$, is defined to be a reachable DES $\mathbf{G}$ with event set $\Sigma = \Sigma_1 \cup \Sigma_2$ and properties:

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

Definition 2.2.14. Let $\mathbf{G}$ be a DES defined over Σ and Σ' be another set of events such that $\Sigma \cap \Sigma' = \emptyset$. The *selfloop* operation on $\mathbf{G}$ is defined as

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

where $\delta' : Q \times (\Sigma \cup \Sigma') \to Q$ is a partial function defined as

$$\delta'(q, \sigma) := \begin{cases} \delta(q, \sigma) & \sigma \in \Sigma, \delta(q, \sigma)! \\ q & \sigma \in \Sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

For DES $\mathbf{G}'_i$ ($i = 1, 2$) defined over event set Σ_i, we will always assume that the synchronous product operator is implemented by first extending each DES to be over Σ by adding selfloops, and then using the meet operator. More formally, we take $\Sigma = \Sigma_1 \cup \Sigma_2$, and $\mathbf{G}_i = \mathbf{selfloop}(\mathbf{G}'_i, \Sigma - \Sigma_i)$. We then have $G'_1||G'_2 = \mathbf{meet}(\mathbf{G}_1, \mathbf{G}_2)$.

In the algorithms we develop in this thesis, we will always assume all DES are combined with the product DES operator. If a portion of the system is actually combined together using the synchronous product operator as is commonly done for plant components, we will first add selfloops as above, and then use these new DES from then on in our algorithms.

2.2.3 Controllability and Supervision

We will take language K to represent the desired safe behavior of our plant represented by DES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$. We want to make sure that the closed loop behavior of the system – that is the behavior of plant $\mathbf{G}$ under control of K – is a subset of $\overline{K}$.

As we mentioned earlier, our system's event set Σ is partitioned into controllable and uncontrollable events. If an undesirable controllable event is possible in $\mathbf{G}$ that will cause the system to leave the behavior represented by $\overline{K}$, we disable it and prevent it from occurring. We cannot do this with an uncontrollable event, so we need to make sure the plant never reaches a state where it can leave the desired behavior by an uncontrollable event. We now express this formally below.

Definition 2.2.15. K is said to be *controllable* with respect to $\mathbf{G}$ if

$$(\forall s \in \overline{K})(\forall \sigma \in \Sigma_u) s\sigma \in L(\mathbf{G}) \implies s\sigma \in \overline{K}$$

We typically give this definition in the form of $\overline{K}\Sigma_u \cap L(\mathbf{G}) \subseteq \overline{K}$ where $\overline{K}\Sigma_u$ denotes the string $s\sigma$ for $s \in \overline{K}$ and $\sigma \in \Sigma_u$. In other words, if the plant reaches a state where uncontrollable event σ is possible, then σ must also be accepted by $\overline{K}$. By definition, $\emptyset$, $L(\mathbf{G})$ and Σ^* are all controllable with respect to $\mathbf{G}$.

Another way to express this definition is

$$(\forall s \in \overline{K} \cap L(\mathbf{G}))\, \mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq \mathrm{Elig}_{\overline{K}}(s)$$

which is used in **Point i** of Definition 3.2.1 in Section 3.2.

As we prefer to work with finite state automata than typically infinite languages, we want to be able to express K as a DES supervisor.

Definition 2.2.16. Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a DES. Let $K \subseteq \Sigma^*$. We say $\mathbf{G}$ *represents* K if

$$K = L_m(\mathbf{G}) \text{ and } \overline{K} = L(\mathbf{G})$$

Definition 2.2.17. Let $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a DES. Let $K \subseteq \Sigma^*$, we say $\mathbf{S}$ *implements* K, if

$$K = L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \text{ and } \overline{K} = L(\mathbf{S}) \cap L(\mathbf{G})$$

Recall that $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$, where Σ_c is a set of controllable events which can be enabled or disabled by external agents; and Σ_u is a set of uncontrollable events which cannot be disabled. We refer to such an external agent as a *supervisor*, which will formally define shortly.

Definition 2.2.18. Let $L(\mathbf{S})$ be the language represented by DES $\mathbf{S}$. We say $\mathbf{S}$ is a *supervisor* for $\mathbf{G}$, if

1. $L(\mathbf{S})$ is controllable with respect to $\mathbf{G}$, and
2. $\overline{L_m(\mathbf{S}) \cap L_m(\mathbf{G})} = L(\mathbf{S}) \cap L(\mathbf{G})$

For convenience, we say $\mathbf{S}$ is *controllable* for $\mathbf{G}$ if $L(\mathbf{S})$ is controllable with respect to $\mathbf{G}$.

We can think of a supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ as a state machine that tracks all the events generated by plant $\mathbf{G}$. Together with current state $x \in X$ as source state, it takes each event as an input to its transition function ξ, then moves to the destination state $x' \in X$. Events in $\mathbf{G}$ are only allowed to occur when the event is not disabled in $\mathbf{S}$. We refer to the closed loop behavior of the system as the behavior of our plant $\mathbf{G}$ under the control of supervisor $\mathbf{S}$. This is typically represented as the meet of $\mathbf{G}$ and $\mathbf{S}$. If we modeled the system only using the synchronous product, then this would be represented as $\mathbf{G}||\mathbf{S}$.

As noted by Balemi in [2], controllable events tend to be events fully under the control of our supervisor's implementation. They may be a software function we call, an output we set to true, or a message we send. That means that we can make these events occur whenever we want. It is not unusual that a plant might be modeled such that these events are suppose to only occur under certain situations. This might be for flexibility (some implementations have these restrictions, for example) or to make the system easier to model or understand. However, the reality for some supervisor implementations is that these events could occur even when the plant said they cannot. We refer to such situations as *illegal transitions.* The requirement is formally defined in [2] as follows.

Definition 2.2.19. A plant $\mathbf{G}$ is *complete* for its supervisor $\mathbf{S}$ if

$$(\forall s \in L(\mathbf{G}) \cap L(\mathbf{S}))(\forall \sigma \in \Sigma_c) s\sigma \in L(\mathbf{S}) \implies s\sigma \in L(\mathbf{G})$$

The definition states that, at each state in plant $\mathbf{G}$, every controllable event enabled by supervisor $\mathbf{S}$ must be accepted by $\mathbf{G}$ as well. This condition can be seen as a dual to the definition of a supervisor $\mathbf{S}$ being controllable for plant $\mathbf{G}$. This definition will be very useful for implementing DES supervisors, as it says that they do not require additional supplementary information from the plant to decide when a controllable event can occur and not violate the plant model.

2.3 Timed Discrete Event Systems

So far we have only discussed untimed DES. As we wish to use a richer modeling framework that includes timing requirements of our system, we will now discuss Timed DES (TDES) introduced by Brandin et al [5] [6].

TDES extends untimed DES theory by adding a new tick event, corresponding to the tick of a global clock. The event set of a TDES contains the tick event as well as other non-tick events called *activity* events (Σ_{act}). The occurrence of a tick event provides us with a concept of time passing, allowing us to model upper and lower time bounds for the occurrence of activity events. A lower time bound for a given activity event can be modeled as requiring a certain number of tick events to first occur before the activity event is eligible. Once an activity event is eligible to occur in the TDES and the desired number of tick events have occurred, we can model an upper bound for the event by not allowing a tick event to occur until either the event has occurred, or another activity event has occur such that the first event is no longer eligible.

The addition of a tick event also allows us to introduce a new type of events called *forcible events* (Σ_{for}), which we guarantee to occur and preempt the next clock tick. This means that now we cannot only prevent some events (referred to as *prohibitable events* (Σ_{hib}) in TDES terminology) from occurring by disabling them, but we can also choose to have certain events occur before the next clock tick. As a convention, we sometimes refer to tick as τ for brevity.

2.3.1 Basic Structure

We formally define a TDES as the tuple

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

where,

Q is the *state set*

$\Sigma = \Sigma_{act} \,\dot{\cup}\, \{\tau\}$ is the set of all events, including activity events and the tick event.

$\delta : Q \times \Sigma \rightarrow Q$ is the (partial) transition function.

$q_0 \in Q$ is the initial state.

$Q_m \subseteq Q$ is the set of marked states.

For convenience, we extend δ to function $\delta : Q \times \Sigma^* \rightarrow Q$ in the same way as we did in the untimed DES definition.

2.3.2 Controllability and Supervision

Control action for timed DES is achieved in an analogous fashion as that of untimed DES, by disabling controllable events. As for untimed DES, we also partition our event set Σ into controllable and uncontrollable events. The set of controllable events is defined to be

$$\Sigma_c := \Sigma_{hib} \cup \{\tau\}$$

where $\Sigma_{hib} \subseteq \Sigma_{act}$ the set of activity events that can disabled by an external agents. These event are referred to as prohibitable events to distinguish them from controllable events that include the tick event. As we will see when we define controllability in the TDES setting, we will use disabling the tick event by the supervisor to model forcing an event. A forcible event is an event in the system that we can make occur before the next clock tick, assuming it is not first preempted by another event. The set of *uncontrollable* events for $\mathbf{G}$ is then defined to be

$$\Sigma_u := \Sigma - \Sigma_c$$

In Section 2.2.3, we introduced Balemi's concept of completeness of a plant for a given supervisor. Unfortunately, that definition was given in terms of controllable events, which includes the tick event in TDES. As we are only concerned about the occurrence of activity events, we need to define a version of this definition for TDES. When discussing this concept, we will not specify whether or not we mean the timed or untimed version, as this will be clear by the context.

Definition 2.3.1. Let TDES $\mathbf{G}$ be a plant and TDES $\mathbf{S}$ be a supervisor. $\mathbf{G}$ is *TDES complete* for $\mathbf{S}$ if

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

We now need to add a technical condition that we most enforce to ensure that our TDES does not allow the physically unrealistic situation where a tick event could be preempted indefinitely by the continued execution of an activity event loop within a given fixed unit time. Formally, a TDES is said to have an activity loop if it satisfies the following definition.

Definition 2.3.2. TDES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ has an *activity loop* if

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

We thus require that a TDES be *activity loop free* (ALF). We can formalize the ALF concept as defined below.

Definition 2.3.3. TDES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ is *activity loop free* if

$$(\forall q \in Q_{reach})(\forall s \in \Sigma_{act}^{+})\delta(q, s) \neq q$$

We only look at states that are reachable (i.e. in Q_{reach}), because we do not care about unreachable states as they do not contribute to the automaton's closed and marked behavior. These unreachable activity loops can be safely ignored. An example that fails the ALF property is shown in Figure 2.2 where the $\alpha\beta$ loop could indefinitely preempt the tick event from occurring.

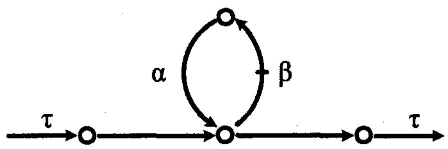


Figure 2.2: An Example Failing ALF Property

We will not require that supervisors be ALF, as they may contain self-loops that are not possible in the plant. We will instead require that the system's closed loop behavior (typically the meet of plant $\mathbf{G}$ and supervisor $\mathbf{S}$) be ALF.

For the FSM translation of individual supervisors in Section 4.2, we need a more specific definition as follows.

Definition 2.3.4. Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a TDES, and let $\mathbf{G}'$ be $\mathbf{G}$ with all activity event selfloops removed. $\mathbf{G}$ is *non-selfloop activity loop free* if $\mathbf{G}'$ is ALF.

Essentially, if we remove the selfloops of any activity events in the TDES, the rest of the TDES must be ALF. This will be a key definition that will allow us to translate the TDES to a Moore finite state machine.

The proposition below states that if individual DES are all ALF, it implies that the synchronous product of these DES is also ALF. This means that we can simply check the individual DES.

Proposition 2.1. For TDES $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{0,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2, \delta_2, q_{0,2}, Q_{m,2})$, if $\mathbf{G}_1$ and $\mathbf{G}_2$ are each ALF, then their synchronous product $\mathbf{G} = \mathbf{G}_1||\mathbf{G}_2$, is ALF.

Proof. Let $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{0,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2, \delta_2, q_{0,2}, Q_{m,2})$ be two TDES and let $P_1 : \Sigma^* \to \Sigma_1^*$ and $P_2 : \Sigma^* \to \Sigma_2^*$ be natural projections.

Define $\Sigma_{act,i} = \Sigma_{act} \cap \Sigma_i$, $i = 1, 2$.

By ALF Definition 2.3.3, for $i = 1, 2$

$$(\forall q_i \in Q_{reach,i})(\forall s_i \in \Sigma_{act,i}^+)\delta_i(q_i, s_i) \neq q_i \qquad (*)$$

where $Q_{reach,i}$ is the set of reachable states for $\mathbf{G}_i$

Let $\mathbf{G} = \mathbf{G}_1 || \mathbf{G}_2 = (Q, \Sigma, \delta, q_0, Q_m)$

Must show

$$(\forall q \in Q_{reach})(\forall s \in \Sigma_{act}^+)\delta(q, s) \neq q$$

We will use proof by contradiction. Assume

$$(\exists q \in Q_{reach})(\exists s' \in \Sigma_{act}^+)\delta(q, s') = q$$

Let $q = (q_1, q_2) \in Q_{reach}$ be this state and let $s' \in \Sigma_{act}^+$ such that $\delta(q, s') = q$.

We know that q is a reachable state if and only if $q_1 \in Q_1$ and $q_2 \in Q_2$ are reachable states in $\mathbf{G}_1$ and $\mathbf{G}_2$, respectively, by Definition of the $||$ operator. We thus have

$$\delta(q, s') = q \implies \delta((q_1, q_2), s') = (q_1, q_2)$$
$$\implies \delta((q_1, q_2), s') = (\delta_1(q_1, P_1(s')), \delta_2(q_2, P_2(s'))) \qquad \text{by Definition of } ||.$$

This implies

$$\delta_1(q_1, P_1(s')) = q_1$$
$$\delta_2(q_2, P_2(s')) = q_2$$

As $s' \in \Sigma_{act}^+$ we thus have $s' \neq \epsilon$. As $\Sigma = \Sigma_1 \cup \Sigma_2$, it follows that either $P_1(s') \neq \epsilon$ or $P_2(s') \neq \epsilon$ This implies that either $\mathbf{G}_1$ or $\mathbf{G}_2$ is not ALF, which contradicts(*).

Therefore it must be that

$$(\forall q \in Q_{reach})(\forall s \in \Sigma_{act}^+)\delta(q, s) \neq q$$

□

The above proposition can be applied to two TDES combined using the meet operator as meet is a special case of the synchronous product.

We next present a proposition that says that to ensure the synchronous product is ALF, it is sufficient that only one of the two TDES is ALF, as long as the event set of the ALF TDES contains all of the events in the event set of the second TDES. It means that if plant is over Σ and the supervisor introduces no new events, then we can just check if the plant is ALF. As indicated by Proposition 2.1, we can check that the plant is ALF by checking if each individual plant component is ALF. Therefore an ALF algorithm does not have to check that the closed loop system is ALF, but can check that the event set of the plant is a superset of the supervisor's event set, then do an ALF check on each individual TDES that makes up the plant. If the check passes, then we are done. Otherwise, we do an ALF check on the entire system.

Proposition 2.2. Let $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2, \delta_2, q_{o,2}, Q_{m,2})$ be two TDES. If $\mathbf{G}_1$ is ALF and $\Sigma_1 \supseteq \Sigma_2$, then $\mathbf{G}_1||\mathbf{G}_2$ is also ALF.

Proof. Assume $\mathbf{G}_1$ is ALF and $\Sigma_1 \supseteq \Sigma_2$. (1)

Let $\mathbf{G} = \mathbf{G}_1||\mathbf{G}_2 = (Q, \Sigma, \delta, q_o, Q_m)$ with $\Sigma = \Sigma_1 \cup \Sigma_2$ and $P_i : \Sigma^* \to \Sigma_i^*$ for $i = 1, 2$. Must show $\mathbf{G}$ is ALF.

We will do so by proof of contradiction.

Assume $\mathbf{G}$ is not ALF, then

$$(\exists q \in Q_{reach})(\exists s' \in \Sigma^+ act)\delta(q, s') = q$$

Let $q = (q_1, q_2) \in Q_{reach}$, and $s' \in \Sigma_{act}^+$ such that $\delta(q, s') = q$. (2)

We first note that q is reachable in $\mathbf{G}$, which implies q_1 is reachable in $\mathbf{G}_1$ and q_2 is reachable in $\mathbf{G}_2$.

We next note that as $\Sigma_1 \supseteq \Sigma_2$, we have $\Sigma = \Sigma_1 \cup \Sigma_2 = \Sigma_1$. This implies that $P_1^{-1}L(\mathbf{G}_1) = L(\mathbf{G}_1)$. (3)

From (2), we have

$$\begin{array}{lcr} & \delta(q, s') = q \implies \delta((q_1, q_2), s') = (q_1, q_2) & \\ \implies & \delta_1(q_1, P_1(s')) = q_1 & \\ \implies & \delta_1(q_1, s') = q_1 & \text{by (3)} \end{array}$$

This contradicts (1) as it implies $\mathbf{G}_1$ is not ALF.

We thus conclude that $\mathbf{G}$ must be ALF. □

We are also want to make sure that the plant is not modeled in such a way that our closed loop system could reach a state where no more tick events are possible, as this "stopping the clock" would be physically unrealistic. To help prevent this, we will require that our plant TDES have *proper time behavior*, as defined by Kai Wong et al. [28].

Definition 2.3.5. TDES $\mathbf{G}$ has a *proper time behavior* if

$$(\forall s \in L(\mathbf{G}))\mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u = \emptyset \implies \tau \in \mathrm{Elig}_{L(\mathbf{G})}(s)$$

This definition can be rewritten as

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

In other words, this TDES must guarantee that at all of its reachable states, either a tick event or an uncontrollable event must be possible. This serves two purposes. Combined with TDES $\mathbf{G}$ being ALF and having a finite state space, this ensures that we call always reach a state where a tick is possible after at most a finite number of activity events. We prove this shortly in Proposition 2.3. This condition will also ensure we do not stop the clock when we combine our plant with a controllable supervisor. An example that fails the proper time behavior property is shown in Figure 2.3 where after the first tick event, neither an uncontrollable event or a tick are possible, only the prohibitable event β.

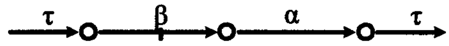


Figure 2.3: An Example Failing the Proper Time Behavior Property

Consider the case where we have a reachable state where tick was ineligible, but only controllable events were possible. If the supervisor disabled these controllable events, there would now be no events possible at all. Proper time behavior ensures that if tick was not possible at this state in the plant, there would be an uncontrollable event possible, even if all the controllable events were disabled. The restriction of proper time behavior applies only to plant TDES. It does not apply to supervisor TDES or the meet of the plant and supervisor (i.e. the closed loop behavior of the system).

If a TDES G has a finite state space, is activity loop free and has proper time behavior, then we expect that at any reachable state, we can always do a tick event after at most a finite number of activity events. In other words, we will never "stop the clock." The following proposition shows that this is indeed the case.

Proposition 2.3. If a TDES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ has a finite statespace, is activity loop free and has proper time behavior, then

$$(\forall q \in Q_{reach})(\exists s \in \Sigma^*)\delta(q, s\tau)!$$

where Q_{reach} is the set of reachable states.

Proof. Assume that $\mathbf{G}$ has a finite statespace, is activity loop free, and has proper time behavior

Let $q \in Q_{reach}$.

Must show implies $(\exists s \in \Sigma^*)\delta(q, s\tau)!$

We first note that as $\mathbf{G}$ has a finite statespace and is non-empty, there exists $n \in \{1, 2, \ldots\}$ such that $n = |Q|$.

As $\mathbf{G}$ is ALF and has n states, it follows that

$$\begin{aligned} &(\exists s \in \Sigma_{act}^{+})|s| \leq n-1 && \text{and} \\ &(\exists q' \in Q_{reach})\delta(q, s) = q' && \text{and} \\ &(\forall \sigma \in \Sigma_{act})\delta(q', \sigma) \not! && (1) \end{aligned}$$

The above follows from the fact that starting at q, we can do at most $n-1$ activity event transitions before we have visited all n states. At this point, there must be no activity event transition or we would have to visit a state twice, creating an activity loop and failing the ALF definition.

As $\Sigma_u \subseteq \Sigma_{act}$, (1) asserts that there are no uncontrollable events at state q'. It thus follows that $\delta(q', \tau)!)$ as $\mathbf{G}$ has proper time behavior.

We thus have:

$$\delta(q, s\tau)!$$

as required. □

We now present the controllability definition for timed DES. Normally, we drop the "TDES" and just say "controllable" as the meaning is clear from the context.

Definition 2.3.6. We define the arbitrary language $K \subseteq L(\mathbf{G})$ to be *TDES controllable* with respect to $\mathbf{G}$ if,

$$(\forall s \in \overline{K})\mathrm{Elig}_{\overline{K}}(s) \supseteq \begin{cases} \mathrm{Elig}_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } \mathrm{Elig}_{\overline{K}}(s) \cap \Sigma_{for} = \emptyset \\ \mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } \mathrm{Elig}_{\overline{K}}(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

Definition 2.3.6 says that a $\overline{K}$ must accept an uncontrollable event if the event is possible in the plant, and it must accept a tick event if it is possible in the plant, unless there exists an eligible forcible event that can preempt the tick.

Note that the closed and marked behavior of a TDES is defined in the same way as for an untimed DES. A TDES is said to be *nonblocking* if Definition 2.2.6 is satisfied.

Proposition 2.4. If TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ both have finite statespaces, $\mathbf{G}$ has proper time behavior, $\mathbf{G}_{cl} = \mathbf{meet}(\mathbf{G}, \mathbf{S}) = (Q, \Sigma, \delta, q_0, Q_m)$ is ALF, and $\mathbf{S}$ is controllable for $\mathbf{G}$, then

$$(\forall q \in Q_{reach})(\exists s \in \Sigma^*)\delta(q, s\tau)!$$

Proof. Assume:

- $\mathbf{G}$ and $\mathbf{S}$ have finite statespaces
- $\mathbf{G}$ has proper time behavior
- $\mathbf{G}_{cl}$ is ALF
- $\mathbf{S}$ is controllable for $\mathbf{G}$

Let $q \in Q_{reach}$. Must show $(\exists s \in \Sigma^*)\delta(q, s\tau)!$

As $\mathbf{G}$ and $\mathbf{S}$ have finite statespaces, it follows from Definition 2.2.12 of the **meet** operator, that $\mathbf{G}_{cl}$ has a finite statespace. Let $n = |Q|$.

As $\mathbf{G}_{cl}$ is ALF and has n states, it follows that

$$\begin{aligned} (\exists s \in \Sigma_{act}^{+})|s| \leq n-1 & \qquad \text{and} \\ (\exists q' \in Q_{reach})\delta(q,s) = q' & \qquad \text{and} \\ (\forall \sigma \in \Sigma_{act})\delta(q',\sigma) \not! & \qquad (1) \end{aligned}$$

The above follows from the fact that starting at q, we can do at most $n-1$ activity event transitions before we have visited all n states. At this point, there must be no more activity event transitions or we would have to visit a state twice, creating an activity loop and failing the ALF definition.

We now need to show tick is defined at q'. From (1), we know that there are no untimed events possible in $\mathbf{G}_{cl}$ at q' as $\Sigma_u \subseteq \Sigma_{act}$. As $\mathbf{S}$ is controllable for $\mathbf{G}$, this implies there are no untimed events possible at the corresponding state in $\mathbf{G}$. As $\mathbf{G}$ has proper time behavior, this implies that τ is possible at this state in $\mathbf{G}$. As (1) asserts there are no activity event at q' and thus no forcible events, $\mathbf{S}$ must accept that tick event as $\mathbf{S}$ is controllable for $\mathbf{G}$.

$$\begin{aligned} \Longrightarrow \quad & \delta(q',\tau)! \\ \Longrightarrow \quad & \delta(q,s\tau)! \end{aligned}$$

$\square$

Chapter 3

Sampled-Data Systems

In this thesis, we will focus on implementing our TDES supervisors as sample-data (SD) controllers. An SD controller is driven by a periodic clock and sees the system as a series of inputs and outputs. On each clock edge, it samples its inputs, changes states, and updates its outputs. For simplicity, we will assume inputs and outputs of an FSM can only take the value of true or false.

When we are using an SD controller to manage a given system, we associate an input with each event, and an output with each controllable event. We consider an event has occurred when its corresponding input has gone true during a given clock period. We consider a controllable event to be enabled when its corresponding output has been set true by the controller, disabled otherwise.

As mentioned above, an SD controller samples the value of its inputs on each clock edge, and uses this value to decide what its next internal state will be. This means the SD controller knows nothing about its inputs until the clock edge, and then all it learns is whether a given input is true or false, signifying that the corresponding event has occurred sometime in the clock period that just ended. This means that for the given clock period, all information about event ordering (which event occurred first etc) is lost, as well as how often a given event occurred if it has occurred more than once. The only ordering information that remains is which *sampling period* (clock period) a given event occurred in.

Another important aspect of an SD controller is that it only changes state on a clock edge, and the value of its outputs are a function of its current state. That means its outputs can only change at a clock edge, and then must stay constant for the rest of the clock period.

In this chapter, we will define the sampled-data setting formally, and develop a new condition to address the issues we identified in Section 1.1.

We will be making a few assumptions about the systems we work with. They are:

- The set of prohibitable events is exactly equal to the set of forcible events for our system. This is a reasonable assumption that will greatly simplify things. As discussed in the introduction, this is basically a matter of how the system is modeled.

- Our SD controllers will be implemented centrally with a common clock, such that they all sample inputs, and update outputs at the same time. Furthermore, their source of inputs and outputs is common such that their outputs exit to the system at the same place, and their inputs enter from the system at the same place. For their inputs, this means they will always all receive the same results from the sampling inputs. We will never have the case that one controller sees input α go true in a given sampling period, while another does not.

- When a prohibitable event is enabled, we will interpret this to mean we should force the event once in the current clock period. Even if we could cause it to occur twice in one clock period, we will not do that.

- To partially address timing issues, we will assume an event has occurred when its input to the controllers goes true. One exception is if the input goes true so close to a clock edge that it is missed and shows up in the next sampling period. In this case, the event is considered to have occurred at the start of the next sampling period. This should be taken into account in the modeling of the system.

- We are also assuming that when we decide to force an event in a given sampling period, not only will the event physically occur in that sampling period, but it will reach our controller's inputs in time to be detected as occurring in that sampling period, and never in the following one. It is up to the designer and user of this theory to make sure that the system they apply it to satisfies these assumptions.

- The input signal should be of an appropriate length so that it will not be missed by the SD controllers (i.e. if its pulse width is shorter than the clock period), nor should it be so long that it is seen at multiple clock edges, unless it is suppose to represent that number of sequential occurrences. For example, if the input is true for two clock edges in a row, it will be considered to have occurred twice, once per clock period. It is the designers responsibility to make sure that the inputs are properly conditioned to ensure this.

3.1 Sampling Inputs

To make the TDES theory work with SD controllers, we identify a tick event occurring with the clock edge that the SD controller uses for sampling and state change. This means for a TDES $\mathbf{G}$ over event set Σ, the strings an SD controller can observe from the closed behavior of $\mathbf{G}$ are strings ending with a tick and the empty string, ϵ. We will refer to such strings as *sampled strings.* The reason the empty string is included is that it represents the initial state of the system, which is usually known. Note also that a non-empty sampled string may contain one or more tick events in addition to the tick event at the end of the string.

Definition 3.1.1. Given a event set Σ, the set of *sampled strings* is denoted by L_{samp} and is define as

$$L_{samp} = \Sigma^{*}.\tau \cup \{\epsilon\}$$

As an SD controller will change from state at each clock edge (tick occurring), the next state of the SD controller will thus be determined by the strings containing a single tick at the end that are possible in the system immediately after the last tick event that brought us to our current state. We will refer to such strings as *concurrent strings*, defined as below. Essentially, an SD controller starts at its initial, or *reset* state (corresponding to the empty string), and then transitions from state to state as concurrent strings occur in the corresponding TDES.

Definition 3.1.2. Given an event set Σ, we denote the set of *concurrent strings* as L_{conc}, defined as

$$L_{conc} = \Sigma_{act}^{*}.tick \subset L_{samp}$$

Obviously, L_{conc} is a strict subset of L_{samp} since the empty string is not found in L_{conc}.

Next, we want to capture the idea that an SD controller cannot tell the difference between two nonidentical concurrent strings if they contain exactly the same activity events but in a different order, and/or one or more event have a different number of occurrences. For example, strings $\alpha\beta\tau$, $\beta\alpha\tau$ and $\alpha\beta\alpha\tau$ would all appear the same to an SD controller. We now give the definition of the occurrence operator. It takes a string and returns the set of events (the *occurrence image*) that make up the string. Essentially, if two concurrent strings have the same occurrence image, they are indistinguishable to an SD controller.

Definition 3.1.3. For $s \in \Sigma^*$, the *occurrence operator* is a function Occu : $\Sigma^* \rightarrow \mathrm{Pwr}(\Sigma)$ defined as below

$$\mathrm{Occu}(s) := \{\sigma \in \Sigma \mid s \in \Sigma^*.\sigma.\Sigma^*\}$$

As an SD controller only gets information about the system it is controlling at sampling instances (ticks), sampled strings represent observable points in the system. Considering a TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, states reached by sampling strings represents states in $\mathbf{S}$ that are at least partially observable. We refer to such states as sampling states, and define them formally below.

Definition 3.1.4. A state $x \in X$ from TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, is a *sampling state* for $\mathbf{S}$ if

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

We refer to $X_{samp} \subseteq X$ as the set of sampling states for $\mathbf{S}$. Note that since $\epsilon \in L_{samp}$, $x_o \in X_{samp}$ by definition. In other words, the initial state is always observable at least once. It is worth noting that their could exist strings in $L(\mathbf{S})$ that take us to a sampled state x, but the strings are not sampled strings. These do not represent observable points, and means that a given sampled state may not always be observable relative to $L(\mathbf{S})$. As far as an SD controller is concerned, the system it is observing starts in its initial state, and then goes from sampled state to sampled state via concurrent strings.

If we wished to convert a TDES **S** into an SD controller, we make the initial state of **S** the start state of the SD controller. We would then determine which concurrent strings are possible from this state. The sampled states of **S** reached by these strings will become states of the controller, and the occurrence image of the concurrent strings would define our next state conditions.

Our translation has a problem if we have two concurrent strings with the same occurrence image, but that take us to different states of **S**. This would mean our SD controller would be nondeterministic. To prevent this, we introduce the concept of CS deterministic, stated formally below. In essence, it requires that if the two concurrent strings possible at a sampled state in **S** have the same occurrence image, they take us to the same next state in **S**. It's possible that the two strings could take us to two different states, but the states are λ-equivalent. If we determine that the strings satisfy the nerode equivalence portion of the requirement, but do not take us to the same state, we can simply merge these states in **S** as they are equivalent. Note that we do not require that **S** be minimal, just minimal with respect to the states we care about which is a cheaper condition to check. The CS deterministic definition will also be useful in making sure a given TDES has the correct structure such that we can represent its sampled-data behavior.

Definition 3.1.5. A TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ is *concurrent string deterministic* or *CS deterministic*, if

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

It is worth noting that SD controllers are concerned with enabling and forcing prohibitable events, and not with marking strings. All an SD controller cares about is that two strings have the same future with respect to the system's closed behavior.

Following Definition 3.1.5 will ensure our controller is deterministic, but we may end up with some redundant states that we can later minimize using standard digital logic techniques [7] for synchronous finite state machines.

For CS deterministic TDES, we now wish to define some of the tools we will need to express the sampled-data behavior of a TDES. This will be useful when we want to talk about the behavior of a plant under the control of an SD controller, and compare it to the TDES behavior of the plant under the control of its TDES supervisor. The first thing we need to do is define for a given TDES, a *next sampling state function.* This will represent how a TDES will move from sampling state to sampling state via concurrent strings.

Definition 3.1.6. For the CS deterministic TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, we define the partial function, *next sampling state function*

$$\Delta : X_{samp} \times \mathrm{Pwr}(\Sigma_{act}) \rightarrow X_{samp}$$

as follows. For $x \in X_{samp}$ and $\Sigma' \subseteq \Sigma_{act}$,

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

For the special case $\Sigma' = \emptyset$, $\Delta(x, \Sigma')$ can still be defined according to the definition. It just returns a sampling state $x' = \xi(x, \tau)$, which means that no event except a tick has occurred during the last sampling period. In analogy to the DES transition function, we write $\Delta(x, \Sigma')!$ if $\Delta(x, \Sigma')$ is defined.

As a precondition for the definition of Δ, we require that the TDES be CS deterministic. This means that two concurrent strings with the same occurrence image will take us to exactly the same state in $\mathbf{S}$. For CS deterministic TDES, this means that Δ is well defined.

To see how a non CS deterministic TDES would cause problems, consider Figure 3.1. For this example, let $\alpha, \beta \in \Sigma_{act}$ and $x_n, x', x'' \in X_{samp}$ for some TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$. In Figure 3.1, part (a) shows the only portion of $\mathbf{S}$ that is not minimized, such that $s' = \alpha\beta\tau$ and $s'' = \beta\alpha\tau$ end up at two different states, x' and x'' respectively. But (b) shows the minimized version where x' and x'' have been

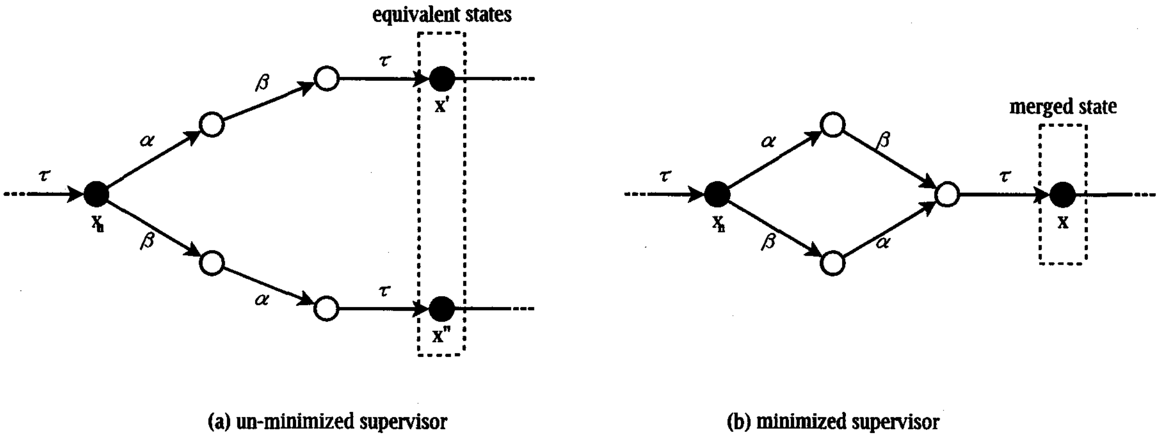


Figure 3.1: Nonminimal Example

merge into a single state x. Clearly in (a), $\mathrm{Occu}(s') \cap \Sigma_{act} = \mathrm{Occu}(s'') \cap \Sigma_{act}$ but $\xi(x_n, s') \neq \xi(x_n, s'')$, which would mean that δ is not well-defined. However in (b), everything is fine. Another problem would be if x' and x'' were not λ-equivalent. This would mean that we cannot merge the two states, and again δ would not be well defined.

3.2 SD Controllable Languages

So far, we have required that our TDES system have a finite statespace, be ALF and nonblocking, that our plant have proper time behavior and be complete for our supervisor. and that our supervisor be controllable for our plant. However, these conditions are not sufficient to address the concerns that we raised in Section 1.1. In particular, we saw that even though the above conditions are met, our actual system behavior under the control of the corresponding SD controller could block, violate our control law, or even exhibit behavior not contained in our plant model.

To address these issues, we now introduce a new concept called *SD controllable languages*, defined below. Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a TDES where $\Sigma = \Sigma_c \dot\cup \Sigma_u$ for controllable and uncontrollable events. Of course, for a TDES system, $\Sigma_c = \Sigma_{hib} \cup \{\tau\}$. As we will see, this new condition implies TDES controllability, thus we do not have to test for this condition separately.

It should be noted that the condition we are presenting is a bit conservative. If a system fails it, there may be some situations where things are still fine. Our goal here is to provide a set of conditions that should ensure correct behavior when we implement our TDES supervisors, and be general and flexible enough to apply to a wide range of systems, yet be reasonable conditions to evaluate.

Definition 3.2.1. A language $K \subseteq \Sigma^*$ is *SD Controllable* with respect to $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ if, $\forall s \in \overline{K} \cap L(\mathbf{G})$, the following statements are satisfied:

i) $\mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq \mathrm{Elig}_{\overline{K}}(s)$

ii) If $\tau \in \mathrm{Elig}_{L(\mathbf{G})}(s)$ then

$$\tau \in \mathrm{Elig}_{\overline{K}}(s) \Leftrightarrow \mathrm{Elig}_{\overline{K} \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$$

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

1. $(\forall s' \in \Sigma_{act}^*)[ss' \in \overline{K} \cap L(\mathbf{G})] \Rightarrow$
$$[\mathrm{Elig}_{\overline{K} \cap L(\mathbf{G})}(ss') \cup \mathrm{Occu}(s')] \cap \Sigma_{hib} = \mathrm{Elig}_{\overline{K} \cap L(\mathbf{G})}(s) \cap \Sigma_{hib}$$
2. $(\forall s', s'' \in L_{conc})\ [ss', ss'' \in \overline{K} \cap L(\mathbf{G}) \wedge \mathrm{Occu}(s') = \mathrm{Occu}(s'')] \Rightarrow$
$$ss' \equiv_{\overline{K} \cap L(\mathbf{G})} ss'' \wedge ss' \equiv_{K \cap L_m(\mathbf{G})} ss''$$

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

Point i This is the standard untimed controllability definition and is part of TDES controllability. Intuitively, any uncontrollable events eligible in $\mathbf{G}$ may not be disabled.

Point ii If both a prohibitable event and tick event are enabled and eligible, it will be ambiguous in which clock period the event should occur in. Also, a supervisor must not disable a tick unless there exists a prohibitable (forcible)[1] event to preempt the tick. The if and only if part only applies if the tick event is eligible in the plant.

The $\Rightarrow$ part states that a tick event must be disabled by $\overline{K}$ if there is an eligible prohibitable event. This is done to ensure that prohibitable events are disabled

[1]Remember, we have required that the set of prohibitable events be equal to the set of forcible events.

until they should occur and then they are immediately forced. In other words, it means forcing and enabling are essentially one and the same. This is to make it clear which clock period a prohibitable event should occur in. This in turn will make translating to an SD controller much simpler and straightforward. Part of the goal of this definition is to make the behavior specified by the TDES as close as possible to that which is possible with the actual SD controller. In this case, the SD controller needs to know exactly when to force an event. A range of possible clock periods is no good to it.

The $\Leftarrow$ part states that a tick event cannot be disabled unless there exists an eligible prohibitable event to preempt the tick. Together with **Point i**, this is equivalent to TDES controllability (Definition 2.3.6).

Point iii The following two points are needed when s is a sampled string.

1) This condition says that the set of prohibitable events eligible in $\overline{K}$ and $L(\mathbf{G})$ after sampled string s (i.e. immediately after a tick occurs (clock edge)) must stay equal to the union of the prohibitable events still eligible, and the prohibitable events that have already occurred since the last tick. In other words, the prohibitable events eligible after the tick must stay eligible until they occur, and no new prohibitable events may become eligible until after the next tick.

This condition is meant to capture two concepts. The first is that since an SD controller only can observe the system at a clock edge (tick event), its enablement and forcing decisions are determined by its current state, and must be constant until the next tick occurs. These cannot change during the current clock cycle in response to events occurring, as it will not know they have occurred until after the next tick, which would be too late.

The second concept is that an SD controller decides to force an event immediately after a tick, based on the information it has at that point (i.e. whether the event is currently enabled and eligible in the plant). Once it decides to force the event, it will occur at some point during the current clock period. So as to not violate the control law or the plant model, this event must stay eligible and enabled until it occurs. This is important as we do not know exactly when this event will actually occur, due to the fact that different implementations of our controller could have different timing characteristics. We thus have to ensure that when it does occur, it does not violate our control law, nor exceed the behavior of our plant model.

A side effect of this condition is that it means that we only have to look at the eligibility and enabling information for prohibitable events at the state reached by a tick, and this determines the information for the clock cycle. This makes the conversion to an SD controller easier.

2) This condition says that if sampled string s can be extended by concurrent strings s' and s'' which have the same occurrence image (and thus indistinguishable to an SD controller), then string ss' will be Nerode equivalent to string ss'' with respect to the system's closed and marked behavior. In other words strings ss' and ss'' will have the same closed and marked future. From a TDES perspective, this means that strings ss' and ss'' will go to states that are λ-equivalent. If the TDES is minimal, this will mean the same state. Otherwise, we may need to check that the two states are λ-equivalent.

This condition is intended to address two issues. The first is the fact that since the SD controller cannot tell the difference between strings s' and s'', it must take the same control action following either string, both now and in the future. We can capture this by requiring them to have the same future with respect to the system's closed behavior.

The second issue has to do with nonblocking. Depending on the implementation of our SD controller, it maybe the case that we may either always get the string s' and never s'', or vice-versa. If s'' never actually occurs in the physical system and it is part of the only path back to a marked state, the physical system would block despite the fact the TDES system is nonblocking. By requiring the two strings to have the same marked future, it will not matter which one we actually get, as long as all of the marked strings in the system are also sampled strings (see **Point iv** for more info on this). In a way, we are ensuring that our system will still be nonblocking for a set of possible closed loop behaviors, that differ by which of these concurrent strings can actually happen in the physical system.

Point iv This point says that all marked strings in the closed loop system must be sampled strings. The primary reason is that sampled strings represent observable points in the system. This makes sure that we do not mark a non empty strict substring of a concurrent string accepted by the system. We saw in **Point iii.2** that two concurrent strings with the same occurrence image have the same marked future, but the condition says nothing about Σ_{act}^{+} substrings of these concurrent strings. **Point iii.2** basically says that even if we only get one of the two concurrent strings, we can still get to a new sampled state with an equivalent marked future. i.e. we might lose one of the paths to this sampled state, but we can still get there. However, if we allow marking along the path between sampled states and that is the path we lose, we may no longer be able to reach a marked state. Hence, we require all marked strings to take us to sampled states.

So far, we have only discussed controllable languages. To extend this concept to a TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, we identify $K = L_m(\mathbf{S})$ and $\overline{K} = L(\mathbf{S})$ in Definition 3.2.1.[2] This gives us the definition below. Note that the definition is implicitly assuming that $\mathbf{G}$ and $\mathbf{S}$ are combined using the **meet** operator. If instead we had a plant $\mathbf{G}'$ and supervisor $\mathbf{S}'$ combined using the synchronous product operator resulting in system event set Σ, we would first construct plant $\mathbf{G}$ from $\mathbf{G}'$ by adding selfloops of any events missing from Σ, and supervisor $\mathbf{S}$ from $\mathbf{S}'$ by again adding needed selfloops. We can then apply the definition below to the these new TDES.

Definition 3.2.2. A supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ is said to be *SD controllable* with respect to $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ if, $\forall s \in L(\mathbf{S}) \cap L(\mathbf{G})$, the following statements are satisfied:

i) $\mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq \mathrm{Elig}_{L(\mathbf{S})}(s)$

ii) If $\tau \in \mathrm{Elig}_{L(\mathbf{G})}(s)$ then

$$\tau \in \mathrm{Elig}_{L(\mathbf{S})}(s) \Leftrightarrow \mathrm{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$$

[2]By "identify," we mean make the indicated replacements in the original definition to get the new definition. We do not mean to imply that we require that **S** be nonblocking.

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

1. $(\forall s' \in \Sigma^*_{act})[ss' \in L(\mathbf{S}) \cap L(\mathbf{G})] \implies$
$$[\mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(ss') \cup \mathrm{Occu}(s')] \cap \Sigma_{hib} = \mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib}$$
2. $(\forall s', s'' \in L_{conc})\ [ss', ss'' \in L(\mathbf{S}) \cap L(\mathbf{G}) \wedge \mathrm{Occu}(s') = \mathrm{Occu}(s'')] \Rightarrow$
$$ss' \equiv_{L(\mathbf{S})\cap L(\mathbf{G})} ss'' \wedge ss' \equiv_{L_m(\mathbf{S})\cap L_m(\mathbf{G})} ss''$$

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

We now discuss a few examples to illustrate the above definition, starting with **Point ii**. We do not give an example for **Point i** or **Point iii.2** since the first is essentially untimed controllability, and the second is similar to the CS Deterministic property discussed in Section 3.1.

Figure 3.2 shows an example where prohibitable event α and a tick are both possible at the same state in the plant. When our supervisor decided to enable α here, **Point ii** required that tick must be disabled. Also, **Point ii** only allowed us to disable tick here as forcible event α was possible in both the plant and supervisor to preempt the tick.

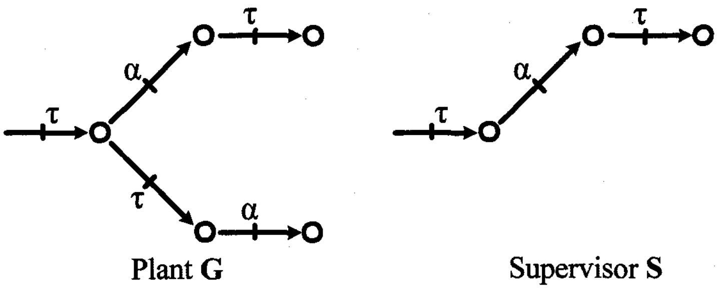
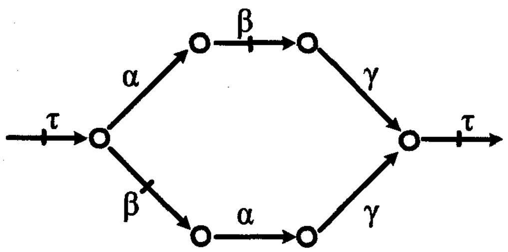


Figure 3.2: An Example for **Point ii**

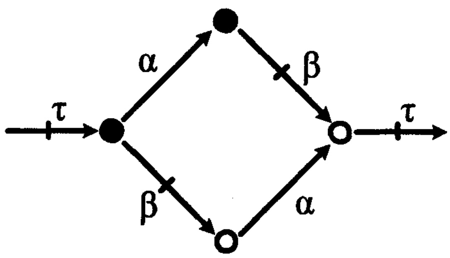
Figure 3.3 shows an example for **Point iii.1**. In the diagram, we see that the only prohibitable event possible after the tick is β. We see that β stays possible until it occurs on both paths, and no new prohibitable events become eligible before the next tick.

Figure 3.4 shows an example that fails **Point iv**. Here we see that the state reached by the first tick is marked which is allowed, but then the state reached by α is also marked, which is not.



meet(G, S)

Figure 3.3: An Example for **Point iii.1**



meet(G, S)

Figure 3.4: An Example Failing **Point iv**

Note that Definition 3.2.2 is not closed under arbitrary union. An example is shown in Figure 3.5, where (a) and (b) are two TDES supervisors that enable and force only one event respectively. In (a), α is forced and β is disabled. In (b), β is forced and α is disabled. It can be shown that both (a) and (b) are SD controllable for our plant shown in (d), but the union of these two languages, shown in (c), is not. The supervisor in (c) fails **Point iii.1** as both α and β are possible at the initial state, but once one occurs, the other is disabled before the next tick has occurred.

This example suggests that in general, there may not exist a supremal SD controllable sublanguage. For this example, there appears to be two maximal sublanguages but no supremal sublanguage. This likely follows from the fact that in normal TDES controllability, the maximally permissive supervisor might allow several choices as they are each safe, and leave it up to an unmodeled agent to decide which option occurs. As they are all possible, eventually we should get all choices. However for SD controllers, we make the choice with respect to which clock cycle an event gets forced in, meaning that some of these choices might vanish. If two choices are mutual disjoint yet equal in terms of size of behavior we would get, we end up with two or more maximal solutions, and no supremal solution.

We now add another tool that we will need to express the sampled-data behavior of a TDES. We will now define the control action that will take place at a sampling state for our TDES. This is the action the SD controller will take during the corresponding sampling period.

Definition 3.2.3. Let TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be SD controllable with respect to plant $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$. The *control action* $\zeta : X_{samp} \rightarrow \mathrm{Pwr}(\Sigma_{hib})$ is defined for $x \in X_{samp} \subseteq X$ as follows:

$$\zeta(x) := \{\sigma \in \Sigma_{hib} | \xi(x, \sigma)!\}$$

Proposition 3.1. For TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ which is SD controllable with respect to plant $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$, we have

$$(\forall s \in L(\mathbf{S}) \cap L_{samp})\zeta(x) = \mathrm{Elig}_{L(\mathbf{S})}(s) \cap \Sigma_{hib}$$

where $x = \xi(x_o, s)$.

Proof. This follows immediately from the definition of $L(\mathbf{S})$ and the Elig operator. □

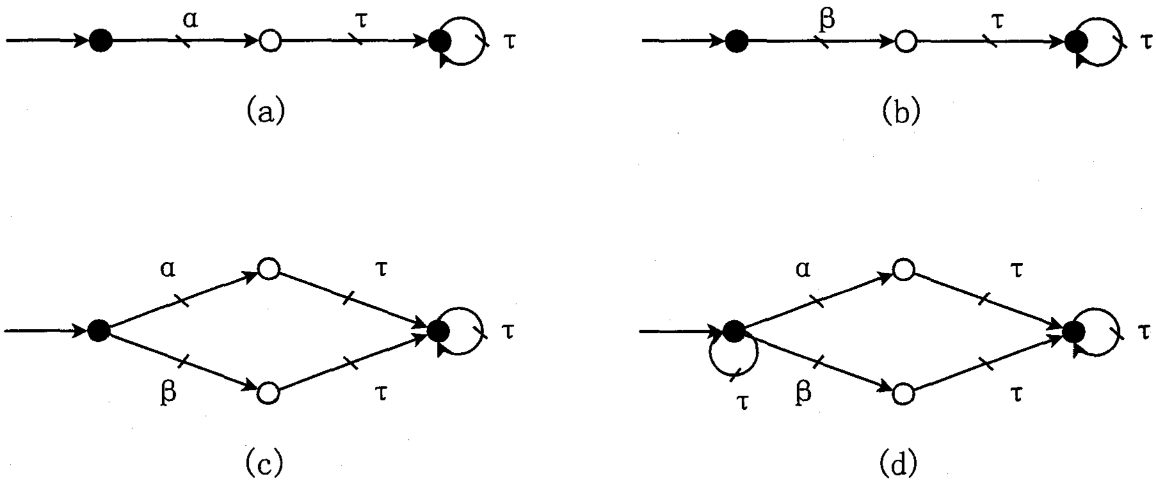


Figure 3.5: SD Controllability and Arbitrary Union.

We close this chapter with a proposition pointing out the connection of our CS deterministic definition and **Point iii.2** of the SD controllability definition.

Proposition 3.2. If TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ is SD controllable for plant $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$, then $\mathbf{meet}(\mathbf{S}, \mathbf{G})$ is CS deterministic if it is minimal.

Proof. Follows automatically from **Point iii.2** in Definition 3.2.2. □

However, an SD controllable supervisor $\mathbf{S}$ with respect to plant $\mathbf{G}$ does not imply that $\mathbf{S}$ is CS deterministic by itself, because of the dependency of plant $\mathbf{G}$ in the definition of SD controllability. We use the CS deterministic property when we wish to only discuss the supervisor, instead of the closed loop behavior of the system.

3.3 Future Work

In this thesis, we have presented some new conditions and methods that are intended to address the concurrency and implementation issues raised in Section 1.1. However, we only partly dealt with time delay issues which we have left as future work due to time considerations.

We have tried to mitigate potential time delay problems by the assumptions we have made at the beginning of Chapter 3. Here, we have required that our controllers be implemented on a single machine, that they use a common clock, that they all see the result of a common sampling of the inputs, and that their outputs change at about the same time. These restrictions should protect against time delay issues caused by a distributed implementation of controllers, where they could sample inputs at different times, update enablement information at different times, and this information could reach the plant at different times.

Another potential time delay problem is the difference between when an event physically occurs (say a part arrives at a machine), and when a controller sees that the event has occurred. For instance, the event might physically occur in sampling period k, but due to transmission delay, it does not reach the input of the controller until the next clock cycle, so the controller "sees" it one clock cycle late. It is even possible that the signal could reach the input right at the clock edge, and thus is not noticed till the next clock edge. All of these issues could cause the system that the controller "sees" to have slightly different timing information from the formal model.

We have tried to compensate for this by assuming that an event has occurred when its corresponding input goes true at the controller, with one exception. The exception is when the input goes true so close to the clock edge, it does not show up till the next sampling period. In this case, the event is assumed to happen just after the clock edge. We then model the system with this interpretation of what it means for an event to occur, in particular with respect to the timing of the events.

Whereas the steps we have taken to compensate for timing delay are not ideal, they should handle the more pressing issues. However, research needs to be done to identify the existing timing delay issues, and address them directly in a more flexible manner.

Chapter 4

Moore Synchronous Finite State Machines

A Moore state machine is a type of finite state machines introduced by Edward F. Moore in [17]. It chooses its next states based on its current state and inputs. Its outputs are determined by its current state only. We will use Moore state machines with clocked systems whose states change only on a rising or falling edge of the clock. Its current output remains the same until the state is changed again. A Moore state machine used in this way is called a *Moore Synchronous Finite State Machine.* In the following discussion, we simply use *Moore machine* or FSM for convenience.

By the properties defined in Chapter 3, an SD Controller can be modeled as a Moore machine. In the following pages, we will first define a formal model for our SD controller in Section 4.1. Then, in Section 4.2 we will introduce translations methods for a centralized controller and for modular controllers. The translation methods require that the given supervisors be CS deterministic and non-selfloop ALF, as defined in Section 3.1 and Section 2.3. Note that we can translate a supervisor as long as its CS deterministic, but it would likely be very hard to evaluate the CS deterministic condition if the TDES is not ALF or non-selfloop ALF, as we would essentially have an infinite number of concurrent strings to evaluate. It is also quite likely such a system would fail the CS deterministic condition. Requiring that the TDES also be ALF or the weaker non-selfloop ALF makes everything easier, and still gives us a general solution as a non ALF system is not physically realistic.

4.1 Formal Model

In this chapter, we will often be discussing vectors of information that will change periodically with respect to some clock. Let $k \in \{0, 1, 2, ..\}$. We will say "at time k" to indicate the point of time at which k clock ticks have gone by since our starting reference point, which we represent as $k = 0$. For any vector $\mathbf{v} = [v_1, v_2, ..., v_n] \in V$ or any of its element v_j , we write "$\mathbf{v}(k)$" and "$v_j(k)$" to denote the value of $\mathbf{v}$ and v_j at time k. Note that $\mathbf{v}(k)$ is not a function of k, but a notation to differentiate the value of $\mathbf{v}$ at different points in time. For $k = 0$, $\mathbf{v}(0)$ represents the initial or starting value of $\mathbf{v}$. When we are discussing an SD controller, we can think of $k = 0$ as representing the time when the controller has just been turned on.

We can think of when k is incremented as the occurrence of a tick from our clock. With respect to a TDES system, this would correspond to the occurrence of the tick event. As such, k induces a sequence for vector $\mathbf{v}$ with respect to these clock ticks, which we define to be $\{\mathbf{v}(k)|k = 0, 1, ...\}$, and is denoted as $\{\mathbf{v}(k)\}$ as a shorthand.

Assumption 4.1. *For convenience, we assume every controller is operating based on the same global clock, so that they change state at the same time.*

Given a TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, we will refer to the implementation of $\mathbf{S}$ as its corresponding *SD controller*. We now give a formal definition of SD controllers.

Definition 4.1.1. An SD controller $\mathbf{C}$ is represented by a Moore machine defined as follows.

$$\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$$

where,

I is the set of possible Boolean vectors that the inputs to our controller can take on. Each vector $\mathbf{i} \in I$ has v input variables, such that

$$\mathbf{i} = [i_0, i_1, .., i_{v-1}]; \;\; i_j \in \{0, 1\}; \;\; j = 0, 1, .., v - 1$$

Each input vector $\mathbf{i}(k') \in \{\mathbf{i}(k)\}$ is sampled at the occurrence of a tick event, except for $k = 0$ which occurs when the controller is turned on.

Each element of I corresponds to a unique activity event in our system. If that element equals "1" at time k, then that means the event has occurred at least once since that last clock tick. If it equals zero, then it means the corresponding event has not occurred at all since the last clock tick.

Z is the set of possible Boolean vectors that the controller outputs can take on. Each vector $\mathbf{z} \in Z$ has r output variables, such that

$$\mathbf{z} = [z_0, z_1, .., z_{r-1}];\ z_j \in \{0, 1\};\ j = 0, 1, .., r-1$$

Each input vector $\mathbf{z}(k') \in \{\mathbf{z}(k)\}$ is generated at the occurrence of the tick event, except for $k = 0$ which occurs when the controller is turned on. Note that we do not provide separate outputs for forcing, because the forcing of an event is already implied by enabling the event.

The values of vector Z represent enablement information for our prohibitable events. A value of '1' means the event is enabled, while '0' means the event is disabled.

Q is the set of possible Boolean vectors that the state of our controller can take on. Each vector $\mathbf{q} \in Q$ has l state variables for state identification, such that

$$\mathbf{q} = [q_0, q_1, .., q_{l-1}];\ q_j \in \{0, 1\};\ j = 0, 1, .., l-1$$

Each state $\mathbf{q}(k') \in \{\mathbf{q}(k)\}$ changes to next state $\mathbf{q}(k'+1) \in \{\mathbf{q}(k)\}$ at the occurrence of the tick event, starting at $k = 1$.

$\mathbf{q}_{res}$ is the default state when the machine is reset or initialized. We take $\mathbf{q}(0) = q_{res}$.

$\Omega : Q \times I \rightarrow Q$ is a next state function which takes the current state $\mathbf{q}(k) \in Q$ and an input vector $\mathbf{i}(k+1) \in I$, and returns the next state $\mathbf{q}(k+1) \in Q$.

$$\mathbf{q}(k+1) = \Omega(\mathbf{q}(k), \mathbf{i}(k+1))$$

$\Phi : Q \rightarrow Z$ is the state to output map. For state $\mathbf{q} \in Q$, the output $\mathbf{z} \in Z$ at this state is:

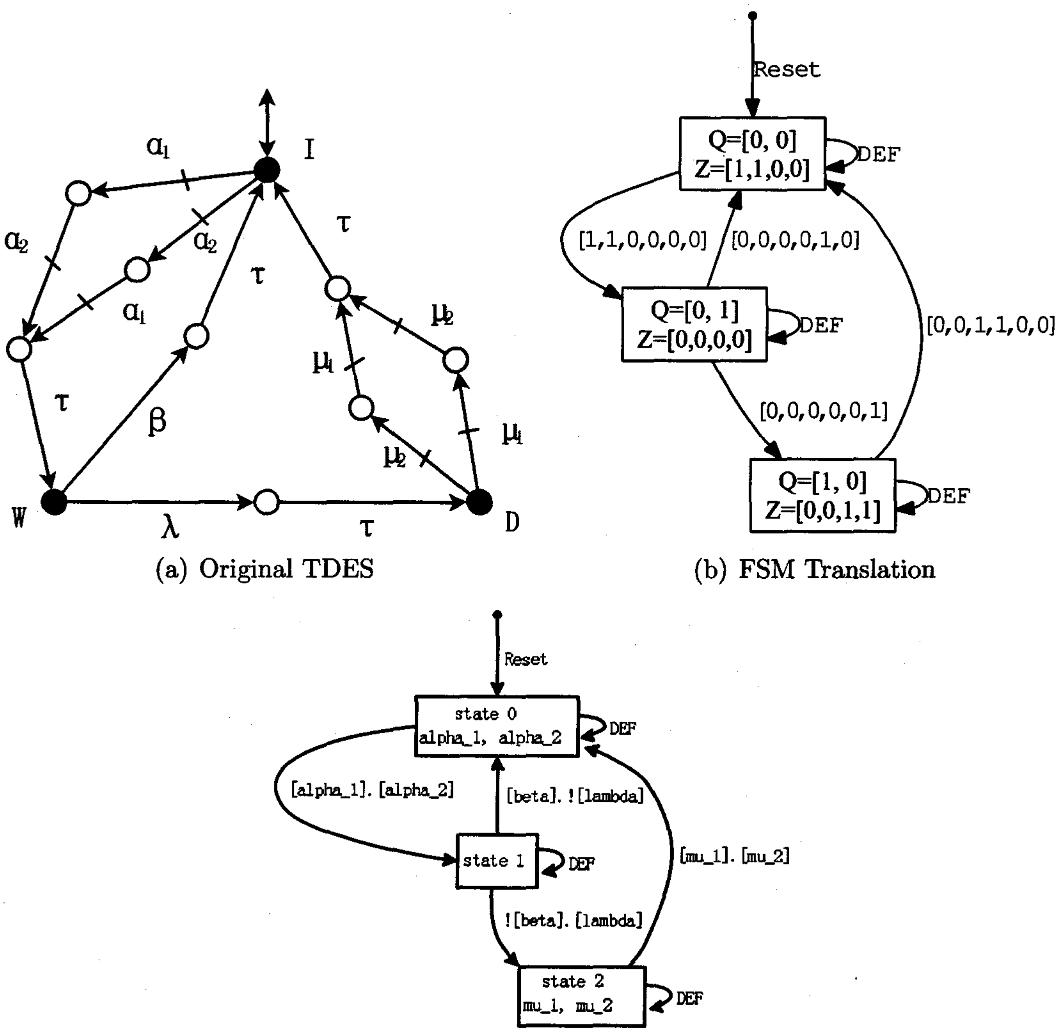
$$\mathbf{z} = \Phi(\mathbf{q})$$

A few comments are worthwhile here to clarify our notation. We will discuss the notation used for states, but the same applies for input and output variables. If we use $\mathbf{q}$ by itself (i.e. $\mathbf{q} \in Q$), then it represents a single instance of Q (i.e. some specific vector of zeros and ones with j elements). When we use $\mathbf{q}(k')$, then this is the k'-th element of the sequence $\{\mathbf{q}(k)\}$ where each element of the sequence is some member of Q. Obviously, we can construct many different possible $\{\mathbf{q}(k)\}$ sequences. If we wish to label different sequences, we will use different labels for $\mathbf{q}$, such as $\{\mathbf{q}(k)\}$ and $\{\mathbf{q}'(k)\}$.

With respect to our input, a specific sequence $\{\mathbf{i}(k)\}$ would represent a specific pattern of inputs we received for a specific run of the system. If we ran the system again, we could get a completely different sequence. From our definition of $\mathbf{C}$, we see that our state sequence is completely determined by q_{res}, Ω, and $\{\mathbf{i}(k)\}$. If we get a different input sequence, we could get a different state sequence, depending on how our next state function responds to the input values. As our output is a function of our current state, this means we could also get a different output sequence as well. In other words, input sequence $\{\mathbf{i}(k)\}$ might induce state and output sequences $\{\mathbf{q}(k)\}$ and $\{\mathbf{z}(k)\}$, while input sequence $\{\mathbf{i}'(k)\}$ might induce state and output sequences $\{\mathbf{q}'(k)\}$ and $\{\mathbf{z}'(k)\}$ which may or may not be the same as the other sequences of the same type..

Example 4.1. *Inspired by the DES shown in Figure 2.1, we take Figure 4.1 as an example to see how to apply our formal SD controller model.*

Figure 4.1(a) shows an example of a TDES and Figure 4.1(b) shows the Moore machine representing this TDES. Our ordering for the input variables is $I = [\alpha_1, \alpha_2, \mu_1, \mu_2, \beta_1, \lambda_1]$



(a) Original TDES

(b) FSM Translation

(c) Abbreviated FSM

Figure 4.1: FSM Translation Example

and for our outputs is $Z = [\alpha_1, \alpha_2, \mu_1, \mu_2]$. *We have also added a* ***DEF*** *or default transition to cover input combinations that we have not explicitly specified. The reason is that the transition function for a TDES is a partial function, but that of a FSM must be a complete function. The actual translation from the TDES in (a) to the controller in (b) will be presented after the translation method for centralized controllers is introduced in the next section.*

In (b), we showed the SD controller for our example in the format of the formal

SD controller model we just defined. Typically when we give a diagram of an FSM, we use the more compact and readable notation shown in Figure 4.1(c). Here we have given states meaningful names, and we only list at a state those prohibitable events whose outputs are true (1) at that state. Also, rather than listing input vectors on transitions, we use boolean equations that are true for the required input vector. We use '!' as NOT, '+' as OR, and '·' as AND[1]. We also only use in the equations those events that could occur at a given state, to simplify the equations.

4.2 Translation Method

To translate a supervisor to Moore FSM, we require that the supervisor be CS deterministic. CS deterministic is necessary because, for SD systems, we lose the ordering information for the events that occur during a given sampling period. Event sequences that have the same occurrence image must all go to the same next state in the state machine implementation or our controller will be nondeterministic. We can ensure this if we require the supervisors to be CS deterministic before being translated.

We also require that the supervisor be non-selfloop ALF. The reason is to make sure we have a manageable set of next state conditions. If we have activity loops that are not selfloops, then our supervisor does not have enough information for us to determine a reasonable set of concurrent strings to use to define our next state condition. We would thus potentially have a large choice of strings, most of which are not possible in the closed loop system. By requiring that the supervisor be non-selfloop ALF, we should have a reasonable set of possible concurrent strings at a given state. As we discussed earlier, technically the CS deterministic condition is strong enough, however, this condition is hard to evaluate if the system is not ALF or non-selfloop ALF. So, what we would do in practice is first check that our TDES is ALF or non-selfloop ALF, and if so, we will then check if it is CS deterministic.

We note that we require that a supervisor **S** be CS deterministic before we can translate it to a controller, but we do not need the supervisor be SD controllable for our plant **G** for the conversion process itself. We also note that if we are translating **S** to a controller, the fact that **S** is SD controllable for **G** is not sufficient to be able to

[1] In the following FSM graphs, this operator is represented by '.(period)' instead of '·' due to a technical difficulty.

do the conversion, as it implies that $\mathbf{S}||\mathbf{G}$ is CS deterministic if $\mathbf{S}||\mathbf{G}$ is minimal, not $\mathbf{S}$ itself. If $\mathbf{G}$ is not complete for $\mathbf{S}$, we may wish to instead convert $\mathbf{S}||\mathbf{G}$ instead of $\mathbf{S}$, but typically we prefer to construct modular controllers for the component supervisors that make up $\mathbf{S}$, as they usually are far more compact.

In the following sections, we introduce event mapping functions, and how to translate a CS deterministic TDES supervisor into a centralized controller. We then discuss the translation of modularized CS deterministic supervisors.

4.2.1 Event Mapping Functions

As we will often be discussing vectors of boolean values whose elements refer to specific events in Σ_{act}, we will need a way to map events to a vector's elements and vice versa. Let $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be the TDES plant to be controlled and let $\mathbf{S} = (X, \Sigma_\mathbf{S}, \xi, x_o, X_m)$ be an arbitrary CS deterministic TDES supervisor for $\mathbf{G}$. We define $\Sigma_{act} \subset \Sigma$ to be the set of all the activity events and $\Sigma_{hib} \subseteq \Sigma_{act}$ to be the set of all prohibitable events. We consider Σ, Σ_{act} and Σ_{hib} to be *global* event sets that can always be referred to in the following discussion.

We first define a bijective map between an activity event set and an index set we will use for labeling the events.

Definition 4.2.1. Let bijective map $\gamma_g : \Sigma_{act} \rightarrow \{0, .., |\Sigma_{act}| - 1\}$ be the *canonical event mapping function* such that

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{act})\sigma_1 = \sigma_2 \iff \gamma_g(\sigma_1) = \gamma_g(\sigma_2)$$

For the controller implementation $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ of $\mathbf{S}$, we include its event mapping information in our translation methods in the following sections, which are the two event mapping functions defined below. The reason we impose the ordering requirement is so that essentially the function γ_g will induce a single way to define the mapping functions.

Definition 4.2.2. The *input event mapping function* for $\mathbf{C}$ is defined to be a bijective map $\gamma : \Sigma_\mathbf{S} \cap \Sigma_{act} \rightarrow \{0, 1, .., v - 1\}$ where $v = |\Sigma_\mathbf{S} \cap \Sigma_{act}|$. It is defined such that

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

Definition 4.2.3. The *output event mapping function* for $\mathbf{C}$ is defined to be a bijective map $\eta : \Sigma_\mathbf{S} \cap \Sigma_{hib} \to \{0, 1, .., r-1\}$ where $r = |\Sigma_\mathbf{S} \cap \Sigma_{hib}|$. It is defined such that

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

Since γ_g is globally available, two input event mapping functions for different controllers will always have the same mapping pairs for the same event domain. In other words, because of the ordering requirement, there is only one way to define the input mapping. Similar logic applies to the output mapping for same event domain. An example is shown below.

Example 4.2. *For different controllers $\mathbf{C}_1$ and $\mathbf{C}_2$ whose supervisors $\mathbf{S}_1$ and $\mathbf{S}_2$ are defined over $\Sigma = \Sigma_{act} \cup \{\tau\} = \{\alpha, \beta, \lambda, \tau\}$. If $\gamma_g(\alpha) < \gamma_g(\beta) < \gamma_g(\lambda)$, then we always have the input event mapping function $\gamma_1 = \gamma_2 = \{(\alpha, 0), (\beta, 1), (\lambda, 2)\}$ for $\mathbf{C}_1$ and $\mathbf{C}_2$.*

Sometimes we want to find out which event an index in an input or output vector corresponds to. This can be easily done by applying the inverse event mapping function, since the event mapping functions we have defined are all bijective. i.e. to find the index of event α in the input event index used by the controller, use $\gamma^{-1}(\alpha)$.

For event $\sigma \in \Sigma_\mathbf{S} \cap \Sigma_{act}$, we can use the inverse event mapping functions to locate the element in a vector that corresponds to σ. For example, the corresponding element for σ in the input vector would be $i_{\gamma^{-1}(\sigma)}$. For convenience, we may write i_σ instead of $i_{\gamma^{-1}(\sigma)}$ and z_σ instead of $z_{\eta^{-1}(\sigma)}$.

4.2.2 Output Equivalence

If we have two or more controllers for system $\mathbf{G}$, we may wish to determine if they will produce equivalent output (i.e. enablement information) for the same input sequence. The problem is that each controller may care about a slightly different event set, thus we likely cannot use a single $\{\mathbf{i}(k)\}$ input sequence for them. As defined in our formal model, for n controllers $\mathbf{C}_1, \mathbf{C}_2, ..., \mathbf{C}_n$, each controller $\mathbf{C}_j$ for $1 \leq j \leq n$ has its own input vector $\mathbf{i}_j \in I_j$ and will generate its own output vector based on the input sequence $\{\mathbf{i}_j(k)\}$ it receives.

Before we check that their output sequences are equivalent, we need each input sequence $\{\mathbf{i}_j(k)\}$ to contain equivalent input information. However, their input vectors might be incompatible with each other, because their event mapping for the inputs can be different. Therefore, we will provide a single input vector $\mathbf{i}_g$ globally available to every controller, and let each controller extract its own input vector $\mathbf{i}_j$ from $\mathbf{i}_g$. Essentially, $\mathbf{i}_g$ represents the inputs the system sees, where each $\mathbf{i}_j$ represents the inputs that each controller sees (which may be a strict subset of the system inputs) and is formatted for the input index that controller is using.

Definition 4.2.4. Let $\Sigma_{act} \subset \Sigma$ be the set of global activity events, we require $\mathbf{i}_g = [i_{g,0}, i_{g,1}, .., i_{g,v_g-1}]$ to be defined over Σ_{act} where $v_g = |\Sigma_{act}|$. That is, for any event $\sigma \in \Sigma_{act}$, there is an element in $\mathbf{i}_g$ corresponds to σ and only σ. We call $\{\mathbf{i}_g(k)\}$ a *canonical input sequence* and $\mathbf{i}_g \in \{\mathbf{i}_g(k)\}$ a *canonical input vector*[2].

To extract input vector $\mathbf{i}_j = [i_{j,0}, i_{j,1}, .., i_{j,v_j-1}]$ from $\mathbf{i}_g$ for controller $\mathbf{C}_j$, for $0 \leq l < v_j$ we have $i_{j,l} = i_{g,l'}$ where $l' = \gamma_g((\gamma_j^{-1}(l)))$.

Definition 4.2.5. For $j = 1, 2$, let $\mathbf{C}_j = (I_j, Z_j, Q_j, \Omega_j, \Phi_j, \mathbf{q}_{res,j})$ be a controller. We say $\mathbf{C}_1$ and $\mathbf{C}_2$ are *output equivalent* if for any canonical input sequence $\{\mathbf{i}_g(k)\}$ and induced output $\mathbf{z}_j(k') = [z_{j,1}(k'), z_{j,2}(k'), .., z_{j,r_j}(k')] \in Z_j$ at time $k' = \{0, 1, 2, \ldots\}$, the follow conditions are satisfied.

1. $r_1 = r_2$
2. $(\forall 0 \leq i < r_1)\eta_1^{-1}(i) = \eta_2^{-1}(i)$
3. $(\forall k' \in \{0, 1, ..\})\mathbf{z}_1(k') = \mathbf{z}_2(k')$

In the above definition, by Point 1, 2 we are essentially requiring the outputs of the two controllers be of the same size, and represent the same events in the same order. We could have been more general and only required that they represent the same events but in possibly different order, but this does not gain much and complicates our notation. In Point 3, we are requiring that one controller enables a prohibitable event if and only if the other does, for any value of k'. In other words, they agree at the reset state, and will continue to agree in the future.

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

A common situation is that controllers $\mathbf{C}_1$ and $\mathbf{C}_2$ have been defined relative to a CS deterministic supervisor $\mathbf{S} = (X, \Sigma_{\mathbf{S}}, \xi, x_o, X_m)$, and we are only interested that they generate the same output with respect to input sequences that represent valid input strings to the supervisor (i.e. $s \in L(\mathbf{S}) \cap L_{samp}$). We first provide a definition for valid input sequences relative to TDES $\mathbf{S}$, and then a form of output equivalence definition for these sequences.

Definition 4.2.6. For system event set Σ, with canonical event mapping function γ_g, activity event set Σ_{act}, and CS deterministic TDES supervisor $\mathbf{S} = (X, \Sigma_{\mathbf{S}}, \xi, x_o, X_m)$, we say a canonical input sequence $\{i_g(k)\}$ is *input valid* for $\mathbf{S}$, if
$(\forall k \in \{1, 2, ...\})(\exists s_1, s_2, \ldots, s_k \in L_{conc})$
$[s_1 s_2 .. s_k \in L(\mathbf{S})] \wedge [(\forall n \in \{1, 2, ..., k\})(\forall \sigma \in \Sigma_{act})\, i_{g,\gamma_g(\sigma)}(n) = 1 \text{ iff } \sigma \in \mathrm{Occu}(s_n)]$

Essentially in the above definition, we are requiring the sequence $\{i_g(k)\}$ to correspond to a sequence of concurrent strings that supervisor $\mathbf{S}$ will accept. We are specifically excluding input sequences that our supervisor says will never occur. As we will see in the next section, when we translate a CS deterministic supervisor into a controller we will define next state information in an arbitrary manner for invalid input sequences. We will thus not be interested in whether two controllers generate the same output sequences for invalid input sequences.

We now provide a new output equivalence definition that is only concerned about input sequences that are valid for our supervisor.

Definition 4.2.7. For system event set Σ, with canonical event mapping function γ_g, activity event set Σ_{act}, and CS deterministic TDES supervisor $\mathbf{S} = (X, \Sigma_{\mathbf{S}}, \xi, x_o, X_m)$, let $\mathbf{C}_j = (I_j, Z_j, Q_j, \Omega_j, \Phi_j, \mathbf{q}_{res,j})$, $j = 1, 2$, be a controller. We say $\mathbf{C}_1$ and $\mathbf{C}_2$ are *output equivalent with respect to* $\mathbf{S}$ if for any canonical input sequence $\{\mathbf{i}_g(k)\}$ that is input valid for $\mathbf{S}$, and induced output $\mathbf{z}_j(k') = [z_{j,1}(k'), z_{j,2}(k'), .., z_{j,r_j}(k')] \in Z_j$ at time $k' = \{0, 1, 2, \ldots\}$, the follow conditions are satisfied.

1. $r_1 = r_2$
2. $(\forall 0 \leq i < r_1)\eta_1^{-1}(i) = \eta_2^{-1}(i)$
3. $(\forall k' \in \{0, 1, ..\})\mathbf{z}_1(k') = \mathbf{z}_2(k')$

4.2.3 Centralized Controller

We will now discuss how to translate a TDES supervisor into a centralized controller.

Let TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be CS deterministic and non-selfloop ALF. To translate $\mathbf{S}$ into a controller $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$, we need to introduce a few definitions.

We start by defining how many state variables are needed for Q. Let $X_{samp} \subseteq X$ be the set of sampling states for $\mathbf{S}$. To map each sampling state to a state in the controller, we define the state size of Q, l, to satisfy $2^{l-1} < |X_{samp}| \leq 2^l$. There are l state variables in vector $\mathbf{q} \in Q$. A state in $\mathbf{S}$ which is not found in X_{samp}, does not correspond to any state variable assignment in Q.

We now define a function to map the sampling states of our TDES supervisor, onto states of the controller.

Definition 4.2.8. Let $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor. Let $\Lambda : X_{samp} \to Q$ be an arbitrary injective map where $X_{samp} \subseteq X$. We say Λ is a *state mapping function* for controller $\mathbf{C}$ if, for all $x \in X_{samp}$, $\Lambda(x)$ returns a vector of state variables $\mathbf{q} = [q_0, q_1, .., q_{l-1}]$ such that,

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

Recall that the initial state is also a sampling state, and it is mapped to be $\Lambda(x_o) = \mathbf{q}_{res} = \mathbf{q}(0)$.

We now define a function that will map subsets of Σ_{act} to a particular assignment of the variables for I (called a *valuation* of I) that will represent the events present in the subset, according to the mapping defined by γ, the controller's input event mapping function. This will be useful for taking the occurrence image of a concurrent string and identifying the corresponding valuation that represents that subset in I.

Definition 4.2.9. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the corresponding controller for CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$. The size of each input vector $\mathbf{i} \in I$ is defined to be $v = |\Sigma_{act}|$.

Let γ be the input event mapping function for controller $\mathbf{C}$. Then we have a bijective map

$$\Gamma_I : Pwr(\Sigma_{act}) \to I$$

defined as follows. For arbitrary $\Sigma_I \subseteq \Sigma_{act}$, we have $\Gamma_I(\Sigma_I) = [i_0, i_1, .., i_{v-1}]$ such that for $j = 0, 1, .., v-1$,

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

We call Γ_I the *input set mapping function* for controller **C**.

The motivation for the above mapping is that at each sampling state, it will be observed which activity events have occurred, and which have not. Since the order of event occurrences is not stored, activity events are observed as if they are concurrent. Thus the occurrence of each event can be represented as a binary value in the corresponding position of the input vector **i**.

We now define a function that will map subsets of Σ_{hib} to a particular assignment of the variables for Z that will represent the events present in the subset, according to the mapping defined by η, the controller's output event mapping function. This will be useful for taking the set of prohibitable events eligible at a sampling state of the supervisor, and identifying the corresponding valuation that represents that subset in Z.

Definition 4.2.10. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the corresponding controller for CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$. The size of each vector in $\mathbf{z} \in Z$ is defined to be $r = |\Sigma_{hib}|$. Let η be the output event mapping function for controller **C**. Then we have a bijective map

$$\Gamma_Z : \mathrm{Pwr}(\Sigma_{hib}) \to Z$$

defined as follows. For arbitrary $\Sigma_Z \subseteq \Sigma_{hib}$, we have $\Gamma_Z(\Sigma_Z) = [z_0, z_1, .., z_{r-1}]$ such that for $j = 0, 1, .., r-1$,

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

We call Γ_Z the *output set mapping function* for controller **C**.

We now discuss how to define the next state function Ω for our controller, using our CS deterministic supervisor as our starting point. Note that the Δ function was defined in Section 3.1.

Definition 4.2.11. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the corresponding controller for CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$. Let $X_{samp} \subseteq X$. For state $\mathbf{q} \in Q$ and arbitrary input $\mathbf{i} \in I$, the *next state function* Ω is defined to be

$$\Omega(\mathbf{q}, \mathbf{i}) = \Lambda(\Delta(x, \Gamma_I^{-1}(\mathbf{i}))) \quad \text{if } (\exists x \in X_{samp})\mathbf{q} = \Lambda(x) \;\&\; \Delta(x, \Gamma_I^{-1}(\mathbf{i}))!$$

All remaining values of Ω are assigned arbitrarily.

Essentially, we define Ω in terms of ξ, the next state function of TDES $\mathbf{S}$. For the given state $\mathbf{q}$ of our controller and input $\mathbf{i}$ which are some valuations of sets Q and I, we define the next state of the controller to match that of the supervisor. We define $\Omega(\mathbf{q}, \mathbf{i})$ arbitrarily unless our state $\mathbf{q}$ corresponds to a sampled state x in $\mathbf{S}$, there exists a concurrent string s whose occurrence image matches the set of activity events represented by $\mathbf{i}$, and $\xi(x, s)!$ in our supervisor. In that case, our new state is $\mathbf{q}' = \Lambda(\xi(x, s))$ as per the definition of Δ. If there does not exist such an x and s, that means $\mathbf{q}$ and $\mathbf{i}$ do not correspond to possible behavior of our system, so we can define the next state as we like (note ξ is a partial function, but Ω must be a total function).

In practice, we would not assign the next state randomly. Most likely, we would choose $\mathbf{q}'$ to either make our controller simpler, or we would choose $\mathbf{q}'$ in a failsafe manner. By failsafe, we mean that we do not believe the combination $\mathbf{q}$ and $\mathbf{i}$ should ever be seen in the physical system, but we will choose our next state in a way to maximize safety should it actually ever occur.

We now discuss how to define the output map Φ for our controller, using our CS deterministic supervisor as our starting point. Note that the ζ function was defined in Section 3.2.

Definition 4.2.12. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the corresponding controller for CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$. Let $\zeta(x)$ be the control action for any sampling state $x \in X_{samp} \subseteq X$ as defined in Definition 3.2.3. For any $\mathbf{q} \in Q$, the *output map* Φ is defined to be

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

The definition states that if state $\mathbf{q}$ in controller $\mathbf{C}$ has a corresponding state $x \in X_{samp}$ in $\mathbf{S}$, then $\Phi(\mathbf{q})$ specifies an output vector based on the control action $\zeta(x)$. $\zeta(x)$ is equal to the set of prohibitable events enabled at state x in $\mathbf{S}$. Otherwise, $\Phi(\mathbf{q})$ leaves all prohibitable events disabled at state $\mathbf{q}$.

Let TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor. Then Figure 4.2 shows the control equivalence diagram for $\mathbf{S}$ and its controller $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$, as defined in this section. If, for arbitrary $\Sigma' \subseteq \Sigma_{act}$ and state $x \in X_{samp}$ of $\mathbf{S}$, $\Delta(x, \Sigma')$ is defined, it is easy to see that this diagram commutes.

$$\begin{tikzcd}
X_{samp} \arrow[d, "\Lambda"'] & \times & Pwr(\Sigma_{act}) \arrow[r, "\Delta"] \arrow[d, "\Gamma_I"'] & X_{samp} \arrow[r, "\zeta"] \arrow[d, "\Lambda"'] & Pwr(\Sigma_{hib}) \arrow[d, "\Gamma_Z"'] \\
Q & \times & I \arrow[r, "\Omega"] & Q \arrow[r, "\Phi"] & Z
\end{tikzcd}$$

Figure 4.2: Centralized Control Equivalence Diagram

Essentially, the diagram says that as long as $\Delta(x, \Sigma')!$, then $\Gamma_Z(\zeta(\Delta(x, \Sigma'))) = \Phi(\Omega(\Lambda(x), \Gamma_I(\Sigma'))$ meaning that we can just use the next state function and output map of the controller, and we will produce the correct enablement. Note that the Σ' represent the occurrence image (minus tick) of the concurrent strings defined at the given sampled state. The figure also says that if $\Delta(x, \Sigma')!$, then $\Lambda((\Delta(x, \Sigma'))) = \Omega(\Lambda(x), \Gamma_I(\Sigma')$, meaning that we can simply use the controller's next state function to determine the correct next state.

Example 4.3. *Let* $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ *be represented by the Moore machine shown in Figure 4.1(b). We see from Figure 4.1(a), that our set of activity events is* $\{\alpha_1, \alpha_2, \beta, \mu_1, \mu_2, \lambda\}$*, and our set of prohibitable events are* $\{\alpha_1, \alpha_2, \mu_1, \mu_2\}$*. We can also see that the TDES is ALF, and CS deterministic.*

We have each $\mathbf{i} \in I$ *in the form of*

$$\mathbf{i} = [i_0, i_1, i_2, i_3, i_4, i_5]$$

For $j = 0, 1, .., 5$*,* i_j *corresponds to the occurrence of events* $[\alpha_1, \alpha_2, \mu_1, \mu_2, \beta, \lambda]$ *respectively, when* $i_j = 1$*.*

We have each $\mathbf{z} \in Z$ *in the form of*

$$\mathbf{z} = [z_0, z_1, z_2, z_3]$$

For $j = 0, 1, .., 3$*,* z_j *corresponds to the enablement of prohibitable events* $[\alpha_1, \alpha_2, \mu_1, \mu_2]$*, when* $z_j = 1$*.*

We see from Figure 4.1(a) that our TDES has three sampled states. Our state size, l*, must thus satisfy* $2^{l-1} < 3 \leq 2^l$*. As only* $l = 2$ *satisfies this equation, our state set must have two binary elements. We thus have each* $\mathbf{q} \in Q$ *in form of*

$$\mathbf{q} = [q_0, q_1]$$

We will let state $(q_0, q_1) \in \{(0,0), (0,1), (1,0)\}$ *represent states* $\{I, W, D\}$ *respectively. The fourth state* $(1,1)$ *is unused and will be unreachable, so we can define transition leaving this state arbitrarily.*

Examining Figure 4.1(a), we can determine which concurrent strings are defined at each sampled state. For instance, at state I *we could only get strings* $\alpha_1\alpha_2\tau$ *or* $\alpha_2\alpha_1\tau$*. Both have occurrence image* $\{\alpha_1, \alpha_2, \tau\}$ *and take us to sampled state* W*. As this subset corresponds to* $\mathbf{i} = [1, 1, 0, 0, 0, 0]$*, we can see where the transition at state (0,0) in Figure 4.1(b) comes from. Continuing this logic, we can derive the remaining transitions for the SD controller shown in Figure 4.1(b). Note, that we have added the **DEF** default transitions as we discussed in Section 4.1.*

Next,

$$\mathbf{q}_{res} = \mathbf{q}(0) = [0, 0]$$

Using the information we have derived for Figure 4.1(b), we can define the next state function, Ω, as below:

$$\mathbf{q}(k+1) = \Omega(\mathbf{q}(k), \mathbf{i}(k+1)) = \Omega([q_0(k), q_1(k)], [i_0(k+1), i_1(k+1), .., i_5(k+1)])$$

such that

$$\begin{aligned}
\Omega([0,0],[1,1,0,0,0,0]) &= [0,1] \\
\Omega([0,0],\mathbf{i}) &= [0,0] \textit{ for all other } \mathbf{i} \in I \\
\Omega([0,1],[0,0,0,0,1,0]) &= [0,0] \\
\Omega([0,1],[0,0,0,0,0,1]) &= [1,0] \\
\Omega([0,1],\mathbf{i}) &= [0,1] \textit{ for all other } \mathbf{i} \in I \\
\Omega([1,0],[0,0,1,1,0,0]) &= [0,0] \\
\Omega([1,0],\mathbf{i}) &= [1,0] \textit{ for all other } \mathbf{i} \in I
\end{aligned}$$

and $\Omega([1,1], \mathbf{i})$, for any $\mathbf{i} \in I$, can be defined arbitrarily as state $[1,1]$ is unreachable.

Lastly, we define the output function to be

$$\mathbf{z} = \Phi(\mathbf{q})$$

such that

$$\begin{aligned}
\Phi([0,0]) &= [1,1,0,0] \\
\Phi([0,1]) &= [0,0,0,0] \\
\Phi([1,0]) &= [0,0,1,1]
\end{aligned}$$

We can define $\Phi([1,1])$ arbitrarily, say $\Phi([1,1]) = [0,0,0,0]$.

The execution of a centralized controller $\mathbf{C}$ is as follows.

1. Initialize the controller by setting $\mathbf{q}(0) = \mathbf{q}_{res}$, $\mathbf{z} = \Phi(\mathbf{q}_{res})$. We have $k = 0$.

2. At the next clock pulse

   i) sample inputs and set $\mathbf{i}(k+1)$ equal to these values.

   ii) calculate our new state and output as follows:

   i.e. $\mathbf{q}(k+1) = \Omega(\mathbf{q}(k), \mathbf{i}(k+1))$ and $\mathbf{z}(k+1) = \Phi(\mathbf{q}(k+1))$

3. Set $k = k + 1$. Go to step 2.

We say **C** *acts on* **G** when controller **C** enables or disables events from plant **G**. Also, since an SD controller forces a prohibitable event as soon as its enabled, the controller is also forcing these events to occur in that clock period. To be consistent, if any controller **C** is discussed from now on, we will assume that it has been converted from some CS deterministic supervisor **S**, using the translation method defined in this section.

Before we close this section, we would like to briefly discuss the case that our TDES supervisor **S** is defined over a subset Σ_S of the system event set, Σ. This would mean that some activity events would not affect the next state of the controller and could be ignored, thus simplifying the next state logic of the controller. The output for the controller would still cover all events in Σ_{hib}. The difference would be that for all $\sigma \in \Sigma_{hib} - \Sigma_{\mathbf{S}}$, their corresponding output would always be set to 1.

4.2.4 Modular Controllers

For large systems, the centralized supervisor for the system is quite likely large and complex. This would mean that its corresponding controller would also be large and complex, making implementing it directly undesirable. Just as we design modular TDES supervisors for systems to make the design more manageable, we can also implement our controllers by directly translating these modular supervisors into their own controllers. We can then combine the outputs of these controllers together, to create the overall output that would be equivalent to the output provided by a centralized controller.

To implement the composition of modular controllers, we need the following two operations on vectors.

Definition 4.2.13. Let V be the set of Boolean vectors with each vector of size n. For $\mathbf{u} = [u_1, u_2, .., u_n], \mathbf{v} = [v_1, v_2, .., v_n] \in V$, the logical AND operator $\wedge : V \times V \to V$ is

$$\mathbf{u} \wedge \mathbf{v} = [u_1 \wedge v_1, u_2 \wedge v_2, .., u_n \wedge v_n].$$

Definition 4.2.14. Let $\mathbf{u}$ be a Boolean vector of i variables, and $\mathbf{v}$ be another Boolean vector of j variables. The concatenation operator $. : V \times V \to V$ is defined as follows.

$$\mathbf{u}.\mathbf{v} = [u_1, u_2, .., u_i, v_1, v_2, .., v_j]$$

For convenience, we will often just write $\mathbf{uv}$ instead.

Let the TDES $\mathbf{S} = \mathbf{S}_1||\mathbf{S}_2||..||\mathbf{S}_n$ be a supervisor where each modular supervisor $\mathbf{S}_i$, for $1 \leq i \leq n$, is CS deterministic.

To avoid implementing the likely large $\mathbf{S}$ directly, we wish to implement each supervisor $\mathbf{S}_i$ as controller $\mathbf{C}_i$, then combine the controllers $\mathbf{C}_1, \mathbf{C}_2, .., \mathbf{C}_n$ (referred to as the composite controller) to generate the actual final output. We call each $\mathbf{C}_i$ the *modular controller* for supervisor $\mathbf{S}_i$. To be able to reuse the implementation technique discussed in the previous section, we assume each supervisor $\mathbf{S}_i$ is CS deterministic.

When comparing a centralized controller implementation to a modular controller implementation, all we care about is the output equivalence of the centralized controller and the composite controller created from $\mathbf{C}_1, \mathbf{C}_2, ..., \mathbf{C}_n$. If we take $\mathbf{S}$ and implement it directly as a controller $\mathbf{C}$, we want the composition of the outputs from $\mathbf{C}_1, \mathbf{C}_2, .., \mathbf{C}_n$ to be equivalent to the output from $\mathbf{C}$.

We will now discuss how to implement the modular supervisors as individual controllers, and then combine them into a composite controller to handle the system. It is key to note that the modular supervisors may be defined over strict subsets of the system event set, Σ. Essentially, supervisor $\mathbf{S}_j$ will have activity event set $\Sigma_{act,j} \subseteq \Sigma_{act}$ and prohibitable event set $\Sigma_{hib,j} \subseteq \Sigma_{hib}$. To translate the CS deterministic supervisor $\mathbf{S}_j$ to a controller, we will use the method defined in Section 4.2.3, but the key difference is that we replace every Σ_{act} in the definitions with $\Sigma_{act,j}$, and each Σ_{hib} with $\Sigma_{hib,j}$. This means that the input and the output sets for the controller may only represent a subset of Σ_{act} and Σ_{hib}, respectively.

Definition 4.2.15. Let $\mathbf{G}$ be the plant to be controlled, γ_g be the canonical event mapping function, Σ be the system event set, Σ_{act} the system activity event set, and Σ_{hib} the prohibitable event set. For $j = 1, 2, .., n$, let $\mathbf{S}_j = (X_j, \Sigma_j, \xi_j, x_{o,j}, X_{m,j})$ be the j-th CS deterministic supervisor, where $\Sigma_j = \Sigma_{act,j} \dot{\cup} \{\tau\} \subseteq \Sigma$. Here we have $\Sigma_{act,j} \subseteq$

Σ_{act} the activity event set for supervisor $\mathbf{S}_j$, $\Sigma_{hib,j} \subseteq \Sigma_{hib}$ the prohibitable event set for $\mathbf{S}_j$. We also require that $\Sigma = \bigcup_{j\in\{1,2,..,n\}} \Sigma_j$. Then we define the composition of modular controllers as follows.

Let $\mathbf{C}_j = (I_j, Z_j, Q_j, \Omega_j, \Phi_j, q_{res,j})$ be the controller for $\mathbf{S}_j$ with the following configuration:

- l_j is the number of state variables for each $\mathbf{q}_j = [q_{j,0}, q_{j,1}, .., q_{j,l_j-1}] \in Q_j$
- $v_j = |\Sigma_{act,j}|$ is number of input variables for each $\mathbf{i}_j = [i_{j,0}, i_{j,1}, .., i_{j,v_j-1}] \in I_j$
- $r_j = |\Sigma_{hib,j}|$ is number of output variables for each $\mathbf{z}_j = [z_{j,0}, z_{j,1}, .., z_{j,r_j-1}] \in Z_j$

The *composition of* $\mathbf{C}_1, \mathbf{C}_2, .., \mathbf{C}_n$,

$$\mathbf{C} = (I, Z, Q, \Omega, \Phi, q_{res}) = \mathbf{comp}(\mathbf{C}_1, \mathbf{C}_2, .., \mathbf{C}_n)$$

is defined as follows.

1. $\Sigma_{act} = \bigcup_{j=1,2,..,n} \Sigma_{act,j}$ and $\Sigma_{hib} = \bigcup_{j=1,2,..,n} \Sigma_{hib,j}$, thus $\Sigma_{hib} \subseteq \Sigma_{act} \subset \Sigma$ is guaranteed.

2. The number of state variables for vectors $\mathbf{q} \in Q$ is defined to be $l = \sum_{j=1}^{n} l_j$. The state vector $\mathbf{q}$ is defined to be

$$\begin{aligned}\mathbf{q} &= \mathbf{q}_1\mathbf{q}_2..\mathbf{q}_n \\ &= [q_{1,0}, q_{1,1}, .., q_{1,l_1-1}][q_{2,0}, q_{2,1}, .., q_{2,l_2-1}]..[q_{n,0}, q_{n,1}, .., q_{n,l_n-1}] \\ &= [q_{1,0}, q_{1,1}, .., q_{1,l_1-1},\ q_{2,0}, q_{2,1}, .., q_{2,l_2-1},\ ..,\ q_{n,0}, q_{n,1}, .., q_{n,l_n-1}]\end{aligned}$$

3. The size of each input vector $\mathbf{i} \in I$ is defined to be $v = |\Sigma_{act}|$.

   Then we define $\gamma : \Sigma_{act} \to \{0, 1, .., v-1\}$ to be the input event mapping function for $\mathbf{C}$ such that

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{act})\gamma_g(\sigma_1) < \gamma_g(\sigma_2) \implies \gamma(\sigma_1) < \gamma(\sigma_2)$$

4. The size of each output vector $\mathbf{z} \in Z$ is defined to be $r = |\Sigma_{hib}|$.

   Then we define $\eta : \Sigma_{hib} \to \{0, 1, .., r-1\}$ to be the output event mapping function for $\mathbf{C}$ such that

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{hib})\gamma_g(\sigma_1) < \gamma_g(\sigma_2) \implies \eta(\sigma_1) < \eta(\sigma_2)$$

5. The next state function $\Omega : Q \times I \rightarrow Q$ is defined such that, for $\mathbf{q}(k) = \mathbf{q}_1(k)\mathbf{q}_2(k)..\mathbf{q}_n(k) \in Q$ and $\mathbf{i}(k+1) \in I$,

$$\begin{aligned}\mathbf{q}(k+1) &= \Omega(\mathbf{q}(k), \mathbf{i}(k+1)) \\ &= \Omega_1(\mathbf{q}_1(k), \mathbf{i}_1(k+1))\ \Omega_2(\mathbf{q}_2(k), \mathbf{i}_2(k+1))\ ..\ \Omega_n(\mathbf{q}_n(k), \mathbf{i}_n(k+1))\end{aligned}$$

For above, the input vector $\mathbf{i}(k+1)$ is in canonical form with respect to γ_g. To use it as an input to each controller $\mathbf{C}_j$, we need to map it to input vector $\mathbf{i}_j(k+1)$ using γ_j, the input event mapping function for Controller $\mathbf{C}_j$. To do this, we need to map input vector

$$\mathbf{i}(k+1) = [i_0(k+1), i_1(k+1), .., i_{v-1}(k+1)]$$

onto input vectors $\mathbf{i}_j(k+1) = [i_{j,0}(k+1), i_{j,1}(k+1), .., i_{j,v_j-1}(k+1)]$ for modular controller $\mathbf{C}_j$, as follows

$$(\forall \sigma \in \Sigma_{act,j}) i_{j,\gamma_j(\sigma)}(k+1) = i_{\gamma_g(\sigma)}(k+1)$$

6. The output map $\Phi : Q \rightarrow Z$ is defined as follows.

Given $\mathbf{q} = \mathbf{q}_1\mathbf{q}_2..\mathbf{q}_n \in Q$, let

$$\mathbf{z}_j = \Phi_j(\mathbf{q}_j) = [z_{j,0}, z_{j,1}, .., z_{j,r_j-1}] \in Z_j$$

For each $\mathbf{z}_j$ we expand it to

$$\mathbf{z}'_j = [z'_{j,0}, z'_{j,1}, .., z'_{j,r-1}] \in Z$$

such that,

$$(\forall \sigma \in \Sigma_{hib}) z'_{j,\eta(\sigma)} = \begin{cases} z_{j,\eta_j(\sigma)} & \text{if } \sigma \in \Sigma_{hib,j} \\ 1 & \text{otherwise} \end{cases}$$

In above, η_j is the output event mapping for controller $\mathbf{C}_j$. Essentially, what we are doing is mapping the output value for $\mathbf{C}_j$ to the corresponding position in $\mathbf{z}'_j$ if $\sigma \in \Sigma_{hib,j}$, else we always set the value equal to 1.

With expanded output vectors $\mathbf{z}'_1, \mathbf{z}'_2, .., \mathbf{z}'_n \in Z$ defined, the next state function is then defined to be

$$\Phi(\mathbf{q}) = \bigwedge_{j \in \{1,2,..,n\}} \mathbf{z}'_j$$

We simply logically AND each $\mathbf{z}'_j$ together to obtain the output vector.

In Definition 4.2.15, we assumed that when the supervisors are combined together, they are defined over Σ, the systems event set (i.e. $\bigcup_{j=1,2,..,n} \Sigma_j$). As for the centralized supervisor, it may be the case that the supervisors only care about a subset of Σ. This would mean that some activity events would not affect the next state of the controller and could be ignored, thus simplifying the next state logic of the controller. The output for the composite controller would still cover all events in Σ_{hib}. The difference would be that for all $\sigma \in \Sigma_{hib}$ but not covered by any modular supervisor, their corresponding output would always be set to 1.

We now present a theorem that shows that we can either implement our supervisor centrally or modularly, and we will get the same enablement information for valid input sequences.

Theorem 4.1. Let $\mathbf{G}$ be the plant to be controlled, γ_g be the canonical event mapping function, Σ be the system event set, Σ_{act} the system activity event set, and Σ_{hib} the prohibitable event set. Also, let CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be composed of n component CS deterministic supervisor $\mathbf{S}_j = (X_j, \Sigma_j, \xi_j, x_{o,j}, X_{m,j})$ for $j = 1, 2, .., n$, such that $\mathbf{S} = \mathbf{S}_1 || \mathbf{S}_2 || .. || \mathbf{S}_n$. Let $\Sigma_{act,j} \subseteq \Sigma_{act}$ and $\Sigma_{hib,j} \subseteq \Sigma_{hib}$ be the activity event set and prohibitable event set for $\mathbf{S}_j$.

For $j = 1, 2, .., n$, let $\mathbf{C}_j = (I_j, Z_j, Q_j, \Omega_j, \Phi_j, \mathbf{q}_{res,j})$ be the controller translated from $\mathbf{S}_j$ using translation method defined in Section 4.2.3 but replacing every Σ_{act} in the definitions with $\Sigma_{act,j}$, and each Σ_{hib} with $\Sigma_{hib,j}$. Let $\mathbf{C}' = \mathbf{comp}(\mathbf{C}_1, \mathbf{C}_2, .., \mathbf{C}_n)$ be the composed controller of $\mathbf{C}_1, \mathbf{C}_2, .., \mathbf{C}_n$. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the controller translated from $\mathbf{S}$ using the translation method defined in Section 4.2.3.

Then $\mathbf{C}$ and $\mathbf{C}'$ are output equivalent with respect to $\mathbf{S}$.

Proof. Assume the required initial conditions for the proof.

Next, we need to define the following items for our proof, to ensure clarity.

Let $X_{samp} \subseteq X$ and $X_{samp,j} \subseteq X_j$ be the sets of sampling states for $\mathbf{S}$ and $\mathbf{S}_j$, respectively.

Let $\Delta : X_{samp} \times \mathrm{Pwr}(\Sigma_{act}) \rightarrow X_{samp}$ and $\Delta_j : X_{samp,j} \times \mathrm{Pwr}(\Sigma_{act,j}) \rightarrow X_{samp,j}$ be the next sampling state functions for $\mathbf{S}$ and $\mathbf{S}_j$, respectively.

Let $\Lambda : X_{samp} \rightarrow Q$ and $\Lambda_j : X_{samp,j} \rightarrow Q_j$ be the state mapping functions for $\mathbf{C}$ and $\mathbf{C}_j$, respectively.

Let $\Gamma_I : \mathrm{Pwr}(\Sigma_{act}) \rightarrow I$ and $\Gamma_{I,j} : \mathrm{Pwr}(\Sigma_{act,j}) \rightarrow I_j$ be the input set mapping functions for $\mathbf{C}$ and $\mathbf{C}_j$, respectively.

Let $\Gamma_Z : \mathrm{Pwr}(\Sigma_{hib}) \rightarrow Z$ and $\Gamma_{Z,j} : \mathrm{Pwr}(\Sigma_{hib,j}) \rightarrow Z_j$ be the output set mapping functions for $\mathbf{C}$ and $\mathbf{C}_j$, respectively.

Let $\gamma : \Sigma_{act} \rightarrow \{0, 1, .., v-1\}$, $\gamma' : \Sigma_{act} \rightarrow \{0, 1, .., v'-1\}$, $\gamma_j : \Sigma_{act,j} \rightarrow \{0, 1, ..v_j-1\}$ be the input event mapping functions for $\mathbf{C}$, $\mathbf{C}'$, and $\mathbf{C}_j$, respectively. We note that since γ and γ' have domain Σ_{act}, they must both equal γ_g due to how they are defined i.e. given a specific γ_g, there is only one way to define the other two functions and it must be the same as γ_g, if Definition 4.2.2 is to be satisfied. (1)

Let $\eta : \Sigma_{hib} \rightarrow \{0, 1, .., r-1\}$, $\eta' : \Sigma_{hib} \rightarrow \{0, 1, ..r'-1\}$, $\eta_j : \Sigma_{hib,j} \rightarrow \{0, 1, ..r_j-1\}$ be the output event mapping functions for C, C', and C_j, respectively. We note that since η and η' have domain Σ_{hib}, they must be equal due to how they are defined (see Definition 4.2.3). This means that Z and Z' represent the same prohibitable events, in the same order, and can be directly compared. (2)

Given the above setting, we will now show that $\mathbf{C}$ and $\mathbf{C}'$ are output equivalent with respect to $\mathbf{S}$.

Let $\{\mathbf{i}(k'')\}$ be a canonical input sequence with respect to γ_g, the canonical event mapping function for the system, and let the sequence be input valid with respect to $\mathbf{S}$. From (1), we have $\gamma = \gamma' = \gamma_g$. This means that $\{\mathbf{i}(k'')\}$ can be used as an input for both $\mathbf{C}$ and $\mathbf{C}'$ directly, without any mapping required.

Let $\mathbf{z}'(k) = [z'_1(k), z'_2(k), .., z'_{r'}(k)] \in Z'$ be the induced output vector in $\mathbf{C}'$ at time k, from input sequence $\{\mathbf{i}(k'')\}$.

Let $\mathbf{z}(k) = [z_1(k), z_2(k), .., z_r(k)] \in Z$ be the induced output vector in $\mathbf{C}$ at time k, from input sequence $\{\mathbf{i}(k'')\}$.

We now need to show the following three points from Definition 4.2.7.

1. Show $r' = r$

   As both $\mathbf{C}$ and $\mathbf{C}'$ are defined relative to Σ, their outputs are both defined relative to Σ_{hib}. It follows immediately from the definition of r' and r in Definition 4.1.1 that $r' = r$.

2. Show $(\forall 0 \leq i < r)\, \eta(i) = \eta'(i)$

   Let $i \in \{0, 1, .., r-1\}$, show $\eta(i) = \eta'(i)$.

   This follows immediately from (2).

3. Show $(\forall k \in \{0, 1, ..\})\, \mathbf{z}(k) = \mathbf{z}'(k)$

   (A) First we will show that if $\mathbf{C}$ is in state $\mathbf{q} = \Lambda(x)$ for some $x \in X_{samp}$, and each $\mathbf{C}_j$ is in state $\mathbf{q}_j = \Lambda_j(x_j)$ for some $x_j \in X_{samp}$ such that $x = (x_1, x_2, .., x_n)$, then $\mathbf{C}$ at state $\mathbf{q}$ and $\mathbf{C}'$ at state $\mathbf{q}_1\mathbf{q}_2..\mathbf{q}_n$ will have the same output.

   (B) Then we will show for all $k \in \{0, 1, ..\}$ that after inputs $\mathbf{i}(1), \mathbf{i}(2), .., \mathbf{i}(k)$ from our input sequence $\{\mathbf{i}(k'')\}$, $\mathbf{C}$ will be in state $\mathbf{q}(k) = \Lambda(x(k))$ for some $x(k) \in X_{samp}$, and $\mathbf{C}_j$ will be in state $\mathbf{q}_j(k) = \Lambda_j(x_j(k))$ for some $x_j(k) \in X_{samp,j}$ such that $x(k) = (x_1(k), x_2(k), .., x_n(k))$.

   Combining the two points will give the desired result.

   Claim A: We will now prove point (A).

   Let $\mathbf{C}$ be in state $\mathbf{q} = \Lambda(x)$ for some $x \in X_{samp}$.

   Let each $\mathbf{C}_j$ be in state $\mathbf{q}_j = \Lambda_j(x_j)$ for some $x_j \in X_{samp,j}$.

   Assume $x = (x_1, x_2, .., x_n)$.

   We thus have $\mathbf{C}'$ at state $\mathbf{q}' = \mathbf{q}_1\mathbf{q}_2..\mathbf{q}_n$.

   Let $\mathbf{z} = \Phi(\mathbf{q})$ and $\mathbf{z}' = \Phi'(\mathbf{q}')$. Must show $\mathbf{z}' = \mathbf{z}$.

   By Definition 4.2.12 of the output map, we have for $\mathbf{C}$ that $\Phi(\mathbf{q}) = \Gamma_Z(\zeta(x))$

The set of prohibitable events enabled at $\mathbf{q}$ can be represented as

$$\begin{aligned}\Sigma_Z =& \zeta(x) \\ =& \{\sigma \in \Sigma_{hib} | \xi(x,\sigma)!\} && \text{by definition of } \zeta(x) \\ =& \bigcap_{j\in\{1,2,..,n\}} \{\sigma \in \Sigma_{hib} | (\sigma \notin \Sigma_j) \vee (\xi_j(x_j,\sigma)!)\} \\ & \text{by definition of synchronous product}\end{aligned}$$

We next note that by point (1) and (2), $\mathbf{C}$ and $\mathbf{C}'$ represent exact the same events in Σ_{hib} in exactly the same order. It is sufficient to show that $\mathbf{C}'$ enables the same event as $\mathbf{C}$.

By Definition 4.2.15, we have

$$\mathbf{z}' = \Phi'(\mathbf{q}') = \bigwedge_{i\in\{1,2,..,n\}} \mathbf{z}'_j$$

where $\mathbf{z}'_j$ is the expanded output from controller $\mathbf{C}_j$.

As defined, an event is enabled in $\mathbf{z}'_j$ if the event is enabled in $\mathbf{z}_j = \Phi_j(\mathbf{q}_j)$, or the event is not in $\Sigma_{hib,j}$ and thus not in Σ_j. Otherwise, the event is disabled.

Therefore, the set of events enabled by $\mathbf{z}'_j$ can be represented as

$$\begin{aligned}\Sigma'_{Z,j} =& \zeta_j(x_j) \cup \{\Sigma_{hib} - \Sigma_{hib,j}\} \\ =& \{\sigma \in \Sigma_{hib} | (\sigma \notin \Sigma_j) \vee (\xi'_j(x_j,\sigma)!)\} && \text{by definition of } \zeta_j(x_j)\end{aligned}$$

The set of events enabled by $\mathbf{z}'$ and thus $\mathbf{C}'$ can be represented as

$$\begin{aligned}\Sigma'_Z =& \bigcap_{j\in\{1,2,..,n\}} \Sigma'_{Z,j} \\ =& \bigcap_{j\in\{1,2,..,n\}} \{\sigma \in \Sigma_{hib} | (\sigma \notin \Sigma_j) \vee (\xi_j(x_j,\sigma)!)\} \\ =& \Sigma_Z\end{aligned}$$

Claim A proven.

Claim B: We will now prove point (B).

We first consider $k = 0$

By definition, $\mathbf{q}(0) = \mathbf{q}_{res} = \Lambda(x_o)$ and for each $\mathbf{C}_j$, $\mathbf{q}_j(0) = \mathbf{q}_{res,j} = \Lambda_j(x_{o,j})$

We next note that initial states are always sampled states, so we have $x(0) = x_o$ and $x_j(0) = x_{o,j}$. Also, $x_o = (x_{o,1}, x_{o,2}, .., x_{o,n})$ by definition of the synchronous product. We note that input $\mathbf{i}(0)$ is ignored as the controller always starts at its reset state.

We now consider $k \in \{1, 2, ..\}$

As $\{\mathbf{i}(k'')\}$ is input valid for $\mathbf{S}$, we know by definition that:

$$\begin{aligned}&(\forall k \in \{1,2,...\})(\exists s_1, s_2, \ldots, s_k \in L_{conc})\ [s_1 s_2..s_k \in L(\mathbf{S})]\wedge \\ &\qquad [(\forall t \in \{1,2,...,k\})(\forall \sigma \in \Sigma_{act}) i_{g,\gamma_g(\sigma)}(t) = 1 \Leftrightarrow \sigma \in \mathrm{Occu}(s_t)] \end{aligned} \tag{3}$$

This implies that for $t \in \{1, 2, .., k\}$, $\mathrm{Occu}(s_t) = \Gamma_I^{-1}(\mathbf{i}(t))$

We thus have $\Delta(x(0), \Gamma_I^{-1}(\mathbf{i}(1))) = x(1) \in X_{samp}$.

We note that as $\mathbf{S}$ is CS deterministic, $x(1) = \xi(x_o, s_1)$ as any concurrent string with same occurrence image would come to the same state. We thus have $x(2) = \Delta(x(1), \Gamma_I^{-1}(\mathbf{i}(2)))$ with $x(2) = \xi(x_o, s_1 s_2) \in X_{samp}$ and so on, until we have $x(k) = \Delta(x(k-1), \Gamma_I^{-1}(\mathbf{i}(k)))$ with $x(k) = \xi(x_o, s_1 s_2..s_k) \in X_{samp}$.

Let $P_j : \Sigma^* \to \Sigma_j^*$, where $j = 1, 2, .., n$, be a natural projection.

By (3) and definition of the synchronous product, it follows that for $t \in \{1, 2, .., k\}$, $\mathrm{Occu}(P_j(s_t)) = \Gamma_{I,j}^{-1}(\mathbf{i}_j(t))$ and $P_j(s_1)P_j(s_2)..P_j(s_t) \in L(\mathbf{S}_j)$

By a similar logic as above, we have

$$\begin{aligned}&\Delta_j(x(0), \Gamma_{I,j}^{-1}(\mathbf{i}_j(1))) \\ =&x_j(1) \\ =&\xi_j(x_{o,j}, P_j(s_1)) \in X_{samp,j}\end{aligned}$$

until we get

$$x_j(k) = \Delta_j(x(k-1), \Gamma_{I,j}^{-1}(\mathbf{i}_j(k)))$$

with $x_j(k) = \xi_j(x_{o,j}, P_j(s_1)P_j(s_2)..P_j(s_k)) \in X_{samp,j}$

By Definition 4.2.11 for Ω, it is easy to see that $(\forall t \in \{1, 2, .., k\})\mathbf{q}(t) = \Lambda(x(t))$ and $\mathbf{q}_j(t) = \Lambda_j(x_j(t))$.

By definition of the synchronous product

$$\begin{aligned} x(k) =& \xi(x_o, s_1 s_2 .. s_k) \\ =& (\xi_1(x_{o,1}, P_1(s_1)P_1(s_2)..P_1(s_k)), \\ & \xi_2(x_{o,2}, P_2(s_1)P_2(s_2)..P_2(s_k)), .., \\ & \xi_n(x_{o,n}, P_n(s_1)P_n(s_2)..P_n(s_k))) \\ =& (x_1(k), x_2(k), .., x_n(k)) \end{aligned}$$ as required.

Claim B proven.

Let $k \in \{0, 1, ..\}$

We are now ready to show that $\mathbf{z}(k) = \mathbf{z}'(k)$

We next note that by Claim B, that after inputs $\mathbf{i}(0), \mathbf{i}(1), \ldots \mathbf{i}(k)$ from $\{\mathbf{i}(k'')\}$, controller $\mathbf{C}$ is in state $\mathbf{q}(k) = \Lambda(x(k))$ for some $x(k) \in X_{samp}$ and each $\mathbf{C}_j$ is in state $\mathbf{q}_j(k) = \Lambda_j(x_j(k))$ for some $x_j(k) \in X_{samp,j}$ and $x(k) = (x_1(k), x_2(k), .., x_n(k))$.

We can now apply Claim A with $\mathbf{q} = \mathbf{q}(k)$ and each $\mathbf{q}_j = \mathbf{q}_j(k)$ for $j = 1, 2, .., n$, and conclude that for $\mathbf{C}$ at state $\mathbf{q}(k)$ and $\mathbf{C}'$ at state $\mathbf{q}_1(k)\mathbf{q}_2(k)..\mathbf{q}_n(k)$, they will produce the same output. In other words, $\mathbf{z}(k) = \mathbf{z}'(k)$, as required.

By steps **1.**, **2.**, and **3.**, we can thus conclude that $\mathbf{C}$ and $\mathbf{C}'$ are output equivalent with respect to $\mathbf{S}$.

□

Chapter 5

Control and Nonblocking Verification

A controller is more constrained than a supervisor. Every time an event occurs, the supervisor changes its state, but a controller reacts only on sampling instances (tick event). This means it is possible that the enablement information from the controller may not always be exactly the same as that of the supervisor's, as a supervisor can be more expressive in this regard. We want to make sure that the corresponding enablement information that the controller applies to the plant is such that the system's closed loop behavior (the actual behavior of the plant reacting to the controller's enablement information and the event forcing initiated by the controller) stays a subset of the desired behavior specified by the supervisor.

5.1 Supervisory Control Construction

First we have the following definition from [6].

Definition 5.1.1. A *TDES supervisory control* for $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ is any map $V : L(\mathbf{G}) \to \mathrm{Pwr}(\Sigma)$, such that,

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

From now on, we will just use the term *supervisory control* when it is clear by our context that we are referring to TDES.

We will be requiring that prohibitable events can only occur at most once per sampling period. This is to simplify things a bit, but is primarily as we only decide to force an event once per clock cycle, it makes sense that the event only occurs once per clock cycle. If the controller has full control over when the event occurs, this is what will happen so the TDES behavior should reflect this. It makes it easier to keep track of things. Also, **Point iii.1** of the SD controllability definition does not say anything about eligibility of Σ_{hib} events after they have occurred once. As we will see in the proofs in this section, this assumption will be a key part in making the proofs work.

Definition 5.1.2. For TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, we say that $\mathbf{G}$ has *singular prohibitable behavior* if,

$$(\forall s \in L(\mathbf{G}) \cap L_{samp})(\forall s' \in L_{conc})ss' \in L(\mathbf{G}) \\ \implies (\forall \sigma \in \mathrm{Occu}(s') \cap \Sigma_{hib})(\exists s_1, s_2 \in (\Sigma_{act} - \{\sigma\})^*)s' = s_1 \sigma s_2 \tau$$

In other words, the above condition says that for TDES $\mathbf{G}$, a prohibitable event is allowed to occur at most once per sampling period.

If TDES $\mathbf{G}$ is our plant and TDES $\mathbf{S}$ is our supervisor, we likely only care about checking this condition for strings in $L(\mathbf{S}) \cap L(\mathbf{G})$. We thus introduce the definition below. An example that fails the $\mathbf{S}$-singular prohibitable behavior property is shown in Figure 5.1. Here we see the prohibitable event α occurring twice in a sampling period.

Definition 5.1.3. For TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, we say that $\mathbf{G}$ has $\mathbf{S}$-*singular prohibitable behavior* if

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp})(\forall s' \in \Sigma_{\mathrm{act}}^*)ss' \in L(\mathbf{S}) \cap L(\mathbf{G}) \\ \implies (\forall \sigma \in \mathrm{Occu}(s') \cap \Sigma_{hib})\ \sigma \notin \mathrm{Elig}_{L(\mathbf{G})}(ss')$$

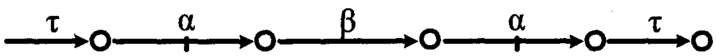


Figure 5.1: An Example Failing $\mathbf{S}$-singular Prohibitable Behavior Property

Let $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a TDES plant. For the rest of this chapter, we will require plant $\mathbf{G}$ to be complete for our supervisor $\mathbf{S}$, have proper time behavior and

S-singular prohibitable behavior, and that **meet**$(\mathbf{G}, \mathbf{S})$ be ALF. This will ensure that for any string $s \in L(\mathbf{G})$ (or $L(V/\mathbf{G})$ if **G** is not ALF on its own), we will always be able to reach a state where tick is possible after at most a finite number of activity events. In other words, we will not "stop the clock." This is important as it ensures that after every sampled string in our system has occurred, all new behavior can be represented as a series of concurrent strings.

Definition 5.1.4. We write $V/\mathbf{G}$ to represent $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ under the supervision of V. The *closed behavior* of $V/\mathbf{G}$ is defined to be $L(V/\mathbf{G}) \subseteq L(\mathbf{G})$ such that

1. $\epsilon \in L(V/\mathbf{G})$;
2. if $s \in L(V/\mathbf{G})$, $\sigma \in V(s)$ and $s\sigma \in L(\mathbf{G})$, then $s\sigma \in L(V/\mathbf{G})$;
3. no other strings are in $L(V/\mathbf{G})$.

It follows from the above definition, that $L(V/\mathbf{G})$ is prefix closed.

Let supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be CS deterministic and SD controllable with respect to our plant **G**. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be a centralized controller translated from **S** using the method described in Section 4.2.3, with input and output event mapping functions γ (see Definition 4.2.2) and η (see Definition 4.2.3), and input and output set mapping functions Γ_I (see Definition 4.2.9) and Γ_Z (see Definition 4.2.10).

To verify that our controller **C** will generate the correct enablement information for our plant, we construct the corresponding supervisory control V for **G**. The idea is to express the enablement information that the controller would provide to the plant as a supervisory control. In particular, we wish to capture the idea that enablement information only changes after a tick, and then stays constant till the next tick. We also want to express the forcing information the controller provides to the plant, in particular the fact that as soon as a prohibitable event is enabled, the controller will force the event to occur within the current sampling period.

The construction of our supervisory control V will be presented as an algorithm. We will use the logic in the algorithm to do the verification. First, we need to have the following definition. An important aspect of sampled strings is that they delineate the concurrent behavior of $\mathbf{G}$, which interprets how $\mathbf{G}$ moves from one sampling state to another.

Definition 5.1.5. For TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, the *concurrent behavior* of $\mathbf{G}$ is defined to be a map $CB_{\mathbf{G}} : L(\mathbf{G}) \cap L_{samp} \rightarrow L_{conc}$, such that for $s \in L(\mathbf{G}) \cap L_{samp}$,

$$CB_{\mathbf{G}}(s) := \{s' \in L_{conc} | ss' \in L(\mathbf{G})\}$$

It states that the possible concurrent behavior for a TDES $\mathbf{G}$ after sampled string $s \in L(\mathbf{G}) \cap L_{samp}$, is the set of concurrent strings that can extend s to a string in the closed behavior of $\mathbf{G}$.

We now discuss our conversion algorithm, labeled Algorithm 5.1. Given $\mathbf{G}$ and the controller $\mathbf{C}$ acting on $\mathbf{G}$, our algorithm constructs our supervisory control map V, by keeping track of how our controller changes state in response to strings generated by our plant. In our next section, we will show the map V is well defined. We first describe some variables that we will use in our algorithm.

Pend $\subseteq L_{samp} \times Q$ is the set of pending $(s, \mathbf{q})$ pairs to be analyzed, where s is a sampled string in $L(\mathbf{G})$, and $\mathbf{q}$ is the corresponding state in the controller that the sequence of inputs that would match the concurrent strings that make up s, unless of course $s = \epsilon$. If $s = \epsilon$, then $\mathbf{q}$ would be our reset state.

Σ_V is the set of prohibitable events enabled by $V(s)$, for current sampled string s that we are processing.

Σ_{temp} is a copy of Σ_V that we make when we are processing a concurrent string that extends the sampled string, s, that we are currently processing. This will be used to keep track of which prohibitable events in Σ_V have not yet occurred in substrings of the concurrent strings that extend s in $L(\mathbf{G})$.

Next, we will explain the statements in Algorithm 5.1 in detail. Note that Algorithm 5.1 may never terminate as the language $L(\mathbf{G})$ may not be finite, thus giving us a non-finite number of string-state pairs to evaluate. The algorithm merely describes abstractly how map V is related to controller $\mathbf{C}$. We will then use this to compare the control behavior of V to that of our supervisor, $\mathbf{S}$, that $\mathbf{C}$ was translated from.

Algorithm 5.1 Obtaining V from controller $\mathbf{C}$, acting plant $\mathbf{G}$

for all $s \in L(\mathbf{G})$ **do**
  $V(s) \leftarrow \Sigma_u \cup \{\tau\}$
end for
$Pend \leftarrow \{(\epsilon, \mathbf{q}_{res})\}$
while $Pend \neq \emptyset$ **do**
  $(s, \mathbf{q}) \leftarrow$ a member from $Pend$
  $Pend \leftarrow Pend - \{(s, \mathbf{q})\}$
  $\mathbf{z} \leftarrow \Phi(\mathbf{q})$
  $\Sigma_V \leftarrow \Gamma_Z^{-1}(\mathbf{z})$
  if $\Sigma_V \neq \emptyset$ **then**
    $V(s) \leftarrow (V(s) \cup \Sigma_V) - \{\tau\}$
  end if
  for all $s' \leftarrow \sigma_1\sigma_2..\sigma_j \in CB_{\mathbf{G}}(s)$ **do** // $\sigma_j = \tau$ *by definition*
    if $(\mathrm{Occu}(s') \cap \Sigma_{hib} \subseteq \Sigma_V) \wedge (ss' \in L(\mathbf{S}))$ **then**
      $\Sigma_{temp} \leftarrow \Sigma_V$
      $\mathbf{i} \leftarrow \Gamma_I(\mathrm{Occu}(s') - \{\tau\})$
      $\mathbf{q}' \leftarrow \Omega(\mathbf{q}, \mathbf{i})$
      $Pend \leftarrow Pend \cup \{(ss', \mathbf{q}')\}$
      if $j > 1$ **then**
        for $i \leftarrow 1$ to $j - 1$ **do**
          $\Sigma_{temp} \leftarrow \Sigma_{temp} - \sigma_i$
          if $\Sigma_{temp} \neq \emptyset$ **then**
            $V(s\sigma_1\sigma_2..\sigma_i) \leftarrow (V(s\sigma_1\sigma_2..\sigma_i) \cup \Sigma_V) - \{\tau\}$
          else
            $V(s\sigma_1\sigma_2..\sigma_i) \leftarrow (V(s\sigma_1\sigma_2..\sigma_i) \cup \Sigma_V)$
          end if
        end for
      end if
    end if
  end for
end while
return V

Initially, the **for-loop** from **line 1** to **line 3** includes all events $\sigma \in \Sigma_u \cup \{\tau\}$ in $V(s)$ for all $s \in L(\mathbf{G})$, to ensure all uncontrollable events are eligible in $V(s)$. This is needed to satisfy the controllability definition. This is the default setting for each possible string s. The tick event will be removed later, if we are suppose to be forcing an event.

A controller always starts operating at its reset state, so this will be the first state we will examine. As this corresponds to the empty string, our starting place is thus the tuple $(\epsilon, \mathbf{q})$. On **line 4**, we thus initialize our set of pending tuples to $(\epsilon, \mathbf{q})$.

The set *Pend* contains all the state-string pairs that have not been analyzed, and its members will be extracted one by one in the **while-loop** running from **line 5** to **line 31**. There are two parts in the **while-loop**, where we process $V(s)$ and then $V(s\sigma_1\sigma_2..\sigma_i)$ for $i < |s'|$, $s' \in CB_{\mathbf{G}}(s)$.

At **line 6** in the **while-loop**, a member $(s, \mathbf{q})$ is extracted from the set *Pend.* This is the next tuple to be analyzed.

At **line 8** output vector $\mathbf{z}$ is obtained from the current controller state $\mathbf{q}$ by applying output function Φ. Vector $\mathbf{z}$ represents all the prohibitable events that the controller enables while it is at state Q. Then at **line 9**, all the prohibitable events enabled by the controller at current state $\mathbf{q}$ are included in Σ_V. This is done by using the inverse of the output set mapping function $\Gamma_Z^{-1}(\mathbf{q})$ from Definition 4.2.10.

At **line 11** the enablement information Σ_V is included in $V(s)$ for current sampled string s. As mentioned, the tick event included at **line 2** is removed here in accordance to **Point ii** of the SD controllability definition (Definition 3.2.2). Basically, it says if we have eligible prohibitable events enabled, we must disable a tick and force the event. Of course, when we later show that the map V we have defined is indeed a TDES supervisor control, we will have to show that these prohibitable events were eligible in $L(\mathbf{G})$ at this point.

The **for-loop** from **line 13** to **line 30** loops through all possible concurrent strings $s' = \sigma_1\sigma_2..\sigma_j \in CB_{\mathbf{G}}(s)$ (i.e. those that can extend s in $L(\mathbf{G})$). First, it calculates the input vector, $\mathbf{i}$, that would correspond to s' occurring. This is done by using the controller's input event mapping function, Γ_I. We then use the controller's next state function, Ω, to calculate $\mathbf{q}'$, the state reached from $\mathbf{q}$ by input vector $\mathbf{i}$. Recall that $CB_{\mathbf{G}}(s)$ from Definition 5.1.5 is the concurrent behavior at state $\delta(y_o, s)$ in $\mathbf{G}$.

At **line 14**, we ignore concurrent strings whose occurrence images contain prohibitable events that are not in Σ_V. The reason is that these events have been disabled by the controller, so this represents behavior that will not occur in the closed loop system, so we just leave it at the default enablement information specified at **line 2**.

We also ignore concurrent strings that do not represent behavior in $L(\mathbf{S})$, thus restricting the strings we can change from their **line 2** defaults, to strings in $L(\mathbf{S}) \cap L(\mathbf{G})$. The reason is that we later need to prove that our V satisfies Definition 5.1.1. We will do this later by first showing that $L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$, and then use the fact that $\mathbf{S}$ is SD controllable for $\mathbf{G}$.

At **line 15**, all prohibitable events in Σ_V are copied to Σ_{temp}, which stores prohibitable events in Σ_V that have not yet occurred in this sampling period. At **line 18**, the new string-state pair $(ss', \mathbf{q}')$ is added to set *Pend*.

At **line 19**, the **if** statement checks if s' contains events other than tick. Since the only tick event in a concurrent string is the ending event, it only checks if $j > 1$ for $j = |s'|$. If so, we execute **lines 20** to **27**.

In the inner most loop from **line 20** to **line 27**, we analyze each substring $\sigma_1\sigma_2..\sigma_i$, $i < j$.

For **lines 22** to **line 26**, if there are still prohibitable events in Σ_{temp} that have not yet occurred, the map $V(s\sigma_1\sigma_2..\sigma_i)$ has to remove the tick event since in our setting, enabling a prohibitable event also means we want to force it. Otherwise we leave the tick event in $V(s\sigma_1\sigma_2..\sigma_i)$. In either case, we add Σ_V to $V(s\sigma_1\sigma_2..\sigma_i)$ since the enablement information of a controller is constant until the next tick event.

In the rest of the chapter, when we are discussing a system with plant $\mathbf{G}$, and CS deterministic TDES supervisor $\mathbf{S}$ that is SD controllable for $\mathbf{G}$, we will be concerned about the SD controller $\mathbf{C}$ that is constructed from $\mathbf{S}$ using the translation method described in Section 4.2, and TDES supervisory control V[1] that is constructed from $\mathbf{C}$ using Algorithm 5.1.

Definition 5.1.6. For plant $\mathbf{G}$, and CS deterministic supervisor $\mathbf{S}$ that is SD controllable for $\mathbf{G}$, let $\mathbf{C}$ be the SD controller that is constructed from $\mathbf{S}$ using the translation method described in Section 4.2, and V be the map that is constructed from $\mathbf{C}$ using

[1]We still need to prove that our map V is indeed a TDES supervisory control, and that the map is well defined. We will prove this in the following sections.

Algorithm 5.1. The *marked behavior* of $V/\mathbf{G}$ is defined to be

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

We say V is *nonblocking* for $\mathbf{G}$ if

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

That is, a nonblocking supervisory control V for $\mathbf{G}$ can always reach a marked state in both $\mathbf{G}$ and $\mathbf{S}$ by extending the current string $s \in L(V/\mathbf{G})$.

5.2 Map V Is Well Defined

We want to show that the map V constructed using Algorithm 5.1 is well defined for any possible string $s \in L(\mathbf{G})$ so that it can be considered as a possible supervisory control.

For example, Let TDES $\mathbf{G}$ be a plant defined over $\Sigma = \{\alpha, \beta, \gamma, \omega, \tau\}$ where $\tau = tick$. Let $\Sigma_{hib} = \{\alpha, \beta, \gamma\}$. Let $\mathbf{C}$ be the controller acting on $\mathbf{G}$. Imagine a part of $\mathbf{G}$ as shown in Figure 5.2.

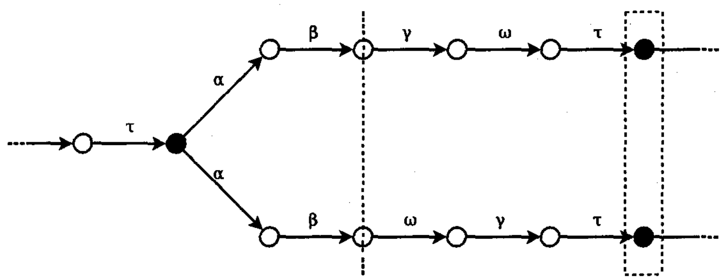


Figure 5.2: Part of a TDES plant

In the figure, let $s \in L(\mathbf{G})$ be the string taking us to the left most sampling state. We see there are two concurrent strings $s'_1 = \alpha\beta\gamma\omega\tau$ and $s'_1 = \alpha\beta\omega\gamma\tau$ extending s in different paths so that $ss'_1, ss'_2 \in L(\mathbf{G})$. Let $\hat{s} = \alpha\beta$ to be the prefix of both s'_1 and s'_2. Since Algorithm 5.1 will evaluate $V(s\hat{s})$ twice (**lines 22** to **26**), we want to make sure each time $V(s\hat{s})$ is assigned the same control action for both paths in the figure. We also need to make sure that every string $s \in L_{samp} \cap L(\mathbf{G})$ is either evaluated once, or is always associated with the same state $\mathbf{q}$ of the controller. We then have the following proposition to be proven.

Proposition 5.1. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let $\mathbf{C}$ be the SD controller that is constructed from $\mathbf{S}$ using the translation method described in Section 4.2.3, and V be map that is constructed from $\mathbf{C}$ using Algorithm 5.1. Then, map V is well defined.

Proof. Assume initial conditions for proposition.

To show that V is well defined, we need to show that for every $s \in L(\mathbf{G})$, our algorithm will define $V(s)$ in only one way.

From the definition of Algorithm 5.1, it is clear that for all $s \notin \overline{L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp}}$, the algorithm only defines $V(s)$ exactly once on **line 2**.

This means we only have to examine strings $s \in \overline{L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp}}$.

Let $s \in \overline{L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp}}$.

Further examination of Algorithm 5.1 shows that $V(s)$ is only updated on **line 11**, **line 23** and **line 25**, if at all.

Examining these cases, we see that if $s \in L_{samp}$, it will only be updated at **line 11**. Otherwise, it could be updated once or more at **line 23** or **line 25**.

Case A) $s \in L_{samp}$

We first note that we only care about sampled strings that are added to *Pend*. If s is never added to *Pend*, it is only defined at **line 2** and is never updated, thus is uniquely defined. We can thus assume that s is added at some point to *Pend*, without loss of generality.

To show that there is only one way to define $V(s)$, it is sufficient to show that whenever **line 11** was executed for s, Σ_V was always the same. Clearly, as long as Σ_V is the same, then executing **line 11** again will produce the same result as the first time. As Σ_V is uniquely defined by state $\mathbf{q}$ of controller $\mathbf{C}$, it is thus sufficient to show that string s will always be paired with state $\mathbf{q}$.

If $s = \epsilon$, then by definition this is always paired with state $\mathbf{q}_{res}$. Studying the algorithm, it is easy to see this is the case. We thus need only consider the case of $s \in \Sigma^*.\tau$.

Examining Algorithm 5.1, we can see that every such string in *Pend*, is constructed by concatenating one or more concurrent strings together.

For s, we thus have:

$$(\exists n \in \{1, 2, ..\})(\exists s_1, s_2, .., s_n \in L_{conc}) s_1 s_2 .. s_n = s$$

As $L_{conc} = \Sigma_{act}^{+}.\tau$, there is only way to define strings s_1 to s_n.

Examining **line 16** and **line 17** of the algorithm, we see how starting with $\mathbf{q}_{res}$, each new state would be calculated using the next concurrent string in the list. Examining the definition of Γ_I and Ω from Section 4.2.3, and Δ from Section 3.1, we can see that since supervisor $\mathbf{S}$ is CS deterministic and $s \in L(\mathbf{S}) \cap L_{samp}$, this sequence of states is unique, meaning the final state $\mathbf{q}$ associated with s is unique for controller $\mathbf{C}$.

We thus conclude that we will always associate the same state $\mathbf{q}$ with s, thus the same set Σ_V.

Case B) $s \notin L_{samp}$

This implies $(\exists t \in L_{samp})(\exists \hat{t} \in L_{conc}) t < s < t\hat{t}$

We note that this implies $(\exists j > 1)(\exists \sigma_1, \sigma_2, .., \sigma_j \in \Sigma)\hat{t} = \sigma_1\sigma_2..\sigma_j$

We thus have $(\exists i \in \{1, 2, .., j-1\}) t\sigma_1, \sigma_2, .., \sigma_i = s$

Note that in above, we have $j > 1$ since as $t < s < t\hat{t}$, $j = 0, 1$ would cause a contradiction. Basically, $j = 0$ would mean $\hat{t} = \epsilon$, thus $t\hat{t} = t$ and we could not have $t < s < t$. If we had $j = 1$, we would have $\hat{t} = \tau$ as $\hat{t} \in L_{conc}$. As we require $t < s$, s must contain at least one event more than t, but that would not also allow $s < t\hat{t}$ as $\hat{t}$ only contains one event. We thus must have $j > 1$.

We note if for all such $\hat{t}$ they fail the condition on **line 14**, or if t was never added to *Pend*, then $V(s)$ will never be updated again, and will retain the value it was assigned on **line 2**. Thus, with no loss of generality, we can assume that t was added to *Pend* and our $\hat{t}$ passes the condition on **line 14**. We thus have $t, t\hat{t} \in L(\mathbf{S}) \cap L(\mathbf{G})$.

Given the definition of L_{conc} and sampled strings, it is easy to see that there is only one way to define $\sigma_1, \sigma_2, .., \sigma_i$, and thus sampled string t.

From Part A, we saw that for a given sampled string, there is only one way to define the corresponding Σ_V set. Of course, it is possible that there are multiple ways to define $\sigma_{i+1}..\sigma_j$.

Examining Algorithm 5.1, we see that the portion that we are concerned with corresponds to **line 19** to **line 28**. Examining these lines, we see that the definition of $V(s)$ is determined only by Σ_V and $t\sigma_1, \sigma_2, .., \sigma_i$, which are unique for s.

It thus follows that $V(s)$ is unique defined for our s.

By Case A and Case B, we have shown that V is well defined. □

5.3 Supervisory Control and SD Supervisors

Given the map V constructed from $\mathbf{C}$ by Algorithm 5.1, we want show that the closed loop behavior $L(V/\mathbf{G})$ equals the behavior of $\mathbf{meet}(\mathbf{G}, \mathbf{S})$, i.e.

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

By Definition 5.1.4 for $L(V/\mathbf{G})$, we find that $\{\epsilon\} \subseteq L(V/\mathbf{G}) \subseteq L(\mathbf{G})$. We thus need to make sure that $L(\mathbf{G}) \neq \emptyset$. This is automatic as long as $\mathbf{G}$ has an initial state.

Theorem 5.1. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \delta, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite statespaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, have proper time and $\mathbf{S}$-singular prohibitable behavior, let $\mathbf{meet}(\mathbf{G}, \mathbf{S})$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$ using the translation method described in Section 4.2.3, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 5.1. Then,

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

Proof. Assume assumptions in proposition setup.

To show $L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$, we must

1. show $L(V/\mathbf{G}) \subseteq L(\mathbf{S}) \cap L(\mathbf{G})$

2. show $L(V/\mathbf{G}) \supseteq L(\mathbf{S}) \cap L(\mathbf{G})$

To show **1**, must show **1.1** and **1.2** as follows.

1.1 show $L(V/\mathbf{G}) \subseteq L(\mathbf{G})$

This is automatic by Definition of $L(V/\mathbf{G})$ and the fact $\mathbf{G}$ contains an initial state.

1.2 show $L(V/\mathbf{G}) \subseteq L(\mathbf{S})$

To show this, we must show

$$(\forall s \in L(V/\mathbf{G}))s \in L(\mathbf{S}) \qquad (1)$$

Let $s \in L(V/\mathbf{G})$. We can show it by induction as follows.

base case $s = \epsilon$.

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

inductive step We assume that $s = \sigma_1..\sigma_k \in L(V/\mathbf{G}) \cap L(\mathbf{S})$ and $s\sigma_{k+1} \in L(V/\mathbf{G})$ for some $k \geq 0$. We will now show this implies that

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

Since $\Sigma = \Sigma_u \dot{\cup} \Sigma_c = \Sigma_u \dot{\cup} \Sigma_{hib} \dot{\cup} \{\tau\}$ by definition of TDES, we have 3 cases for $\sigma_{k+1} \in \Sigma$

(i) $\sigma_{k+1} \in \Sigma_u$

As $s\sigma_{k+1} \in L(V/\mathbf{G})$, it follows that $\sigma_{k+1} \in \text{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u$ As $\mathbf{S}$ is

SD controllable for $\mathbf{G}$, it follows that $\sigma_{k+1} \in \text{Elig}_{L(\mathbf{S})}(s)$, thus $s\sigma_{k+1} \in L(\mathbf{S})$

(ii) $\sigma_{k+1} = \tau$

To show $\tau \in \text{Elig}_{L(\mathbf{S})}(s)$, by **Point ii** in Definition 3.2.2 of SD controllability (since $\mathbf{S}$ is SD controllable for $\mathbf{G}$) we need to show

$$(\tau \in \text{Elig}_{L(\mathbf{G})}(s)) \wedge (\text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset)$$

Since $L(V/\mathbf{G}) \subseteq L(\mathbf{G})$ as shown in **1.1**, we have

$$\begin{aligned} &\tau \in \text{Elig}_{L(V/\mathbf{G})}(s) \\ \Longrightarrow\ &\tau \in \text{Elig}_{L(\mathbf{G})}(s) \end{aligned}$$

Now we need to show

$$\text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$$

By default, the tick event is included in $V(s)$ at **line 2**. In the algorithm, tick is only removed if $\Sigma_V \neq \emptyset$ at **line 11** or $\Sigma_{temp} \neq \emptyset$ at **line 23**.

We thus have four possibilities: **a)** $s \in L_{samp}$, $\Sigma_V = \emptyset$ and s was added to *Pend*, **b)** $s \notin L_{samp}$, $\Sigma_{temp} = \emptyset$, and $V(s)$ is re-evaluated **c)** $s \in L_{samp}$ and s was not added to *Pend*, or **d)** $s \notin L_{samp}$ and $V(s)$ is not re-evaluated. We now examine these cases.

(ii.a) $s \in L_{samp}$, $\Sigma_V = \emptyset$, and s was added to *Pend*.

As $s \in L_{samp}$, either it is the empty string, or $s \in \Sigma^*.\tau$. For the case $s = \epsilon$, we have $\Lambda(x_o) = \mathbf{q}_{res}$ (see Definition 4.2.8), which matches the state-string association that Algorithm 5.1 makes.

Otherwise, s is composed of one or more concurrent strings. We thus have

$$(\exists n \in 1, 2, ..)(\exists s_1, s_2, .., s_n \in L_{conc})s_1 s_2..s_n = s$$

As $L_{conc} = \Sigma_{act}^{+}.\tau$, there is only way to define strings s_1 to s_n.

Based on the definitions from Section 4.2.3, we can determine the state in $\mathbf{C}$ that will correspond to string s, by starting with $\mathbf{q}_{res}$, and evaluating

$$\Omega(\mathbf{q}_{res}, \Gamma_I(\mathrm{Occu}(s_1) - \{\tau\})) = \mathbf{q}_1$$

As $\mathbf{S}$ is CS deterministic, it follows from the definitions of Γ_I and Ω that $\mathbf{q}_1 = \Lambda(x_1)$, where $x_1 = \xi(x_o, s_1)$. Note that we have $s_1 \in L(\mathbf{S})$ as the language is closed.

By the same logic we have

$$\Omega(\mathbf{q}_1, \Gamma_I(\mathrm{Occu}(s_2) - \{\tau\})) = \mathbf{q}_2$$

and $\mathbf{q}_2 = \Lambda(x_2)$, where $x_2 = \xi(x_o, s_1 s_2)$.

Extending this logic to the end, we have

$$\Omega(\mathbf{q}_{n-1}, \Gamma_I(\mathrm{Occu}(s_n) - \{\tau\})) = \mathbf{q}_n$$

and $\mathbf{q}_n = \Lambda(x_n)$, where $x_n = \xi(x_o, s_1 s_2..s_n)$. To simplify the notation, we will take $\mathbf{q} = \mathbf{q}_n$ and $x = x_n$.

We thus have $\mathbf{q}$ the state the controller $\mathbf{C}$ will be in after string s, and $x \in X_{samp}$ the state that $\mathbf{S}$ will be in, while $\mathbf{q} = \Lambda(x)$. It is easy to see by the logic of Algorithm 5.1, that string s will be paired with state $\mathbf{q}$. See proof of Proposition 5.1 for more details.

We next note that the outputs at state $\mathbf{q}$ are $\mathbf{z} = \Phi(\mathbf{q})$. We thus have by Definition 4.2.12 that $\mathbf{z} = \Gamma_Z(\zeta(x))$.

By the definition of control action given in Definition 3.2.3, it follows that $\zeta(x) = \{\sigma \in \Sigma_{hib} | \xi(x, \sigma)!\}$. As $\Sigma_V = \Gamma_Z^{-1}(\mathbf{z})$ as per **line 9** of Algorithm 5.1, we thus have $\Sigma_V = \{\sigma \in \Sigma_{hib} | \xi(x, \sigma)!\}$. As we have $\Sigma_V = \emptyset$ by assumption, we thus have

$$\{\sigma \in \Sigma_{hib} | \xi(x, \sigma)!\} = \emptyset$$

which implies

$$\begin{aligned} & \mathrm{Elig}_{L(\mathbf{S})}(s) \cap \Sigma_{hib} = \emptyset \qquad\qquad \text{as } \xi(x_o, s) = x \\ \Longrightarrow\ & \mathrm{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \end{aligned}$$

as required.
Part **(ii.a)** complete.

(ii.b) $s \notin L_{samp}$, $\Sigma_{temp} = \emptyset$ and $V(s)$ is re-evaluated.
As $V(s)$ is re-evaluated and $s \notin L_{samp}$, then it follows from the logic of Algorithm 5.1 that

$$\begin{aligned} & (\exists t \in L_{samp} \cap L(\mathbf{S}) \cap L(\mathbf{G}))(\exists \hat{t} \in L_{conc}) \\ & \qquad (t < s < t\hat{t}) \wedge (t\hat{t} \in L(\mathbf{S}) \cap L(\mathbf{G})) \end{aligned} \tag{1}$$

It also follows that:

$(\exists l \in \{1,2,..\})(\exists \sigma_1, \sigma_2, .., \sigma_l \in \Sigma_{act} \subset \Sigma)\, t\sigma_1\sigma_2..\sigma_l = s$

Now, from the logic of part **(ii.a)**, we know that string t will be paired in *Pend* with a state $\mathbf{q}$, such that $\mathbf{q} = \Lambda(x)$, where $x = \xi(x_o, t)$. We thus have

$$\Sigma_V = \mathrm{Elig}_{L(\mathbf{S})}(t) \cap \Sigma_{hib}$$
$$\Longrightarrow \mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(t) \cap \Sigma_{hib} \subseteq \Sigma_V$$

We now note that as **S** is SD controllable for **G**, we have by **Point iii.1** of Definition 3.2.2,

$$[\mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cup \mathrm{Occu}(\sigma_1\sigma_2..\sigma_l)] \cap \Sigma_{hib}$$
$$= \mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(t) \cap \Sigma_{hib} \subseteq \Sigma_V$$

$$\Rightarrow (\mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cup \mathrm{Occu}(\sigma_1\sigma_2..\sigma_l)) \cap \Sigma_{hib} \subseteq \Sigma_V$$
$$\Rightarrow \mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} \subseteq \Sigma_V \qquad (2)$$

It follows from the logic of Algorithm 5.1 and fact that $\Sigma_{temp} = \emptyset$, that

$$\mathrm{Occu}(\sigma_1\sigma_2..\sigma_l) \cap \Sigma_{hib} \supseteq \Sigma_V$$

In other words, every prohibitable event in Σ_V has occurred at least once since t (i.e. this sampling period).

As $t \in L(\mathbf{S}) \cap L(\mathbf{G})$ by (1), and **G** has **S**-singular prohibitable behavior by our initial assumptions, it follows that the prohibitable events in Σ_V cannot occur again in $\hat{t}$ (i.e. not until after next tick). This implies

$$\mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_V = \emptyset$$

Since $\mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} \subseteq \Sigma_V$ by (2), it follows that

$$\mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$$

as required.

Part **(ii.b)** complete.

(ii.c) $s \in L_{samp}$ and s not added to *Pend*

We will show that $s \in L_{samp}$ causes a contradiction and thus s must be added to *Pend*. This means that case **(ii.a)** represents the only valid possibility, if $s \in L_{samp}$

We note that $L_{samp} = \Sigma^*.tick \cup \{\epsilon\}$. If $s = \epsilon$, then we know ϵ is always added to *Pend* (**line 4** of Algorithm 5.1), so this section does not apply. We can thus assume $s \neq \epsilon$, and thus $s \in \Sigma^*.\tau$

Our goal is to show that s will always be added to *Pend*, thus **(ii.c)** never applies.

As $s \in \Sigma^*.\tau$, if follows

$$(\exists n \in \{1, 2, ..\})(\exists s_1, s_2, .., s_n \in L_{conc}) s_1 s_2 .. s_n = s$$

To show that s must be added to *Pend*, we need to show:

$$(\forall l \in \{1, 2, \ldots, n\})$$
$$(s_1 s_2 .. s_l \in L(\mathbf{S}) \cap L(\mathbf{G})) \wedge (\mathrm{Occu}(s_l) \cap \Sigma_{hib} \subseteq \Sigma_V(s_1 s_2 .. s_{l-1}))$$

where $\Sigma_V(s_1 s_2 .. s_{l-1})$ is the value of Σ_V at **line 14** in the algorithm when sampled string $s_1 s_2 .. s_{l-1}$ is being evaluated.

Let $l \in \{1, 2, \ldots, n\}$.

As $s \in L(\mathbf{S}) \cap L(\mathbf{G})$ by assumption, and $L(\mathbf{S})$ and $L(\mathbf{G})$ are closed languages, $s_1 s_2 .. s_l \in L(\mathbf{S}) \cap L(\mathbf{G})$ is automatic.

All that remains is showing

$$\mathrm{Occu}(s_l) \cap \Sigma_{hib} \subseteq \Sigma_V(s_1 s_2 .. s_{l-1})$$

We know from part **(ii.a)** that sampled string $s_1 s_2 .. s_{l-1}$ will always be paired with state $\mathbf{q}$ of the controller, with $\mathbf{q} = \Lambda(x)$, where $x = \xi(x_o, s_1 s_2 .. s_{l-1})$. This state $\mathbf{q}$ is the state the controller will be in after this string, thus $\Sigma_V(s_1 s_2 .. s_{l-1})$ will be the enablement output (as per definition of Algorithm 5.1) of the controller until after the next tick occurs. That means for all $\sigma \in \Sigma_{hib} - \Sigma_V(s_1s_2..s_{l-1})$, σ will be disabled until after concurrent string s_l has occurred. As $s \in L(V/\mathbf{G})$ by assumption, we also have $s_1s_2..s_l \in L(V/\mathbf{G})$ as $L(V/\mathbf{G})$ is closed and $s_1s_2..s_l \leq s$. This means s_l cannot contain any events in $\Sigma_{hib} - \Sigma_V(s_1s_2..s_{l-1})$, thus

$$\mathrm{Occu}(s_l) \cap \Sigma_{hib} \subseteq \Sigma_V(s_1s_2..s_{l-1})$$

We have thus shown that for $s = \epsilon$ or $s \in \Sigma^*.\tau$, it must have been added to *Pend*. This means that **(ii.c)** does not apply to s at all, so string s must be covered by case **(ii.a)**.
Part **(ii.c)** complete.

(ii.d) $s \notin L_{samp}$ and $V(s)$ is not re-evaluated.

We now examine case of $s \notin L_{samp}$ and show that it must have been re-evaluated at **line 23** or **line 25**, thus case **(ii.b)** is the only valid possibility for $s \notin L_{samp}$.

As $s \notin L_{samp}$, it follows that: $(\exists t \in L_{samp})(\exists \hat{t} \in L_{conc})t < s < t\hat{t}$

It also follows that

$$(\exists l \in \{1, 2, ..\})(\exists \sigma_1, \sigma_2, .., \sigma_l \in \Sigma_{act} \subset \Sigma)\, t\sigma_1\sigma_2..\sigma_l = s \qquad (3)$$

To show that $V(s)$ must have been re-evaluated at **line 23** or **line 25**, it is sufficient to show that t must be added to *Pend*, and that there exist a $\hat{t}$ that will pass the condition on **line 14**.

We first note that as $s \in L(\mathbf{S}) \cap L(\mathbf{G})$, it follows that $t \in L(\mathbf{S}) \cap L(\mathbf{G})$ as $L(\mathbf{G})$ and $L(\mathbf{S})$ are closed languages. Similarly, as $s \in L(V/\mathbf{G})$, we also have $t \in L(V/\mathbf{G})$.

We can thus apply the logic from **(ii.c)**, and conclude that t must be added to *Pend*, and it will be paired with state $\mathbf{q}$ of the controller with $\mathbf{q} = \Lambda(x)$ where $x = \xi(x_o, t)$. State $\mathbf{q}$ is the state the controller will be in after string t, thus Σ_V will be the enablement output of the controller, where

$$\Sigma_V = \text{Elig}_{L(\mathbf{S})}(t) \cap \Sigma_{hib} \tag{4}$$

We now need to show:

$$(\exists \hat{t} \in L_{conc})$$
$$(s < t\hat{t}) \wedge (t\hat{t} \cap L(\mathbf{S}) \cap L(\mathbf{G})) \wedge \text{Occu}(\hat{t}) \cap \Sigma_{hib} \subseteq \Sigma_V$$

We start by constructing a string $\hat{t} \in L_{conc}$ that satisfies the first two conditions.

We note that by assumption, $\mathbf{G}$ and $\mathbf{S}$ have finite statespaces, $\mathbf{G}$ has proper time behavior, $\mathbf{meet}(\mathbf{G}, \mathbf{S})$ is ALF, and that $\mathbf{S}$ is controllable for $\mathbf{G}$ (this is implied by fact $\mathbf{S}$ is SD controllable for $\mathbf{G}$). We can thus apply Proposition 2.4 and conclude

$$(\exists s' \in \Sigma^*) ss'\tau \in L(\mathbf{S}) \cap L(\mathbf{G})$$

We can thus take $\hat{t} = \sigma_1\sigma_2..\sigma_l s'\tau$ and we have $s < t\hat{t}$ (by (3)) and $t\hat{t} \in L(\mathbf{S}) \cap L(\mathbf{G})$.

All that remains is to show

$$\text{Occu}(\hat{t}) \cap \Sigma_{hib} \subseteq \Sigma_V$$

From (4), we have

$$\Sigma_V = \text{Elig}_{L(\mathbf{S})}(t) \cap \Sigma_{hib}$$
$$\Longrightarrow \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(t) \subseteq \Sigma_V$$

We now note that as $\mathbf{S}$ is SD controllable for $\mathbf{G}$, we have by **Point iii.1** of Definition 3.2.2

$$(\forall t' \in \Sigma_{act}^*)(t' < \hat{t}) \Longrightarrow$$
$$[\text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(tt') \cup \text{Occu}(t')] \cap \Sigma_{hib} = \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(t) \cap \Sigma_{hib} \subseteq \Sigma_V$$

If we take $t' = \sigma_1\sigma_2..\sigma_l s' < t$ we have

$$\begin{aligned}&[\mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(tt') \cup \mathrm{Occu}(t')] \cap \Sigma_{hib} \subseteq \Sigma_V \\ \Longrightarrow\ &\mathrm{Occu}(t') \cap \Sigma_{hib} \subseteq \Sigma_V\end{aligned}$$

As $t'\tau = \hat{t}$, we thus have $\mathrm{Occu}(\hat{t}) \cap \Sigma_{hib} \subseteq \Sigma_V$

We have now shown, that for $s \notin L_{samp}$, we must have re-evaluated $V(s)$ at **line 23** or **line 25**, so **(ii.d)** does not apply. This means that string s must be covered under **(ii.b)**.

Part **(ii.d)** complete.

We thus have shown by **(ii.a-d)**, that

$$\begin{aligned}&\mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \ \wedge \ \tau \in \mathrm{Elig}_{L(\mathbf{G})}(s) \\ \Longrightarrow\ &\tau \in \mathrm{Elig}_{L(\mathbf{S})}(s) \qquad \text{by } \textbf{Point ii} \text{ of Definition 3.2.2} \\ \Longrightarrow\ &s\sigma_{k+1} \in L(\mathbf{S})\end{aligned}$$

(iii) $\sigma_{k+1} \in \Sigma_{hib}$

For $s \in L(\mathbf{S}) \cap L(\mathbf{G})$ and $s \in L(V/\mathbf{G})$, we know there exists $t \in L_{samp}$ and $t' \in \Sigma_{act}^*$ such that $s = tt'$.

From **(ii.a)**, we know that Algorithm 5.1 will pair sampled string t with state $\mathbf{q}$ in the controller, with $\mathbf{q} = \Lambda(x)$, where $x = \xi(x_o, t)$.

Also, we have

$$\Sigma_V = \mathrm{Elig}_{L(\mathbf{S})}(t) \cap \Sigma_{hib}$$

$\Rightarrow \mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(t) \cap \Sigma_{hib} \subseteq \Sigma_V$

We now will show that $\sigma_{k+1} \in \Sigma_V$.

We note that as $s\sigma_{k+1} \in L(V/\mathbf{G})$, we have $\sigma_{k+1} \in V(s)$.

From **line 2**, we see $V(s)$ is initially set to $\Sigma_u \cup \{\tau\}$. This means that σ_{k+1} must have been added at **line 11** if $t' = \epsilon$ and $t = s$, or at **line**

23 or **line 25**. In either case it implies our prohibitable event is in Σ_V.

We thus have

$$\begin{aligned} \sigma_{k+1} \in \Sigma_V &\implies \sigma_{k+1} \in \mathrm{Elig}_{L(\mathbf{S})}(t) \\ &\implies \sigma_{k+1} \in \mathrm{Elig}_{L(\mathbf{G})}(t) \qquad \text{as } \mathbf{G} \text{ is complete for } \mathbf{S}. \end{aligned} \tag{5}$$

As **S** is SD controllable for **G**, we have from **Point iii.1** of Definition 3.2.2 that

$$[\mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(tt') \cup \mathrm{Occu}(t')] \cap \Sigma_{hib} = \mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(t) \cap \Sigma_{hib} \tag{6}$$

We note that as $s\sigma_{k+1} \in L(V/\mathbf{G})$, we have $s\sigma_{k+1} \in L(\mathbf{G})$. As **G** has **S**-singular prohibitable behavior, this implies σ_{k+1} has not yet occurred in this sampling period. Thus

$$\sigma_{k+1} \notin \mathrm{Occu}(t') \tag{7}$$

From (5), we have $\sigma_{k+1} \in \mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(t)$.

From (6), we thus have

$$\sigma_{k+1} \in [\mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(tt') \cup \mathrm{Occu}(t')]$$

As $\sigma_{k+1} \notin \mathrm{Occu}(t')$ from (7), it follows that

$$\begin{aligned} &\sigma_{k+1} \in \mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(s) && (\text{as } tt' = s) \\ \implies &s\sigma_{k+1} \in L(\mathbf{S}) && \text{as required} \end{aligned}$$

By **(i)**, **(ii)** and **(iii)**, we have shown $s\sigma_{k+1} \in L(\mathbf{S})$ for any $\sigma_{k+1} \in \Sigma$, thus our inductive step is complete.

Thus by our base case and our inductive step, we have $s \in L(\mathbf{S})$ for arbitrary $s \in L(V/\mathbf{G})$. Therefore, **1.2** is complete.

By step **1.1** and **1.2**, we have shown $L(V/\mathbf{G}) \subseteq L(\mathbf{S}) \cap L(\mathbf{G})$.

2. Show $L(V/\mathbf{G}) \supseteq L(\mathbf{S}) \cap L(\mathbf{G})$

Let $s \in L(\mathbf{S}) \cap L(\mathbf{G})$. We need to show this implies

$$s \in L(V/\mathbf{G})$$

We will show this using proof by induction.

base case $s = \epsilon$

Automatic that $s \in L(V/\mathbf{G})$, by Definition 5.1.4 for $L(V/\mathbf{G})$.

inductive step We assume that $s = \sigma_1\sigma_2..\sigma_k \in L(V/\mathbf{G}) \cap L(\mathbf{S}) \cap L(\mathbf{G})$ and $s\sigma_{k+1} \in L(\mathbf{S}) \cap L(\mathbf{G})$ for some $k \geq 0$.

We will now show this implies $s\sigma_{k+1} \in L(V/\mathbf{G})$.

Sufficient to show $\sigma_{k+1} \in V(s)$ by Definition of $L(V/\mathbf{G})$, and fact we already have $s\sigma_{k+1} \in \cancel{L(\mathbf{G})}$

Again, since $\Sigma = \Sigma_u \dot\cup \Sigma_{hib} \dot\cup \{\tau\}$ by definition of TDES, we have 3 cases for $\sigma_{k+1} \in \Sigma$.

(i) $\sigma_{k+1} \in \Sigma_u$

This is automatic by **line 2** in Algorithm 5.1, where all uncontrollable events are included in $V(s)$ for each possible string s by default. Examining the algorithm, it is clear that uncontrollable events are never later removed.

(ii) $\sigma_{k+1} = \tau$

As we have $s\tau \in L(\mathbf{G})$ and $\mathbf{S}$ is SD controllable for $\mathbf{G}$ by assumption, we can conclude by **Point ii** in Definition 3.2.2 that

$$\tau \in \text{Elig}_{L(\mathbf{S})}(s) \iff \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$$

As we have $s\tau \in L(\mathbf{S})$ by assumption, we thus have

$$\text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \tag{8}$$

$$\Longrightarrow \text{Elig}_{L(\mathbf{S})}(s) \cap \Sigma_{hib} = \emptyset \qquad \text{as } \mathbf{G} \text{ is complete for } \mathbf{S} \tag{9}$$

Essentially, **G** complete for **S** means that if a prohibitable event was accepted by **S**, it must also be accepted by **G**, thus in $L(\mathbf{S}) \cap L(\mathbf{G})$. Thus, the only way there could be no eligible prohibitable events in both, is if there are none in $L(\mathbf{S})$, otherwise we would have a contradiction.

We next note that τ is initially added to $V(\mathbf{S})$ at **line 2** of Algorithm 5.1, thus we would have $\tau \in V(s)$ unless it is removed at **line 11** or **line 23**. Now, it is possible that s will never be added to *Pend* if s is a sampled string, or that it will never be processed in the **for-loop** from **line 20** to **line 27** if s is not a sampled string. If that was the case, $V(s)$ would have the default value and we have $\tau \in V(s)$ as required. We can thus, without any loss of generality, assume that s is added to *Pend* if $s \in L_{samp}$, or s is processed by the **for-loop** from **line 20** to **line 27** if $s \notin L_{samp}$.

It is thus sufficient to show that the tick event is not removed at **line 11** when $s \in L_{samp}$ or at **line 23** when $s \notin L_{samp}$. The two situations are discussed individually below.

(ii.a) $s \in L_{samp}$

If $s \in L_{samp}$, τ could only be removed at **line 11**. To show that it is not, it is sufficient to show that $\Sigma_V = \emptyset$.

As we know from **(ii.a)** in the proof of part 1, Algorithm 5.1 will always associate with s in *Pend*, the state $\mathbf{q}$ in the controller with $\mathbf{q} = \Lambda(x)$ where $x = \xi(x_o, s) \in X_{samp}$. Also, we will have $\Sigma_V = \text{Elig}_{L(\mathbf{S})}(s) \cap \Sigma_{hib}$

From (9) we know $\text{Elig}_{L(\mathbf{S})}(s) \cap \Sigma_{hib} = \emptyset$, thus $\Sigma_V = \emptyset$, as required.

(ii.b) $s \notin L_{samp}$

As $s \notin L_{samp}$, we know: $(\exists t \in L_{samp})(\exists \hat{t} \in L_{conc}) t < s < t\hat{t}$.

Also, we know: $(\exists i \in \{1, 2, ..\})(\exists \sigma_1, \sigma_2, .., \sigma_i \in \Sigma_{act} \subset \Sigma) t\sigma_1\sigma_2..\sigma_i = s$

As s is being processed by the **for-loop** from **line 20** to **line 27**, by assumption we have $t\hat{t} \in L(\mathbf{S}) \cap L(\mathbf{G})$.

Examining from **line 22** to **line 26** of Algorithm 5.1, we see that to show τ is not removed, it is sufficient to show that when s is processed, $\Sigma_{temp} = \emptyset$

From the logic of Algorithm 5.1, we see that initially $\Sigma_{temp} = \Sigma_V$, and $\Sigma_{temp} = \Sigma_V - \{\sigma_1, \sigma_2, .., \sigma_i\}$ by the time s is evaluated.

As we know from the logic of **(ii.a)** in part 1, Algorithm 5.1 will pair string t in *Pend* with state $\mathbf{q}$ from the controller, where $\mathbf{q} = \Lambda(x)$ and $x = \xi(x_o, t) \in X_{samp}$. Also,

$$\Sigma_V = \text{Elig}_{L(\mathbf{S})}(t) \cap \Sigma_{hib}$$
$$\Longrightarrow \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(t) \cap \Sigma_{hib} \subseteq \Sigma_V$$

We will now show that $\Sigma_V \subseteq \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(t) \cap \Sigma_{hib}$, and thus $\Sigma_V = \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(t) \cap \Sigma_{hib}$

Let $\sigma \in \Sigma_V = \text{Elig}_{L(\mathbf{S})}(t) \cap \Sigma_{hib}$

As σ is prohibitable, we immediately know $\sigma \in \text{Elig}_{L(\mathbf{G})}(t)$ as $\mathbf{G}$ is complete for $\mathbf{S}$, which implies $\sigma \in \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(t) \cap \Sigma_{hib}$.

$$\Rightarrow \Sigma_V = \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(t) \cap \Sigma_{hib} \tag{10}$$

As $t < s < t\hat{t}$, and $\text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$ by (8), it thus follows that all prohibitable events that were possible at t, are no longer possible at s in $\mathbf{meet}(\mathbf{G}, \mathbf{S})$.

As $\mathbf{S}$ is SD controllable for $\mathbf{G}$, we can apply **Point iii.1** of Definition 3.2.2, and conclude

$$(\text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cup \text{Occu}(\sigma_1 \sigma_2 .. \sigma_i)) \cap \Sigma_{hib} = \text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(t) \cap \Sigma_{hib}$$

$\Rightarrow (\text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cup \text{Occu}(\sigma_1 \sigma_2 .. \sigma_i)) \cap \Sigma_{hib} = \Sigma_V$ by (10).

As $\text{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) = \emptyset$, by (8) we have

$$\text{Occu}(\sigma_1 \sigma_2 .. \sigma_i) \cap \Sigma_{hib} = \Sigma_v$$

This means, after string $t\sigma_1\sigma_2..\sigma_i$ has occurred, every event in Σ_V has occurred at least once in $\sigma_1\sigma_2..\sigma_i$. Thus, by the time s is evaluated,

$$\Sigma_{temp} = \Sigma_V - \{\sigma_1, \sigma_2, .., \sigma_i\} = \emptyset$$

This means that for s, **line 25** is executed in Algorithm 5.1, not **line 23**, so tick is not removed from $V(s)$. Thus $\tau \in V(s)$, as required.

Part **(ii.b)** complete.

By **(ii.a)** and **(ii.b)** we have shown $s\sigma_{k+1} \in V(s)$ and thus

$$s\sigma_{k+1} \in L(V/\mathbf{G})$$

(iii) $\sigma_{k+1} \in \Sigma_{hib}$

We thus have by assumption

$$\sigma_{k+1} \in \mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib}$$

Examining Algorithm 5.1, we see no prohibitable event is added to $V(s)$ at **line 2**. This means, σ_{k+1} could only be added at **line 11** if $s \in L_{samp}$, or at **line 23** or **line 25**, if $s \notin L_{samp}$. We thus have two cases to examine.

(iii.a) $s \in L_{samp}$

First, we have to show that s will be added to *Pend*, or it will never get to **line 11**.

As we have $s \in L(\mathbf{S}) \cap L(\mathbf{G}) \cap L(V/\mathbf{G})$ by assumption, we can apply the same logic that we used in **(ii.c)** in part 1, to show that s will always be added to *Pend*.

We next note that from the logic of **(ii.a)** in part 1, Algorithm 5.1 will always associate with s in *Pend* the state $\mathbf{q}$ in the controller with $\mathbf{q} = \Lambda(x)$, where $x = \xi(x_o, s) \in X_{samp}$. Also we will have

$$\Sigma_V = \mathrm{Elig}_{L(\mathbf{S})}(s) \cap \Sigma_{hib}$$

As $\sigma_{k+1} \in \mathrm{Elig}_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib}$, we thus have $\sigma_{k+1} \in \Sigma_V$. This means that the condition at **line 10** of Algorithm 5.1 is satisfied, and thus $V(s) \leftarrow (V(s) \cup \Sigma_V) - \{\tau\}$.

Therefore, $\sigma_{k+1} \in V(s)$ as required.

(iii.b) $s \notin L_{samp}$

First, we need to show that we will reach **line 23** or **line 25** for s, or s could only be assigned the default value at **line 2**.

As we have $s \in L(\mathbf{S}) \cap L(\mathbf{G}) \cap L(V/\mathbf{G})$ by assumption, we can apply the logic of **(ii.d)** in part 1, and conclude

$$(\exists t \in L_{samp})(\exists \hat{t} \in L_{conc})t < s < t\hat{t}$$

such that t will be added to *Pend* and associated with state $\mathbf{q}$ in the controller with $\mathbf{q} = \Lambda(x)$, where $x = \xi(x_o, t)$. Also, $\Sigma_V = \mathrm{Elig}_{L(\mathbf{S})}(t) \cap \Sigma_{hib}$ (11)

Also, $\hat{t}$ is such that the condition at **line 14** will be satisfied, and thus **line 23** or **line 25** will be reached.

We also note

$$(\exists i \in \{1, 2, ..\})(\exists \sigma_1, \sigma_2, .., \sigma_i \in \Sigma_{act} \subset \Sigma)t\sigma_1\sigma_2..\sigma_i = s$$

Now, since either **line 23** or **line 25** will be executed, Σ_V will be added to $V(s)$. It is thus sufficient to show $\sigma_{k+1} \in \Sigma_V$.

Since by assumption $\sigma_{k+1} \in \Sigma_{hib}$, and $\sigma_{k+1} \in \mathrm{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s)$, it follows that

$$\sigma_{k+1} \in \mathrm{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} \tag{12}$$

As $\mathbf{S}$ is SD controllable for $\mathbf{G}$, we can apply **Point iii.1** of Definition 3.2.2, and conclude

$$(\mathrm{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cup \mathrm{Occu}(\sigma_1\sigma_2..\sigma_i)) \cap \Sigma_{hib} = \mathrm{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(t) \cap \Sigma_{hib}$$

$\Rightarrow \sigma_{k+1} \in (\mathrm{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cup \mathrm{Occu}(\sigma_1\sigma_2..\sigma_i)) \cap \Sigma_{hib}$, by (12).

$\Rightarrow \sigma_{k+1} \in \mathrm{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(t) \cap \Sigma_{hib}$

$\Rightarrow \sigma_{k+1} \in \mathrm{Elig}_{L(\mathbf{S})}(t) \cap \Sigma_{hib} = \Sigma_V$, by (11)

Thus $\sigma_{k+1} \in V(s)$, as required.

By part **(iii.a)** and **(iii.b)**, we have $\sigma_{k+1} \in V(s)$, as required.

Part **iii** complete.

By cases **i-iii**, we have $\sigma_{k+1} \in V(s)$. We thus have $s\sigma_{k+1} \in L(V/\mathbf{G})$, thus our inductive step is complete.

Thus by our **base case** and our **inductive step**, we have shown $s \in L(V/\mathbf{G})$ for arbitrary $s \in L(S) \cap L(G)$.

Part **2** is complete.

We have shown **1** and **2**, thus we have shown $L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$. □

We are now ready to show that the V we constructed in Algorithm 5.1 with the given system requirements, is indeed a TDES supervisory control.

Proposition 5.2. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \delta, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite statespaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, have proper time and $\mathbf{S}$-singular prohibitable behavior, let $\mathbf{meet}(\mathbf{G}, \mathbf{S})$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$ using the translation method described in Section 4.2.3, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 5.1. Then map V is a TDES supervisory control for $\mathbf{G}$.

Proof. Let $s \in L(\mathbf{G})$.

To show that V satisfies Definition 5.1.1, we need to show 1) $V(s) \supseteq \Sigma_u$ and 2) $[(\tau \in \mathrm{Elig}_{L(\mathbf{G})}(s)) \wedge (V(s) \cap \mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset)] \implies \tau \in V(s)$.

1) Show $V(s) \supseteq \Sigma_u$

   This is automatic as $V(s) = \Sigma_u \cup \{\tau\}$ is set at **line 2** of Algorithm 5.1, and as $\tau \notin \Sigma_u$, no $\sigma \in \Sigma_u$ is ever removed from $V(s)$.

2) Show $[(\tau \in \mathrm{Elig}_{L(\mathbf{G})}(s)) \wedge (V(s) \cap \mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset)] \implies \tau \in V(s)$

   Assume $\tau \in \mathrm{Elig}_{L(\mathbf{G})}(s)$ and $V(s) \cap \mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

We will now show this implies $\tau \in V(s)$

We first note that as the assumptions of Theorem 5.1 are satisfied, we can conclude $L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$

We next note that τ is initially added to $V(s)$ at **line 2** of Algorithm 5.1. If s is not processed again at **line 11**, **line 23** and **line 25**, we have $\tau \in V(s)$

As we initializes *Pend* to $(\epsilon, \mathbf{q}_{res})$, and we see that only strings in $L(\mathbf{S}) \cap L(\mathbf{G})$ will ever be added to *Pend* or processed at **line 23** or **line 25** (this can be seen by **line 13** and **line 14**). This means if $s \notin L(\mathbf{S}) \cap L(\mathbf{G})$, we get the default value from **line 2** and thus have $\tau \in V(s)$.

We only need to still consider $s \in L(\mathbf{S}) \cap L(\mathbf{G})$. (1)

We will now show that $\mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$.

By definition of $L(V/\mathbf{G})$ (Definition 5.1.4), for $\sigma \in \Sigma_{hib}$, we would only have $s\sigma \in L(V/\mathbf{G})$ if $s \in L(V/\mathbf{G})$, $\sigma \in V(s)$ and $s\sigma \in L(\mathbf{G})$.

We have $s \in L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$ from (1), so for $s\sigma \in L(V/\mathbf{G})$, we would need $\sigma \in V(s) \cap \mathrm{Elig}_{L(\mathbf{G})}(s)$. However, we have $V(s) \cap \mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$ by assumption, thus

$$\begin{aligned} &(\forall \sigma \in \Sigma_{hib}) s\sigma \notin L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G}) \\ \Longrightarrow\ &\mathrm{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \end{aligned}$$

As $\mathbf{S}$ is SD controllable for $\mathbf{G}$, we can conclude by **Point ii** of Definition 3.2.2 that, $\tau \in \mathrm{Elig}_{L(\mathbf{S})}(s)$. Since by assumption, we have $\tau \in \mathrm{Elig}_{L(\mathbf{G})}(s)$, we have

$$\begin{aligned} &\tau \in \mathrm{Elig}_{L(\mathbf{S}) \cap L(\mathbf{G})}(s) = \mathrm{Elig}_{L(V/\mathbf{G})}(s) \\ \Longrightarrow\ &s\tau \in L(V/\mathbf{G}) \end{aligned}$$

As $s \in L(V/\mathbf{G})$, $s\tau \in L(\mathbf{G})$ and $s\tau \in L(V/\mathbf{G})$, it follows from Definition 5.1.4 that $\tau \in V(s)$, as required.

From points 1) and 2), we thus conclude that V is a TDES supervisory control. □

We have now captured the enablement and forcing behavior of our controller as a map V, and shown that if $\mathbf{G}$ and $\mathbf{S}$ have the appropriate properties, V is indeed a TDES supervisory control. We have also shown that the closed behavior of $\mathbf{G}$ under the control of V, $L(V/\mathbf{G})$, is exactly that of the closed behavior of the $\mathbf{meet}(\mathbf{G},\mathbf{S})$, namely $L(\mathbf{S}) \cap L(\mathbf{G})$.This means that the behavior we get when our controller acts on our plant is what we expect, at least with respect to enablement and forcing behavior. Of course, this is assuming that none of the time delay issues we discussed in Section 3.3 occur.

We will now show that V is nonblocking for $\mathbf{G}$ if and only if the meet of $\mathbf{G}$ and $\mathbf{S}$ are nonblocking.

Proposition 5.3. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \delta, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite statespaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, have proper time and $\mathbf{S}$-singular prohibitable behavior, let $\mathbf{meet}(\mathbf{G},\mathbf{S})$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$ using the translation method described in Section 4.2.3, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 5.1. Then V is non-blocking for $\mathbf{G}$ if and only if $\mathbf{meet}(\mathbf{G},\mathbf{S})$ is non-blocking.

Proof. To show this, it is sufficient to show that $L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$, and $L_m(V/\mathbf{G}) = L_m(\mathbf{S}) \cap L_m(\mathbf{G})$.

As the assumptions of Theorem 5.1 are satisfied, we can conclude $L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$.

We next note that by Definition 5.1.6, we have

$$\begin{aligned} L_m(V/\mathbf{G}) =& L(V/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \\ =& L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G}) && \text{after substitution for } L(V/\mathbf{G}) \\ =& L_m(\mathbf{S}) \cap L_m(\mathbf{G}) && \text{as } L_m(\mathbf{G}) \subseteq L(\mathbf{G}) \text{ and } L_m(\mathbf{S}) \subseteq L(\mathbf{S}) \end{aligned}$$

as required. □

5.4 Concurrent Supervisory Control Equivalent

In general, the order that events occur in the physical plant during a given sampling period, are that dictated by the plant model, and are allowed by the enablement and forcing behavior of the plant's SD controller. However, in practice time delay restrictions and the particular implementation of our controller might mean that all concurrent strings that should be possible in a given sampling period according to our plant model, may not actually be possible in practice.

For instance, we may be expecting that we could either get string $\alpha\beta\tau$ or $\beta\alpha\tau$ ($\alpha, \beta \in \Sigma_{hib}$), yet it may be that only string $\alpha\beta\tau$ will ever occur due to time delay or the specific implementation of our controller. With respect to time delay, it could be possible, for example, that α always reaches our controller's inputs first. With respect to implementation, our controller might have to execute the events sequentially and always chooses to first do an α then a β as it must choose an execution order (people typically would not design a controller that randomly chooses an execution order each time). Another possibility is that the controller could start α and β tasks at about the same time, but β always takes longer (in this particular implementation of our controller) to occur.

This could be a problem if, for example, only string $\beta\alpha\tau$ lead back to a marked state. In such a case, our TDES system would be nonblocking and controllable, but our implementation would block. We want to ensure that if our TDES system is nonblocking, and in our actual controlled system where we have a set of possible concurrent strings with the same occurrence image that could occur (according to our TDES model) in a given sampling period, if at least one of these strings can actually occur, our implementation would still be nonblocking. In other words, we wish our system to be robust with respect to nonblocking and this uncertainty.

We will now show that the conditions that we have developed will in fact guarantee this. We will frame our argument in terms of supervisory controls. Given a TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ and supervisor control V for $\mathbf{G}$, we want to be able define a supervisor control V' such that if V allows a set of concurrent strings with the same occurrence image to occur in $\mathbf{G}$ in a given sampling instance, V' will always allow at least one of them to occur, but not necessarily all of them. We want to make sure that as long as our actual controlled system exhibits the behavior of at least one of these V', it will still be nonblocking. Note that we are only modeling variations in which prohibitable events are enabled and possibly forced. We capture this notation in the following definition.

Definition 5.4.1. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a TDES plant and let V and V' be supervisory controls for $\mathbf{G}$. We say V' is *concurrent supervisory control equivalent* to V if

1. $(\forall s \in L(\mathbf{G}))V'(s) \subseteq V(s)$

2. $(\forall s \in L(V'/\mathbf{G}) \cap L_{samp})(\forall s' \in L_{conc})ss' \in L(V/\mathbf{G})$
   $\implies (\exists s'' \in L_{conc})ss'' \in L(V'/\mathbf{G}) \cap \mathrm{Occu}(s') = \mathrm{Occu}(s'')$

By point 1 in the definition above, we require each event that $V'(s)$ allows is also allowed by $V(s)$, so that $L(V'/\mathbf{G})$ does not include unwanted behavior.

By point 2, we require that if $V'/\mathbf{G}$ accepts sampled string s, and $V/\mathbf{G}$ accepts concurrent string s' after it accepts string s, then $V'/\mathbf{G}$ must accept a concurrent string s'' that has the same occurrence image as s'. We use the the term *concurrent equivalent* because strings s' and s'' in the definition could both occur in the same sampling period and would be indistinguishable to an SD controller.

Figure 5.3 shows an example for the concurrent supervisory control equivalence definition. Here we see that for $V/\mathbf{G}$, we have two paths with the same occurrence image. For $V'/\mathbf{G}$, only one of the two paths are still possible, but that is enough to satisfy the definition.

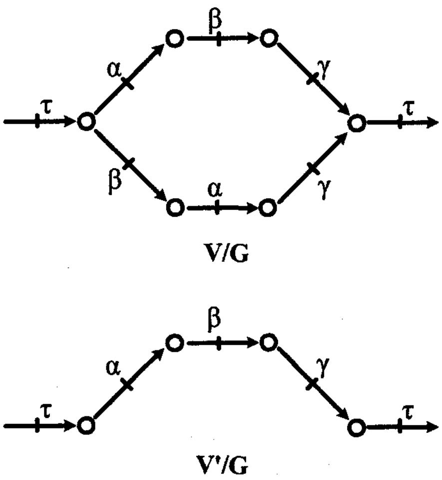


Figure 5.3: An Example for Concurrent Supervisory Control Equivalence

Proposition 5.4. For TDES plant $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$, let V and V' be supervisory controls for $\mathbf{G}$. If V' is concurrent supervisory control equivalent to V, then

$$L(V'/\mathbf{G}) \subseteq L(V/\mathbf{G})$$

Proof. Let V and V' be supervisory controls for $\mathbf{G}$.

Assume V' is concurrent supervisory control equivalent to V.

Must show

$$(\forall s \in L(V'/\mathbf{G}))s \in L(V/\mathbf{G})$$

We will show this using proof by induction.

base case Let $s = \epsilon$

This automatically implies $\epsilon \in L(V/\mathbf{G})$ by definition of $L(V/\mathbf{G})$.

inductive step Let $s \in L(V'/\mathbf{G}) \cap L(V/\mathbf{G})$. Let $\sigma \in \Sigma$ such that $s\sigma \in L(V'/\mathbf{G})$.

Need to show implies $s\sigma \in L(V/\mathbf{G})$.

As we already have $s \in L(V/\mathbf{G})$ by assumption, it is sufficient to show

1. $\sigma \in V(s)$
   As $s\sigma \in L(V'/\mathbf{G})$, we have by definition of $L(V'/\mathbf{G})$ that $\sigma \in V'(s)$ and $s \in L(\mathbf{G})$. This implies $\sigma \in V(s)$ by point 1 of Definition 5.4.1.
2. $s\sigma \in L(\mathbf{G})$
   By assumption, we have $s\sigma \in L(V'/\mathbf{G})$, which implies $s\sigma \in L(\mathbf{G})$ by definition of $L(V'/\mathbf{G})$.

Thus by definition 5.1.4 of supervisory control, we have $s\sigma \in L(V/\mathbf{G})$.

Thus by our base case and inductive step, we conclude

$$(\forall s \in L(V'/\mathbf{G}))s \in L(V/\mathbf{G})$$

which implies

$$L(V'/\mathbf{G}) \subseteq L(V/\mathbf{G})$$

□

We will now show that if V is the TDES supervisor control we constructed with Algorithm 5.1 and V' is a TDES supervisor control that is concurrent control equivalent to V, then V nonblocking for $\mathbf{G}$ implies that V' is also nonblocking for $\mathbf{G}$ (per Definition 5.1.6).

Theorem 5.2. For plant $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$, and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, and have proper time and $\mathbf{S}$-singular prohibitable behavior, let $\mathbf{meet}(\mathbf{G}, \mathbf{S})$ be ALF, let $\mathbf{C}$ be the SD controller that is constructed from $\mathbf{S}$ using the translation method described in Section 4.2.3, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 5.1. Let V' be a supervisor control for $\mathbf{G}$. If V is nonblocking for $\mathbf{G}$ and V' is concurrent supervisory control equivalent to V, then V' is also nonblocking for $\mathbf{G}$.

Proof. Assume the initial conditions for the proposition, including that V is nonblocking and V' is concurrent supervisory control equivalent to V.

We must show this implies

$$L(V'/\mathbf{G}) = \overline{L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})}$$

It is sufficient to show points 1 and 2 as follows.

1. $L(V'/\mathbf{G}) \supseteq \overline{L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})}$

   Let $s \in \overline{L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})}$.

   Must show implies $s \in L(V'/\mathbf{G})$.

   Since $s \in \overline{L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})}$, there exists $s'' \in \Sigma^*$ such that

$$\begin{aligned} & ss'' \in L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \\ \Longrightarrow\ & ss'' \in L(V'/\mathbf{G}) \\ \Longrightarrow\ & s \in \overline{L(V'/\mathbf{G})} \\ \Longrightarrow\ & s \in L(V'/\mathbf{G}) \qquad \text{as } L(V'/\mathbf{G}) \text{ is prefix closed, by Definition 5.1.4} \end{aligned}$$

2. $L(V'/\mathbf{G}) \subseteq \overline{L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})}$

Let $s \in L(V'/\mathbf{G})$. (1)

Must show $s \in \overline{L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})}$.

Sufficient to show

$$(\exists s'' \in \Sigma^*) ss'' \in L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})$$

For the case that

$$s \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})$$

We have $ss'' \in L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})$, with $s'' = \epsilon$.

We now examine the case

$$s \notin L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \tag{2}$$

We first note that the assumptions of Theorem 5.1 have been met, so we can conclude that

$$L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G}) \tag{3}$$

Also, as $L_m(\mathbf{G}) \subseteq L(\mathbf{G})$ and $L_m(\mathbf{S}) \subseteq L(\mathbf{S})$, it follows that

$$L_m(\mathbf{G}) \cap L_m(\mathbf{S}) \subseteq L(V/\mathbf{G}) \tag{4}$$

We have two sub-cases for s: 1) $s \in L_{samp}$ and 2) $s \notin L_{samp}$.

2.1 $s \in L_{samp}$ (5)

By Proposition 5.4 we have $L(V'/\mathbf{G}) \subseteq L(V/\mathbf{G})$, we thus have

$$s \in L(V/\mathbf{G}) \qquad \text{by (1)}$$

Since V is nonblocking for $\mathbf{G}$, we have

$$\begin{aligned} s \in L(V/\mathbf{G}) &\implies (\exists s' \in \Sigma^*)\, ss' \in L(V/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \\ &\implies (\exists s' \in \Sigma^*)\, (ss' \in L(V/\mathbf{G})) \wedge (ss' \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})) \end{aligned}$$

Let $s' \in \Sigma^*$ such that

$$(ss' \in L(V/\mathbf{G})) \wedge (ss' \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})) \tag{6}$$

As $\mathbf{S}$ is SD controllable for $\mathbf{G}$, we have by point **iv** in Definition 3.2.2 that

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

We can thus divide s' into consecutive strings $s'_1, s'_2, .., s'_n \in L_{conc}$ such that

$$s' := s'_1 s'_2 .. s'_n \tag{7}$$

We note that $n \geq 1$ as $s \notin L_m(\mathbf{S}) \cap L_m(\mathbf{G})$, by (2). We thus have

$$ss'_1 s'_2 .. s'_n \in L(V/\mathbf{G}) \tag{8}$$

We will now use $s'_1, s'_2, .., s'_n$ to construct $s''_1, s''_2, .., s''_n \in L_{conc}$ such that $ss''_1 s''_2 .. s''_n \in L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})$

We will use a proof by induction to show that, for all $k \in \{2, 3, .., n\}$

$$\begin{aligned}
&(\exists s''_1, s''_2, .., s''_{k-1} \in L_{conc})\\
&\qquad [ss''_1 s''_2 .. s''_{k-1} s'_k .. s'_n \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})] \wedge [ss''_1 s''_2 .. s''_{k-1} \in L(V'/\mathbf{G})]\\
&\qquad\qquad \Longrightarrow\\
&(\exists s''_k \in L_{conc})\\
&\qquad [ss''_1 s''_2 .. s''_k s'_{k+1} .. s'_n \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})] \wedge [ss''_1 s''_2 .. s''_k \in L(V'/\mathbf{G})]
\end{aligned} \tag{9}$$

base case Show: $(\exists s''_1 \in L_{conc})$

$$[ss''_1 s'_2 .. s'_n \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})] \wedge [ss''_1 \in L(V'/\mathbf{G})]$$

From (6), (7) and (8) we have $s'_1, s'_2, .., s'_n \in L_{conc}$ and

$$\begin{aligned}
&ss'_1 s'_2 .. s'_n \in L(V/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})\\
\Longrightarrow\ &ss'_1 \in L(V/\mathbf{G}) \qquad\qquad \text{as } L(V/\mathbf{G}) \text{ is prefix closed}
\end{aligned}$$

By (1) and (5), we have $s \in L(V'/\mathbf{G}) \cap L_{samp}$

Putting this together we see we have

$$(s \in L(V'/\mathbf{G}) \cap L_{samp}) \wedge (s'_1 \in L_{conc}) \wedge (ss'_1 \in L(V/\mathbf{G}))$$

As V' is concurrent supervisory control equivalent to V, we can apply Definition 5.4.1 and conclude

$$(\exists s_1'' \in L_{conc})(ss_1'' \in L(V'/\mathbf{G})) \wedge (\text{Occu}(s_1') = \text{Occu}(s_1'')) \tag{10}$$

As $L(V'/\mathbf{G}) \subseteq L(V/\mathbf{G})$ by Proposition 5.4, we have $ss_1'' \in L(V/\mathbf{G})$

As $s, ss_1', ss_1'' \in L(V/\mathbf{G})$, we have

$$s, ss_1', ss_1'' \in L(\mathbf{S}) \cap L(\mathbf{G}) \qquad \text{by (3)}$$

As $s \in L_{samp}$ and $\mathbf{S}$ is SD controllable for $\mathbf{G}$, we can apply **Point iii.2** and conclude: $ss_1' \equiv_{L_m(\mathbf{S}) \cap L_m(\mathbf{G})} ss_1''$.

As $ss_1's_2'..s_n' \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})$ from (6) and (7), we have $ss_1''s_2'..s_n' \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})$ as ss_1' and ss_1'' are Nerode equivalent mod $L_m(\mathbf{S}) \cap L_m(\mathbf{G})$.

We have thus shown

$$(\exists s_1'' \in L_{conc})[ss_1''s_2'..s_n' \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})] \wedge [ss_1'' \in L(V'/\mathbf{G})]$$

Base case complete.

inductive step Let $k \in \{2, 3, .., n\}$.

Assume:

$(\exists s_1'', s_2'', .., s_{k-1}'' \in L_{conc})$
$[ss_1''s_2''..s_{k-1}''s_k'..s_n' \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})] \wedge [ss_1''s_2''..s_{k-1}'' \in L(V'/\mathbf{G})]$

We will show this implies condition (9) is satisfied for this k.

We first note that as $s_1'', s_2'', .., s_{k-1}'' \in L_{conc}$, $ss_1''s_2''..s_{k-1}'' \in L_{samp}$.

As $L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \subseteq L(V/\mathbf{G})$ by (4), we have $ss_1''s_2''..s_{k-1}''s_k'..s_n' \in L(V/\mathbf{G})$.

As $L(V/\mathbf{G})$ is prefix closed, we have $ss_1''..s_{k-1}''s_k' \in L(V/\mathbf{G})$. Also, $s_k' \in L_{conc}$ by (7).

We thus have:

$$\begin{aligned}&(ss''_1s''_2..s''_{k-1} \in L(V'/\mathbf{G}) \cap L_{samp})\\ &\wedge(s'_k \in L_{conc})\\ &\wedge(ss''_1s''_2..s''_{k-1}s'_k \in L(V/\mathbf{G}))\end{aligned}$$

As V' is concurrent supervisory control equivalent to V, we can thus apply point 2 of Definition 5.4.1 and conclude

$$(\exists s''_k \in L_{conc})ss''_1s''_2..s''_k \in L(V'/\mathbf{G}) \text{ and } \mathrm{Occu}(s'_k) = \mathrm{Occu}(s''_k)$$

As $L(V'/\mathbf{G}) \subseteq L(V/\mathbf{G})$, we have $ss''_1s''_2..s''_k \in L(V/\mathbf{G})$

We thus have

$$\begin{aligned}&ss''_1s''_2..s''_{k-1} \in L(V/\mathbf{G}) \cap L_{samp}\\ \Longrightarrow\ &ss''_1s''_2..s''_{k-1} \in L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp} \qquad \text{by (3)}\end{aligned}$$

and

$$ss''_1s''_2..s''_{k-1}s'_k,\ ss''_1s''_2..s''_k \in L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$$

As $\mathbf{S}$ is SD controllable for $\mathbf{G}$, we can apply **Point iii.2** of Definition 3.2.2 and conclude

$$ss''_1s''_2..s''_{k-1}s'_k \equiv_{L_m(\mathbf{S})\cap L_m(\mathbf{G})} ss''_1s''_2..s''_k$$

As $ss''_1s''_2..s''_{k-1}s'_k..s'_n \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})$ by assumption, we thus have $ss''_1s''_2..s''_ks'_{k+1}..s'_n \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})$ as $ss''_1s''_2..s''_{k-1}s'_k$ and $ss''_1s''_2..s''_k$ are Nerode equivalent mod $L_m(\mathbf{S}) \cap L_m(\mathbf{G})$.

We have thus shown

$$\begin{aligned}&(\exists s''_k \in L_{conc})\\ &\quad [ss''_1s''_2..s''_ks'_{k+1}..s'_n \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})] \text{ and } [ss''_1s''_2..s''_k \in L(V'/\mathbf{G})]\end{aligned}$$

Inductive step complete.

Combining our base case and inductive step, we can take $k = n$, and conclude

$$(\exists s_1'', s_2'', .., s_n'' \in L_{conc})\, ss_1''s_2''..s_n'' \in L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \cap L(V'/\mathbf{G})$$

We thus take $s'' = s_1''s_2''..s_n''$ and Case 2.1 is complete.

2.2 $s \notin L_{samp}$

As we want to reuse the result from **2.1** for this part, we first need to extend s to a string in $L(V'/\mathbf{G}) \cap L_{samp}$.

As $\mathbf{G}$ and $\mathbf{S}$ have finite statespaces, and $\mathbf{meet}(\mathbf{G}, \mathbf{S})$ is activity loop free, it follows that $\mathbf{meet}(\mathbf{G}, \mathbf{S})$ will accept at most a finite number of non-tick events, before no more non-tick events can occur. Note $L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$ by (3).

This means that at the state reached by s in $\mathbf{meet}(\mathbf{G}, \mathbf{S})$, either there are no activity events possible, or after at most a finite number of activity events occur, we will be in a state where no activity events are possible.

The reason is that we have a finite number of states in $\mathbf{meet}(\mathbf{G}, \mathbf{S})$, thus after at most a finite number of non-tick transitions, we will have either reached a state where no activity events are possible, or we will have visited each state once as $\mathbf{meet}(\mathbf{G}, \mathbf{S})$ is ALF. If we have visited each state once, we can't have another activity event possible, or it would create a loop, violating the assumption that $\mathbf{meet}(\mathbf{G}, \mathbf{S})$ is ALF.

As $L(V'/\mathbf{G}) \subseteq L(V/\mathbf{G})$ by Proposition 5.4, it thus follows

$$(\exists \hat{t} \in \Sigma_{act}^*)\,(s\hat{t} \in L(V'/\mathbf{G})) \wedge (\mathrm{Elig}_{L(V'/\mathbf{G})}(s\hat{t}) \cap \Sigma_{act} = \emptyset)$$

We will now show that: $s\hat{t}\tau \in L(V'/\mathbf{G}) \cap L_{samp}$.

We first note that by definition of $L(V'/\mathbf{G})$,

$$\mathrm{Elig}_{L(V'/\mathbf{G})}(s\hat{t}) = V'(s\hat{t}) \cap \mathrm{Elig}_{L(\mathbf{G})}(s\hat{t})$$

As V' is a TDES supervisory control, we have $V'(s\hat{t}) \supseteq \Sigma_u$. Thus

$$\begin{aligned} & V'(s\hat{t}) \cap \mathrm{Elig}_{L(\mathbf{G})}(s\hat{t}) \cap \Sigma_{act} = \emptyset \\ \Longrightarrow\ & \mathrm{Elig}_{L(\mathbf{G})}(s\hat{t}) \cap \Sigma_u = \emptyset \\ \Longrightarrow\ & \tau \in \mathrm{Elig}_{L(\mathbf{G})}(s\hat{t}) \qquad \text{as } \mathbf{G} \text{ has proper time behavior} \end{aligned}$$

We next note

$$\begin{aligned} & V'(s\hat{t}) \cap \mathrm{Elig}_{L(\mathbf{G})}(s\hat{t}) \cap \Sigma_{act} = \emptyset \\ \Longrightarrow\ & V'(s\hat{t}) \cap \mathrm{Elig}_{L(\mathbf{G})}(s\hat{t}) \cap \Sigma_{hib} = \emptyset \\ \Longrightarrow\ & \tau \in V'(s\hat{t}) \qquad \text{as } V' \text{ is a TDES supervisory control} \end{aligned}$$

Combining the two results, we have $s\hat{t}\tau \in L(V'/\mathbf{G})$.

Taking $t = s\hat{t}\tau$, we first note that if $t \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})$ we can take $s'' = \hat{t}\tau$ and we have

$$ss'' \in L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})$$

and we are done.

We then consider the case $t \notin L_m(\mathbf{S}) \cap L_m(\mathbf{G})$.

As $t = s\hat{t}\tau$, we thus have $t \in L_{samp} \cap L(V'/\mathbf{G})$

We can now apply the logic of part 2.1, but use t instead of s as our starting place.

We can thus conclude

$$(\exists s_1'', s_2'', .., s_n'' \in L_{conc}) t s_1'' s_2'' .. s_n'' \in L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \cap L(V'/\mathbf{G})$$

We thus take $s'' = \hat{t}\tau s_1'' s_2'' .. s_n''$ and part 2.2 is complete.

By both part 2.1 and 2.2, we have constructed a string $s'' \in \Sigma^*$, where

$$\begin{aligned} & ss'' \in L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \cap L(V'/\mathbf{G}) \\ \Longrightarrow\ & s \in \overline{L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \cap L(V'/\mathbf{G})} \end{aligned}$$

as required.

Part 2 is complete.

By part 1 and 2, we thus have

$$L(V'/\mathbf{G}) = \overline{L(V'/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})}$$

i.e. V' is non-blocking for $\mathbf{G}$. $\square$

Chapter 6

Symbolic Verification for SD System

In this section, we will present algorithms to verify nonblocking, untimed controllability, ALF, proper time behavior, plant completeness, S-singular prohibitable behavior, and SD controllability. To ensure scalability, we will develop predicate based algorithms that are built upon the work of Song [26]. We will first introduce predicates, and then discuss how we can use them to verify properties of interest. We then present our new algorithms, as well as a few that we will re-use from [26].

All the data representations, computations and verifications are based on ordered binary decision diagram [8]. For simplicity, we will just use the term BDD. In the appendix, you will find the source code for the software tool we developed to implement our algorithms. The code is based on the software developed by Song [26], and uses his BDD variable ordering algorithm. The code also uses the *BuDDy* library [13] which is a C++ library that implements standard BDD structures and operations.

6.1 Predicates and Predicate Transformers

6.1.1 State Predicates

From now on, we will use '$\equiv$' to mean logical equivalence between state predicates. We will also use 'T' and 'F' for logical true and false.

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a TDES.

Definition 6.1.1. A *predicate* P defined on state set Q is a function

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

identified by the corresponding state subset

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

We identify state predicate *true* by Q, state predicate *false* by $\emptyset$, and state predicate P_m by Q_m.

We write $q \models P$ if $q \in Q_P$ and say "*q satisfies P*" or "*P includes q*". Thus we have

$$q \models P \iff P(q) = T$$

We write *Pred*(Q) for the set of all predicates defined on Q; thus *Pred*(Q) is identified by Pwr(Q). 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.

Definition 6.1.2. For $P, P_1, P_2 \in Pred(Q)$ and $q \in Q$, we can build boolean expressions by using the following predicate operations.

$$\begin{aligned}
(\neg P)(q) = T &\iff P(q) = F \\
(P_1 \wedge P_2)(q) = T &\iff P_1(q) = T \text{ and } P_2(q) = T \\
(P_1 \vee P_2)(q) = T &\iff P_1(q) = T \text{ or } P_2(q) = T \\
(P_1 - P_2)(q) = T &\iff P_1(q) = T \text{ and } P_2(q) = F
\end{aligned}$$

Definition 6.1.3. The partial order relation $\preceq$ over *Pred*(Q) is defined as

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

It is obvious that $Q_{P_1} \subseteq Q_{P_2} \iff P_1 \preceq P_2$. In this case,

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

Definition 6.1.4. Let $P_1, P_2 \in Pred(Q)$ for some state set Q. P_1 is a *subpredicate* of P_2 if $P_1 \preceq P_2$. We say P_1 is *stronger* than P_2 and P_2 is *weaker* than P_1.

We write *Sub*(P) to be the set of all the subpredicates of $P \in Pred(Q)$ such that *Sub*(P) is identified by Pwr(Q_P).

6.1.2 Predicate Transformers

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a TDES and $P \in Pred(Q)$. A *predicate transformer* is a function $f : Pred(Q) \to Pred(Q)$. Here we introduce several basic predicate transformers from [26] which are required by the following sections.

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

  The *reachability predicate* $R(\mathbf{G}, P)$ is true for exactly the states in $\mathbf{G}$ that can be reached from q_o by states satisfying P. It is inductively defined as follows.

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

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

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

  The *coreachability predicate* $CR(\mathbf{G}, P)$ is true for exactly the states in $\mathbf{G}$ that can reach a marked state by states satisfying P. It is inductively defined as follows.

  1. $P_m \wedge P \equiv false \implies CR(\mathbf{G}, P) \equiv false$
  2. $q \models P_m \wedge P \implies q \models CR(\mathbf{G}, P)$
  3. $q \models CR(\mathbf{G}, P)$ & $q' \models P$ & $\sigma \in \Sigma$ & $\delta(q', \sigma)!$ & $\delta(q', \sigma) = q \implies q' \models CR(\mathbf{G}, P)$
  4. No other states satisfy $CR(\mathbf{G}, P)$.

  It says that a state $q \models CR(\mathbf{G}, P)$ if and only if there exists a path from q to some marked state in $\mathbf{G}$ and each state in that path satisfies P. To represent the set of all coreachable states in Q, we use $CR(\mathbf{G}, true)$.

- $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$

  Let $P' \in Pred(Q)$ and $\Sigma' \subseteq \Sigma$. Once we fix $\mathbf{G}$, P' and Σ', $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ is then a predicate transformer. The predicate $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ is true for exactly the states in $\mathbf{G}$ that can reach a state in $\mathbf{G}$ satisfying P', by states that satisfy P and by transition with events in Σ'. It is inductively defined as follows.

  1. $P' \wedge P \equiv false \implies \mathcal{CR}(\mathbf{G}, P', \Sigma', P) \equiv false$
  2. $q \models P' \wedge P \implies q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$
  3. $q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ & $q' \models P$ & $\sigma \in \Sigma'$ & $\delta(q', \sigma)!$ & $\delta(q', \sigma) = q$ $\implies q' \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$
  4. No other states satisfy $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$.

  By comparing with definition of coreachablity predicate CR, we have

$$\mathcal{CR}(\mathbf{G}, P_m, \Sigma, P) \equiv CR(\mathbf{G}, P)$$

6.2 Symbolic Representation

For symbolic verification of SD systems, we need to have a representation for states and transitions. We will use the symbolic representation from Song [26], who in turn based his work on Ma [14]. In this section, we only introduce the necessary definitions from this representation that are needed for the computation and verification in the following sections.

6.2.1 State Subsets

Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n$ be the product TDES of component TDES $\mathbf{G}_i$ where $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ for $i = 1, 2, .., n$. For any state $q \in Q$, we have $q = (q_1, q_2, .., q_n)$ where $q_i \in Q_i$.

In later sections we will be evaluating the meet of component TDES for some of the verifications. The only difference between the meet and the product of these TDES is that, the product might contain unreachable states but the meet does not. However, the checking of unreachable states is expensive and therefore the reachability check is performed over the entire system at the end. In addition, since including unreachable states does not effect the closed loop behavior, using the product TDES will not introduce any error.

Definition 6.2.1. For $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n$, let $i = 1, 2, .., n$ and $q_i \in Q_i$. The *state variable* v_i for the i-th component TDES $\mathbf{G}_i$ is a variable of domain Q_i. If v_i has assigned value q_i, then $v_i = q_i$ returns T; otherwise it returns F.

Here we use '=' to if v_i has been assigned value q_i, because '$\equiv$' has been used for logical equivalence between state predicates.

Definition 6.2.2. For $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n$, the *state variable vector* $\mathbf{v}$ is a vector $[v_1, v_2, .., v_n]$ of state variables v_i from each component TDES $\mathbf{G}_i$. For state subset $A \subseteq Q$, we write predicate

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

or P_A if $\mathbf{v}$ is understood.

6.2.2 Transitions

Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n$ be the product TDES of component TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ for $i = 1, 2, .., n$ as defined in previous section.

Definition 6.2.3. For $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n$, let $\sigma \in \Sigma$. A *transition predicate* $N_\sigma : Q \times Q \to \{T, F\}$ identifies all the transitions for σ in $\mathbf{G}$ and is defined as follows.

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

To distinguish between source states and destination states, we need to have two different vectors of state variables, as defined below.

Definition 6.2.4. For $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n$, let $i = 1, 2, .., n$. For each $\mathbf{G}_i$, we have the *normal state variable* v_i (source state) and the *prime state variable* v'_i (destination state), both with domain Q_i. For $\mathbf{G}$, we have the *normal state variable vector* $\mathbf{v} = [v_1, v_2, .., v_n]$ and the *prime state variable vector* $\mathbf{v}' = [v'_1, v'_2, .., v'_n]$.

For each $\sigma \in \Sigma$, we can write the transition predicate for σ, N_σ, as below. Essentially, if we set $\mathbf{v} = q$ and $\mathbf{v}' = q'$ such that $\delta(q, \sigma) = q'$, then $N_\sigma(\mathbf{v}, \mathbf{v}')$ will return T.

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

However, when designing a system with multiple component TDES defined over different event set, such as when we use the synchronous product operator, each component TDES must be selflooped at each state with events that are not in its own event set. This of course makes the transition predicate much more complicated. A new representation to avoid this issue is defined as below. Note that the size of v_σ and v_σ' will always be the same.

Definition 6.2.5. We use the *transition tuple* $(\mathbf{v}_\sigma, \mathbf{v}_\sigma', N_\sigma)$ to represent the transition on σ, where $\mathbf{v}_\sigma = \{v_i \in \mathbf{v} | \sigma \in \Sigma_i\}$, $\mathbf{v}_\sigma' = \{v_i' \in \mathbf{v}' | \sigma \in \Sigma_i\}$ and

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

Although selflooped transitions are not specified in the definition, the selfloop information is still expressed. For those state variables that are not in $\mathbf{v}_\sigma$, we know that the corresponding component TDES must be selflooped with event σ on each state, so we do not need to express this explicitly. Definition 6.2.5 will work fine with systems where these self-loops have already been added.

Since BDD [8] does not support first order logic by itself, to compute state transitions we will need the following definition taken from the existential quantifier elimination method for finite domain [1].

Definition 6.2.6. For $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n$, let $\sigma \in \Sigma$ and $(\mathbf{v}_\sigma, \mathbf{v}_\sigma', N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. For $i = 1, 2, .., n$, if $v_i \in \mathbf{v}_\sigma$ and $v_i' \in \mathbf{v}_\sigma'$, then define

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

where $N_\sigma[q_i/v_i]$ is the predicate N_σ with each term v_i substituted by q_i, and $N_\sigma[q_i/v_i']$ is defined analogously.

We use the above method to eliminate either the normal or prime variable, so that we can express the statement using propositional logic that we can represent as a BDD.

Let $\mathbf{v}_\sigma = \{v_1, v_2, .., v_m\}$ for $m > 0$. For convenience, we write $\exists \mathbf{v}_\sigma N_\sigma$ to represent $\exists v_1(\exists v_2..(\exists v_m N_\sigma)..)$ and the resulting predicate should contain only prime variables in $\mathbf{v}'_\sigma$. For any computation of state predicates, we need all input variables to be consistent. That is, either all predicates in the computation have to be expressed as normal variables or prime variables. We thus need to substitute all the prime variables by normal variables, denoted as $\exists \mathbf{v}_\sigma N_\sigma[\mathbf{v}'_\sigma \rightarrow \mathbf{v}_\sigma]$. The substitution should return the predicate for the set of *target* states for σ transitions in $\mathbf{G}$. This means that each state in this set has a σ transition entering it.

Let $\mathbf{v}'_\sigma = \{v'_1, v'_2, .., v'_m\}$. For convenience, we also write $\exists \mathbf{v}'_\sigma N_\sigma$ to represent $\exists v'_1(\exists v'_2..(\exists v'_m N_\sigma)..)$ and the resulting predicate should contain only normal variables in $\mathbf{v}_\sigma$, which represents the set of *source* states for σ transitions in $\mathbf{G}$. This means that each state in this set has a σ transition leaving it.

6.3 Symbolic Computation

We will now discuss symbolic computation based on the symbolic representation we just introduced. This work is based on the work of Song [26] who in turn based his work on Ma [14].

6.3.1 Transitions and Inverse Transitions

Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n$ be a TDES plant. For a state $q \in Q$ and a event $\sigma \in \Sigma$, we want to compute the transition $\delta(q, \sigma)$ using the symbolic representation introduced previously. To do this, for $Q_P \subseteq Q$, where $P \in Pred(Q)$, we can compute

$$Q'_P = \bigcup_{q \in Q_P} \{\delta(q, \sigma)\}$$

and then find $P' := pr(Q'_P)$. However, computing q' one by one is time consuming for systems with large statespaces. Instead, we can directly compute the predicate of the set of next states from the predicate of the set of current states.

The computation is based on a function $\hat{\delta} : Pred(Q) \times \Sigma \rightarrow Pred(Q)$ defined to be

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

As discussed in previous section, the formula $\exists \mathbf{v}_\sigma N_\sigma[\mathbf{v}'_\sigma \rightarrow \mathbf{v}_\sigma]$ returns a predicate representing the set of target states $\{q' \in Q | (\exists q \in Q)\delta(q, \sigma) = q'\}$. We thus have the following definition.

Definition 6.3.1. Let $\sigma \in \Sigma$ and $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. For $P \in Pred(Q)$,

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

By first computing $N_\sigma \wedge P$ in the above definition, we are restricting the source states to those satisfying P.

We also need an inverse function $\hat{\delta}^{-1} : Pred(Q) \times \Sigma \rightarrow Pred(Q)$ to compute the predicate of the set of source states from the predicate representing the set of target states, where $\hat{\delta}^{-1}$ is defined to be

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

Since the formula $\exists \mathbf{v}'_\sigma N_\sigma$ returns a predicate representing the set of source states $\{q \in Q | \delta(q, \sigma)!\}$, we have the following definition.

Definition 6.3.2. Let $\sigma \in \Sigma$ and $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. For $P \in Pred(Q)$,

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

In the definition, $P[\mathbf{v}_\sigma \rightarrow \mathbf{v}'_\sigma]$ returns predicate P with its normal variables substituted by prime variables. As prime variables represent target states, this has the effect of restricting the target states to those satisfying P.

6.3.2 Computation of Predicate Transformers

Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n$ be the cross product TDES of component TDES $\mathbf{G}_i$ for $i = 1, 2, .., n$. Let $P \in Pred(Q)$. To compute the predicate transformers R and CR introduced in Section 6.1.2, we have the following algorithms which are taken from [26].

Reachability Check

Algorithm 6.1 $R(\mathbf{G}, P)$

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

In Algorithm 6.1, procedure $R(\mathbf{G}, P)$ takes a TDES $\mathbf{G}$ and a predicate P, then returns a predicate which holds a set of states in $\mathbf{G}$ that can be reached from q_o by states satisfying P.

At **line 1**, P_1 is initialized to be the predicate which represents the initial state q_o or $\emptyset$ if $q_o \not\models P$.

From **line 2** to **line 10**, for $i \in 1, .., n$, we loop over $\sigma \in \Sigma_i$ and determine states that satisfy P, and are reachable from a state that satisfies P_1 by a σ transition.

Due to the *intermediate logic formula expansion problem* described in [26], that intermediate logic formula can become large and complicated even though the final predicate might be relatively small, the **for** loop on **line 4** to **line 9** repeatedly modifies P_1 on a component TDES basis. We start with a specific TDES, $\mathbf{G}_i$, and determine next states using only events from Σ_i until no more changes. Then move onto next TDES. For each component TDES $\mathbf{G}_i$, P_1 is modified until it is logical equivalent to its previous value, P_3. We cycle through all the TDES until no further changes.

Coreachability Check

Algorithm 6.2 $CR(\mathbf{G}, P', \Sigma', P)$

1: $P_1 \leftarrow P' \wedge P$
2: **repeat**
3:   $P_2 \leftarrow P_1$
4:   **for** $i \leftarrow 1$ to n **do**
5:     **repeat**
6:       $P_3 \leftarrow P_1$
7:       $P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma' \cap \Sigma_i} (\hat{\delta}^{-1}(P_1, \sigma) \wedge P) \right)$
8:     **until** $P_1 \equiv P_3$
9:   **end for**
10: **until** $P_1 \equiv P_2$
11: **return** P_1

In Algorithm 6.2, procedure $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ takes a TDES $\mathbf{G}$, a predicate P', an event set Σ' and a predicate P, then returns a predicate which represents a set of states in $\mathbf{G}$ that can reach a state in $\mathbf{G}$ satisfying P' by states that satisfy P and by transition with events in Σ'. We do not present an algorithm for $CR(Q, P)$ as it is a special case which is equivalent to $\mathcal{CR}(\mathbf{G}, P_m, \Sigma, P)$.

At **line 1**, P_1 is initialized to be the predicate which represents the set of states in $Q_{P'}$ which satisfies predicate P as well.

Like in Algorithm 6.1, **line 4** to **line 9** focus on one TDES event set at a time to reduce the complexity of intermediate logic formulas. In **line 7**, we are adding to P_1 the states in P that can be reached by a state in P_1 via an event in $\Sigma' \cap \Sigma_i$.

We iterate until there are no more changes.

6.4 Symbolic Verification

The TDES systems we are interested in are composed of a plant $\mathbf{G}$ and a supervisor $\mathbf{S}$, with system event set Σ.

Given $\mathbf{G}'_i = (Y_i, \Sigma_i, \delta_i, y_{o,i}, Y_{m,i})$ and $\mathbf{G}' = \mathbf{G}'_1||\mathbf{G}'_2||..||\mathbf{G}'_n$, for $i = 1, 2, .., n$, let $\mathbf{G}_i = \mathbf{selfloop}(\mathbf{G}'_i, \Sigma - \Sigma_i)$. The plant is defined as:

$$\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n := (Y, \Sigma, \delta, y_o, Y_m)$$

Given $\mathbf{S}_i = (X_i, \Sigma, \xi_i, x_{o,i}, X_{m,i})$, the supervisor is defined as

$$\mathbf{S} = \mathbf{S}_1 \times \mathbf{S}_2 \times .. \times \mathbf{S}_m := (X, \Sigma, \xi, x_o, X_m)$$

Therefore both $\mathbf{G}$ and $\mathbf{S}$ are defined over the global event set Σ. If our component supervisors were defined over subsets of Σ and combined together using the synchronous product, we would add selfloops of the missing events as we did for the plant components, and then use these new DES from then on.

The closed-loop system, $\mathbf{G}_{cl}$, is the product of the plant and supervisor

$$\mathbf{G}_{cl} = \mathbf{G} \times \mathbf{S} := (Q, \Sigma, \eta, q_o, Q_m)$$

where $Q = Y \times X = Y_1 \times Y_2 \times .. \times Y_n \times X_1 \times X_2 \times .. \times X_m$, $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$, $\eta = \delta \times \xi$, $q_o = (y_o, x_o)$ and $Q_m = Y_m \times X_m$. See Definition 2.2.11 for more details.

Note that we cannot use $\mathbf{G}_{cl} = \mathbf{meet}(\mathbf{G}, \mathbf{S})$ as **meet** by definition only contains reachable states, which is too restrictive. The product DES is the same as **meet**, but it can include unreachable states.

Definition 6.4.1. Let $\mathbf{G}_{cl} = \mathbf{G} \times \mathbf{S} := (Q, \Sigma, \eta, q_o, Q_m)$ where $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times .. \times \mathbf{G}_n = (Y, \Sigma, \delta, y_o, Y_m)$ and $\mathbf{S} = \mathbf{S}_1 \times \mathbf{S}_2 \times .. \times \mathbf{S}_m = (X, \Sigma, \xi, x_o, X_m)$. For a given event $\sigma \in \Sigma$, the σ plant transition predicate $N_{\mathbf{G},\sigma} : Q \times Q \rightarrow \{T, F\}$ can be written as

$$N_{\mathbf{G},\sigma}(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \le i \le n\}} \left(\bigvee_{\{y_i, y'_i \in Y_i | \delta_i(y_i, \sigma) = y'_i\}} (v_i = y_i) \wedge (v'_i = y'_i) \right)$$

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

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

$N_{\mathbf{G},\sigma}$ and $N_{\mathbf{S},\sigma}$ are state predicates defined on $Q \times Q$ and use the $\mathbf{v}$ and $\mathbf{v}'$ variables like N_σ. We use $N_{\mathbf{G},\sigma}$ when we wish to determine if there is a σ defined at the plant portion of the indicated states, say for when we are checking controllability. Similarly, we use $N_{\mathbf{S},\sigma}$ when we wish to determine if there is a σ defined at the supervisor portion of the indicated states. They must be defined over $Q \times Q$ so the results of each can be compared and combined with other state predicates on Q.

Definition 6.4.2. Let $\sigma \in \Sigma$ and $N_{\mathbf{G},\sigma}$ be the σ transition predicate for plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$. We define $\hat{\delta}_{\mathbf{G}} : Pred(Q) \times \Sigma \rightarrow Pred(Q)$, for $P \in Pred(Q)$, to be

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

and we also define $\hat{\delta}_{\mathbf{G}}^{-1} : Pred(Q) \times \Sigma \rightarrow Pred(Q)$ to be

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

Definition 6.4.3. Let $\sigma \in \Sigma$ and $N_{\mathbf{S},\sigma}$ be the σ transition predicate for supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$. We define $\hat{\xi} : Pred(Q) \times \Sigma \rightarrow Pred(Q)$, for $P \in Pred(Q)$, to be

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

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

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

6.4.1 Untimed Controllability

To verify that a supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ is controllable with respect to plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, we need the closed loop system $\mathbf{G}_{cl} = (Q, \Sigma, \eta, q_o, Q_m)$ as defined in Section 6.4. For $q \in Q$, there must exist a state $x \in X$ and $y \in Y$ such that $q = (y, x)$.

According to Definition 2.2.15 for untimed controllability, we can express the states that could cause $\mathbf{S}$ to be uncontrollable for $\mathbf{G}$ (if they are reachable), as follows:

Definition 6.4.4. Let $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant, then

$$Q_{bad} = \{q = (y, x) \in Q | (\exists \sigma_u \in \Sigma_u) \delta(y, \sigma_u)! \ \& \ \xi(x, \sigma_u) \not! \}$$

By this definition, the state set Q_{bad} includes all states $q \in Q$ in system $\mathbf{G}_{cl}$ that an uncontrollable event is eligible at the corresponding state in plant $\mathbf{G}$ but not eligible in the corresponding state in supervisor $\mathbf{S}$. We consider such states *bad.* Of course, not all states in Q_{bad} are necessarily reachable. Therefore $\mathbf{S}$ is controllable with to respect to $\mathbf{G}$ if $Q_{bad} \cap Q_{reach} = \emptyset$ where Q_{reach} is the set of reachable states.

The corresponding predicate $P_{bad} := pr(Q_{bad})$ is defined to be

$$P_{bad} = \bigvee_{\sigma_u \in \Sigma_u} \left(\hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma_u) \wedge \neg\hat{\xi}^{-1}(true, \sigma_u) \right)$$

where $\hat{\delta}_{\mathbf{G}}^{-1}$ and $\hat{\xi}^{-1}$ are the inverse transition predicate functions for $\mathbf{G}$ and $\mathbf{S}$ respectively. We thus have $\mathbf{S}$ is controllable with respect to $\mathbf{G}$ if $P_{bad} \wedge P_{reach} \equiv false$ where $P_{reach} := pr(Q_{reach})$ holds the set of reachable states. Otherwise, $P_{bad} \wedge P_{reach}$ represents the set of *bad* states where supervisor S has disabled an uncontrollable event.

Algorithm 6.3, from [26], checks untimed controllability. For each uncontrollable event σ_u, it looks for the reachable composite state at which σ_u is eligible in $\mathbf{G}$ but not eligible in $\mathbf{S}$. If such a state exists, then $\mathbf{S}$ is not controllable with respect to $\mathbf{G}$. The algorithm returns $True$[1] if the supervisor $\mathbf{S}$ is controllable with respect to $\mathbf{G}$ and $False$ otherwise.

Algorithm 6.3 CheckUntimedControllability($\mathbf{G}$, $\mathbf{S}$)

$P_{bad} \leftarrow false$
for all $\sigma_u \in \Sigma_u$ **do**
  $P_{bad} \leftarrow P_{bad} \vee (\hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma_u) \wedge \neg\hat{\xi}^{-1}(true, \sigma_u))$
end for
$P_{bad} \leftarrow P_{bad} \wedge R(\mathbf{G} \times \mathbf{S}, true)$
if ($P_{bad} \not\equiv false$) **then**
  return $False$
end if
return $True$

[1]We use $True$ and $False$ here because it is a boolean returned by the algorithm, instead of a state predicate.

6.4.2 Plant Completeness

Similar to checking untimed controllability, we have the following definition for plant completeness.

Definition 6.4.5. Let $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a DES supervisor. Let $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a DES plant, then

$$Q_{incomplete} = \{q = (y, x) \in Q \mid (\exists \sigma \in \Sigma_{hib})\xi(x, \sigma)! \ \& \ \delta(y, \sigma) \not!\}$$

By this definition, the state set $Q_{incomplete}$ includes all states q in system $\mathbf{G}_{cl}$ that a prohibitable event is eligible at the corresponding state in supervisor $\mathbf{S}$ but not eligible in the corresponding state in plant $\mathbf{G}$. Plant $\mathbf{G}$ is complete for its supervisor $\mathbf{S}$ only if $Q_{incomplete} \cap Q_{reach} = \emptyset$. We only care about states in $Q_{incomplete}$ that are reachable.

The corresponding predicate $P_{incomplete} := pr(Q_{incomplete})$ is defined to be

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

where $\hat{\delta}_{\mathbf{G}}^{-1}$ and $\hat{\xi}^{-1}$ are the inverse transition predicate functions for $\mathbf{G}$ and $\mathbf{S}$ respectively. Therefore the plant $\mathbf{G}$ is complete for its supervisor $\mathbf{S}$ only if $P_{incomplete} \wedge P_{reach} \equiv false$. Otherwise, $P_{incomplete} \wedge P_{reach}$ represents the set of states which fail the condition.

Algorithm 6.4 CheckPlantCompleteness(**G**, **S**)

$P_{incomplete} \leftarrow false$
for all $\sigma \in \Sigma_{hib}$ **do**
    $P_{incomplete} \leftarrow P_{incomplete} \vee (\hat{\xi}^{-1}(true, \sigma) \wedge \neg \hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma))$
end for
$P_{incomplete} \leftarrow P_{incomplete} \wedge R(\mathbf{G} \times \mathbf{S}, true)$
if $(P_{incomplete} \not\equiv false)$ **then**
    return $False$
end if
return $True$

Algorithm 6.4 checks for plant completeness. For each prohibitable event σ, it looks for reachable composite states at which σ is eligible in $\mathbf{S}$ but not eligible in

G. If such a state exists, then plant **G** fails to be complete for supervisor **S** and the algorithm returns *False*. Otherwise it returns *True*.

6.4.3 Non-blocking

Algorithm 6.5 checks for non-blocking as defined in Definition 2.2.6. It compares the set of reachable states with the set of coreachable states, then returns *True* if there is no reachable state that is not coreachable and *False* otherwise.

Algorithm 6.5 Nonblocking(**G**)

$P_{reach} \leftarrow R(\mathbf{G}, true)$
$P_{coreach} \leftarrow \mathcal{CR}(\mathbf{G}, P_{reach})$
if $(P_{reach} \wedge \neg P_{coreach} \not\equiv false)$ **then**
  return *False*
end if
return *True*

6.4.4 Activity Loop Free

By Definition 2.3.3 of Activity Loop Free (ALF), we require that for each reachable state in a TDES there will not be a non-empty string of activity events leaving from that state and back to itself. This is to prevent the TDES from "stopping the clock". Algorithm 6.6 checks the given TDES **G** and returns *True* if it is ALF and *False* otherwise.

At **line 1**, Algorithm 6.6 first calculates all the reachable states. Then for each state q in P_{chk}, it starts from any states P_{visit} reached via activity events from q at **line 4**. From there, in the following loop from **line 7** to **line 17** it traverses to next states P_{next} until no more state can be reached by activity events.

At each iteration of the loop, the algorithm first checks if there is an overlap between P_{visit} and P_{next}. Then it checks if state q has been reached again. If state q has been reached again, then the system is not ALF. Otherwise, the loop continues.

Once the check is done for state q, this state is removed from P_{chk}. If there is no overlap found in the loop, all the visited states are removed from P_{chk}. After that,

Algorithm 6.6 ALF(G)

```
P_chk ← R(G, true)
P_tmp ← false
for (q ⊨ P_chk) do
  P_visit ← ( ⋁_{σ∈Σ_act} δ̂(pr({q}), σ) ) ∧ P_chk
  overlap ← False
  P_next ← P_visit
  repeat
    P_next ← ( ⋁_{σ∈Σ_act} δ̂(P_next, σ) ) ∧ P_chk
    P_tmp ← P_visit
    if (P_visit ∧ P_next ≢ false) then
      overlap ← True
    end if
    P_visit ← P_visit ∨ P_next
    if (q ⊨ P_visit) then
      return False
    end if
  until (P_visit ≡ P_tmp)
  P_chk ← P_chk − pr({q})
  if (¬overlap) then
    P_chk ← P_chk − P_visit
  end if
end for
return True
```

the algorithm moves to next state in P_{chk}. If there was no *False* returned during the loop, the algorithm will consider it to be ALF and returns *True*.

6.4.5 Proper Time Behavior

By Definition 2.3.5 for Proper Time Behavior, we require that at each reachable state in a TDES plant, either an uncontrollable event or a tick event is eligible. Algorithm

6.7 checks the given TDES plant **G** and returns $True$ if it has a proper time behavior and $False$ otherwise.

Algorithm 6.7 ProperTimeBehavior(**G**)

$P_1 \leftarrow \bigvee_{\sigma \in \Sigma_u \cup \{\tau\}} \delta^{-1}(true, \sigma)$
$P_2 \leftarrow R(\mathbf{G}, true)$
if $P_2 - P_1 \not\equiv false$ **then**
  return $False$
end if
return $True$

Algorithm 6.7 first calculates P_1, the set of all states that have a $\Sigma_u \cup \{\tau\}$ transition leaving it. It then compares P_1 to the set P_2 of reachable states. When there is a state in P_2 but not in P_1, it implies that the state is reachable and neither a tick or an uncontrollable event is eligible at this state.

6.4.6 SD Controllability and S-Singular Prohibitable Behavior

Algorithm 6.8 evaluates SD controllability for supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ with respect to plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, where **G**, **S**, and the closed loop system, $\mathbf{G}_{cl} = \mathbf{G} \times \mathbf{S}$ are as defined in Section 6.4. In addition, the algorithm's subroutine, Algorithm 6.11, also checks that **G** has **S**-singular prohibitable behavior. As checking **Point i** of the SD controllability definition is the same as checking untimed controllability (Algorithm 6.3), we will not mention it explicitly here.

Σ is defined to be $\Sigma = \Sigma_{hib} \dot{\cup} \Sigma_u \dot{\cup} \{\tau\}$, where Σ_{hib} is the set of prohibitable events in **G** and Σ_u is the set of uncontrollable events in **G**. The set of controllable events is $\Sigma_c = \Sigma_{hib} \cup \{\tau\}$, and the set of activity events is $\Sigma_{act} = \Sigma_{hib} \dot{\cup} \Sigma_u$.

The algorithm makes the following assumptions:

- The set Σ_{hib} of prohibitable events equals the set Σ_{for} of forcible events
- The plant has proper time behavior (checked by Algorithm 6.7)
- All TDES are finite and deterministic

- The closed loop system, $\mathbf{G}_{cl}$, is activity loop free (ALF) (checked by Algorithm 6.6)

The algorithm uses certain variables as it executes.

P_{reach}: The predicate of the set of reachable states of $\mathbf{G}_{cl}$.

P_{SF}: The predicate of the set that contains sampling states of $\mathbf{G}_{cl}$ found by the algorithm.

Z_{SP}: This set contains the predicates of sampling states in $\mathbf{G}_{cl}$ found and not yet analyzed by the algorithm.

$N_{\mathbf{G},\sigma}, N_{\mathbf{S},\sigma}$: Transition predicates for σ for $\mathbf{G}$ and $\mathbf{S}$ as in Definition 6.4.1.

N_σ: Transition predicate for σ for $\mathbf{G}_{cl}$ as in Definition 6.2.5.

$\hat{\delta}$: Transition function for state predicates for $\mathbf{G}_{cl}$ as in Definition 6.3.1.

$\hat{\delta}_{\mathbf{G}}$: Transition function for state predicates for $\mathbf{G}$ only as in Definition 6.4.2.

$\hat{\xi}$: Transition function for state predicates for $\mathbf{S}$ only as in Definition 6.4.3.

$pNerFail$: This set $pNerFail \subseteq \mathrm{Pwr}(Pred(Q))$ is a set of sets of predicates that stores information where **Point iii.2** in Definition 3.2.2 of SD controllability may have failed.

$SDControllable$: This flag asserts if $\mathbf{S}$ is SD controllable with respect to $\mathbf{G}$.

Algorithm 6.8 starts at the initial state, which is always a sampling state. Then it analyzes the concurrent behavior of this state by creating a reachability tree with the initial state as a node. It expands the tree until all paths terminate at a tick event. Since we first check that the closed loop system is activity loop free, the system has a finite state space and that the plant has proper time behavior, we are either guaranteed that we will reach a tick after a finite number of events, or the system will fail **Point ii** of the SD controllability definition. Any new sampling states found are then analyzed as above, until all reachable sampling states have been analyzed.

As the reachability tree for a given sampling period is created, conformance to Definition 3.2.2 of SD controllability is tested. We also test here that $\mathbf{G}$ has $\mathbf{S}$-singular

Algorithm 6.8 CheckSDControllability($\mathbf{G}, \mathbf{S}$)

$\mathbf{G}_{cl} \leftarrow \mathbf{G} \times \mathbf{S}$
$P_{reach} \leftarrow R(\mathbf{G} \times \mathbf{S}, true)$
if (CheckSDContii($\mathbf{G}, \mathbf{S}, P_{reach}$) $= False$) **then**
    return $False$
end if
$SDControllable \leftarrow True$
$P_{SF} \leftarrow pr\{z_0\}$
$Z_{SP} \leftarrow \{pr\{z_0\}\}$
$pNerFail \leftarrow \emptyset$
while ($Z_{SP} \neq \emptyset$) **do**
    $P_{ss} \leftarrow$Pop(Z_{SP})
    $SDControllable \leftarrow$ AnalyseSampledState($\mathbf{G}, \mathbf{S}, P_{SF}, Z_{SP}, P_{reach}, P_{ss}, pNerFail$)
    if ($\neg SDControllable$) **then**
        return $False$
    end if
end while
if ($pNerFail \neq \emptyset$) **then**
    $SDControllable \leftarrow$ RecheckNerodeCells($pNerFail$)
    if ($\neg SDControllable$) **then**
        return $False$
    end if
end if
if ($\neg$ CheckSamplingMarkingStates(P_{reach})) **then**
    return $False$
end if
return $True$

prohibitable behavior. With the exception of **Point iii.2**, evaluation stops if the test for any of the other points fail. If the test for **Point iii.2** fails, the problem area is noted and the algorithm continues until all reachable sampling states have been analyzed. Nerode cells will be rechecked and then **Point iii.2** is tested again.

In the algorithm, $pNerFail$ represents states reached by concurrent strings with the same occurrence image, thus should belong to the same equivalence classes for $\equiv_{L(\mathbf{S})\cap L(\mathbf{G})}$ and $\equiv_{L_m(\mathbf{S})\cap L_m(\mathbf{G})}$. It contains the states these strings ended up in, and we will now check to see if these states actually represent the same equivalence cells. i.e. they are equivalent mod λ (Definition 2.2.7).

Finally, the algorithm checks **Point iv** in Definition 3.2.2 of SD controllability by comparing the set of marked states, implied by P_m, with the set of states reached by a tick event. If not all states implied by P_m are reached by a tick and if that state not reached by a tick is not the initial state z_o, then it returns $False$.

If all tests pass, the algorithm returns $True$ at the end.

See following sections for subroutines in Algorithm 6.8. The subroutine *CheckSDContii* is defined in Algorithm 6.9. The subroutine *AnalyseSampledState* is defined in Algorithm 6.10. The subroutine *RecheckNerodeCells* is defined in Algorithm 6.13. The subroutine *CheckSamplingMarkingStates* is defined in Algorithm 6.15.

Point ii of SD Controllability

Algorithm 6.9 checks **Point ii** of the SD Controllability definition. The algorithm takes the following three parameters: a plant $\mathbf{G}$, a supervisor $\mathbf{S}$ and a predicate P_{reach} of all reachable states in $\mathbf{G}_{cl}$.

Algorithm 6.9 CheckSDContii($\mathbf{G}, \mathbf{S}, P_{reach}$)

1: $P_{q-hib} \leftarrow \bigvee_{\sigma\in\Sigma_{hib}} \exists \mathbf{v}' N_\sigma$
2: $P_{bad} \leftarrow \exists \mathbf{v}' N_{tick} \wedge P_{q-hib}$
3: **if** $P_{bad} \wedge P_{reach} \not\equiv false$ **then**
4:     **return** $False$
5: **end if**
6: $P_{bad} \leftarrow \exists \mathbf{v}' N_{\mathbf{G},tick} \wedge \neg(\exists \mathbf{v}' N_{\mathbf{S},tick}) \wedge \neg P_{q-hib}$
7: **if** $P_{bad} \wedge P_{reach} \not\equiv false$ **then**
8:     **return** $False$
9: **end if**
10: **return** $True$

From **line 1** to **line 5** the algorithm checks the "$\Rightarrow$" part of **Point ii**. It checks for any reachable states in $\mathbf{G}_{cl}$ that has both a prohibitable event and tick event enabled. If such a state exists, then it returns *False*.

Then from **line 6** to **line 9**, the algorithm checks "$\Leftarrow$" part of **Point ii**. It checks to see if a reachable state exists in $\mathbf{G}_{cl}$ where no prohibitable events are eligible, but a tick is eligible in $\mathbf{G}$ but not in $\mathbf{S}$. If such a state exists, then it returns *False*.

AnalyzeSampledState

Algorithm 6.10 analyzes the concurrent behavior for sampling state q_{ss}, represented by predicate P_{ss}. The algorithm takes seven parameters. See Algorithm 6.8 for their definitions.

During the execution, the algorithm uses the following variables:

Σ_{Elig}: The set of prohibitable events eligible in both $\mathbf{G}$ and $\mathbf{S}$ at q_{ss}, the sampling state in $\mathbf{G}_{cl}$ that we are processing.

P_q: The predicate of current state in $\mathbf{G}_{cl}$.

Σ_{poss}: The set of events eligible in both $\mathbf{G}$ and $\mathbf{S}$ at predicate P_q of current state in $\mathbf{G}_{cl}$.

$\Sigma_{\mathbf{G}poss}$: The set of prohibitable events eligible in $\mathbf{G}$ at predicate P_q of current state in $\mathbf{G}_{cl}$.

nextLabel: This number represents the next unused node in B_{map}. It is used to name newly discovered nodes of the reachability tree.

B_{map}: This partial function $B_{map} : \mathcal{N} \rightarrow Pred(Q)$ maps the nodes of the reachability tree to the predicates of the states of $\mathbf{G}_{cl}$ which the nodes represent. This function will sometimes be treated like the set $B_{map} \subseteq \mathcal{N} \times Pred(Q)$. Note, $\mathcal{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers.

B_p: This is the set of nodes pending to be expanded in the reachability tree.

B_{conc}: The set $B_{conc} \subseteq \mathcal{N} \times Pred(Q)$ contains nodes that represent concurrent strings and the sampled states the strings lead to. For $(b, q) \in B_{conc}$, the node b is a node at which tick is eligible in $\mathbf{G}$ and $\mathbf{S}$, and q is the sampling state of $\mathbf{G}_{cl}$ that the tick leads to.

Algorithm 6.10 AnalyseSampledState($\mathbf{G}, \mathbf{S}, P_{SF}, Z_{SP}, P_{reach}, P_{ss}, pNerFail$)

$B_{map} \leftarrow \{(0, P_{ss})\}$
$B_{conc} \leftarrow \emptyset$
$B_p \leftarrow \{0\}$
$nextLabel \leftarrow 1$
$\text{Occu}_B \leftarrow \{(0, \emptyset)\}$
while $B_p \neq \emptyset$ **do**
  $b \leftarrow \text{Pop}(B_p)$
  $P_q \leftarrow B_{map}(b)$
  $\Sigma_{poss} \leftarrow \emptyset$
  $\Sigma_{\mathbf{G}poss} \leftarrow \emptyset$
  for all $\sigma \in \Sigma$ **do**
    if $(\hat{\delta}(P_q, \sigma) \not\equiv false)$ **then**
      $\Sigma_{poss} \leftarrow \Sigma_{poss} \cup \{\sigma\}$
    end if
    if $(\hat{\delta}_{\mathbf{G}}(P_q, \sigma) \not\equiv false)$ **then**
      $\Sigma_{\mathbf{G}poss} \leftarrow \Sigma_{\mathbf{G}poss} \cup (\{\sigma\} \cap \Sigma_{hib})$
    end if
  end for
  if $(P_q \equiv P_{ss})$ **then**
    $\Sigma_{Elig} \leftarrow \Sigma_{poss} \cap \Sigma_{hib}$
  end if
  if $((\Sigma_{poss} \cup Occu_B(b)) \cap \Sigma_{hib} \neq \Sigma_{Elig})$ **then**
    return $False$
  end if
  if $(\neg\text{NextState}(b, \Sigma_{poss}, \Sigma_{\mathbf{G}poss}, P_q, nextLabel, B_{map}, B_p, B_{conc}, P_{SF}, Z_{SP}, Occu_B(b)))$ **then**
    return $False$
  end if
end while
$\text{CheckNerodeCells}(B_{conc}, \text{Occu}_B, pNerFail)$
return $True$

Occu_B: The partial function $\text{Occu}_B : \mathcal{N} \to \text{Pwr}(\Sigma)$ maps the nodes of the reachability tree to the occurrence image of the string that they represent. This function will sometimes be treated like the set $\text{Occu}_B \subseteq \mathcal{N} \times \text{Pwr}(\Sigma)$.

The algorithm builds the reachability tree, starting at q_{ss}, until all nodes terminates at a tick event or one of our checks fail. As we need to evaluate the strings taking us from the sampled state, we need to know how we got to a given state. So we introduce nodes for the states we reach, and associate with the node the occurrence image of the string that brought us to that node. We use map Occu_B to do this. The function B_{map} maps the nodes back to the states in $\mathbf{G}_{cl}$ that they represent. The information is stored per node, not per state of $\mathbf{G}_{cl}$. It means there could be two or more nodes that corresponds to the same state, but have possibly different occurrence images, as they were reached by different strings.

When the algorithm starts, we store the set of prohibitable events that are eligible at our starting sampling state. **Point iii.1** in Definition 3.2.2 for SD controllability is analyzed as the tree is built. In the algorithm, a concurrent string is represented by the label b of the node it is associated with, and a sampled string is represented by the sampling state q_{ss}. From **line 22** to **line 24**, the algorithm checks this condition. If the test fails, the algorithm returns $False$.

After the reachability tree is complete, B_{conc} will represent the concurrent strings leaving the sampling state implied by predicate P_{ss}, and the sampling state each string leads to. We then call *CheckNerodeCells* which will indicate via $pNerFail$ what further checks are needed. This is how **Point iii.2** is checked.

In next section we will discuss subroutine *NextState* (Algorithm 6.11) and subroutine *CheckNerodeCells* (Algorithm 6.12), as both algorithms are called from *AnalyseSampledState.*

NextState

Algorithm 6.11 determines the next states to be processed for Algorithm 6.10. Subroutine *NextState* takes parameters b, Σ_{poss}, $\Sigma_{\mathbf{G}poss}$, P_q, $nextLabel$, B_{map}, B_p, B_{conc}, P_{SF}, Z_{SP}, and $Occu_B(b)$. See Algorithms 6.8 and 6.10 for their definitions.

The algorithm returns if the set of eligible events, Σ_{poss}, at state q (implied by P_q) of $\mathbf{G}_{cl}$, is empty. If tick is possible at state q, we determine the new sampling

Algorithm 6.11 NextState(...)

1: **if** $(\Sigma_{poss} = \emptyset)$ **then**
2:   **return** $True$
3: **end if**
4: **if** $(\tau \in \Sigma_{poss})$ **then**
5:   $P_{q'} \leftarrow \hat{\delta}(P_q, tick)$
6:   Push$(B_{conc}, (b, P_{q'}))$
7:   **if** $(P_{q'} \wedge P_{SF} \equiv false)$ **then**
8:     $P_{SF} \leftarrow P_{SF} \vee P_{q'}$
9:     Push$(Z_{SP}, P_{q'})$
10:   **end if**
11: **end if**
12: **for all** $\sigma \in \Sigma_{\mathbf{G}poss}$ **do**
13:   **if** $(\mathrm{Occu}_B(b) \cap \{\sigma\} \neq \emptyset)$ **then**
14:     **return** $False$
15:   **end if**
16: **end for**
17: **for all** $\sigma \in (\Sigma_{poss} - \{\tau\})$ **do**
18:   $P_{q'} \leftarrow \hat{\delta}(P_q, \sigma)$
19:   $b' \leftarrow nextLabel$
20:   $nextLabel \leftarrow nextLabel + 1$
21:   Push$(B_{map}, (b', P_{q'}))$
22:   Push(B_p, b')
23:   Push$(\mathrm{Occu}_B, (b', Occu_B(b) \cup \{\sigma\}))$
24: **end for**
25: **return** $True$

state that tick takes us to, and then add b and the state to B_{conc}. If we have not yet encountered this state, it is added to P_{SF} and Z_{SP}.

In **lines 12** to **16**, we check that no prohibitable event is currently eligible in **G** if it has already occurred this sampling period. This is part of checking if **G** has **S**-singular prohibitable behavior.

Then for each non-tick event σ, it finds the next state implied by $P_{q'}$, assigns a new node b' to it and pushes (b', q') onto B_{map}, and b' onto the set of pending nodes, B_p. It also associates the occurrence image of the strings that took us to b' with node b', via Occu_B.

CheckNerodeCells

Algorithm 6.12 is used to determine if we have possible violations of **Point iii.2** of the SD controllability definition. Subroutine *CheckNerodeCells* is passed a set of sampled states reached in the recent search, plus information on the occurrence images of the concurrent strings that took us to that state. For more details on these parameters, see Algorithm 6.10.

Point iii.2 of the SD Controllability definition requires that if two concurrent strings have the same occurrence image, they must take us to states representing the same equivalence cell of $\equiv_{L(\mathbf{S}) \cap L(\mathbf{G})}$ and $\equiv_{L_m(\mathbf{S}) \cap L_m(\mathbf{G})}$. In other words, to states that are λ-equivalent (see Definition 2.2.7). If $\mathbf{G}_{cl}$ is minimal, they must go to the same state . If they do not, we add each set of non-equal states, represented by variable $Z_{eqv} \subseteq \mathit{Pred}(Q)$, to $pNerFail$, and we will later check to see if they are indeed λ-equivalent. Note that every state predicate in Z_{eqv} represents a single state.

RecheckNerodeCells

Algorithm 6.13 checks state subsets of $\mathbf{G}_{cl}$ stored in $pNerFail$ to see if the states in a given subset actually are equivalent mod λ (see Definition 2.2.7) to each other. Subroutine *RecheckNerodeCells* is passed parameter $pNerFail$. See Algorithm 6.8 for the definition of $pNerFail$.

At a given sampling state, if we found two or more concurrent strings that had the same occurrence image but terminated in different states, we stored the predicates that identified the states these strings led us to, in $pNerFail$. Variable $pNerFail$ contains all such sets found by Algorithm 6.8 as it processed all the reachable sampling states of $\mathbf{G}_{cl}$. For the system to pass **Point iii.2** of Definition 3.2.2, the states in a given state predicate in $pNerFail$ must all be λ-equivalent to each other. If a single set fails this test, the system fails **Point iii.2** of Definition 3.2.2.

From **line 1** to **line 3**, the algorithm first sees if there is actually any state sets in $pNerFail$ to be checked. If it is empty, it returns *True*.

Algorithm 6.12 CheckNerodeCells(B_{conc}, Occu_B, $pNerFail$)

1: **while** ($B_{conc} \neq \emptyset$) **do**
2:   $(b, P_q) \leftarrow \text{Pop}(B_{conc})$
3:   $Z_{eqv} \leftarrow \emptyset$
4:   $\text{Push}(Z_{eqv}, P_q)$
5:   $sameCell \leftarrow True$
6:   **for all** $(b', P_{q'}) \in B_{conc}$ **do**
7:     **if** ($\text{Occu}_B(b) = \text{Occu}_B(b')$) **then**
8:       $\text{Push}(Z_{eqv}, P_{q'})$
9:       $B_{conc} \leftarrow B_{conc} - \{(b', P_{q'})\}$
10:       **if** ($P_q \not\equiv P_{q'}$) **then**
11:         $sameCell \leftarrow False$
12:       **end if**
13:     **end if**
14:   **end for**
15:   **if** ($\neg sameCell$) **then**
16:     $\text{Push}(pNerFail, Z_{eqv})$
17:   **end if**
18: **end while**
19: **return**

At **line 4**, variable $Visited \subseteq Pred(Q) \times Pred(Q)$ is initialized to the empty set. After each call to *RecheckNerodeCell* (Algorithm 6.14) that returns $True$, $Visited$ will contain tuples of state predicates, where each predicate in the tuple represents a single state in Q. Essentially, a tuple belonging to $Visited$ means that *RecheckNerodeCell* has determined that those two states are λ-equivalent. We pass it back into *RecheckNerodeCell* so that this information can be reused in future checks.

During the **while** loop from **lines 5** to **line 10**, we call *RecheckNerodeCell* for each element $Z_{eqv} \subseteq Pred(Q)$ in $pNerFail$. If *RecheckNerodeCell* returns $False$, then the system fails **Point iii.2** of Definition 3.2.2.

Algorithm 6.13 RecheckNerodeCells($pNerFail$)

```
if (pNerFail = ∅) then
    return True
end if
Visited ← ∅
while pNerFail ≠ ∅ do
    Z_eqv ←Pop(pNerFail)
    if ¬ RecheckNerodeCell(Z_eqv, Visited) then
        return False
    end if
end while
return True
```

RecheckNerodeCell

For each set of state predicates $Z_{eqv} \subseteq Pred(Q)$ that Algorithm 6.14 is called with, we will check that these states identified by the predicates are λ-equivalent to each other, and return *False* if they are not. When Subroutine RecheckNerodeCell is called, parameter $Visited \subseteq Pred(Q) \times Pred(Q)$ represents tuples of states that are known to be λ-equivalent. See Algorithm 6.13 for further details about these parameters.

At **line 1**, a state predicate is popped out of Z_{eqv} and labeled as P_{q_1}.

From **line 2** to **line 6**, the algorithm populates the *Pending* set with all pairs of P_{q_1} and P_{q_2}, where P_{q_2} is also popped from Z_{eqv}. Note that state predicates P_{q_1} and P_{q_2} each represent a single state in Q. Set *Pending* represents all the state pairs that we wish to show to be λ-equivalent. Of course, we will likely finding new state pairs that we will also need to test, as our algorithm progresses.

Two states $q_1, q_2 \in Q$ are λ-equivalent if they have the same future with respect to the marked and closed behavior of $\mathbf{G}_{cl}$. That means that both states are either marked, or neither is marked (**lines 10-12**). It also means that for each $\sigma \in \Sigma$ (**lines 13-28**), there is a σ transition at one state if and only if there is a σ transition at the other (**line 17-18**). Also, if there is a σ transition leaving each state, the two new states reached must be λ-equivalent. Obviously if $q_1 = q_2$ (**line 19**), then the two

Algorithm 6.14 RecheckNerodeCell($Z_{eqv}, Visited$)

$P_{q_1} \leftarrow$ Pop(Z_{eqv})
$Pending \leftarrow \emptyset$
while $Z_{eqv} \neq \emptyset$ **do**
    $P_{q_2} \leftarrow$ Pop(Z_{eqv})
    Push($Pending, (P_{q_1}, P_{q_2})$)
end while
while $Pending \neq \emptyset$ **do**
    $(P_{q_1}, P_{q_2}) \leftarrow$ Pop($Pending$)
    $P \leftarrow P_{q_1} \vee P_{q_2}$
    if $(P \wedge P_m \not\equiv false)$ & $(P \wedge P_m \not\equiv P)$ **then**
        return $False$
    end if
    for all $\sigma \in \Sigma$ **do**
        $P' \leftarrow \hat{\delta}(P, \sigma)$
        $P'_{q_1} \leftarrow \hat{\delta}(P_{q_1}, \sigma)$
        $P'_{q_2} \leftarrow \hat{\delta}(P_{q_2}, \sigma)$
        if $(P' \not\equiv false)$ **then**
            if $(P'_{q_1} \wedge P' \not\equiv false)$ & $(P'_{q_2} \wedge P' \not\equiv false)$ **then**
                if $(P'_{q_1} \not\equiv P'_{q_2})$ & $((P'_{q_1}, P'_{q_2}) \notin Visited)$ **then**
                    Push($Visited, (P'_{q_1}, P'_{q_2})$)
                    Push($Visited, (P'_{q_2}, P'_{q_1})$)
                    Push($Pending, (P'_{q_1}, P'_{q_2})$)
                end if
            else
                return $False$
            end if
        end if
    end for
end while
return $True$

states are λ-equivalent.

Our approach to prove that $q_1, q_2 \in Q$ are λ-equivalent will be to attempt to prove they are not. We will check the per state conditions (**lines 10-12** and **lines 17-27**), and then if the states take us to two different states for a common σ transition (**line 19**), we check to see if the new states already have a tuple in *Visited* (**line 19**). If they do, either they are known to be equivalent or we have already processed the pair and added their requirements to *Pending*. If they do not, we add the pair to *Pending* and *Visited* (**lines 20-22**). This ensures that a state pair is added to pending at most once, so we will terminate after a finite number of iterations as $\mathbf{G}_{cl}$ has a finite statespace. There is no sense in adding the pair to *Pending* twice as processing the pair twice would not provide new information to check.

The idea is that if the state pair are not equivalent, then we must eventually reach a state pair that we need to be equivalent, but the states do not have the same marking information and/or the same possible outgoing event transitions. If we never reach such a pair (and we have a finite number of possible state pairs to check), then the original state pairs must be equivalent. Not only that, then every state pair that we encountered to check, must also be equivalent to each other, or they would have caused the test to fail. This is why all state pairs in *Visited* are known to be equivalent if the algorithm returns true.

As we expect that our plant and supervisor TDES components are typically minimal or close to it, we also expect that $\mathbf{G}_{cl}$ is likely minimal or close to it. As such, we believe that when we start to check that a state pair is equivalent, we expect to either quickly find out it is not, or have the test terminate successfully as the new state pairs we encounter to test are actually the same state.

We now make a few additional comments to clarify a few steps of the algorithm. For **lines 14-16**, predicate P' represents states reached via σ from either state q_1 or state q_2, while P'_{q_1} and P'_{q_1} represents states reached via σ only from the indicated state. The condition on **line 17** will be satisfied if either state q_1 or state q_2 has a σ transition leaving that state. The condition on **line 18** will fail if only one of the two states has a σ transition leaving that state.

Checking Point iv of SD Controllability

Point iv in Definition 3.2.2 for SD Controllability is checked by Algorithm 6.15. Subroutine *CheckSamplingMarkingStates* is passed the state predicate P_{reach}, which represents the set of reachable states of $\mathbf{G}_{cl}$, when it is called by Algorithm 6.8.

Point iv of SD Controllability states that only sampled strings can be marked strings. This implies that every reachable marked state of $\mathbf{G}_{cl}$ can only have at most incoming tick transitions from other reachable states.

Algorithm 6.15 CheckSamplingMarkingStates(P_{reach})

1: $P \leftarrow \bigvee_{\sigma \in \Sigma - \{\tau\}} \hat{\delta}(P_{reach}, \sigma)$
2: **if** $P \wedge P_m \not\equiv false$ **then**
3:     **return** $False$
4: **end if**
5: **return** $True$

At **line 1**, we first identify all states with an incoming non-tick transition from a reachable state. This implies that all of these states are also reachable. At **line 2**, we check to see if any of these states are also marked. If one of them is marked, then $\mathbf{G}_{cl}$ fails this condition and we return $False$.

Chapter 7

Examples

In this chapter we provide illustrative examples for key required conditions we have defined for an SD system (see Section 7.1), as well as a successful example based on Hill's Flexible Manufacturing System (FMS) from [11] (see Section 7.2). Then in Section 7.3, we translate the FMS example into Moore FSM, using the approach we discussed in Chapter 4.

All the DES examples have been verified to be either passing or failing using the software tool we implemented, based on the algorithms from Chapter 6. The examples are illustrated as per the legend shown in Figure 7.1.

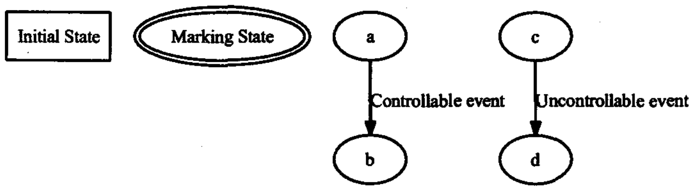


Figure 7.1: Legend Used to Display DES

As shown in Figure 7.1,

- An initial state is a box shape with its border single lined.
- A marked state is a ellipse shape with its border doubled lined.
- By default, a regular state is a ellipse shape with its border single lined.
- A controllable event transition is shown as a bold arrow.

- An uncontrollable event transition is shown as a thin arrow.

7.1 Examples

In this section we provide some examples which fail key conditions that we require, in order to provide a better understanding of these conditions. The conditions we cover include plant completeness, activity loop free, proper time behavior, and SD controllability. We have not included examples for untimed controllability and nonblocking conditions since these two conditions are already well studied.

7.1.1 Plant Completeness

Figures 7.2 and 7.3 show a plant and a supervisor such that the plant fails to be complete for the supervisor, as per Definition 2.3.1. This is because event **repair.2** is not eligible at state **down** in the plant, while this event is eligible at state **down** in the supervisor. This could be a problem if event **repair.2** is being generated by the controller, and can occur whenever it is enabled. This would mean that the event could potentially occur when the plant model says it can't, resulting in unmodeled behavior.

Listing 7.1: Output

```
Checking proper timed behavior Condition...
CLowSub::VeriBalemiBad():306: iTick = 3
VERI_BALEMI: 0seconds.
(-206) State size of the synchronous product: 7
Number of bdd nodes to store the synchronous product: 20
Computing time: 0 seconds.
failed: proper timed behavior Condition checking failed at following state(s):
        <mach:down, sup:down>

Causing controllable event:repair.2
```

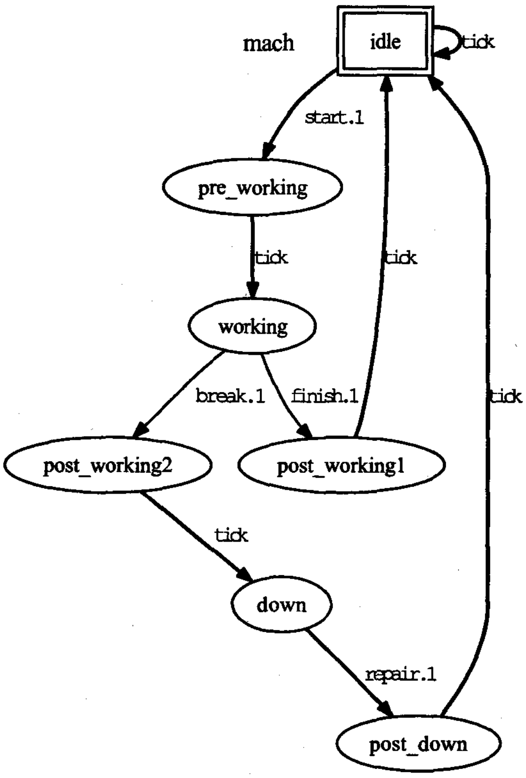


Figure 7.2: Plant Completeness Example: Plant

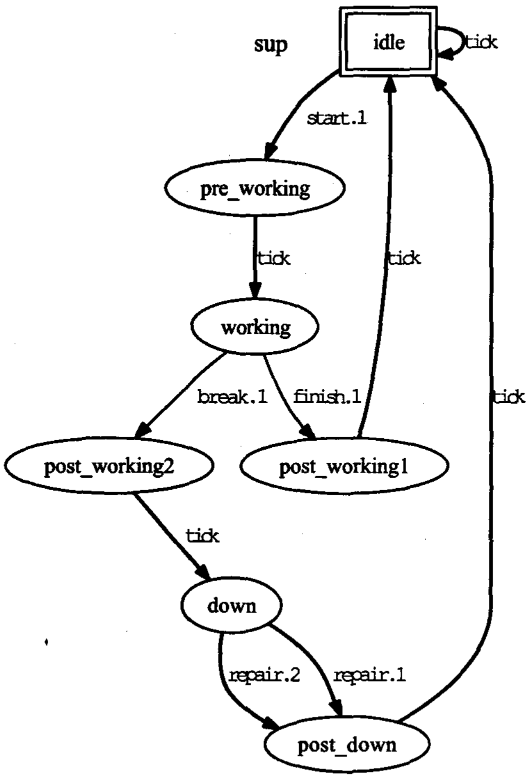


Figure 7.3: Plant Completeness Example: Supervisor

7.1.2 Activity Loop Free

Figure 7.4 shows a TDES which is not activity loop free, as per Definition 2.3.3. This is because at state **(b)** the event **down.1** is able to preempt the tick event and proceed to state **(c)** and after that to state **(a)**. This creates a tick-less cycle. This cycle of 'start.1-down.1-repair.1' can occur an unlimited number of times. This implies the physically unrealistic situation that we can have an infinite number of these events occur in a finite time period, and thus must not be allowed.

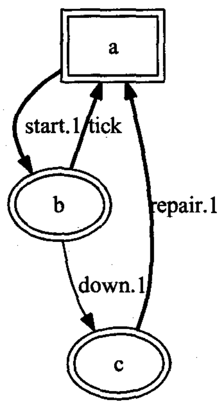


Figure 7.4: Activity Loop Example

7.1.3 Proper Time Behavior

Figure 7.5 shows a plant which fails to satisfy proper time behavior as per Definition 2.3.5. At state **down**, neither a tick event nor an uncontrollable event is eligible, just the controllable event *repair.1*. This causes two problems: First, it implies that the controllable event must occur in a particular time frame, yet the event can be disabled forever by a supervisor, and thus never occur. Second, because its controllable, it can be disabled by a supervisor. Since no other events are possible, if this event is disabled, we effectively "stop the clock", which is physically unrealistic. Note that supervisor could disable *repair.1* here and still be TDES controllable. i.e. this problem is not caught by the TDES controllability definition.

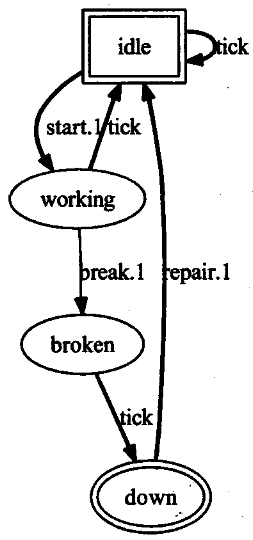


Figure 7.5: Proper Time Behavior Example

7.1.4 SD Controllability

We now examine the the various points of the SD controllability condition from Definition 3.2.2.

Point i and Point ii

As **Point i** and the '$\Leftarrow$' part of **Point ii** are essentially equivalent to the standard TDES controllability condition, we will not provide an example here for them. We will instead focus on the '$\Rightarrow$' part of **Point ii** as this is a new condition introduced bu SD Controllability.

Figure 7.6 and Figure 7.7 show a plant and a supervisor such that **Supervisor** fails to satisfy the '$\Rightarrow$' part of Definition 3.2.2, with respect to **Plant**. The prohibitable event is *job* and the uncontrollable events are *verified* and *done*. We first note that a tick event is eligible at state 3 in the **Plant**. Since the prohibitable event *job* is eligible at state **(Plant:3, Supervisor:3)** in the synchronous product, the supervisor should disable tick at its state 3 since a prohibitable event should only be enabled when it is to be forced. Alternately, if we do not yet wish event *job* to occur, it should be disabled until we are ready for it.

Listing 7.2: Output

```
Checking SD Controllability
VERI_SD: 0seconds.
(-209) State size of the synchronous product: 12
Number of bdd nodes to store the synchronous product: 38
Computing time: 0 seconds.
failed1: Failed SD Controllability condition II at state:
        <failed1_mach1:3, failed1_sup1:3>
```

Point iii.1

Figure 7.8 and Figure 7.9 show a plant and a supervisor such that **Supervisor** fail to satisfy **Point iii.1** of Definition 3.2.2 with respect to **Plant**. The only prohibitable event is *job*. The uncontrollable events are $\{verified1, verified2, done\}$.

In the system, prohibitable event **job** is eligible at sampling state 1 in the **Plant**, so the eligible prohibitable event set for this sampling period is $\{job\}$. However when we reach state 3, event *job* has not yet occurred, but is no longer eligible, violating **Point iii.1**.

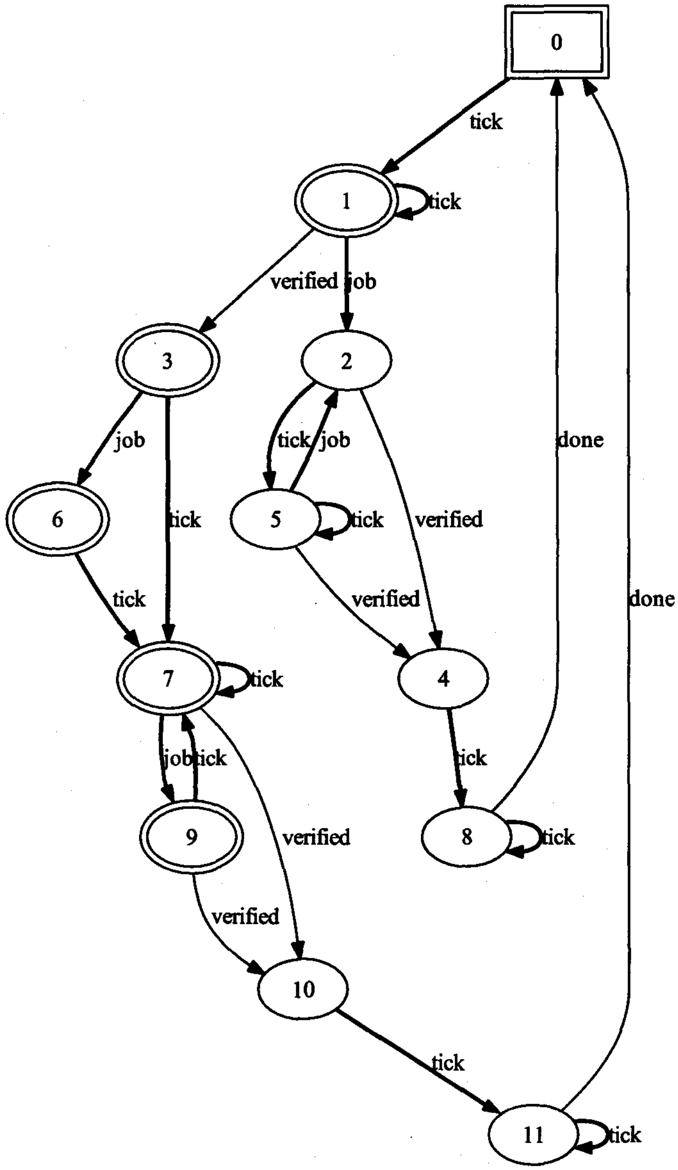


Figure 7.6: SD Controllability i, ii Example: Plant

This is a problem as often when a prohibitable event occurs is completely under the control of the implementation (as discussed before, this is a modeling issue). Also, this event may occur at different times during a sampling period, depending on the implementation used. As an SD controller makes its forcing decisions immediately after a tick, it will cause event *job* to occur at state 1 in the physical system. If the

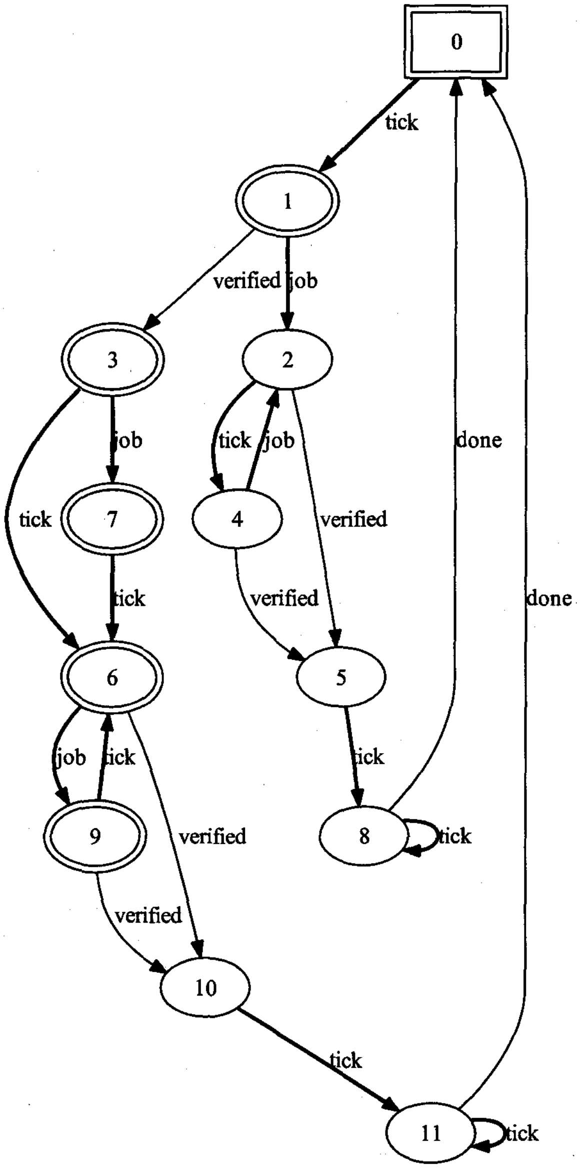


Figure 7.7: SD Controllability Point i, ii Example: Supervisor

implementation is such that event *job* is delayed and event *verified1* occurs first, we could get event *job* after event *verified1* in the physical system, which does not match our plant model.

In this example, it was the plant model that made event *job* become ineligible. A related issue would have been if event *job* was possible at state 3 in the plant, but not in the supervisor. This would imply that the SD controller must detect that event *verified1* has occurred in the current sampling period, and disable event *job* in time to prevent it from occurring. This of course cannot be done as the event has already been initiated after the tick occurred and even if could be stopped, the SD controller will not even see that event *verified1* has occurred until after the next tick, at which point it would be too late. If the implementation is such that event *verified1* occurs before event *job*, we would still get a job transition in the current sampling period in the physical system, violating our control law. For example, if event *job* was "walk through doorway", and event *verified1* was "door closes", this would mean we would walk into a closed door.

A second related problem this condition can catch is when a prohibitable event is not eligible at state 1, but becomes eligible at state 3. The supervisor is trying to express that the event should occur this sampling period, but not until after event *verified1* has occurred. This cannot be implemented as the SD controller would not know event *verified1* had occurred until after the next tick, thus too late to force a new event. If we tried to simply force the prohibitable event at state one in the controller, we might get the situation that the event occurs before event *verified1* (depending on our implementation). Again, this would violate our control law.

Point iii.2

Figure 7.10 and Figure 7.11 show a plant and a supervisor such that **Supervisor** fails to satisfy **Point iii.2** of Definition 3.2.2, with respect to **Plant**. The prohibitable events are $\{job1, job2\}$. The uncontrollable events are $\{done1, done2\}$.

In the system, states 6 and 7 are reached from sampled state 1 by concurrent strings $job1 - job2 - tick$ and $job2 - job1 - \tau$, respectively. As these strings have the same occurrence image, **Point iii.2** requires that states 6 and 7 represent the same Nerode equivalence cells of the closed loop system's closed and marked language's. However, as strings reaching state 6 can be extended by a *done1* event, while strings reaching state 7 can be extended by a *done2* event, the states clearly do not represent the same Nerode equivalence cell of the system's closed behavior. Similarly, as strings reaching state 6 can be extended by a *done1* event to a marked string while strings reaching state 7 can be extended by a *done2* event to a marked string, they do not represent the same Nerode equivalence cell of the system's marked language either.

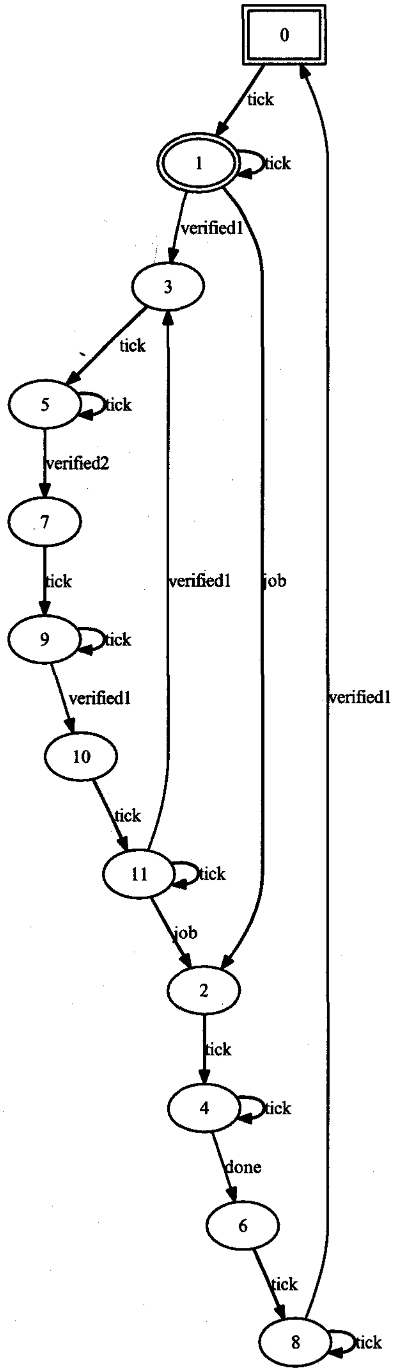


Figure 7.8: SD Controllability Point iii.1 Example: Plant

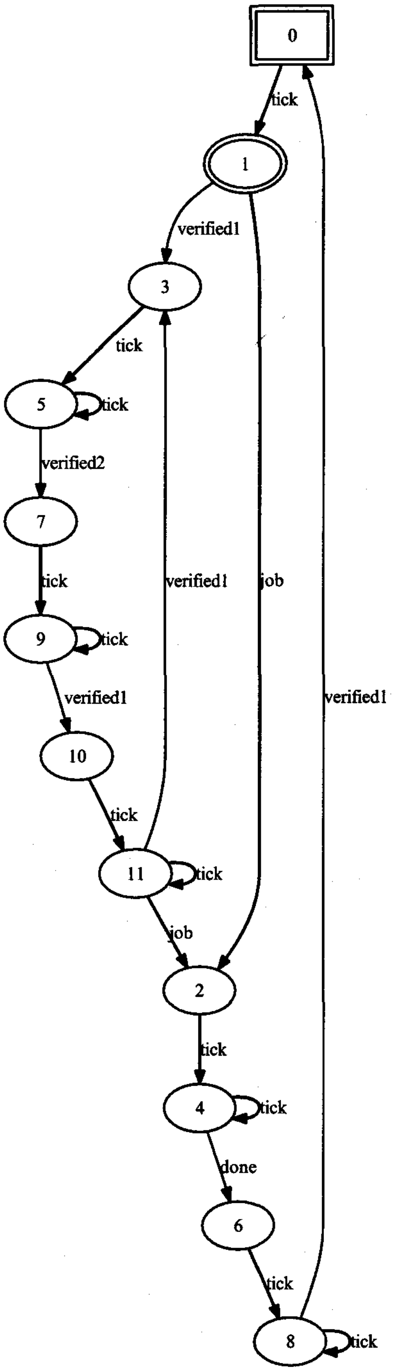


Figure 7.9: SD Controllability Point iii.1 Example: Supervisor

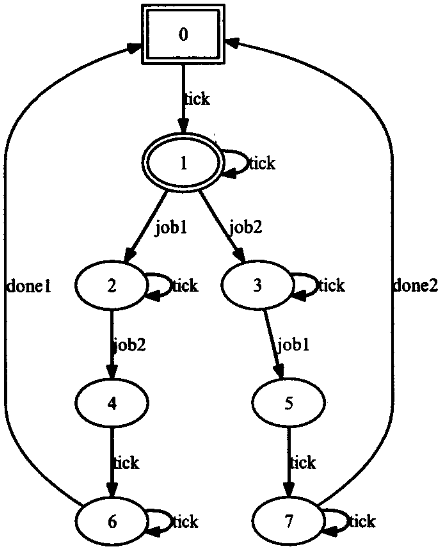


Figure 7.10: SD Controllability Point iii.2 Example: Plant

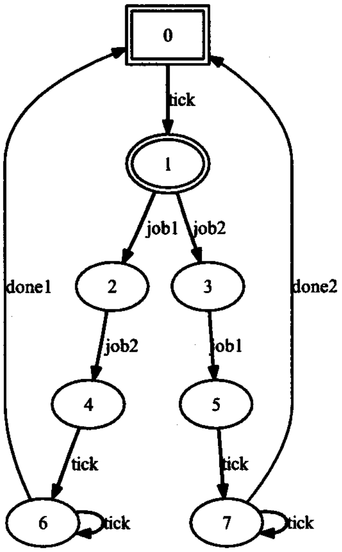


Figure 7.11: SD Controllability Point iii.2 Example: Supervisor

This condition is important for controllability and nonblocking. The reason is that an SD controller cannot tell the difference between the two concurrent strings, so it does not know whether it should be in state 6 or state 7. If events *done1* and *done2* were controllable, it would not know if it should be enabling event *done1* or event *done2*. Clearly, we could not enforce such a control law.

The reason this is important for nonblocking is also that we cannot tell the difference between the two strings. If we had a sequence of possible concurrent strings such that each pair had the same occurrence image and only one path of the pair ever reached a marked state, we would never be able to determine if our system reached a marked state.

A related issue is how our controller is implemented. The control law says that either sequence $job1 - job2$ or sequence $job2 - job1$ is fine, but not which one will actually occur. It might be that we will get a bit of both, but we might always get only one due to timing issues; or perhaps we have a sequential implementation that knows that $job1$ and $job2$ must occur, so its designers choose the order $job1 - job2$, and the implementation always executes these events in this order. If the sequence $job2 - job1$ was the only path back to a marked state, the implementation would block despite the fact the TDES system was nonblocking. This condition, in conjunction with **Point iv** of the SD controllability definition, helps make sure nonblocking does not depend on the order of the events and allows things to function if we only get one of the variations of the possible concurrent strings with the same occurrence image. One can image that we have a family of possible physical systems that we could get based on how we implement our controllers, each differing based on which of the possible variations of the concurrent strings can actually occur. We are assuming we will see at least one variation, possibly more. These conditions are intended to ensure that whichever system we get, it will still be nonblocking if the TDES system was nonblocking.

Listing 7.3: Output

```
...
(-211) State size of the synchronous product: 8
Computing time: 0 seconds.
failed1: Failed SD Controllability condition III.2 at state:
        <failed1_mach:6, failed1_sup:6>
        <failed1_mach:7, failed1_sup:7>

list_NerFail is not empty and RecheckNerodeCells() Failed.
```

Point iv

Figure 7.12 and Figure 7.13 show a plant and a supervisor such that **Supervisor** fails to satisfy **Point iv** in Definition 3.2.2 with respect to **Plant**. Since state **0** is a marked state and is reached from state 6 by activity event *done*, the system does not satisfy the condition as clearly its marked language is not a subset of the sampled strings (empty string and strings ending in a tick).

If a marked state is reachable by a non-tick event, it means the system can reach a marked state in a way that is invisible to the SD controller as it can only observe sampled strings. This by itself is undesirable, as we could have a system that can only reach marked states by non-tick events and we would never be able to tell if we had actually reached a marked state. Also, if we have multiple concurrent strings with the same concurrence image, we could have the situation that only some of them pass through a marked state in that sampling period. Worse, our implementation might be such that we only get the variations that do not pass through a marked state! Note also, that **Point iii.2** of the SD controllability definition only says that concurrent strings with same occurrence image must have same marked future. it does not say

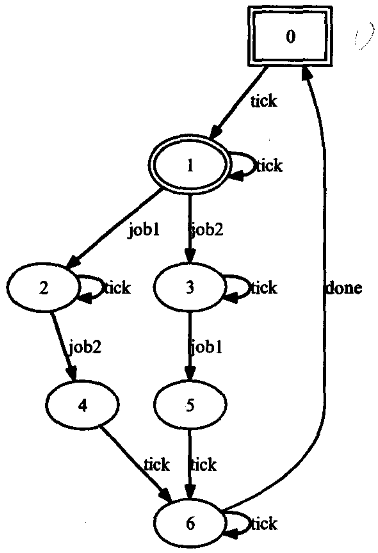


Figure 7.12: SD Controllability Point iv Example: Plant

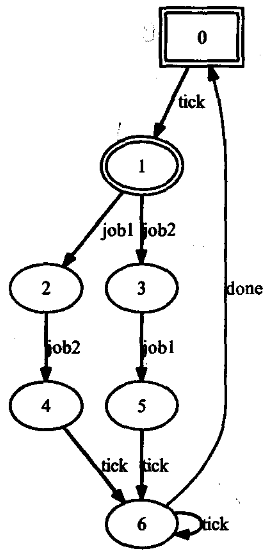


Figure 7.13: SD Controllability Point iv Example: Supervisor

much about the prefixes of these concurrent strings. That is where **Point iv** comes in, making sure the Σ_{act}^{+} prefixes are not marked.

Listing 7.4: Output

```
VERI_SD: 1seconds.
(-212) State size of the synchronous product: 7
Number of bdd nodes to store the synchronous product: 20
Computing time: 1 seconds.
failed1: Failed SD Controllability condition IV at state:
        <failed1_mach:0, failed1_sup:0>

There is a reachable marking state reached by a non-tick event.
```

7.2 SD Controlled Flexible Manufacturing System

In this section we present a working example based on the untimed Flexible Manufacturing System (FMS) from [11]. The system, shown in Figure 7.14, is composed of six plant components and five one slot buffers. We will treat the buffers as specifications, requiring that they do not overflow or underflow. Table 7.1 below shows a mapping from the event labels used in the diagrams to their meaning. The events labeled as numbers are directly from the Hill untimed example. We kept the same labeling to make it easy to see the correspondence.

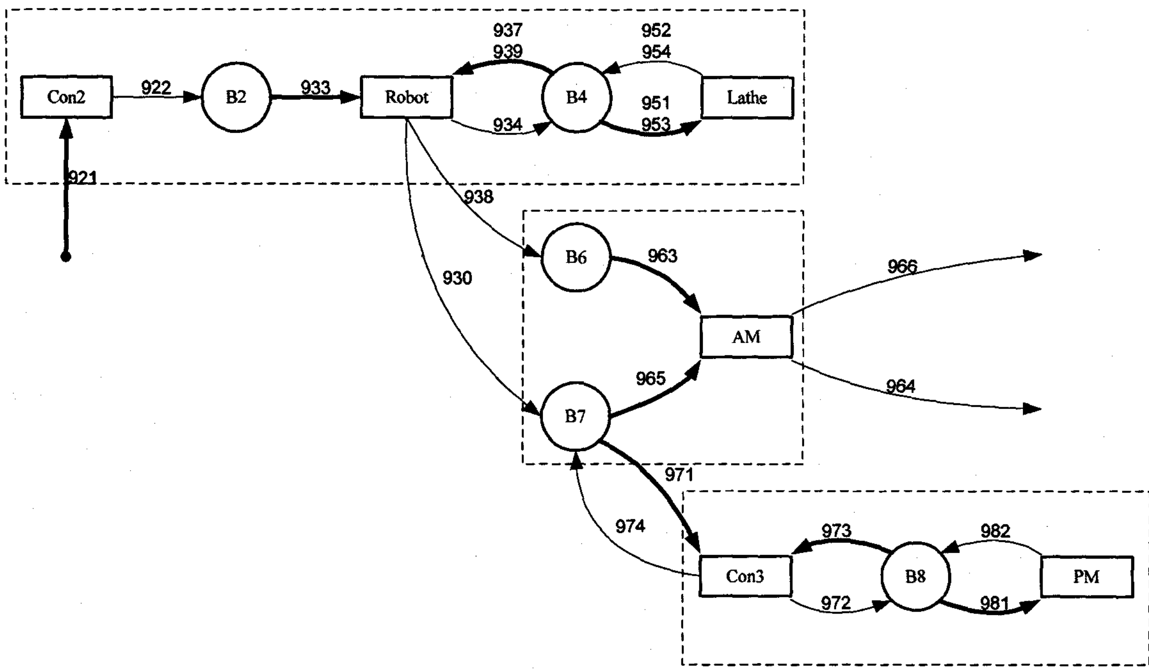


Figure 7.14: Flexible Manufacturing System Overview

7.2.1 FMS Plants

The plant components consist of two conveyors (**Con2** and **Con3**), a handling robot (**Robot**), a lathe that can produce two different parts, a painting machine (**PM**), and a finishing machine (**AM**). The flow of material is illustrated in Figure 7.14. See Figures 7.15 - 7.20 for the TDES models of the components.

Table 7.1: Explanation of Event Labels

Label	Meaning	Label	Meaning	Label	Meaning
921	Part enters system	*922*	Part enters B2	*933*	Robot takes from B2
934	Robot to B4	*937*	B4 to Robot for B6	*939*	B4 to Robot for B7
938	Robot to B6	*930*	Robot to B7	*951*	B4 to Lathe (A)
953	B4 to Lathe (B)	*952*	Lathe to B4 (A)	*954*	Lathe to B4 (B)
971	B7 to Con3	*974*	Con3 to B7	*972*	Con3 to B8
973	B8 to Con3	*981*	B8 to PM	*982*	PM to B8
961	Initialize AM	*963*	B6 to AM	*965*	B7 to AM
966	Finished from B7	*964*	Finished from B6		

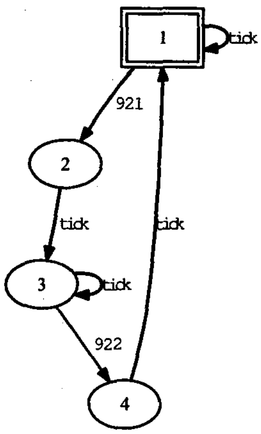


Figure 7.15: Conveyor - **Con2**

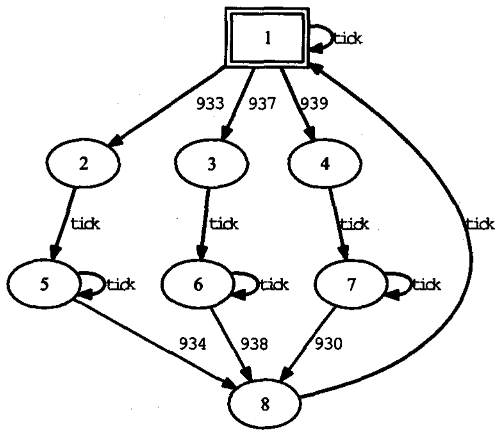


Figure 7.16: **Robot**

7.2.2 Buffer Supervisors

We now discuss the TDES supervisors, shown in Figures 7.21 - 7.25, that control the flow of parts in and out of the buffers. Their goal is to make sure the buffers do not overflow or underflow. They are based on the original untimed buffer specification of [11], but extended to the SD controllable setting. In some of the supervisors in this section such as **B4** in Figure 7.22, we have activity events selflooped (i.e. event *933* at state 0 of **B4**). This will not cause the system to have an activity loop, as it will be combined with the plant TDES which only allow these events to occur once per clock cycle.

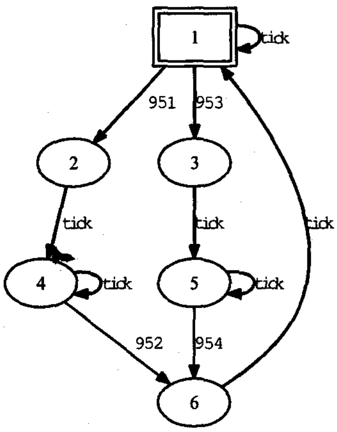


Figure 7.17: **Lathe**

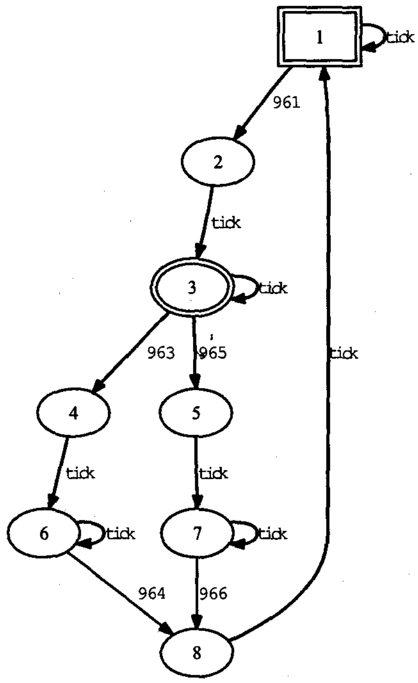


Figure 7.18: Finishing Machine - **AM**

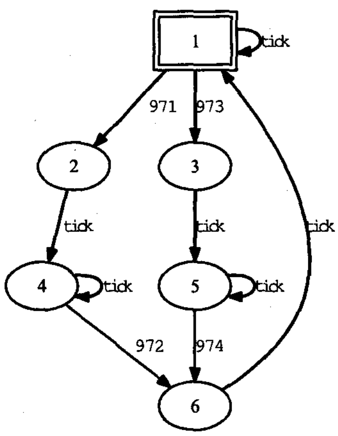


Figure 7.19: Conveyor - **Con3**

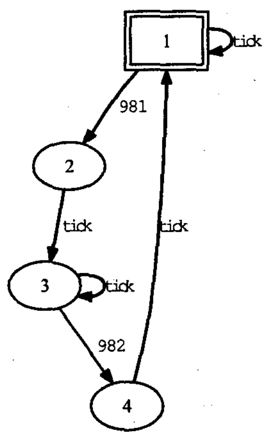


Figure 7.20: Painting Machine - **PM**

Supervisor **B2** not only prevents overflow and underflow of buffer **B2**, it also decides when event *921* should occur. As soon as the system is turned on, it immediately enables and forces *921*, causing **Con2** to accept a new piece into the system. It then waits for the piece to enter **B2**, before it enables event *933*, allowing the

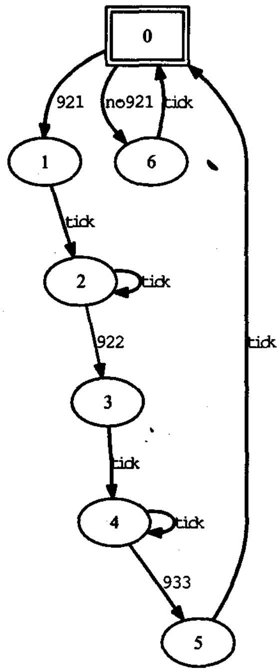


Figure 7.21: Supervisor **B2**

Robot to remove the part. It does not cause another *921* to occur until *933* does, ensuring that the buffer is empty. A few things are worth noting. First, **B2** enables prohibitable event *933*, but does not disable the tick at state 4. This tells us that it wants to prevent the event from occurring too soon, but does not decide when the event will actually occur. This is controlled by another supervisor. Second, **B2** makes sure there is a tick between *933* occurring, and enabling and forcing event *921*. This is to satisfy **Point iii.1** of the SD controllability definition. Third, Supervisor **B2** contains a special event, *no921*, which we will discuss in a later section. This is a "virtual event" that was not part of the original plant, but that we added to aid in communication between supervisors.

Supervisors **B4**, **B6**, and **B7** manage their respective buffers. They strictly disable and enable events to prevent buffer overflow and underflow. They do not force any events, telling us that other supervisors make these decisions. This is because the decision of when these events should occur requires more than just a local view of whether a buffer is empty or not. We will discuss these other supervisors in later sections.

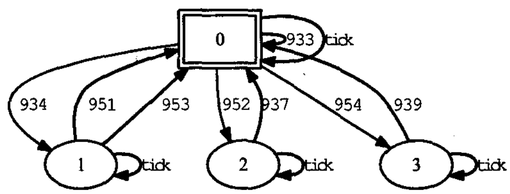


Figure 7.22: Supervisor **B4**

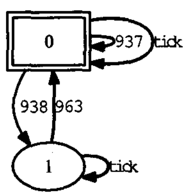


Figure 7.23: Supervisor **B6**

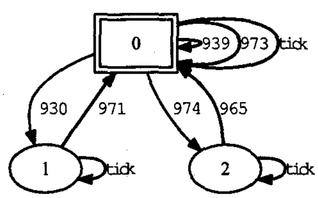


Figure 7.24: Supervisor **B7**

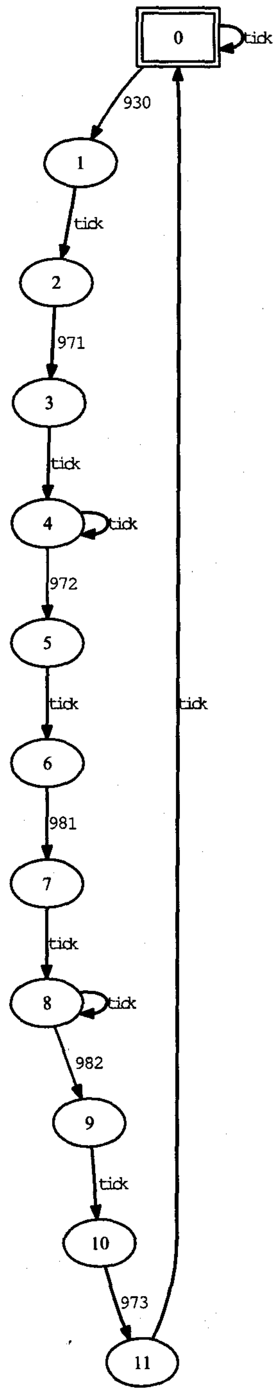


Figure 7.25: Supervisor **B8**

Supervisor **B8** not only prevents overflow and underflow of buffer B8, it also controls the flow of pieces once a part enters buffer **B7** (event *930*), flows to TDES **PM**, and then back to buffer **B7**. It does this by watching the parts progress, and then forcing events *971*, *981*, and *973* as needed. As **B8** determines when these events occur, it disables tick as soon as it enables these events to comply with **Point ii** of the SD controllability definition. In other words, once the event is enabled by all the supervisors and possible in the plant, the event is also forced.

The fact that the supervisor must not only decide when to enable an event, but also when to force the event, makes things more complicated. It must not only decide when to enable the event, but also must know that the event is not disabled by another supervisor and that it is eligible in the plant. Otherwise, it could disable a tick when the desired event cannot occur, either forcing the wrong event or becoming uncontrollable.

7.2.3 B4 to Lathe Path

In addition to the buffer supervisors we represented in Section 7.2.2, we need to add the following supervisors to resolve some nonblocking and concurrency issues on the **B4** to lathe part path of Figure 7.14.

We first need to address a nonblocking issue with respect to buffer **B4** and **B2**. We see from Figure 7.14 and Figure 7.16, that **Robot** takes a piece from buffer **B2** (event *933*), and places it in **B4**. The piece then goes to the **Lathe**, and then back to buffer **B4**. The robot will then take the piece from **B4**, and put it in either buffer **B7** (event *930*), or buffer **B6** (event *938*).

There are two issues here. The first issue is how to decide which action the **Robot** should take if both buffer **B2** and buffer **B4** have a part waiting. In normal supervisory control theory, we can just enable the safe choices, and allow the plant to somehow make the decision. However, we want to be able to convert from a TDES supervisor to an SD controller in an easy, deterministic fashion. This means we must dictate which prohibitable events occur, and in which sampling period they occur in. We thus have to choose to service either buffer **B2** or **B4**, as we cannot do both at the same time.

This issue is handled by supervisor **TakeB2**, shown in Figure 7.26. It forces

Robot to first service buffer **B2**, then buffer **B4**, then back to buffer **B2**. It waits until there is a piece in **B2** (event *922*), then it immediately enables and forces event *933* to move the piece to buffer **B4**. It then waits until the piece goes to the **Lathe**, returns to **B4**, and then moved to either **B6** or **B7**, before it allows the **Robot** to service **B2** again.

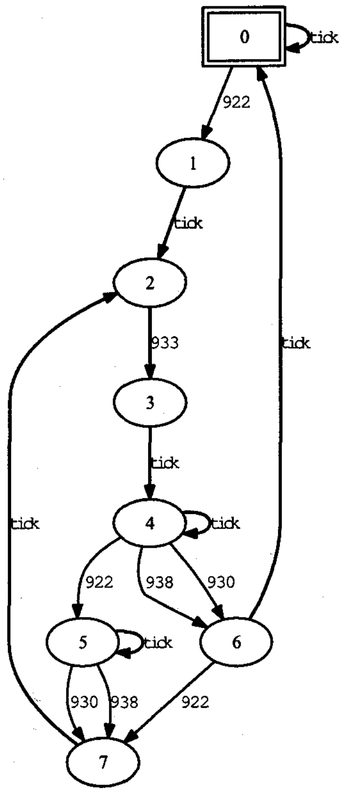


Figure 7.26: **TakeB2**

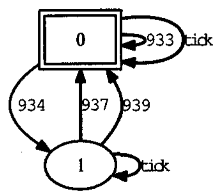


Figure 7.27: **B4Path**

The second issue is to prevent a conflict with respect to buffer **B4**. Once the **Robot** puts a piece in **B4** and the piece is taken by the **Lathe**, the **Robot** could put a second piece in **B4**. This would mean the **Lathe** has no place to return its part, and the system blocks. **TakeB2** prevents this by disabling event 933 until the current part has returned to **B4**, and then removed to either **B6** or **B7**.

We now discuss supervisor **B4Path**. It works with buffer supervisor **B4** to ensure the proper behavior of the **B4** to lathe part path. Supervisor **B4** primarily ensures that buffer **B4** does not overflow or underflow. It serves an additional role in making sure that once a piece is put in **B4**, the correct action is taken when it is taken out. When the robot initially puts a piece in **B4** (event *934*), it makes sure that only events *951* and *953* can be used to take the piece out. This ensures the part goes to the **Lathe** for processing. The **Lathe** can process the piece as type A (event *951*) or type B (event *953*), producing different results. The **Lathe** then returns the part to **B4** using events *952* (part is type A) or event *954* (part is type B). Since type A parts go to buffer **B6** (events *937* then *938*), and type B parts (event *939* then *930*) must go to buffer **B7**, supervisor **B4** ensures only the correct follow up event is possible. **B4Path** contributes to the proper behavior of the **B4** to lathe path, by disabling event *933* once a part is put into **B4** from **B2**, and disabling events *937* and *939* until a part is placed into **B4** from **B2**.

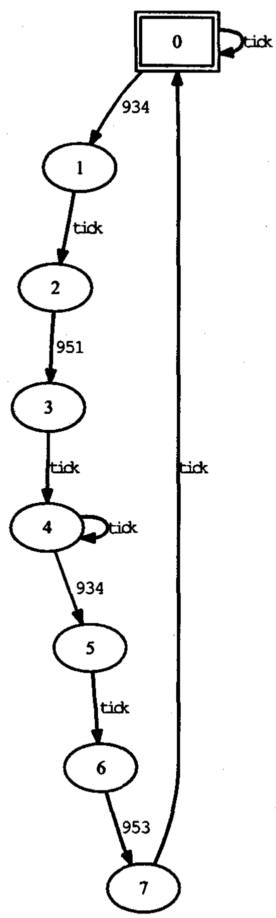


Figure 7.28: **LathePick**

Supervisor **LathePick**, shown in Figure 7.28, also contributes to control of the **B4** to lathe part path. To satisfy **Point ii** of SD controllability, we cannot just enable both event *951* and *953* and let the system "decide." We have to dictate when these events are to occur. That means we have to make a choice. In **LathePick**, we have required that the **Lathe** first produce a type A part, then a type B part, and then alternate. Note that the supervisor has enough information to know when the events are possible in the plant, so it does not try to force them at the wrong time, possibly "stopping the clock."

7.2.4 Moving Parts from B4 to B6/B7

We now discuss some concurrency and blocking issues involved with moving pieces from buffer **B4**, to either buffer **B6** or **B7**. To move a part from buffer **B4** to **B6**, we use event *937*. To satisfy **Point ii** of SD controllability, we need to decide when to enable and force this event. This is handled by supervisor **TakeB4PutB6**, shown in Figure 7.29. It waits for event *952* to occur, which signifies a piece of type A is ready to be transferred to buffer **B6**. It forces event *937* and then waits for event *963* to occur, signifying that the piece has been taken by **AM** and that **B6** is ready for a new part.

We now consider moving a part from **B4** to **B7**. We do this using event *939*. We have to decide when to force *939* in order to satisfy **Point ii** of SD controllability, but we also have to deal with a potential blocking situation. Because a part placed in **B7** first goes to **PM** for processing, it is possible that the robot could put a part in the now empty buffer **B7**, leaving no place for the first part to return to. Supervisor **TakeB4PutB7**, shown in Figure 7.30, handles both issues. It watches for event *954* to occur, signaling that a part of type B has been placed in **B4**, and is ready to be transferred to buffer **B7**. **TakeB4PutB7** forces event *939* to make the transfer. It then waits for event *965* to occur signaling that **AM** has removed the part from **B7**, before allowing another *939* to occur, thus preventing blocking.

7.2.5 AM to Exit Path

We now discuss the paths from **B6** and **B7**, leading through machine **AM** and then to where the parts exit the system. We have several concurrency issues to deal with here.

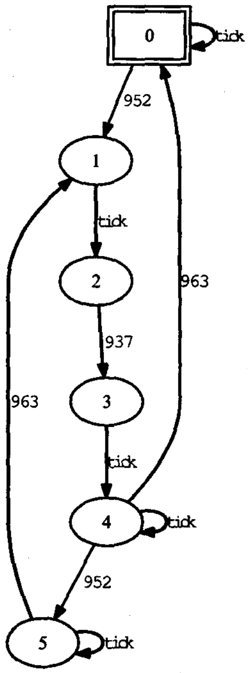


Figure 7.29: **TakeB4PutB6**

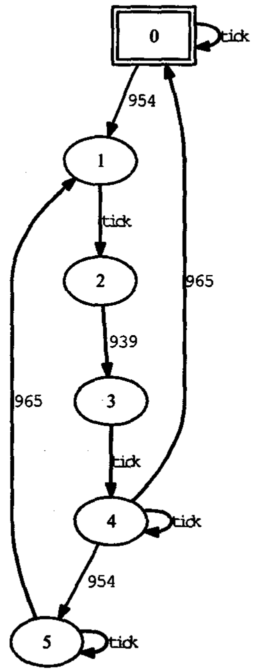


Figure 7.30: **TakeB4PutB7**

First, we have to specify when prohibitable events *961*, *963*, and *965* are suppose to occur in order to satisfy **Point ii** of SD controllability. This is complicated by the fact that a piece could be waiting for **AM** in both **B6** and **B7**, so we need to specify how to choose which buffer to service first.

The problem is that these three events are linked and we have to keep track of several issues in order to decide when to force which event. We could create a single supervisor to do this, but it would be quite large and complicated, thus difficult to design correctly. It would be nice to be able to design several modular supervisors. If we were only enabling and disabling events, this would not be that hard. However, since we must decide when to force the events, we have to make sure we do not try to force an event when it is not possible in the plant, or disabled by another supervisor. It was very non-obvious how to do this modularly, without significant reuse of logic from other supervisors.

We then came up with the solution of using prohibitable "virtual events" *no963a*, *no963b*, *no965a*, and *no965b*. We introduced these new events to the system by adding plants **AddNo963** and **AddNo965**, shown in Figures 7.31 and 7.32. Note that we made sure the plants specify that these events can only occur once per sampling period, so that we do not have to specify this in our supervisors.

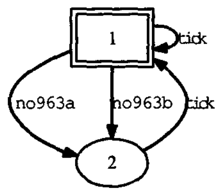


Figure 7.31: Plant **AddNo963**

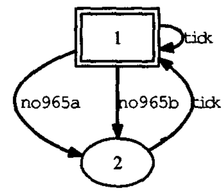


Figure 7.32: Plant **AddNo965**

Let's first discuss how to handle event *963*. The idea is that when we want to disable the tick to force event *963* in one supervisor, events *no963a/b* can be used as an alternate event to force if event *963* is disabled by another supervisor, or not possible yet in the plant. The other supervisors will only enable event *no963a* or *no963b* when they know *963* is not possible, and they will make sure only one of the three events are possible at a given time. The reason there is an 'a' and 'b' event is that there are three supervisors with which we need to coordinate enablement information. This will become clear later.

The primary supervisor for event *963* is **Force963**, shown in Figure 7.33. It watches for event *938* to occur, signifying that there is a part in **B6** waiting to go to **AM**. The supervisor then disables the *tick* to force *963*. Note, that it is the only supervisor that tries to force this event. However, event *963* could be ineligible in plant component **AM**, or disabled by supervisors **Force961** or **AMChooser**, the latter two TDES shown in Figures 7.34 and 7.35. **Force963** has no way of knowing this. It handles this by adding the *no963a/b-tick* loop at state 2. Supervisors **Force961** and **AMChooser** will ensure that out of events *963*, *no963a*, and *no963b*, one and only event will be eligible and enabled immediately after a *tick*. If *963* is ineligible or disabled, then *no963a* or *no963b* gets forced instead, and then we try again after the *tick*. This way we signal we want *963* to occur as soon as it can, but do not stop the clock. We also do not need to repeat information from the plant and other supervisors about when these events are eligible/enabled.

The reason that only one of the three events are ever allowed to be eligible/enabled at the start of a *tick*, is to avoid violating **Point iii.1** of the SD controllability definition. Examining state 2 of **Force963**, we see that once one of the three events occurs, the others are disabled. If more than one was enabled and eligible at state 2, this would cause one of them to change eligibility status between *ticks*, violating **Point iii.1** of the SD controllability definition.

For event *965*, we have similar behavior represented by supervisor **Force965**, shown in Figure 7.36. It interacts in a similar way with plant component **AM**, and supervisors **Force961** and **AMChooser**.

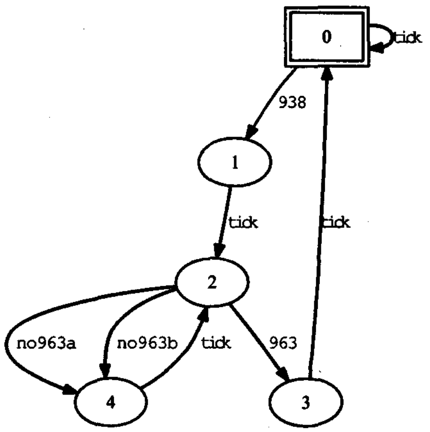


Figure 7.33: **Force963**

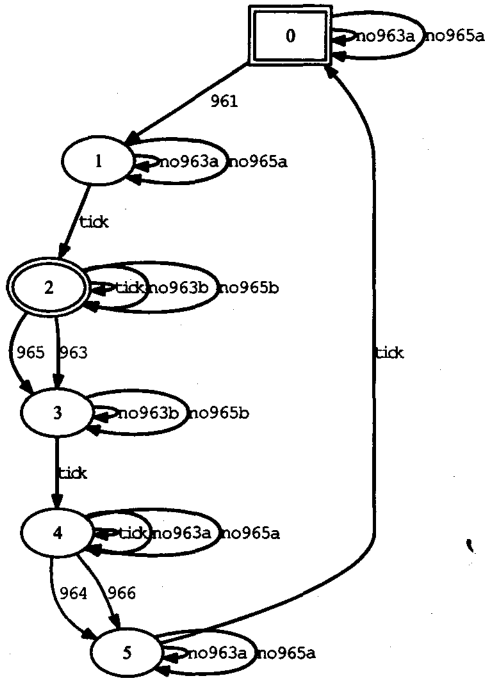


Figure 7.34: **Force961**

We now discuss supervisor **Force961**, shown in Figure 7.34. Its primary task is to determine when to force event *961* which readies **AM** to process parts. **Force961** forces *961* right away, and then waits for events *964* or *966* (signifies **AM** has finished processing the part) to occur, before forcing event *961* again.

The secondary task of **Force961** is to only enable events *no963a* and *no965a* when events *963* and *965* are not possible in the plant component **AM**. When they are possible in the plant, **Force961** enables *no963b* and *no965b* instead. This insures that events *no963a* and *no965a* will always be possible after a tick when events *963* and *965* are ineligible in the plant. It also ensures that the 'a' and 'b' events are never

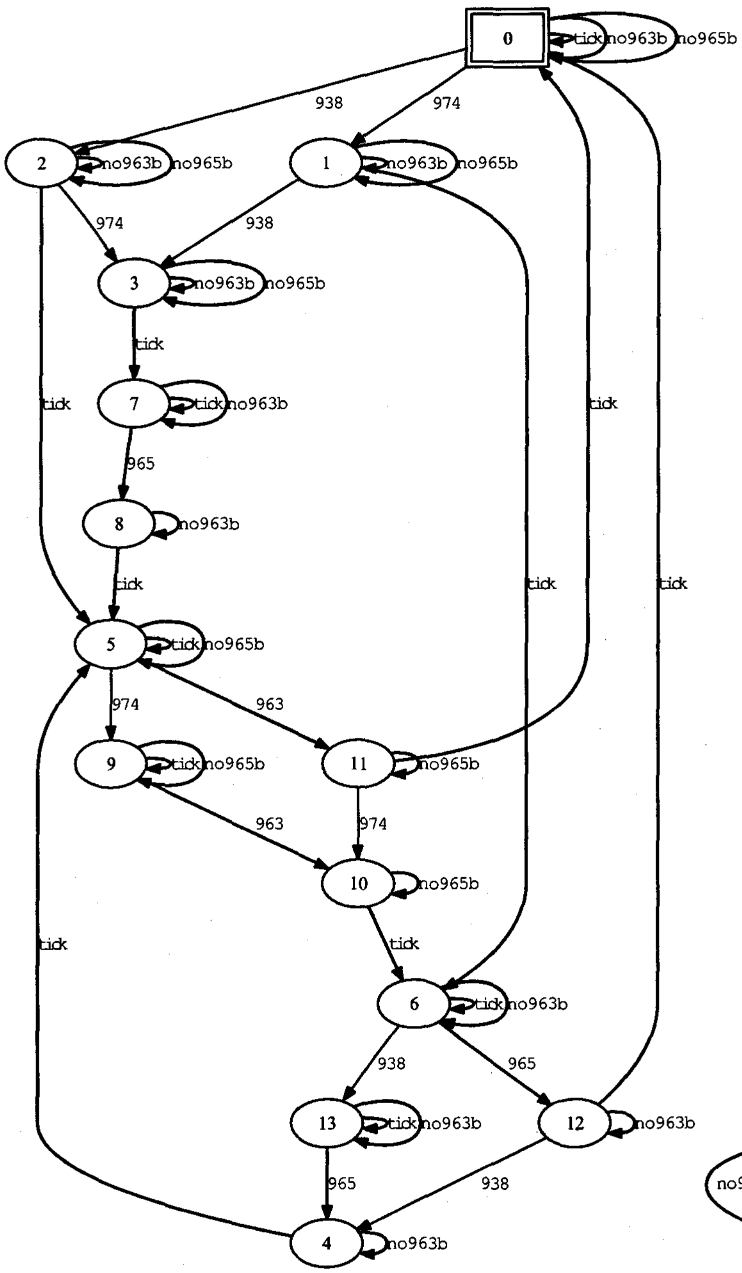


Figure 7.35: **AMChooser**

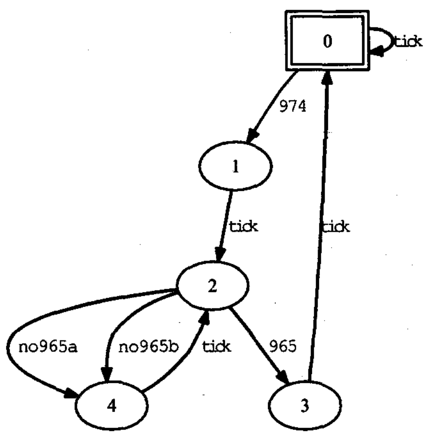


Figure 7.36: **Force965**

eligible at the same time. Also, as supervisor **AMChooser** ignores the 'a' events, they will never be disabled when **Force961** needs them. As **Force961** never disables the 'b' events when *963* and *965* are possible in the plant, this ensures that they will not be disabled when **AMChooser** needs them. This means the two supervisors do not interfere with each other with respect to these events.

Finally, please note that when we switch from the 'a' to the 'b' events in **Force961**, we only do so immediately after a *tick* (consider states 1 to state 4 as an example). This is to not violate **Point iii.1** of the SD controllability definition.

We now consider our last supervisor for this section, **AMChooser**, shown in Figure 7.35. The role of this supervisor is to choose between taking a piece from buffer **B7** (event *965*) or buffer **B6** (event *963*), when both have a waiting part. If both receive a part in the same sampling period, we take the piece from buffer **B7** first as there are other machines to keep busy along the **B7** to PM path. We then take a piece from **B6**. If there is already a new piece from **B7** waiting, we continue in an alternating fashion. If there is only one piece waiting in a given sampling period, then we handle that piece. Because **AMChooser** sometimes disables event *963* or *965* in order to enforce this order, it enables the appropriate *no963b* or *no965b* event as a forcing substitute. It also ensures that event *963* and *no963b* are never enabled at the same time. It behaves similarly for events *965* and *no965b*.

7.2.6 System Shutdown

When we tested the previous supervisors (excepting supervisor **B2** originally did not have its state 6, plant component **AM** was not marked at its state 3, and supervisor **Force961** was not marked at its state 2) we found that the system was blocking. It was not that the system was deadlocking or not completing its tasks, it was simply the fact that, due to forcing events as soon as they were ready, the entire system was never in a marked state at the same time. We could have delayed some events to achieve this, but that would have been less efficient.

The real cause was the fact that the system did not have a shutdown mechanism. Once started, it just kept running. A shutdown mechanism would cause the system to empty out, allowing a non-deadlocked/livelocked system to return to its idle state. The easiest way to cause the system to go idle, is to prevent plant component **Con2** from taking new parts (event *921*). Once new parts stopped coming in, the system would process the existing ones, allow them to leave, and then the TDES should return to their idle states which are marked.

To achieve this, we added a new plant component **SystDownNup**, shown in Figure 7.37. It contains an event *shutdown* to empty the system, and an event *restart* to bring the system back up. This could correspond to a physical switch an operator could throw to control this behavior.

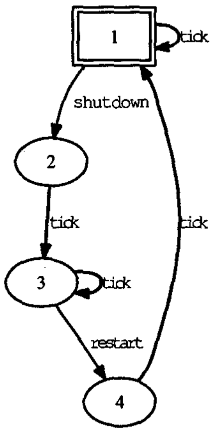


Figure 7.37: Plant **SystDownNup**

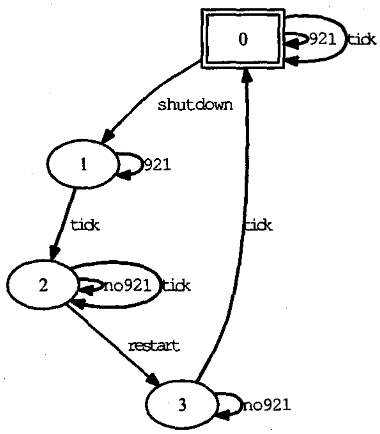


Figure 7.38: Supervisor **handleSystDown**

Our next task was to stop new pieces from entering the system. The problem was that supervisor **B2** forced event *921*, causing **Con2** to take a new part, as soon as buffer **B2** was empty. As we wanted to keep supervisor **B2** simple, we added a new prohibitable "virtual event," *no921*. This was introduced by adding plant **AddNo921**, shown in Figure 7.39. We then added the *no921-tick* loop at state 0 of supervisor **B2**. We would use event *no921* as an alternate event to force, when we disabled event *921*.

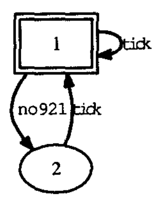


Figure 7.39: Plant **AddNo921**

Finally, we added supervisor **handleSystDown**, shown in Figure 7.38. Its job was to enable event *921* and disable *no921* initially, and then disable *921* and enable *no921* once the *shutdown* event occurs. When the *restart* event occurs, the process is reversed. We also make sure events *921* and *no921* are never enabled at the same time, and that one of the two are always eligible and enabled immediately after a *tick*.

However, after the above, we were still ~~non~~blocking. The culprit was supervisor **Force961**. As soon as event *961* was eligible, it was forced so that **AM** was ready to process a part. We could have created a *no961* event like we did for **B2**, but this would have been trickier as we needed to allow enough *961* events to occur to allow the existing pieces to leave. Rather than do this, we decided that for **AM**, state 3 was a rest state, and it was fine to leave it there. So, we marked state 3 of **AM**, and state 2 of **Force961**, and the system was nonblocking. Note that we could have marked state 2 of TDES **AM**, and state 1 of TDES **Force961**, but that would have caused **Point iv** of the SD controllability definition to fail.

7.2.7 Algorithm Runtime Statistics

To test the performance of the algorithm on this example, the following machine configuration was used:

- 1.8GHz Dual core AMD processor
- 4GB of Dual channel DDR2 RAM
- Cygwin 1.5.25-15 with gcc version 4.3.2

For testing purpose, the source code is compiled with `-O3` optimization[1].

As we can see from the log output for the FMS example, shown in Listing 7.5, our supervisor **S** is SD controllable for our plant. We also see that our plant has proper time behavior, is complete for our supervisor, and has S-singular prohibitable behavior. Finally, we see that our closed loop system is ALF and nonblocking. From the log, the total number of states of the synchronous product is 82,608. The verifications take about 2 minutes and 51 seconds. The memory usage is around 183 megabytes at the highest point, as shown in Figure 7.40. For the input files of all the DES in this example, please see the appendix.

[1] More information can be found by running `man gcc`.

Listing 7.5: Output

```
*************************************************
  Bdd-based TDES Verification Tool
*************************************************
  L - Low Level verification
  F - File the current project
  C - Close the current project
  Q - Quit
*************************************************
Current Project: FMS_1.sub

Procedure desired:
Show the blocking type(may take long time)(Y/N)?
Verbose level (0 - disable, 1 - brief, 2 - full)?

Computing reachable subpredicate...
R: Iteration_1 nodes: 120       time: 0 s       states: 10
R: Iteration_2 nodes: 586       time: 0 s       states: 77
R: Iteration_3 nodes: 1754      time: 0.031 s   states: 772
R: Iteration_4 nodes: 2801      time: 0.093 s   states: 4531
R: Iteration_5 nodes: 3265      time: 0.172 s   states: 26540
R: Iteration_6 nodes: 3310      time: 0.109 s   states: 48300
R: Iteration_7 nodes: 2281      time: 0.094 s   states: 58068
R: Iteration_8 nodes: 2387      time: 0.062 s   states: 62420
R: Iteration_9 nodes: 2132      time: 0.047 s   states: 68242
R: Iteration_10 nodes: 1983     time: 0.047 s   states: 76780
R: Iteration_11 nodes: 1546     time: 0.015 s   states: 82128
R: Iteration_12 nodes: 1330     time: 0.016 s   states: 82608
R: Iteration_13 nodes: 1330     time: 0 s       states: 82608
R: 0seconds.
bddReach states:82608
bddReach Nodes:1330

Verifying controllablity...
VERI_CON: 0seconds.
Verifying Nonblocking...
CR: Iteration_1 nodes: 191      time: 0 s       states: 24
CR: Iteration_2 nodes: 357      time: 0.016 s   states: 70
CR: Iteration_3 nodes: 488      time: 0.015 s   states: 190
CR: Iteration_4 nodes: 540      time: 0.016 s   states: 394
CR: Iteration_5 nodes: 785      time: 0.031 s   states: 540
CR: Iteration_6 nodes: 1143     time: 0.047 s   states: 773
CR: Iteration_7 nodes: 1757     time: 0.093 s   states: 3545
CR: Iteration_8 nodes: 2805     time: 0.281 s   states: 28173
CR: Iteration_9 nodes: 2080     time: 0.203 s   states: 47358
CR: Iteration_10 nodes: 2048    time: 0.172 s   states: 67045
CR: Iteration_11 nodes: 1552    time: 0.109 s   states: 81732
CR: Iteration_12 nodes: 1330    time: 0.031 s   states: 82608
CR: Iteration_13 nodes: 1330    time: 0.047 s   states: 82608
VERI_NONBLOCKING: 2seconds.
Checking Plant Completeness...
VERI_BALEMI: 0seconds.
Verifying Activity Loop Free...
Garbage collection #1: 2000003 nodes / 1996580 free / 0.1s / 0.1s total
VERI_ALF: 7seconds.
Verifying Proper Timed Behavior...
VERI_PTB: 0seconds.
Checking SD Controllability
VERI_SD: 162seconds.
(0) This system has been verified succesfully!
State size of the synchronous product: 82608
Number of bdd nodes to store the synchronous product: 1330
Computing time: 171 seconds.
Total computing time:171 seconds.
```

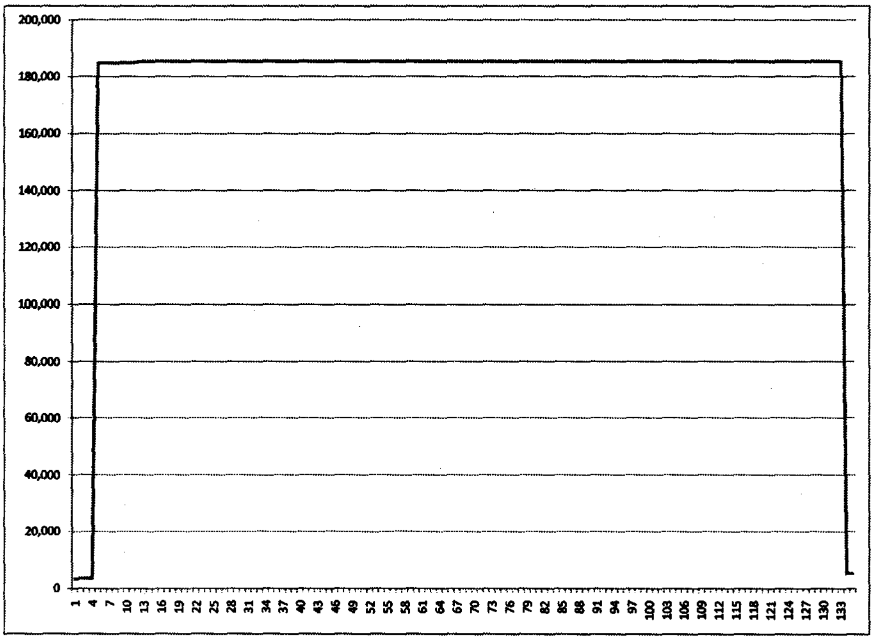


Figure 7.40: Histogram for Memory Usage (Kbytes vs. seconds)

7.3 Translating FSM Supervisors to Moore FSM

In Section 7.2, we presented an example of a Flexible Manufacturing System with SD controllable TDES supervisors. In this section, we apply the method in Section 4.2 to translate individual FMS supervisors into Moore finite state machines (FSM) (see Section 4.1). This is possible because our supervisor is SD controllable, and our plant is complete for our supervisor. If the plant was not complete, we would have had to use additional information from the plant components to determine when the problematic prohibitable events were not possible in the plant. This can be accomplished by converting the plant components that contain the needed information into FSM as well, and combining them with the FSM for the supervisors as modular controllers.

7.3.1 Adding More Timing Information

Before we can translate the individual TDES supervisors into FSM, they must be CS deterministic as in Definition 3.1.5 and non-selfloop ALF. A TDES is non-selfloop ALF if once any activity selfloops are removed, the resulting TDES is ALF. For example, supervisor B4 in Figure 7.22 is neither CS deterministic or non-selfloop ALF. This is a problem as the possible next state transitions of the FSM are too numerous, and many of them are not actually possible in the plant. For example, we could have according to the TDES a *934-tick* sequence, a *934-951-tick*, or even a $\{$*934-951*$\}^*$-*tick* sequence. We simply have too many choices, and this would result in an overly complex FSM. Also, concurrent strings *934-951-tick* and *934-951-934-tick* have the same occurrence image but lead to different states, which would result in a nondeterministic controller. Examining the plant and other supervisors, we see that there will always be a tick between events *934* and *951*, so we can add this to TDES **B4**, as we have done in Figure 7.41.

Making similar observations for the other non-selfloop activity loops, we get the supervisor in Figure 7.41 which should provide us with the same over all closed loop behavior as the original **B4** supervisor. However, we note that prohibitable event *933* is still selflooped at state 0, so the TDES is not ALF. We could modify the supervisor to remove this loop, but we do not need to as the selfloop provides enablement information, but does not affect the next state information. As such, it does not impede our translation. i.e. our next state information is $\{$*933*$\}^*$-*934-tick* to state

1 of **B4**. Essentially, as long as the supervisor is CS deterministic and non-selfloop ALF, we can do the translation. As was discussed in Chapter 4, all we require is that the TDES be CS deterministic, but typically if the TDES is not non-selfloop ALF it will also not be CS deterministic. Also, it is often difficult to even check the CS deterministic condition if the TDES is not non-selfloop ALF.

We then made similar changes to supervisors **B6**, **B7**, and **B4Path**. The new supervisors are shown in Figures 7.42 - 7.44. All remaining supervisors can be converted directly. We reran our software on the FMS system with these new supervisors, and all conditions still passed.

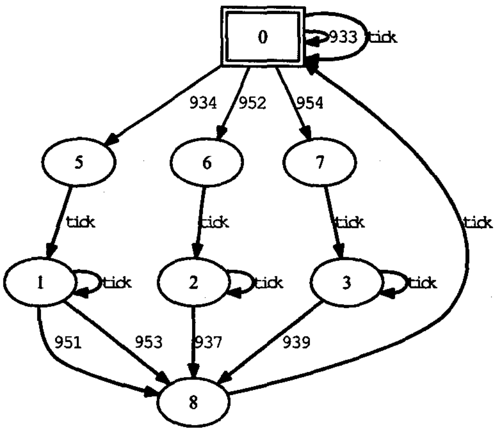


Figure 7.41: New B4

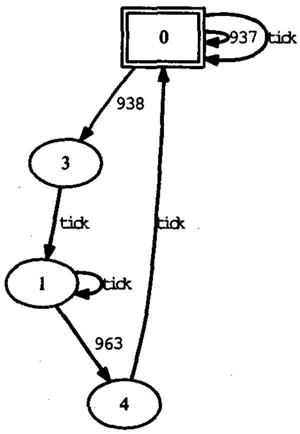


Figure 7.42: New B6

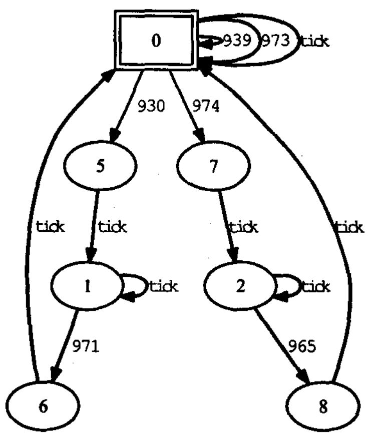


Figure 7.43: New B7

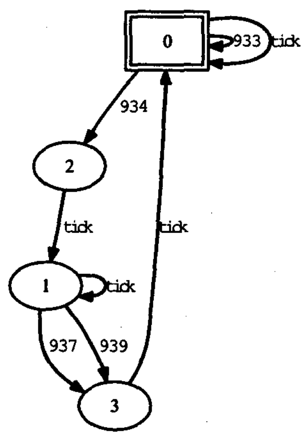


Figure 7.44: New B4Path

7.3.2 FSM Controllers for Flexible Manufacturing System

This section lists all the FSM Controllers for the Flexible Manufacturing System we presented in Section 7.2 and 7.3.1, using the method developed in Section 4.2. We first briefly discuss some implementation and modeling details, as well as introduce some notation that we will use.

Each FSM samples its inputs on the clock edge when tick occurs, and then changes state based on its current state, the value of each relevant input, and the next state arcs for that state. The timing info is implicit as it only changes state on a clock edge. If an input for an event is true when sampled on the clock edge, then it is considered to have occurred during the last clock period. The designer must make sure that the input for a given value has a pulse length equal to the period of our clock so that the input will not be lost. If an input is seen at two clock edges in a row, it is considered to have occurred twice. As such, the designer must make sure an input does not have an overly long pulse length. Remember, except for one exception, an event is considered to occur when its input goes true at the controller. The exception is when the input goes true so close to a sampling edge it is detected in the next sampling period, then it is considered to have occurred in the next clock period. This should be taken in to account in modeling the system.

To represent the FSM visually, the following notations are defined. For the given FSM,

- At each state in the FSM, a prohibitable event is listed if its corresponding output is *true* at that state, which means the controller enables this event at this state. An event is not listed if its output is $false$.
- At each transition, we use logical operators to represent the sampled input. We use '!' as **NOT**, '+' as **OR**, '$\cdot$' as **AND**.
- To distinguish from a DES event label and the event input being *true* at the clock edge, the event name is surrounded by '[ ]' to indicate that the input was *true* at the clock edge.

If the controller is following a concurrent string, for example $\alpha - \beta - \tau$ from one sampling state to the next, we add a transition arc with '$\cdot$'(**AND**) between the non-tick events. For example '$[\alpha]\cdot[\beta]$'.[2] This would be interpreted as events α and β occurred in the last sampling period, and no other activity events. Of course, there is no implied ordering of the two events, nor do we know how many times each event actually occurred during the last clock period.

Technically, if a supervisor has event set $\Sigma = \alpha, \beta, \gamma, tick$, the next state condition for a given concurrent string should include a term for each activity event in the event set. When the event is missing, it is negated. For string $\alpha - \beta - \tau$, this would be '$[\alpha]\cdot[\beta]\cdot![\gamma]$'. However, we can often simplify these equations using Boolean logic. For instance, if none of the possible strings at the current sampled state contain γ, we can leave it out of the equations.

If the controller is getting to the same state by different strings which are not occurrence equivalent, then we can use '+'(**OR**) to combine the conditions together. For example '$[\alpha]+[\beta]$'. This would be interpreted as event α or event β occurred in the last sampling period, but no others.

If at a given state in the controller we can do concurrent string $\alpha-\tau$ and $\alpha-\beta-\tau$, we need to make sure their next state equations do not overlap. Using conditions '$[\alpha]$' and '$[\alpha]\cdot[\beta]$' is not enough as first condition is *true* as long as α occurred, irrespective of β. Instead, the condition for $\alpha - \tau$ should be '$[\alpha]\cdot![\beta]$'.

For each FSM, the initial state is identified by a *Reset* signal. This *Reset* signal represents the "power on" behavior or a restart of the controller. This state is equivalent to the initial state of each TDES. It also explains why the initial state of a timed DES is a sampling state that does not need to be reached by a tick, since the FSM always starts at this state.

For each FSM state, we typically define a default transition **DEF**. This is because a TDES transition function is a partial function and an FSM next state function is a total function. Basically, it is a shorthand for all the next state equations that we have not explicitly specified. It is equivalent to taking the logical **OR** of all existing outgoing next state conditions from that state, and then negating the result. Sometimes, when we are translating a supervisor, we end up with a specified next state equation going to the same place as our **DEF** transition. That means this transition can be removed as it will be covered by the **DEF** condition.

[2]In the following FSM graphs, this operator is represented by '.'(period) instead of '$\cdot$' due to a technical difficulty.

Our first FSM is for supervisor **B2**, and is shown in Figure 7.45. At state 0, we have merged selfloop transition '$![921] \cdot [no921]$' with the **DEF** transition. It is worth noting how much simpler the FSM tends to be than the corresponding supervisor. For **B2**, we went from a 7 state supervisor to a 3 state FSM.

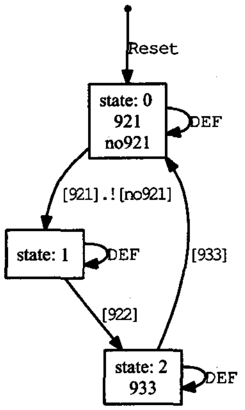


Figure 7.45: FSM B2

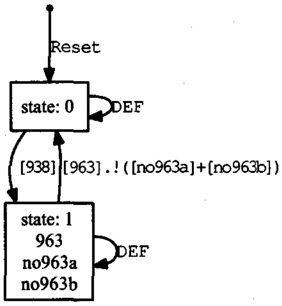


Figure 7.46: FSM Force963

We do a similar simplification for supervisors **Force963** and **Force965**. The translated FSM are shown in Figures 7.46 and 7.47. For **Force963**, we should have a $![963] \cdot ([no963a] + [no963b])$ selfloop at state 1, but we have absorbed this into the **DEF** transition. For **Force965**, we have absorbed the $![965] \cdot ([no965a] + [no965b])$ transition at state 1, into the **DEF** transition.

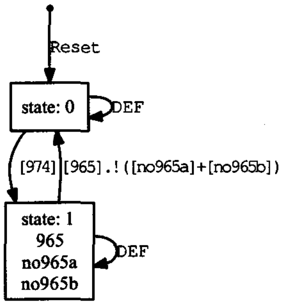


Figure 7.47: FSM Force965

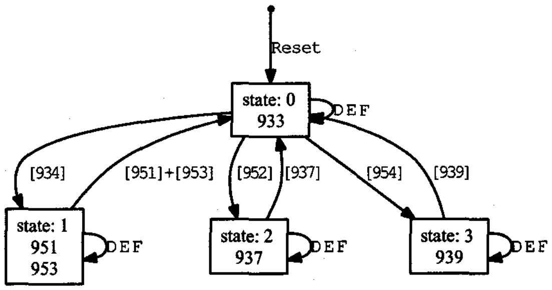


Figure 7.48: FSM B4

The next translation we examine is for **B4**, and the FSM are shown in Figure 7.48. Note at state 0, we have a transition to state 1 with condition '[934]'. Strictly speaking this should be '[934]·![952]·![954]'. However, after examining the plant and other supervisors, we know that these three events can never occur in the same clock period. We can thus simplify this to '[934]' to keep our diagram simple. Similar for the '[952]' and '[954]' transitions. A similar example is at state 1. Here we have transition condition '[951] + [953]'. Strictly speaking, this should be '[951]·![953]+![951] · [953]' but we know from the plant that these events can't occur in the same clock period, so we can simplify things.

The translation of the remaining FSM are straightforward so we do not need to discuss them individually. The translations for supervisors **B6**, **B7**, **B8**, **LathePick**, **TakeB2**, **B4Path**, **Force961**, **handleSystDown**, **TakeB4PutB6**, **TakeB4PutB7**, and **AMChooser** are shown in Figures 7.49 - 7.59.

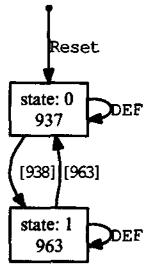


Figure 7.49: FSM B6

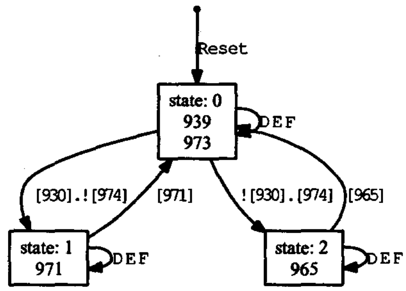


Figure 7.50: FSM B7

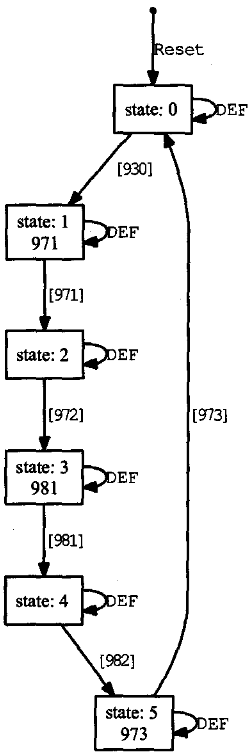


Figure 7.51: FSM B8

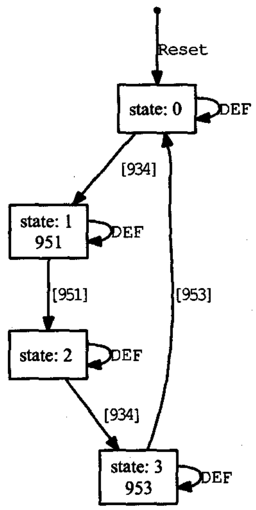


Figure 7.52: FSM LathePick

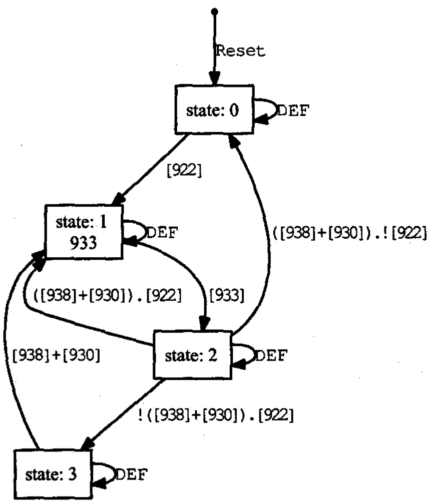


Figure 7.53: FSM TakeB2

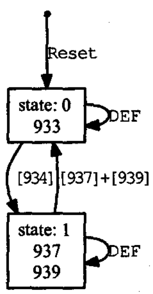


Figure 7.54: FSM B4Path

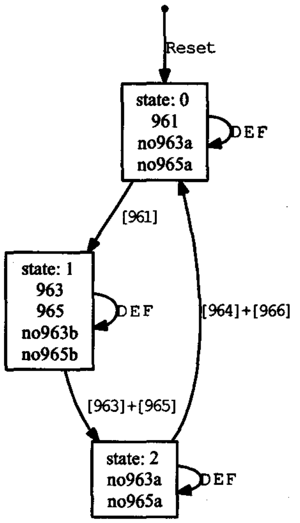


Figure 7.55: FSM Force961

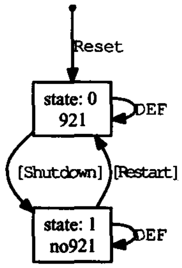


Figure 7.56: FSM handleSystDown

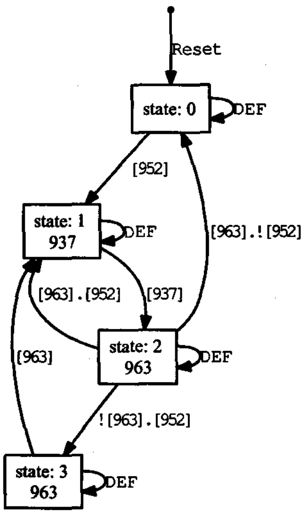


Figure 7.57: FSM TakeB4PutB6

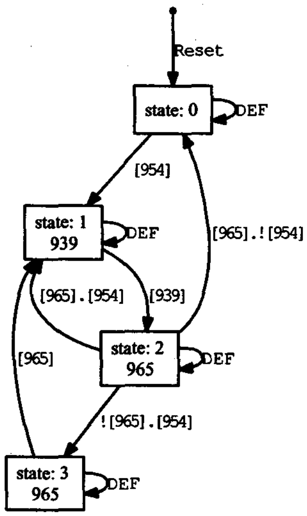


Figure 7.58: FSM TakeB4PutB7

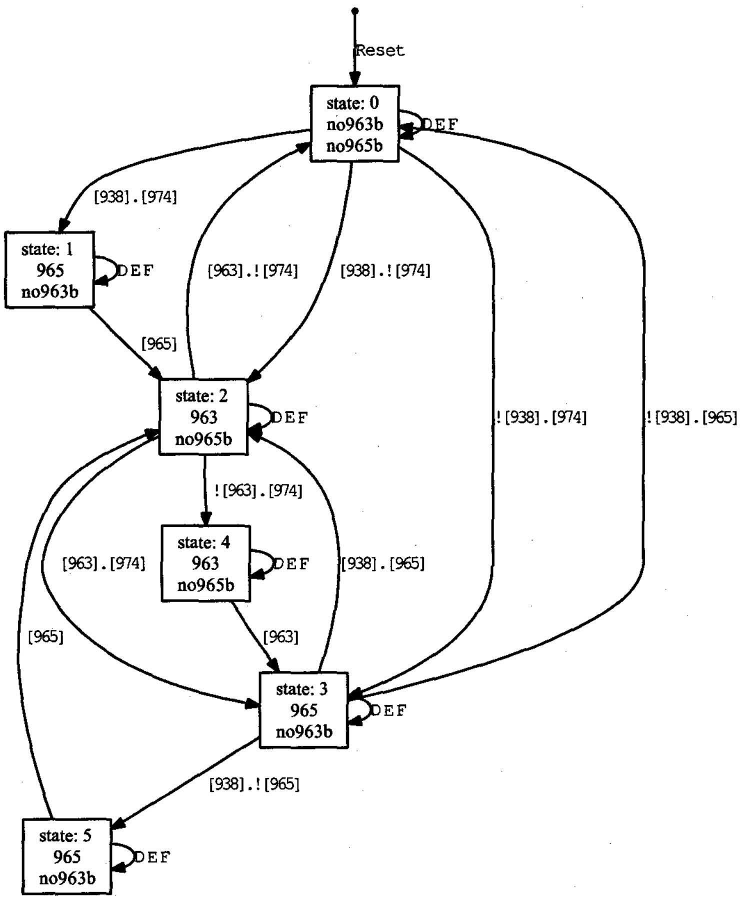


Figure 7.59: FSM AMChooser

Chapter 8

Conclusions

This thesis focuses on issues related to implementing theoretical Discrete-Event Systems (DES) supervisors, and the concurrency and timing delay issues involved.

Sampled-data (SD) supervisory control deals with timed DES (TDES) systems where the supervisors will be implemented as SD controllers. An SD controller is driven by a periodic clock and sees the system as a series of inputs and outputs. On each clock edge (tick event), it samples its inputs, changes states, and updates its outputs. In our introduction, we identified several concurrency issues that are not covered by the standard controllability and nonblocking definitions.

In this thesis, we identify a set of existing TDES properties that will be useful to our work, but not sufficient. We require that our plant have proper time behavior, and is complete for our supervisor. We also require that our closed loop system is activity loop free and nonblocking. To these properties, we add two new conditions. First, we require that the plant have **S**-singular prohibitable behavior, where **S** is our TDES supervisor. This condition restricts plant behavior such that prohibitable events can only occur at most once per clock cycle, but is only concerned with strings that are also accepted by our supervisor.

The main new condition we introduce is the SD controllability definition. This condition extends the standard TDES controllability definition by adding restrictions so that the TDES behavior is consistent with restrictions imposed by SD controllers, making it easier to translate a TDES into an SD controller. It includes conditions to ensure that the enablement and eligibility information is constant across a sampling period, and that when the controller forces an event, it will not occur when the plant model says it can't. It also ensures that when two strings that appear identical to an SD controller occur in the same sampling period, the strings have the same closed and marked future in the system's closed loop behavior. This means the SD controller will take the same control action for both, and either string will be sufficient to get us to a marked state. Finally, we require that only the empty string or a string ending in a tick can be marked. This ensures that marked strings will be observable to the controller.

We then establish a formal representation of an SD controller as a Moore Finite State Machine (FSM), and describe how to translate a TDES supervisor to a FSM. To be able to translate a given TDES into an FSM, we require that the TDES be CS deterministic. This new condition essentially says that if two concurrent strings can occur in the same clock cycle and they contain the same events (possibly in different order or number), then they must take us to the same state in the supervisor. This ensures our FSM is deterministic. We also discuss how to construct a single centralized controller, as well as a set of modular controllers and show that they will produce equivalent output. This is an important result, because we prefer a modularized design of controllers rather than a large, complex, centralized design.

Next, we capture the enablement and forcing action of a translated controller in the form of a TDES supervisory control map, and show that the closed-loop behavior of this map and the plant is the same as that of the plant and the original TDES supervisor. This is important as it shows that the behavior we expect from our TDES model is what we should actually get in the system, at least as far as enablement and forcing goes. As a controller chooses its next state based on which events occurred in the last clock period, this means the enablement and forcing actions the controller takes is irrespective to event ordering or number, but will have equivalent effect as that of our TDES supervisor. As we discuss at the end of Chapter 3, there are several time delay issues that we only partially address, leaving the remaining issues for future work.

We also show that our method is robust with respect to nonblocking and certain variations in the actual behavior of our physical system. Essentially, if there are two or more concurrent strings possible in a given clock cycle and they contain the same events (possibly in different order or number), we showed that as long as at least one of these strings is actually possible in the physical system, then the physical system and our SD controller will be nonblocking if our TDES closed loop system is nonblocking. This result is important as some implementations may be such that we actually get a subset of our expect behavior. This result says that as long as we get this minimal subset, we will remain nonblocking.

We also introduce a set of predicate-based algorithms to verify the SD controllability property, as well as the other conditions that we require. The algorithms are implemented on the top of the preceding code base of Raoguang Song and use binary decision diagrams (BDD). BDD is an efficient structure to store systems with large statespaces and to perform state set operations. The implemented software tool is able to verify a system whose synchronous product has more than 80,000 states, in less than 3 minutes. We expect that it will be able to handle quite large systems, but we did not have time to attempt this ourselves.

Finally to test our algorithms, we have produced a set of illustrative examples which fail the key conditions discussed in this thesis, as well as a successful application example based on a Flexible Manufacturing System (FMS). For all the supervisors in the FMS example, we also translated them into Moore FSM controllers using the translation method we created. Ideally, we would like to see an algorithm that converts these controllers into program source code in some computer language. This is left as future work and is beyond the scope of this thesis.

The source code of the software tool and the input files for the FSM example are included in the appendix. The software is single threaded, which limits its performance. A few choices for the next step for the software tool, are rewriting the code to be multithreaded, and/or implement a mechanism that can distribute the verification over multiple machines. We believe that our algorithms have good parallelizing potential. This is left as future work.

Bibliography

[1] D. S. Arnon, "Bibliography of quantifier elimination for real closed fields," *J. Symb. Comput.*, vol. 5, no. 1-2, pp. 267–274, 1988. 6.2.2

[2] S. Balemi, "Input/output discrete event processes and communication delays," *Discrete Event Dynamical Systems: Theory and Applications*, pp. 41–85, 1994. 1.1, 1.2, 2.2.3

[3] F. Basile and P. Chiacchio, "On the implementation of supervised control of discrete event systems," *Control Systems Technology, IEEE Transactions on*, vol. 15, no. 4, pp. 725–739, 2007. 1.2

[4] W. Bolton, *Programmable Logic Controllers.* Elsevier, 4th ed., 2006. 1

[5] B. Brandin and W. Wonham, "Supervisory control of timed discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 39, pp. 329–342, Feb 1994. 1.1, 1.2, 2.3

[6] B. A. Brandin, *Real-time supervisory control of automated manufacturing systems.* PhD thesis, University of Toronto. Graduate Department of Electrical and Computer Engineering, 1992. 1.1, 1.2, 2.3, 5.1

[7] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design.* McGraw Hill Higher Education, 3rd ed., 7 2008. 1, 3.1

[8] A. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, pp. 293–318, 1992. 1.1, 1.2, 6, 6.2.2

[9] C. Dragert, J. Dingel, and K. Rudie, "Generation of concurrency control code using discrete-event systems theory," in *SIGSOFT '08/FSE-16: Proceedings of*

the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, (Atlanta, Georgia), pp. 146–157, ACM, 2008. 1.2

[10] A. Giua and C. Seatzu, "Modeling and supervisory control of railway networks using petri nets," *IEEE Transactions on Automation Science and Engineering*, vol. 5, no. 3, pp. 431–445, July 2008. 1.2

[11] R. C. Hill, *Modular Verification and Supervisory Controller Design for Discrete-Event Systems Using Abstraction and Incremental Construction.* PhD thesis, Department of Mechanical Engineering, University of Michigan, 2008. 1.1, 7, 7.2, 7.2.2

[12] R. J. Leduc, "PLC implementation of a DES supervisor for a manufacturing testbed: An implementation perspective," Master's thesis, Graduate Department of Computer and Electrical Engineering, University of Toronto, 1996. 1.2

[13] J. Lind-Nielsen, *BuDDy: Binary Decision Diagram Package.* IT-University of Copenhagen, 11 2002. 6

[14] C. Ma, *Nonblocking supervisory control of state tree structures.* PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 2004. Adviser-W. M. Wonham. 1.2, 6.2, 6.3

[15] P. Malik, *From Supervisory Control to Nonblocking Controllers for Discrete Event Systems.* PhD thesis, Dept. of Computer Science, University of Kaiserslautern, Kaiserslautern, 2003. 1.2

[16] J. O. Moody and P. J. Antsaklis, *Supervisory Control of Discrete Event Systems using Petri Nets.* Kluwer Academic Publishers, 1998. 1.2

[17] E. Moore, "Gedanken-experiments on sequential machines," in *Automata Studies* (C. Shannon and J. McCarthy, eds.), pp. 129–153, Princeton, NJ: Princeton University Press, 1956. 1.1, 4

[18] M. Nourelfath and E. Niel, "Modular supervisory control of an experimental automated manufacturing system," *Control Engineering Practice*, vol. 12, no. 2, pp. 205–216, Feb. 2004. 1.2

[19] J. S. Ostroff, "Deciding properties of timed transition models," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 2, pp. 170–183, 1990. 1.2

[20] J. S. Ostroff, *Temporal logic for real time systems.* New York, NY, USA: John Wiley & Sons, Inc., 1989. 1.2

[21] J. Ostroff and W. Wonham, "A framework for real-time discrete event control," *IEEE Transactions on Automatic Control*, vol. 35(4), pp. 386–397, April 1990. 1.2

[22] S. Perk, T. Moor, and K. Schmidt, "Controller synthesis for an I/O-based hierarchical system architecture," *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pp. 474–479, May 2008. 1.2

[23] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Control Optim.*, vol. 25, no. 1, pp. 206–230, 1987. 1

[24] A. Saadatpoor, "State-based control of timed discrete-event systems using binary decision diagrams," Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 2004. 1.2

[25] K. Schmidt and E. Schmidt, "Communication of distributed discrete-event supervisors on a switched network," *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pp. 419–424, May 2008. 1.2

[26] R. Song, "Symbolic synthesis and verification of hierarchical interface-based supervisory control," Master's thesis, Dept. of Computing and Software, McMaster University, 2006. 1.1, 1.2, 6, 6.1.2, 6.2, 6.3, 6.3.2, 6.3.2, 6.4.1

[27] A. Vahidi, B. Lennartson, and M. Fabian, "Efficient analysis of large discrete-event systems with binary decision diagrams," in *Proc. of the 44th IEEE Conf. Decision Contr. and and 2005 European Contr. Conf.*, (Seville, Spain), pp. 2751–2756, 2005. 1.2

[28] K. C. Wong and W. M. Wonham, "Hierarchical control of timed discrete-event systems," *Discrete Event Dynamic Systems*, vol. 6, pp. Pages 275 - 306, July 1996. 2.3.2

[29] W. M. Wonham, *Supervisory Control of Discrete-Event Systems.* Department of Electrical Engineering, University of Toronto, 2005. 1, 2

[30] W. M. Wonham and P. J. Ramadge, "On the supermal controllable sublanguage of a given language," *SIAM J. Control Optim.*, vol. 25, no. 3, pp. 637–659, 1987. 1

[31] S. Xu and R. Kumar, "Asynchronous implementation of synchronous discrete event control," *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pp. 181–186, May 2008. 1.2

[32] Z. Zhang and W. M. Wonham, "STCT: an efficient algorithm for supervisory control design," in *Proc. of SCODES 2001*, (INRIA, Paris), pp. 82–93, July 2001. 1.2

[33] M. Zhou and F. DiCesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems.* Kluwer Academic Publishers, 1993. 1.2

Appendix A

SD Software Program

A.1 FMS Example Input Files

The input files below are all for the FMS example that we presented in Section 7.2.

A.1.1 FMS Plants

Listing A.1: Con2

```
#generated by pds2hsc
[States]
4       # num of states
1
2
3
4

[InitState]
1

[MarkingStates]
1

[Events]
tick    Y       L
921     Y       L
922     N       L

[Transitions]
4
(tick   1)
1
(tick   1)
(921    2)
2
(tick   3)
3
(tick   3)
```

```
(922    4)
```

Listing A.2: Robot

```
#generated by pds2hsc
[States]
8       # num of states
1
2
3
4
5
6
7
8

[InitState]
1

[MarkingStates]
1

[Events]
tick    Y       L
930     N       L
933     Y       L
934     N       L
937     Y       L
938     N       L
939     Y       L

[Transitions]
8
(tick   1)
1
(tick   1)
(933    2)
(937    3)
(939    4)
2
(tick   5)
3
(tick   6)
4
(tick   7)
5
(tick   5)
(934    8)
6
(tick   6)
(938    8)
7
(tick   7)
(930    8)
```

Listing A.3: Lathe

```
#generated by pds2hsc
[States]
6       # num of states
1
2
3
4
5
```

```
6

[InitState]
1

[MarkingStates]
1

[Events]
tick    Y       L
951     Y       L
952     N       L
953     Y       L
954     N       L

[Transitions]
6
(tick   1)
1
(tick   1)
(951    2)
(953    3)
2
(tick   4)
3
(tick   5)
4
(tick   4)
(952    6)
5
(tick   5)
(954    6)
```

Listing A.4: AM

```
#generated by pds2hsc
[States]
8       # num of states
1
2
3
4
5
6
7
8

[InitState]
1

[MarkingStates]
1
3

[Events]
tick    Y       L
961     Y       L
963     Y       L
964     N       L
965     Y       L
966     N       L

[Transitions]
8
(tick   1)
1
```

```
(tick	1)
(961	2)
2
(tick	3)
3
(tick	3)
(963	4)
(965	5)
4
(tick	6)
5
(tick	7)
6
(tick	6)
(964	8)
7
(tick	7)
(966	8)
```

Listing A.5: Con3

```
#generated by pds2hsc
[States]
6	# num of states
1
2
3
4
5
6

[InitState]
1

[MarkingStates]
1

[Events]
tick	Y	L
971	Y	L
972	N	L
973	Y	L
974	N	L

[Transitions]
6
(tick	1)
1
(tick	1)
(971	2)
(973	3)
2
(tick	4)
3
(tick	5)
4
(tick	4)
(972	6)
5
(tick	5)
(974	6)
```

Listing A.6: PM

```
#generated by pds2hsc
```

```
[States]
4       # num of states
1
2
3
4

[InitState]
1

[MarkingStates]
1

[Events]
tick    Y       L
981     Y       L
982     N       L

[Transitions]
4
(tick   1)
1
(tick   1)
(981    2)
2
(tick   3)
3
(tick   3)
(982    4)
```

A.1.2 Helper Plants

Listing A.7: AddNo921

```
#generated by pds2hsc
[States]
2       # num of states
1
2

[InitState]
1

[MarkingStates]
1

[Events]
tick    Y       L
no921  Y   L

[Transitions]
1
(tick   1)
(no921 2)
2
(tick   1)
```

Listing A.8: AddNo963

```
#generated by pds2hsc
[States]
```

```
2       # num of states
1
2

[InitState]
1

[MarkingStates]
1

[Events]
tick     Y        L
no963a   Y   L
no963b   Y   L

[Transitions]
1
(tick   1)
(no963a 2)
(no963b 2)
2
(tick   1)
```

Listing A.9: AddNo965

```
#generated by pds2hsc
[States]
2       # num of states
1
2

[InitState]
1

[MarkingStates]
1

[Events]
tick     Y        L
no965a   Y   L
no965b   Y   L

[Transitions]
1
(tick   1)
(no965a 2)
(no965b 2)
2
(tick   1)
```

Listing A.10: SystDownNup

```
#generated by pds2hsc
[States]
4       # num of states
1
2
3
4

[InitState]
1

[MarkingStates]
1
```

```
[Events]
shutdown        N       L
restart N       L
tick Y   L

[Transitions]
1
(tick   1)
(shutdown 2)
2
(tick   3)
3
(tick   3)
(restart 4)
4
(tick   1)
```

A.1.3 Buffer Supervisors

Listing A.11: B2

```
[States]
7   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
2
3
4
5
6

[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
921     Y   L
no921     Y   L
922     N   L
933     Y   L
tick     Y   L

[Transitions]
0
(921    1)
(no921  6)
1
(tick   2)
2
(tick   2)
(922    3)
3
(tick   4)
4
(tick   4)
(933    5)
5
(tick   0)
6
(tick   0)
```

```
####################################
```

Listing A.12: B4

```
[States]
8   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
2
3
5
6
7
8

[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
933     Y   L
934     N   L
937     Y   L
939     Y   L
951     Y   L
952     N   L
953     Y   L
954     N   L
tick     Y   L

[Transitions]
0
(933    0)
(952  6)
(934   5)
(954  7)
(tick   0)
1
(tick   1)
(951  8)
(953  8)
2
(tick   2)
(937   8)
3
(tick   3)
(939   8)
5
(tick   1)
6
(tick   2)
7
(tick   3)
8
(tick   0)
####################################
```

Listing A.13: B6

```
[States]
4   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
```

```
3
4

[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
937    Y   L
938    N   L
963    Y   L
tick    Y   L

[Transitions]
0
(937    0)
(938   3)
(tick   0)
1
(963   4)
(tick   1)
3
(tick   1)
4
(tick   0)
####################################
```

Listing A.14: B7

```
[States]
7   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
2
5
6
7
8

[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
939    Y   L
930    N   L
965    Y   L
971    Y   L
973    Y   L
974    N   L
tick    Y   L

[Transitions]
0
(939    0)
(930   5)
(973    0)
(974  7)
(tick   0)
1
(971  6)
(tick   1)
```

```
2
(965   8)
(tick   2)
5
(tick   1)
6
(tick   0)
7
(tick   2)
8
(tick   0)
####################################
```

Listing A.15: B8

```
[States]
12   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
2
3
4
5
6
7
8
9
10
11

[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
930      N   L
971     Y   L
972     N   L
973     Y   L
981     Y   L
982     N   L
tick     Y   L

[Transitions]
0
(tick   0)
(930    1)
1
(tick   2)
2
(971    3)
3
(tick   4)
4
(tick   4)
(972    5)
5
(tick   6)
6
(981    7)
7
(tick   8)
8
(tick   8)
```

```
(982    9)
9
(tick   10)
10
(973   11)
11
(tick   0)
##################################
```

A.1.4 Additional Supervisors

Listing A.16: AMChooser

```
[States]
14   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
2
3
4
5
6
7
8
9
10
11
12
13

[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
938      N   L
963      Y   L
965      Y   L
974      N   L
tick     Y   L
no963b   Y   L
no965b   Y   L

[Transitions]
0
(tick    0)
(no963b 0)
(no965b 0)
(974     1)
(938     2)
1
(no963b 1)
(no965b 1)
(tick    6)
(938     3)
2
(no963b 2)
(no965b 2)
(tick    5)
(974     3)
3
```

```
(no963b 3)
(no965b 3)
(tick   7)
4
(no963b 4)
(tick   5)
5
(no965b 5)
(tick   5)
(974    9)
(963    11)
6
(no963b 6)
(tick   6)
(965    12)
(938    13)
7
(no963b 7)
(tick   7)
(965    8)
8
(no963b 8)
(tick   5)
9
(no965b 9)
(tick   9)
(963    10)
10
(no965b 10)
(tick   6)
11
(no965b 11)
(974    10)
(tick   0)
12
(no963b 12)
(938    4)
(tick   0)
13
(no963b 13)
(tick   13)
(965    4)
###################################
```

Listing A.17: B4Path

```
[States]
4   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
2
3

[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
933      Y   L
934     N   L
937      Y   L
939      Y   L
tick     Y   L
```

```
[Transitions]
0
(tick   0)
(933    0)
(934    2)
1
(tick   1)
(937    3)
(939    3)
2
(tick   1)
3
(tick   0)
###################################
```

Listing A.18: Force961

```
#generated by pds2hsc
[States]
6       # num of states
0
1
2
3
4
5

[InitState]
0

[MarkingStates]
0
2

[Events]
tick    Y       L
no963a  Y   L
no963b  Y   L
no965a  Y   L
no965b  Y   L
961     Y   L
963     Y   L
964     N   L
965     Y   L
966     N   L

[Transitions]
0
(no965a 0)
(no963a 0)
(961    1)
1
(no965a 1)
(no963a 1)
(tick   2)
2
(no965b 2)
(no963b 2)
(tick   2)
(963    3)
(965    3)
3
(no965b 3)
(no963b 3)
(tick   4)
4
```

```
(no965a 4)
(no963a 4)
(tick   4)
(964    5)
(966    5)
5
(no965a 5)
(no963a 5)
(tick   0)
```

Listing A.19: Force963

```
#generated by pds2hsc
[States]
5       # num of states
0
1
2
3
4

[InitState]
0

[MarkingStates]
0

[Events]
tick     Y       L
no963a   Y   L
no963b   Y   L
938      N   L
963      Y   L

[Transitions]
0
(tick   0)
(938    1)
1
(tick   2)
2
(963    3)
(no963a 4)
(no963b 4)
3
(tick   0)
4
(tick   2)
```

Listing A.20: Force965

```
#generated by pds2hsc
[States]
5       # num of states
0
1
2
3
4

[InitState]
0

[MarkingStates]
0
```

```
[Events]
tick     Y       L
no965a   Y   L
no965b   Y   L
974      N   L
965      Y   L

[Transitions]
0
(tick   0)
(974    1)
1
(tick   2)
2
(no965a 4)
(no965b 4)
(965    3)
3
(tick   0)
4
(tick   2)
```

Listing A.21: LathePick

```
[States]
8   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
2
3
4
5
6
7

[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
934     N   L
951    Y   L
953    Y   L
tick    Y   L

[Transitions]
0
(tick   0)
(934    1)
1
(tick   2)
2
(951    3)
3
(tick   4)
4
(tick   4)
(934    5)
5
(tick   6)
6
(953    7)
7
```

```
(tick   0)
###################################
```

Listing A.22: TakeB2

```
[States]
8   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
2
3
4
5
6
7

[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
922    N   L
930     N   L
933    Y   L
938     N   L
tick    Y   L

[Transitions]
0
(tick   0)
(922    1)
1
(tick   2)
2
(933   3)
3
(tick   4)
4
(tick   4)
(922    5)
(938    6)
(930    6)
5
(tick   5)
(930    7)
(938    7)
6
(tick   0)
(922    7)
7
(tick   2)
###################################
```

Listing A.23: TakeB4PutB6

```
[States]
6   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
2
3
4
5
```

```
[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
937    Y   L
952    N   L
963     Y   L
tick    Y   L

[Transitions]
0
(tick   0)
(952    1)
1
(tick   2)
2
(937    3)
3
(tick   4)
4
(tick   4)
(952    5)
(963    0)
5
(tick   5)
(963    1)
##################################
```

Listing A.24: TakeB4PutB7

```
[States]
6   #num of states
0   #list of state names. If the list is omitted, then this tool will
1
2
3
4
5

[InitState]
0

[MarkingStates]
0

[Events]  #(event name, controllable, L/R/A)
939    Y   L
954    N   L
965     Y   L
tick    Y   L

[Transitions]
0
(tick   0)
(954    1)
1
(tick   2)
2
(939    3)
3
(tick   4)
4
```

```
(tick   4)
(954    5)
(965    0)
5
(tick   5)
(965    1)
###################################
```

Listing A.25: handleSystDown

```
#generated by pds2hsc
[States]
4       # num of states
0
1
2
3

[InitState]
0

[MarkingStates]
0

[Events]
921     Y   L
no921   Y   L
shutdown        N       L
restart N       L
tick    Y   L

[Transitions]
0
(tick   0)
(921    0)
(shutdown 1)
1
(921    1)
(tick   2)
2
(no921  2)
(tick   2)
(restart 3)
3
(no921  3)
(tick   0)
```

A.2 Source code

The source code files are to be compiled using `gcc 4.3.2` or higher version. Optimization `-O` is suggested for better performance.

A.2.1 Main

main.cpp

```
{
    bool bPrjLoaded = false;
    char ch = '\0';
    char prjfile[MAX_PATH];
    string errmsg;
    prjfile[0] = '\0';
    int iret = 0;
    char prjoutputfile[MAX_PATH];

    char savepath[MAX_PATH];
    savepath[0] = '\0';
    HISC_SUPERINFO superinfo;

    HISC_TRACETYPE tracetype;
    int computetime = 0;

    while (ch != 'q' && ch != 'Q')
    {
        ch = getchoice(bPrjLoaded, prjfile);
        switch (ch)
        {
            case 'q':
            case 'Q':
                iret = close_prj(errmsg);
                bPrjLoaded = false;
                prjfile[0] = '\0';

                break;
            //Load a project
            case 'P':
            case 'p':
                cout << "Sub name:";
```

```
        cin.getline(prjfile, MAX_PATH);
        iret = load_prj(prjfile, errmsg);
        if (iret < 0)
        {
            if (iret > -10000) //error
                bPrjLoaded = false;
            else
                bPrjLoaded = true; //waring
        }
        else
            bPrjLoaded = true;
        break;
    //close the current project
    case 'c':
    case 'C':
        iret = close_prj(errmsg);
        bPrjLoaded = false;
        prjfile[0] = '\0';
        break;
    //File the current project
    case 'f':
    case 'F':
        cout << "file name:";
        cin.getline(prjoutputfile, MAX_PATH);
        iret = print_prj(prjoutputfile, errmsg);
        break;
    //Low Level verification
    case 'l':
    case 'L':
        cout << "Show the blocking type(may take long time)(Y/N)?";
        tracetype = (HISC_TRACETYPE)getchoice_tracetype();

        char verbosechoices[3] = {'0', '1', '2'};
        cout << "Verbose level (0 - disable, 1 - brief, 2 - full)?";
        const char choice[2] = { getkeystroke(verbosechoices,
```

```
3), '\0' };
                iVerbLevel = atoi(choice);

                computetime = 0;

                superinfo.statesize = -1;
                superinfo.nodesize = -1;
                superinfo.time = 0;

                iret = verify_low(tracetype, errmsg, &superinfo);
                cout << "("<< iret << ") ";

                if (iret == 0)
                    cout << "This system has been verified succesfully!"
                         << endl;
                if (superinfo.statesize >= 0)
                    cout << "State size of the synchronous product: " <<
                            superinfo.statesize << endl;
                if (superinfo.nodesize >= 0)
                    cout << "Number of bdd nodes to store" <<
                        " the synchronous product: " << superinfo.nodesize
                        << endl;
                cout << "Computing time: " << superinfo.time <<
                        " seconds." << endl;
                computetime += superinfo.time;

                if (iret < 0)
                {
                    cout << errmsg << endl;
                    cout << "Press any key to continue...";
                    iret = 0;
                    errmsg[0] = '\0';
                    getkeystroke(NULL, 0);
                }

```

```
                    cout << "Total computing time:" << computetime << " seconds."
                            << endl;

                    break;
            }
            if (iret < 0)
            {
                cout << errmsg << endl;
                cout << "Press any key to continue...";
                iret = 0;
                errmsg[0] = '\0';
                getkeystroke(NULL, 0);
            }
    }

    close_hisc();

    return 0;
}

int getchoice(bool bPrjLoaded, const char *prjfile)
{
    char allowed_choice[50];
    int numofchoice = 0;

    cout << endl << endl << endl << endl << endl;
    cout << "************************************************" << endl;
    cout << " Bdd-based HISC Synthesis and Verification Tool " << endl;
    cout << "************************************************" << endl;
    if (!bPrjLoaded)
    {
        allowed_choice[0] = 'p';
        allowed_choice[1] = 'P';
        allowed_choice[2] = 'q';
```

```
        allowed_choice[3] = 'Q';
        numofchoice = 4;
        cout << "  P - Load a HISC project                    " << endl;
    }
    else
    {
        allowed_choice[0] = 'c';
        allowed_choice[1] = 'C';
        allowed_choice[2] = 'q';
        allowed_choice[3] = 'Q';
        allowed_choice[4] = 'F';
        allowed_choice[5] = 'f';
        allowed_choice[6] = 'l';
        allowed_choice[7] = 'L';
        numofchoice = 8;
        cout << "  L - Low Level verification                 " << endl;
        cout << "  F - File the current project               " << endl;
        cout << "  C - Close the current project              " << endl;
    }
    cout << "  Q - Quit                                   " << endl;
    cout << "**********************************************" << endl;
    if (bPrjLoaded)
    {
        cout << "Current Project: " << prjfile << endl;
    }
    cout << endl;
    cout << "Procedure desired:";
    return getkeystroke(allowed_choice, numofchoice);
}

char getkeystroke(char *allowed_choices, int len)
{
    char choice;
    struct termios initial_settings, new_settings;

```

```
    tcgetattr(fileno(stdin), &initial_settings);
    new_settings = initial_settings;
    new_settings.c_lflag &= ~ICANON;

    new_settings.c_cc[VMIN] = 1;
    new_settings.c_cc[VTIME] = 0;
    new_settings.c_lflag &= ~ISIG;

    tcsetattr(fileno(stdin), TCSANOW, &new_settings);
    if (len > 0)
    {
        do {
            choice = fgetc(stdin);
            int i;
            for (i = 0; i < len; i++)
            {
                if (choice == allowed_choices[i])
                    break;
            }
            if (i == len)
                choice = '\n';
        } while (choice == '\n' || choice == '\r');
    }
    else
        choice = fgetc(stdin);

    tcsetattr(fileno(stdin),TCSANOW, &initial_settings);
    cout << endl;
    return choice;
}

int getchoice_savesup()
{
    char allowed_choice[50];
    int numofchoice = 0;
```

```
    char choice;

    allowed_choice[0] = '0';
    allowed_choice[1] = '1';
    allowed_choice[2] = '2';
    allowed_choice[3] = '3';
    numofchoice = 4;
    choice = getkeystroke(allowed_choice, numofchoice);
    return choice - '0';
}

int getchoice_tracetype()
{
    char allowed_choice[50];
    int numofchoice = 0;
    char choice;

    allowed_choice[0] = 'Y';
    allowed_choice[1] = 'y';
    allowed_choice[2] = 'N';
    allowed_choice[3] = 'n';
    numofchoice = 4;
    choice = getkeystroke(allowed_choice, numofchoice);

    if (choice == 'Y' || choice == 'y')
        return 1;
    else
        return 0;
}
```

A.2.2 Global Functions, Typedefs, Variables, Preprocessors symbols

type.h

```
const string sTick = "tick";

enum DESTYPE {PLANT_DES = 0, SPEC_DES = 1};
enum EVENTTYPE {CON_EVENT = 0, UNCON_EVENT = 1};

#define L_EVENT 3

typedef map<string, int> STATES; //state name, index
typedef map<int, string> INVSTATES; //state index, name

typedef map<string, int> EVENTS; //event name, global index
typedef map<int, string> INVEVENTS; //event global index, name

typedef map<string, unsigned short> LOCALEVENTS; //event name, level-wise index
typedef map<unsigned short, string> LOCALINVEVENTS;//event level-wise index,name

typedef set<unsigned short> EVENTSET;

typedef list<int> MARKINGLIST; //link list to save all the marker states index
typedef map<int, int> TRANS; //source state index (key), target state index
#endif //_TYPE_H_
```

errmsg.h

```

#define HISC_BAD_INTERFACE -11
#define HISC_TICK_NOT_FOUND -12

#define HISC_LOWERR_GENCONBAD -20
#define HISC_LOWERR_GENP4BAD -21
#define HISC_LOWERR_SUPCP -22
#define HISC_LOWERR_COREACH -23
#define HISC_LOWERR_REACH -24
#define HISC_LOWERR_P5 -25
#define HISC_LOWERR_P6 -26
#define HISC_LOWERR_GENBALEMIBAD -27
#define HISC_LOWERR_ALF -28
#define HISC_LOWERR_PTB -29
#define HISC_LOWERR_SD  -30
#define HISC_LOWERR_SDIV    -31

#define HISC_VERI_LOW_UNCON -201
#define HISC_VERI_LOW_BLOCKING -202
#define HISC_VERI_LOW_P4FAILED -203
#define HISC_VERI_LOW_P5FAILED -204
#define HISC_VERI_LOW_P6FAILED -205
#define HISC_VERI_LOW_CON -206
#define HISC_VERI_LOW_ALF -207
#define HISC_VERI_LOW_PTB -208
#define HISC_VERI_LOW_SD_II -209
#define HISC_VERI_LOW_SD_III_1 -210
#define HISC_VERI_LOW_SD_III_2 -211
#define HISC_VERI_LOW_SD_IV -212
#define HISC_VERI_LOW_ZERO_LB  -213

#define HISC_HIGHERR_GENCONBAD -30
#define HISC_HIGHERR_GENP3BAD -31
#define HISC_HIGHERR_SUPCP -32
```

```
#define HISC_HIGHERR_COREACH -33
#define HISC_HIGHERR_REACH -34
#define HISC_VERI_HIGH_UNCON -101
#define HISC_VERI_HIGH_P3FAILED -102
#define HISC_VERI_HIGH_BLOCKING -103

#define HISC_BAD_SAVESUPER -97
#define HISC_BAD_PRINT_FILE -98
#define HISC_NOT_ENOUGH_MEMORY -99

#define HISC_WARN_BLOCKEVENTS -10000
#define HISC_INTERNAL_ERR_SUBEVENT -10001

#endif //____ERRMSG_H____


```

pubfunc.h

```
extern string str_upper(const string &str);
extern string str_lower(const string &str);
extern string str_itos(int iInt);
extern string str_ltos(long long lLong);

extern string str_nocomment(const string & str);
extern int scp_err(const string & sErr, const int iErrCode);

extern string GetNameFromFile(const string & vsFile);

extern int IsInteger(const string &str);
extern int CompareInt(const void* pa, const void* pb);

extern void bddPrintStats(const bddStat &stat);
extern void SetBddPairs(bddPair *pPair, const bdd & bddOld, const bdd & bddNew);
```

```
extern int NumofSharedEvents(const int * pEventsArr_a, const int viNumofEvents_a,
            const int * pEventsArr_b, const int viNumofEvents_b);
extern void my_bdd_gbchandler(int pre, bddGbcStat *s);

#endif //____PUBFUNC_H____


```

pubfunc.cpp

```
 * PARA:    str: a string (input)
 * RETURN:  trimmed string
 * */
string str_trim(const string &str)
{
    string sTmp("");
    unsigned int i = 0;

    //trim off the prefix spaces
    for (i = 0; i < str.length(); i++)
    {
        if (str[i] != 32 && str[i] != 9)
            break;
    }
    if (i < str.length())
    {
        sTmp = str.substr(i);
    }
    else
    {
        return sTmp;
    }

    //trim off the suffix spaces
```

```
    int j = 0;
    for (j = sTmp.length() - 1; j >= 0; j--)
    {
        if (sTmp[j] != 32 && sTmp[j] != 9)
            break;
    }
    if (j >= 0)
    {
        sTmp = sTmp.substr(0, j + 1);
    }
    else
    {
        sTmp.clear();
    }

    return sTmp;
}

/**
 * DESCR:   convert all the letters in a string to uppercase
 * PARA:    str: a string (input)
 * RETURN:  converted string
 * */
string str_upper(const string &str)
{
    unsigned int i = 0;
    string sTmp(str);

    for (i = 0; i < str.length(); i++)
    {
        if ((sTmp[i] >= 'a') & (sTmp[i] <= 'z'))
        {
            sTmp[i] = sTmp[i] - 32;
        }
    }
```

```
    return sTmp;
}

/**
 * DESCR:   convert all the letters in a string to lowercase
 * PARA:    str: a string (input)
 * RETURN:  converted string
 * */
string str_lower(const string &str)
{
    unsigned int i = 0;
    string sTmp(str);

    for (i = 0; i < str.length(); i++)
    {
        if ((sTmp[i] >= 'A') & (sTmp[i] <= 'Z'))
        {
            sTmp[i] = sTmp[i] + 32;
        }
    }
    return sTmp;
}

/**
 * DESCR:   convert an integer to a string
 * PARA:    iInt: an integer
 * RETURN:  converted string
 * */
string str_itos(int iInt)
{
    char scTmp[65];
    string str;
    sprintf(scTmp, "%d", iInt);
    str = scTmp;

```

```
    return str;
}

/**
 * DESCR:   convert a long integer to a string
 * PARA:    iInt: a long integer
 * RETURN:  converted string
 * */
string str_ltos(long long lLong)
{
    char scTmp[65];
    string str;
    sprintf(scTmp, "%lld", lLong);
    str = scTmp;

    return str;
}

/**
 * DESCR:   trim off all the characters after a COMMENT_CHAR
 * PARA:    str : a string
 * RETURN:  processed string
 * */
string str_nocomment(const string & str)
{
    int i;
    int iLen = str.length();

    for (i = 0; i < iLen; i++)
    {
        if (str[i] == COMMENT_CHAR)
            break;
    }
    if (i < iLen)
        return str.substr(0, i);
```

```
    else
        return str;
}

/**
* DESCR:   Get sub name or des name from a full path file name
 *            with extension ".sub"/".hsc"
 *            ex: vsFile = "/home/roger/m1.sub" will return "m1"
 * PARA:      vsFile: file name with path
 * RETURN:    sub name or des name
 * */
string GetNameFromFile(const string & vsFile)
{
    assert(vsFile.length() > 4);
    assert(vsFile.substr(vsFile.length() - 4) == ".sub" ||
              vsFile.substr(vsFile.length() - 4) == ".hsc");

    unsigned int iPos = vsFile.find_last_of('/');

    if ( iPos == string::npos)
    {
        return vsFile.substr(0, vsFile.length() - 4);
    }
    else
    {
        return vsFile.substr(iPos + 1, vsFile.length() - 4 - (iPos +
1));
    }
}

/**
* DESCR:   Test if a string could be converted to an integer
 * PARA:      str: a string
 * RETURN:    0: no 1: yes
 * */
```

```
int IsInteger(const string &str)
{
    if (str.length() == 0)
        return 0;
    for (unsigned int i = 0; i < str.length(); i++)
    {
        if (str[i] >= '0' && str[i] <= '9')
            continue;
        else
            return 0;
    }

    return 1;
}

/**
 * DESCR:   Compare two integers which are provided by two general
pointers.
 *          qsort, bsearch will use this function
 * PARA:    pa, pb: general pointers pointing to two integers
 * RETURN:  1: a>b
 *          0: a=b
 *         -1: a<b
 * */
int CompareInt(const void* pa, const void* pb)
{
    int a = *((int *) pa);
    int b = *((int *) pb);

    if (a > b)
        return 1;
    else if (a < b)
        return -1;
    else
        return 0;
```

```
}


/**
 * DESCR:   To print the content of a bddStat variable.
 *          Original BDD package doesn't provide such a function.
 * PARA:    bddStat: see documents of Buddy package
 * RETURN:  None
 * */
void bddPrintStats(const bddStat &stat)
{
    cout << endl;
    cout << "--------------bddStat----------------" << endl;

    cout << "Num of new produced nodes: " << stat.produced << endl;
    cout << "Num of allocated nodes: " << stat.nodenum << endl;
    cout << "Max num of user defined nodes: " << stat.maxnodenum << endl;
    cout << "Num of free nodes: " << stat.freenodes << endl;
    cout << "Min num of nodes after garbage collection: " << stat.minfreenodes
         << endl;
    cout << "Num of vars:" << stat.varnum << endl;
    cout << "Num of entries in the internal caches:" << stat.cachesize <<
endl;
    cout << "Num of garbage collections done until now:" << stat.gbcnum <<
endl;
    return;
}

/**
 * DESCR:   Set bddpairs based on two bdd variable sets.
 *          The original function bdd_setbddpair(...) is not
 *          as the document said.
 * PARA:    pPair: where to add bdd variable pairs
 *          bddOld: variable will be replaced
 *          bddNew: new variable
```

```
 * RETURN:  None
 * */
void SetBddPairs(bddPair *pPair, const bdd & bddOld, const bdd & bddNew)
{
    assert(pPair != NULL);

    int *vOld = NULL;
    int *vNew = NULL;
    int nOld = 0;
    int nNew = 0;

    bdd_scanset(bddOld, vOld, nOld);
    bdd_scanset(bddNew, vNew, nNew);

    assert(nOld == nNew);

    for (int i = 0; i < nOld; i++)
    {
        bdd_setpair(pPair, vOld[i], vNew[i]);
    }

    free(vOld);
    free(vNew);

    return;
}

/**
 * DESCR:   Compute the number of shared events between two DES
 * PARA:    pEventsArr_a:   Event array for DES a (global index, sorted)
 *          viNumofEvents_a: Number of events in array pEventsArr_a
 *          pEventsArr_b:   Event array for DES b (global index, sorted)
```

```
 *           viNumofEvents_b: Number of events in array pEventsArr_b
 * RETURN:  Number of shared events
 * */
int NumofSharedEvents(const int * pEventsArr_a, const int viNumofEvents_a,
            const int * pEventsArr_b, const int viNumofEvents_b)
{
    int iNum = 0;
    int i = 0;

    assert(pEventsArr_a != NULL);
    assert(pEventsArr_b != NULL);

    if (viNumofEvents_a <= viNumofEvents_b)
    {
        for (i = 0; i < viNumofEvents_a; i++)
        {
            if (bsearch(&(pEventsArr_a[i]), pEventsArr_b, viNumofEvents_b,
                            sizeof(int), CompareInt) != NULL)
            {
                iNum++;
            }
        }
    }
    else
    {
        for (i = 0; i < viNumofEvents_b; i++)
        {
            if (bsearch(&(pEventsArr_b[i]), pEventsArr_a, viNumofEvents_a,
                            sizeof(int), CompareInt) != NULL)
            {
                iNum++;
            }
```

```
        }
    }

    return iNum;
}

/**
* DESCR:   Customized Garbage collection handler for this program
 * PARA:     see document of Buddy Package
 * RETURN:  None
 * */
void my_bdd_gbchandler(int pre, bddGbcStat *s)
{
   if (!pre)
   {
      if (s->nodes > giNumofBddNodes)
      {
          printf("Garbage collection #294          s-¿num, s-¿nodes,
s-¿freenodes);
          printf(" / %.1fs / %.1fs total\n",
             (float)s-¿time/(float)(CLOCKS_PER_SEC),
             (float)s-¿sumtime/(float)CLOCKS_PER_SEC);
          giNumofBddNodes = s-¿nodes;
      }
   }
   return;
}



```

BddSd.h

```
int load_prj(const char *prjfile, std::string & errmsg);

```

```
/**
 * DESCR:   close opened HISC project
 * PARA:    errmsg: returned errmsg (output)
 * RETURN:  0: sucess  < 0: fail
 * */
int close_prj(std::string & errmsg);

/**
 * DESCR:   Save the project in the memory to a text file, just for verifying
 *          the loaded project.
 * PARA:    filename: where to save the text file (input)
 *          errmsg: returned errmsg (output)
 * RETURN:  0: sucess  < 0: fail
 * */
int print_prj(std::string filename, std::string & errmsg);

/**
 * A structure for storing computing result information
 * */
typedef struct Hisc_SuperInfo
{
    double statesize;   /*state size*/
    int nodesize;       /*bdd node size*/
    int time;           /*computing time (seconds)*/
}HISC_SUPERINFO;

/**
 * To show a path from the initial state to one bad state or not
 * Currently HISC_SHOW_TRACE is only for telling if a blocking state is
 * deadlock or livelock
 * */
enum HISC_TRACETYPE {HISC_NO_TRACE = 0, HISC_SHOW_TRACE = 1};

/**
```

```
 * To synthesize on reachable statespace or not
 * */
enum HISC_COMPUTEMETHOD{HISC_ONCOREACHABLE = 0, HISC_ONREACHABLE = 1};

/**
 * DESCR:   verify low level
 * PARA:    showtrace: show a trace to the bad state (not implemented)
(input)
 *          subname: low level name ("all" means all the low levels)
(input)
 *          errmsg: returned errmsg (output)
 *          pinfo: returned system infomation (output)
 *          pnextlow: next low level sub index(initially,it must be 0,
mainly
 *                    used for "all") (input)
 *          saveproduct: whether to save syn-product (input)
 *          savepath: where to save syn-product (input)
 * RETURN: 0: successsful < 0: error happened (See errmsg.h)
 * */
int verify_low(
        HISC_TRACETYPE showtrace,
        std::string & errmsg,
        HISC_SUPERINFO *pinfo);

#endif
```

BddSd.cpp

```
 *          errmsg: returned errmsg (output)
 * RETURN: 0: sucess  < 0: fail
 * */
int load_prj(const char *prjfile, string & errmsg)
```

```
{
    int iRet = 0;

    assert(prjfile != NULL);

    pSub = new CLowSub(prjfile);

    iRet = pSub->LoadSub();

    errmsg = pSub->GetErrMsg();
    if (pSub->GetErrCode() < 0)
    {
        if (pSub->GetErrCode() > HISC_WARN_BLOCKEVENTS) //error happened
        {
            delete pSub;
            pSub = NULL;
        }
        //else only a warning
    }
    return iRet;
}

/**
 * DESCR:   close opened HISC project
 * PARA:    errmsg: returned errmsg (output)
 * RETURN:  0: sucess  < 0: fail
 * */
int close_prj(string & errmsg)
{
    int iRet = 0;

    if (NULL != pSub)
    {
        errmsg = pSub->GetErrMsg();
```

```
        iRet = pSub->GetErrCode();
        if (pSub->GetErrCode() < 0)
        {
            delete pSub;
            pSub = NULL;
        }
    }
    return iRet;
}

/**
 * DESCR:  clear the HISC enviorment
 * PARA:   none
 * RETURN: 0
 * */
int close_hisc()
{
    if (pSub != NULL)
    {
        delete pSub;
        pSub = NULL;
    }
    return 0;
}

/**
 * DESCR:   Save the project in the memory to a text file, just for verifying
 *          the loaded project.
 * PARA:    filename: where to save the text file (input)
 *          errmsg: returned errmsg (output)
 * RETURN:  0: sucess  < 0: fail
 * */
int print_prj(string filename, string & errmsg)
{
```

```
    int iRet = 0;
    assert(!filename.empty());
    assert(!errmsg.empty());

    ofstream fout;
    try
    {
        fout.open(filename.data());
        if (!fout)
            throw -1;

        if (pSub->PrintSubAll(fout) < 0)
            throw -1;
        fout.close();
    }
    catch(...)
    {
        if (fout.is_open())
            fout.close();
        pSub->SetErr(filename + ":Unable to create the print file.",
                HISC_BAD_PRINT_FILE);
        return -1;
    }

    return 0;

    errmsg = pSub->GetErrMsg();

    iRet = pSub->GetErrCode();

    pSub->ClearErr();

    return iRet;
}
```

```
/**
 * DESCR:   verify low level
 * PARA:    showtrace: show a trace to the bad state (not implemented)
(input)
 *          subname: low level name ("all" means all the low levels)
(input)
 *          errmsg: returned errmsg (output)
 *          pinfo: returned system infomation (output)
 *          pnextlow: next low level sub index(initially,it must be 0,
mainly
 *                    used for "all") (input)
 *          saveproduct: whether to save syn-product (input)
 *          savepath: where to save syn-product (input)
 * RETURN: 0: successsful < 0: error happened (See errmsg.h)
 * */
int verify_low(
        HISC_TRACETYPE showtrace,
        string & errmsg,
        HISC_SUPERINFO *pinfo)
{
    assert(pinfo != NULL);

    int iRet = 0;

        time_t tstart;
        time(&tstart);

        if (pSub->VeriSub(showtrace, *pinfo) < 0)
        {
            errmsg = pSub->GetErrMsg();
            iRet = pSub->GetErrCode();
            pSub->ClearErr();
        }

        time_t tend;
```

```
        time(&tend);
        pinfo->time = tend - tstart;

    return iRet;
}


```

A.2.3 DES Class

DES.h

```
    virtual ~CDES();

public:
    int LoadDES();
    int PrintDES(ofstream & fout);

public:
    string GetDESName() const {return m_sDESName;};
    int * GetEventsArr() {return m_piEventsArr;};
    int GetNumofEvents() const {return m_DESEventsMap.size();};
    int GetNumofMarkingStates() const {return m_MarkingList.size();};
    MARKINGLIST & GetMarkingList() {return m_MarkingList;};
    int GetNumofStates() const { return m_iNumofStates;};
    int GetInitState() const {return m_iInitState;};
    map<int, int> *GetTrans() const {return m_pTransArr;};
    DESTYPE GetDESType() const {return m_DESType;};
    CSub*  GetSub() {return m_pSub;};
    string GetStateName(int iState) {return m_InvStatesMap[iState];};

    EVENTS m_DESEventsMap;  //A STL Map for events (event name (key),
                            //local event index). Used only for current
DES
                            //(speed reason)

```

```
private: //data memeber
    string m_sDESFile;   //DES file name with path
    string m_sDESName;   //DES name without path and file extension
    DESTYPE m_DESType;   //DES type

    int m_iNumofStates; //Number of States
    int m_iInitState;  //Initial state

    MARKINGLIST m_MarkingList; //Link list containing all marking
states

    STATES m_StatesMap; //A STL Map for states (state name (key), state
index)
    INVSTATES m_InvStatesMap; //A STL Map for states (state index
(key),
                                    //state name)(for printing)

    INVEVENTS m_InvDESEventsMap; //A STL Map for events (localindex
(key),
                                      //event name). Used only for current
DES
                                      //(for printing)
    EVENTS m_UnusedEvents; //A STL Map for blocked events(name: key,
index)

    int *m_piEventsArr;  //Save all the event indices ascendingly.
                         //used for find shared events between two
DESes.

    TRANS *m_pTransArr; //Transiton Map array, indexed by event
indices.
                        //TRANSMAP: first int: source state index
                        //          second int: target state index
    CSub *m_pSub; //which subsystem this DES belongs to
private: //internal function members
```

```
    int AddEvent(const string & vsEventName,
                 const char cControllable);
    int AddTrans(const string & vsLine,
                 const string & vsExitState,
                 const int viExitState);
};

#endif //_DES_H_


```

DES.cpp

```
*           vsDESFile:  DES file name with path (input)
*           vDESType:   DES Type (inpute)
* RETURN: none
* ACCESS: public
*/
CDES::CDES(CSub *vpSub, const string &vsDESFile, const DESTYPE vDESType)
{
    m_pSub = vpSub;
    m_sDESFile = vsDESFile;
    m_sDESName.clear();
    m_DESType = vDESType;
    m_iNumofStates = 0;
    m_iInitState = -1;

    m_MarkingList.clear();
    m_StatesMap.clear();
    m_InvStatesMap.clear();
    m_DESEventsMap.clear();
    m_UnusedEvents.clear();
    m_InvDESEventsMap.clear();

```

```
    m_piEventsArr = NULL;
    m_pTransArr = NULL;
}

/**
 * DESCR:   Destructor
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
CDES::~CDES()
{
    delete[] m_pTransArr;
    m_pTransArr = NULL;

    delete[] m_piEventsArr;
    m_piEventsArr = NULL;
}

/**
 * DESCR:   Loading DES file
 * PARA:    None
 * RETURN:  0: sucess  -1: fail
 * ACCESS:  public
 */
int CDES::LoadDES()
{
    ifstream fin;
    int iRet = 0;
    string sErr;

    int i = 0;

    string sSubName = m_pSub->GetSubName();

```

```
    try
    {
        m_sDESFile = str_trim(m_sDESFile);

        if (m_sDESFile.length() <= 4)
        {
            pSub->SetErr(sSubName + ": Invalid DES file name " +
m_sDESFile,
                                        HISC_BAD_DES_FILE);
            throw -1;
        }

        if (m_sDESFile.substr(m_sDESFile.length() - 4) != ".hsc")
        {
            pSub->SetErr(sSubName + ": Invalid DES file name " +
m_sDESFile,
                                        HISC_BAD_DES_FILE);
            throw -1;
        }

        fin.open(m_sDESFile.data(), ios::in);

        //unable to find DES file
        if (!fin)
        {
            pSub->SetErr(sSubName + ": Unable to open the DES file " +
                                m_sDESFile, HISC_BAD_DES_FILE);
            throw -1;
        }

        m_sDESName = GetNameFromFile(m_sDESFile);

        string sDESLoc = sSubName + ":" + m_sDESName + " : ";
        char scBuf[MAX_LINE_LENGTH];
        string sLine;
```

```
        int iField = -1; //0: States 1: InitState 2: MarkingStates
                         //3: Events 4: Transitions
        char *scFieldArr[] = {"STATES", "INITSTATE", "MARKINGSTATES",
                                "EVENTS", "TRANSITIONS"};
        int iStatesFieldFlag = 0;  //1: just finised reading the [States] line,
                                   //   so next line should be the num of states
                                   //0: otherwise
        int iTmpStateIndex = 0;
        int iTmpEventIndex = 0;
        char cEventSub = '\0';
        char cControllable = '\0';

        string sExitState;
        int iExitState = -1;

        while (fin.getline(scBuf,  MAX_LINE_LENGTH))
        {
            sLine = str_nocomment(scBuf);
            sLine = str_trim(sLine);

            if (sLine.empty())
                continue;

            //Field line
            if (sLine[0] == '[' && sLine[sLine.length() - 1] == ']')
            {
                sLine = sLine.substr(1, sLine.length() - 1);
                sLine = sLine.substr(0, sLine.length() - 1);
                sLine = str_upper(str_trim(sLine));

                iField++;

```

```
            if (iField <= 4)
            {
                if (sLine != scFieldArr[iField])
                {
                    pSub->SetErr(sDESLoc +
                                "Field name or order is incorrect!",
                                HISC_BAD_DES_FORMAT);
                    throw -1;
                }
                if (iField == 0)
                {
                    iStatesFieldFlag = 1;
                }
            }
            else
            {
                pSub->SetErr(sDESLoc + "Two many fields.",
                            HISC_BAD_DES_FORMAT);
                throw -1;
            }
        }
        else  //Data line
        {
            switch (iField)
            {
            case 0:  //States
                if (iStatesFieldFlag == 1) //num of states
                {
                    if (atoi(sLine.data()) <= 0 ||
                       atoi(sLine.data()) >
MAX_STATES_IN_ONE_COMPONENT_DES)
                    {
                        pSub->SetErr(sDESLoc + "Too few or too many
states",
                                        HISC_BAD_DES_FORMAT);
```

```
                    throw -1;
                }

                //initialize the number of states
                m_iNumofStates = atoi(sLine.data());

                //initialize the transition arrary
                m_pTransArr = new TRANS[m_iNumofStates];

                iStatesFieldFlag = 0;
            }
            else
            {
                if (m_StatesMap.find(sLine) != m_StatesMap.end())
                {
                    pSub->SetErr(sDESLoc + "Duplicate state names--" +
                                        sLine, HISC_BAD_DES_FORMAT);
                    throw -1;
                }
                else if (sLine[0] == '(')
                {
                    pSub->SetErr(sDESLoc +
                        "The first letter of state names can not be (",
                        HISC_BAD_DES_FORMAT);
                    throw -1;
                }
                else
                {
                    m_StatesMap[sLine] = m_StatesMap.size() - 1;
                    m_InvStatesMap[m_StatesMap.size() - 1] = sLine;
```

```
                    }
                }

                break;
            case 1:  //InitState

//---------------------------------------------------
                //Must specify the number of states
                if (m_iNumofStates == 0)
                {
                    pSub->SetErr(sDESLoc + "Number of states is absent.",
                                      HISC_BAD_DES_FORMAT);
                    throw -1;
                }

                //If there is no state names specified, generate state
                //names automatically.
                if (m_StatesMap.size() == 0)
                {
                    for (i = 0; i < m_iNumofStates; i++)
                    {
                        m_StatesMap[str_itos(i)] = i;
                        m_InvStatesMap[i] = str_itos(i);
                    }
                }

                //if specify state names, the number of state names must be
                //equal to m_iNumofStates.
                if (((unsigned int)m_iNumofStates) != m_StatesMap.size())
                {
```

```
                pSub->SetErr(sDESLoc + "States are incomplete.",
                        HISC_BAD_DES_FORMAT);
                throw -1;
            }


//-----------------------------------------------------------

            //Initial state name must be valid
            if (m_StatesMap.find(sLine) == m_StatesMap.end())
            {
                pSub->SetErr(sDESLoc + "Initial state is not defined.",
                            HISC_BAD_DES_FORMAT);
                throw -1;
            }

            //only one initial state allowed
            if (m_iInitState != -1)
            {
                pSub->SetErr(sDESLoc + "More than one initial states.",
                            HISC_BAD_DES_FORMAT);
                throw -1;
            }

            m_iInitState = m_StatesMap[sLine];

            break;
        case 2:  //MarkingStates

            if (m_StatesMap.find(sLine) == m_StatesMap.end())
            {
                pSub->SetErr(sDESLoc + "Marking states do not exist.",
```

```
                              HISC_BAD_DES_FORMAT);
                    throw -1;
                }

                iTmpStateIndex = m_StatesMap[sLine];

                for (MARKINGLIST::const_iterator ci =
m_MarkingList.begin();
                    ci != m_MarkingList.end(); ci++)
                {
                    if (*ci == iTmpStateIndex)
                    {
                        pSub->SetErr(sDESLoc + "Duplicate marking
states.",
                                  HISC_BAD_DES_FORMAT);
                        throw -1;
                    }
                }

                m_MarkingList.push_back(iTmpStateIndex);

                break;
            case 3:  //Events

                //Get event type H/R/A/L
                if (sLine.length() < 5)
                {
                    pSub->SetErr(sDESLoc + "Incorrect event definition.",
                              HISC_BAD_DES_FORMAT);
                    throw -1;
                }
                cEventSub = sLine[sLine.length() - 1];
                sLine = str_trim(sLine.substr(0, sLine.length() -
1));

```

```
                //Get controllable or not
                if (sLine.length() < 3)
                {
                    pSub->SetErr(sDESLoc + "Incorrect event definition.",
                                    HISC_BAD_DES_FORMAT);
                    throw -1;
                }
                cControllable = sLine[sLine.length() - 1];
                sLine = str_trim(sLine.substr(0, sLine.length() - 1));

                //Get event name
                if (sLine.empty())
                {
                    pSub->SetErr(sDESLoc + "Incorrect event definition.",
                                HISC_BAD_DES_FORMAT);
                    throw -1;
                }

                if (cEventSub >= 'a')
                    cEventSub -= 32;
                if (cControllable >= 'a')
                    cControllable -= 32;

                iTmpEventIndex = AddEvent(sLine, cControllable);
                if (iTmpEventIndex < 0)
                    throw -1;   //Errmsg generated by AddEvent

                m_DESEventsMap[sLine] = iTmpEventIndex;
                m_UnusedEvents[sLine] = iTmpEventIndex;
                m_InvDESEventsMap[iTmpEventIndex] = sLine;
                break;

            case 4: //Transitions
                //check exiting state
```

```
                if (sLine[0] != '(')
                {
                    if (m_StatesMap.find(sLine) == m_StatesMap.end())
                    {
                        pSub->SetErr(sDESLoc + "Exiting state:" + sLine +
                            " in transitions does not exist",
                            HISC_BAD_DES_FORMAT);
                        throw -1;
                    }
                    iExitState = m_StatesMap[sLine];
                    sExitState = sLine;
                }
                else  //Transitions
                {
                    if (AddTrans(sLine, sExitState, iExitState) < 0)
                        throw -1;
                }
                break;
            default:
                pSub->SetErr(sDESLoc + "Bad DES file format!",
                                HISC_BAD_DES_FORMAT);
                throw -1;
            }
        }
    }

    //No initial state defined
    if (m_iInitState == -1)
    {
        pSub->SetErr(sDESLoc + "No initial state.", HISC_BAD_DES_FORMAT);
        throw -1;
```

```
        }
        //No marking states defined
        if (m_MarkingList.size() == 0)
        {
            pSub->SetErr(sDESLoc + "No marking states", HISC_BAD_DES_FORMAT);
            throw -1;
        }
        //must have all the fields
        if (iField != 4)
        {
            pSub->SetErr(sDESLoc + "Incomplete DES file.", HISC_BAD_DES_FORMAT);
            throw -1;
        }

        //Add event indices into m_piEventsArr;
        m_piEventsArr = new int[m_DESEventsMap.size()];
        i = 0;
        for (EVENTS::const_iterator ci = m_DESEventsMap.begin();
             ci != m_DESEventsMap.end(); ++ci)
        {
            m_piEventsArr[i] = ci->second;
            ++i;
        }
        qsort(m_piEventsArr, m_DESEventsMap.size(), sizeof(int), CompareInt);

        //unused events
        if (m_UnusedEvents.size() > 0)
        {
            string sWarn;
            sWarn = "\nWarning: ";
            sWarn += "Unused events are disabled at every state of DES " + sDESLoc + "\n";
```

```
            for (EVENTS::const_iterator ci = m_UnusedEvents.begin();
                    ci != m_UnusedEvents.end(); ++ci)
            {
                sWarn += ci->first;
                sWarn += "\n";
            }
            pSub->SetErr(sWarn, HISC_WARN_BLOCKEVENTS);
        }

        fin.close();
    }
    catch (int iError)
    {
        if (fin.is_open())
            fin.close();
        iRet = iError;
    }
    return iRet;
}

/*
 * DESCR:   Add an event to CSub event map and CProject event map
 *          For CSub event map: If exists, return local index;
 *                                      Otherwise create a new one.
 *          For CProject event map: If exists, must have same global
index;
 *                                          Otherwise the event sets are not
disjoint
 * PARA:    vsEventName: Event name(input)
 *          cEventSub: Event type ('H", 'L', 'R', 'A')(input)
 *          cControllable: Controllable? ('Y', 'N')(input)
 * RETURN:  >0 global event index
 *          <0 the event sets are not disjoint.
 * ACCESS:  Private
 */
```

```
int CDES::AddEvent(const string & vsEventName, const char
cControllable)
{
    string sErr;

    int iTmpEventIndex = 0;
    int iTmpLocalEventIndex = 0;

    string sDESLoc = m_pSub->GetSubName() + ": " + m_sDESName + ": ";

    //Controllable or uncontrollable
    if (cControllable != 'Y' && cControllable != 'N')
    {
        pSub->SetErr(sDESLoc + "Unknown event controllable type--"
+vsEventName,
                                HISC_BAD_DES_FORMAT);
        return -1;
    }

    //already defined in current DES
    if (m_DESEventsMap.find(vsEventName) != m_DESEventsMap.end())
    {
        pSub->SetErr(sDESLoc + "Duplicate events definition--" + vsEventName,
            HISC_BAD_DES_FORMAT);
        return -1;
    }

    //Compute local event index
    iTmpLocalEventIndex = m_pSub->AddSubEvent(vsEventName,
             (cControllable == 'Y')? CON_EVENT:UNCON_EVENT);

    if ((cControllable == 'Y' && iTmpLocalEventIndex % 2 == 0) ||
       (cControllable == 'N' && iTmpLocalEventIndex % 2 == 1))
    {
        pSub->SetErr(sDESLoc + "Event " + vsEventName +
```

```
                    " has inconsistent controllability definitions.",
                    HISC_BAD_DES_FORMAT);
        return -1;
    }
    //Compute global event index
    iTmpEventIndex = pSub->GenEventIndex(iTmpLocalEventIndex);

    //Add Event to pSub->m_AllEventsMap
    if (pSub->AddPrjEvent(vsEventName, iTmpEventIndex) < 0)
    {
        sErr = "Event conflict--" +  m_pSub->GetSubName() + ":" +
                this->GetDESName() + ":" +
                vsEventName + " is also defined in sub " + " event";
        pSub->SetErr(sErr, HISC_BAD_DES_FORMAT);
        iTmpEventIndex = -1;
    }

    return iTmpEventIndex;
}

/*
 * DESCR:   Add a transition to the m_pTransArr of the current DES.
 * PARA:    vsLine: a text line in [Transition] field(input)
 *          vsExitState: source state name of the transition(input)
 *          viExitState: source state index of the transition(input)
 * RETURN:  0: success -1: fail
 * ACCESS:  private
 */
int CDES::AddTrans(const string & vsLine,
                   const string & vsExitState,
                   const int viExitState)
{
    string sTrans = vsLine;

    string sEnterState;
```

```
    int iEnterStateIndex;

    string sTransEvent;
    int iTransEventIndex;

    unsigned long iSepLoc = string::npos;
    string sErrMsg;

    string sDESLoc = m_pSub->GetSubName() + ": " + m_sDESName + ": ";


    try
    {
        if (viExitState == -1)
        {
            pSub->SetErr(sDESLoc + "No existing state for transitions",
                                                HISC_BAD_DES_FORMAT);
            throw -1;
        }

        //remove '(' and ')'
        sTrans = sTrans.substr(1);
        sTrans = sTrans.substr(0, sTrans.length() - 1);
        sTrans = str_trim(sTrans);

        //find sepration character '\t' or ' '
        iSepLoc = sTrans.find_last_of('\t');
        if (iSepLoc == string::npos)
            iSepLoc = sTrans.find_last_of(' ');

        if (iSepLoc == string::npos)
        {
            pSub->SetErr(sDESLoc +
                        "No event or entering state for transition. (" +
                        sTrans + ")", HISC_BAD_DES_FORMAT);
```

```
            throw -1;
        }
        else
        {
            sEnterState = str_trim(sTrans.substr(iSepLoc + 1));
            sTransEvent = str_trim(sTrans.substr(0, iSepLoc));
        }

        //Check event in transitions
        if (m_DESEventsMap.find(sTransEvent) == m_DESEventsMap.end())
        {
            pSub->SetErr(sDESLoc + "Event " + sTransEvent +
                    " in transitions does not exist.",
                    HISC_BAD_DES_FORMAT);
            throw -1;
        }
        iTransEventIndex = m_DESEventsMap[sTransEvent];
        m_UnusedEvents.erase(sTransEvent);

        //Check entering state
        if (m_StatesMap.find(sEnterState) == m_StatesMap.end())
        {
            pSub->SetErr(sDESLoc + "State " + sEnterState +
                    " in transitions does not exist.",
                    HISC_BAD_DES_FORMAT);
            throw -1;
        }
        iEnterStateIndex = m_StatesMap[sEnterState];

        //Check determinacy
        if (m_pTransArr[viExitState].find(iTransEventIndex) !=
            m_pTransArr[viExitState].end())
        {
            pSub->SetErr(sDESLoc + "ExitState:" + vsExitState +
                    " has nondeterministic transitions on event " +
```

```
sTransEvent,
                        HISC_BAD_DES_FORMAT);
                throw -1;
            }
            m_pTransArr[viExitState][iTransEventIndex] = iEnterStateIndex;
        }
        catch(int)
        {
            return -1;
        }

        return 0;
    }

    /*
     * DESCR:   Print this DES in memory to a file (for checking)
     * PARA:    fout: file stream(input)
     * RETURN:  0: success -1: fail
     * ACCESS:  public
     */
    int CDES::PrintDES(ofstream & fout)
    {
        try
        {
            int i = 0;

            fout << endl << "#——DES: " << m_sDESName << " ---------" <<
endl;
            fout << "[States]" << endl;
            fout << m_iNumofStates << endl;

            for (INVSTATES::const_iterator ci = m_InvStatesMap.begin();
                 ci != m_InvStatesMap.end(); ++ci)
            {
                fout << ci->second << endl;
```

```
        }

        fout << endl;
        fout << "[InitState]" << endl;
        fout << m_InvStatesMap[m_iInitState] << endl;

        fout << endl;
        fout << "[MarkingStates]" << endl;
        for (MARKINGLIST::const_iterator ci = m_MarkingList.begin();
             ci != m_MarkingList.end(); ++ci)
        {
            fout << m_InvStatesMap[*ci] << endl;
        }

        fout << endl;
        fout << "[Events]" << endl;
        for (INVEVENTS::const_iterator ci = m_InvDESEventsMap.begin();
             ci != m_InvDESEventsMap.end(); ++ci)
        {
            if (ci->first % 2 == 0)  //uncontrollable
                fout << ci->second << "\t" << "N" << "\tL" << endl;
            else
                fout << ci->second << "\t" << "Y" << "\tL" << endl;
        }

        fout << endl;
        fout << "[Transitions]" << endl;
        if (m_pTransArr != NULL)
        {
            for (i = 0; i < m_iNumofStates; i++)
            {
                fout << m_InvStatesMap[i] << endl;

                for (TRANS::const_iterator ci =
(m_pTransArr[i]).begin();
```

```
                        ci != (m_pTransArr[i]).end(); ++ci)
                   {
                       fout << "(" << m_InvDESEventsMap[ci->first] << " "
                           << m_InvStatesMap[ci->second] << ")" << endl;
                   }
               }
           }

           fout <<
"################################################"
<< endl;
       }
       catch(...)
       {
           return -1;
       }
       return 0;
}


```

A.2.4 Sub Class

Sub.h

```
virtual ~CSub();
virtual unsigned short AddSubEvent(const string & vsEventName,
                                   const EVENTTYPE vEventType);
virtual int PrintSub(ofstream & fout) = 0;
virtual int PrintSubAll(ofstream& fout) = 0;
virtual string SearchEventName(unsigned short usiLocalIndex) = 0;

virtual int LoadSub() = 0;
virtual int VeriSub(const HISC_TRACETYPE showtrace,
                    HISC_SUPERINFO & superinfo) = 0;

```

```
    void SetErr(const string & vsErrMsg, const int viErrCode);

    int GenEventIndex(const unsigned short vusiLocalEventIndex);
    int SearchPrjEvent(const string & vsEventName);
    int SearchSubEvent(const string & vsEventName);
    INVEVENTS & GetInvAllEventsMap() {return m_InvAllEventsMap;};

    string GetErrMsg() const {return m_sErrMsg;};
    int GetErrCode() const {return m_iErrCode;};
    void ClearErr();

    int AddPrjEvent(const string & vsEventName, const int
viEventIndex);

private:
    string m_sErrMsg; //Error msg during processing this project
    int m_iErrCode; //Error code during processing this project

public:  //access methods
    virtual string GetSubName() const {return m_sSubName;};

    virtual int GetNumofDES() const
                {return m_iNumofPlants + m_iNumofSpecs;};
    virtual unsigned short GetMaxUnCon()
                {return m_usiMaxUnCon;};
    virtual unsigned short GetMaxCon()
                {return m_usiMaxCon;};

private: //DES reorder related memebers
    int ** m_piCrossMatrix;
    int DESReorder_Sift();
    double TotalCross_Sift(double dOldCross, double dSwapCross,
                            int iCur, int iFlag);
    double cross(int i, int j);
    int DESReorder_Force();
```

```
    void UpdatePos();
    void InsertDES(int iCur, int iPos);
    double TotalCross_Force();
    double Force(int i);
    int InitialDESOrder();

protected:  //protected methods
    virtual string GetDESFileFromSubFile(const string & vsSubFile,
                       const string &vsDES);
    virtual int MakeBdd() = 0;
    virtual int InitBddFields();
    virtual int ClearBddFields();

    int DESReorder();

    int PrintStateSet(const bdd & bddStateSet, int viSetFlag);
    void PrintStateSet2(const bdd & bddStateSet);
    bdd GetOneState(const bdd & bddStates);
    int CountStates(const bdd & bddStateSet);

    int PrintEvents(ofstream & fout);
    int PrintTextTrans(ofstream & fout, bdd & bddController,
                           unsigned short usiLocalIndex,
                           const bdd & bddReach, string sEventName,
                           STATES & statesMap);
    bdd SimplifyController(const bdd & bddController, const unsigned short usiIndex);

protected:  //fields
    string m_sSubFile;   //this subsytem file name(".sub") with path.
    string m_sSubName;   //This subsystem name

    int m_iNumofPlants;   //Number of Plant DES
    int m_iNumofSpecs;    //Number of Specification DES
                          //(High: all interface DES; Low: 1)
```

```
    CDES **m_pDESArr;      //DES Array for all the DES in high or low
levels.
                           //(High: including all interface DES,
                           //Low: only including 1 DES for this
subsystem)
    EVENTSET m_SubPlantEvents;
    EVENTSET m_SubSupervisorEvents;
    LOCALEVENTS m_SubEventsMap; //save all the events map in this
subsytems
                                //(name(key), local index(16 bits))
                                //just for compute local event index.
    LOCALINVEVENTS m_InvSubEventsMap;
    EVENTS m_AllEventsMap; //The map containing all the events in this
project
                           //(Event Name (key), Event global index)
    INVEVENTS m_InvAllEventsMap; //The map containing all the events in
this
                                //project (Event global index (key),
Event Name)
    unsigned short m_usiMaxCon; //Max index of controllable events
(1,3,...)
    unsigned short m_usiMaxUnCon;//Max index of uncontrollable
events(2,4,..)
    /*BDD needed fields*/
    int m_iNumofBddNormVar; //Num of BDD normal variables in the sub.
    int *m_piDESOrderArr; //DES indices organized as clusters.
    int *m_piDESPosArr; //DES positions in the m_piDESOrderArr
    bdd m_bddInit;  //Initial state predicate
```

```
    bdd m_bddMarking;  //Marking states predicate
    bdd m_bddSuper; //The generated supervisor

    ////////////////////////////////////////////////////////////////////
    //Transition predicates and its variable sets, variable pairs.
    //0: High level events
    //1: Request events
    //2: Answer events
    //3: Low level events
    ////////////////////////////////////////////////////////////////////
    //Transition predicates
    bdd *m_pbdd_ConTrans;
    bdd *m_pbdd_ConPlantTrans;
    bdd *m_pbdd_ConSupTrans;
    bdd *m_pbdd_UnConTrans;
    bdd *m_pbdd_UnConPlantTrans;
    bdd *m_pbdd_UnConSupTrans;

    //variable(DES index) set for transition predicates
    bdd *m_pbdd_ConVar;
    bdd *m_pbdd_ConVarPrim;
    bdd *m_pbdd_UnConVar;
    bdd *m_pbdd_UnConVarPrim;
    //plant part variables
    bdd *m_pbdd_UnConPlantVar;
    bdd *m_pbdd_UnConPlantVarPrim;
    bdd *m_pbdd_ConPhysicVar;   //for simplifying controller (note: physical)
    bdd *m_pbdd_ConPhysicVarPrim;//for simplifying controller (note:physical)
    //supervisor part variables
    bdd *m_pbdd_UnConSupVar;
    bdd *m_pbdd_UnConSupVarPrim;
    bdd *m_pbdd_ConSupVar;   //for simplifying controller (note: physical)
```

```
    bdd *m_pbdd_ConSupVarPrim;//for simplifying controller (note:physical)
    //variable pairs(normal-prime)
    bddPair **m_pPair_Con;
    bddPair **m_pPair_UnCon;
    bddPair **m_pPair_ConPrim;
    bddPair **m_pPair_UnConPrim;
};
#endif //_SUB_H_
```

Sub.cpp

```
 */
CSub::CSub(const string & vsSubFile)
{
    m_AllEventsMap.clear();
    m_InvAllEventsMap.clear();

    m_iErrCode = 0;
    m_sErrMsg.clear();

    m_sSubFile = vsSubFile;
    m_sSubName.clear();

    m_iNumofPlants = -1;
    m_iNumofSpecs = -1;

    m_pDESArr = NULL;

    m_SubEventsMap.clear();

    m_usiMaxCon = 0xFFFF;
    m_usiMaxUnCon = 0x0;
```

```

    m_piDESOrderArr = NULL;
    m_piDESPosArr = NULL;

    InitBddFields();
}

/**
 * DESCR:   Destructor
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
CSub::~CSub()
{
    if (m_pDESArr != NULL)
    {
        int iNumofDES = this->GetNumofDES();

        for (int i = 0; i < iNumofDES; i++)
        {
            if (m_pDESArr[i] != NULL)
            {
                    delete m_pDESArr[i];
                    m_pDESArr[i] = NULL;
            }
        }
        delete[] m_pDESArr;
        m_pDESArr = NULL;
    }


    delete[] m_piDESOrderArr;
    m_piDESOrderArr = NULL;
    delete[] m_piDESPosArr;
```

```
    m_piDESPosArr = NULL;

    ClearBddFields();
}

/*
 * DESCR:   Initialize BDD related data members
 * PARA:    None
 * RETURN:  0
 * ACCESS:  protected
 */
int CSub::InitBddFields()
{
    m_iNumofBddNormVar = 0;
    m_bddInit = bddtrue;
    m_bddMarking = bddtrue;
    m_bddSuper = bddfalse;

        m_pbdd_ConTrans = NULL;
        m_pbdd_ConPlantTrans = NULL;
        m_pbdd_ConSupTrans = NULL;

        m_pbdd_UnConTrans = NULL;
        m_pbdd_UnConPlantTrans = NULL;
        m_pbdd_UnConSupTrans = NULL;

        m_pbdd_ConVar = NULL;
        m_pbdd_ConVarPrim = NULL;

        m_pbdd_UnConVar = NULL;
        m_pbdd_UnConVarPrim = NULL;

        m_pbdd_UnConPlantVar = NULL;
        m_pbdd_UnConPlantVarPrim = NULL;

```

```
        m_pbdd_UnConSupVar = NULL;
        m_pbdd_UnConSupVarPrim = NULL;

        m_pbdd_ConPhysicVar = NULL;
        m_pbdd_ConPhysicVarPrim = NULL;

        m_pbdd_ConSupVar = NULL;
        m_pbdd_ConSupVarPrim = NULL;

        m_pPair_Con = NULL;
        m_pPair_UnCon = NULL;
        m_pPair_ConPrim = NULL;
        m_pPair_UnConPrim = NULL;

    return 0;
}

/*
 * DESCR:   Release memory for BDD related data members
 * PARA:    None
 * RETURN:  0
 * ACCESS:  protected
 */
int CSub::ClearBddFields()
{
        delete[] m_pbdd_ConTrans;
        m_pbdd_ConTrans = NULL;

        delete[] m_pbdd_ConPlantTrans;
        m_pbdd_ConPlantTrans = NULL;

        delete[] m_pbdd_ConSupTrans;
        m_pbdd_ConSupTrans = NULL;

        delete[] m_pbdd_UnConTrans;
```

```
        m_pbdd_UnConTrans = NULL;

        delete[] m_pbdd_UnConPlantTrans;
        m_pbdd_UnConPlantTrans = NULL;

        delete[] m_pbdd_UnConSupTrans;
        m_pbdd_UnConSupTrans = NULL;

        delete[] m_pbdd_ConVar;
        m_pbdd_ConVar = NULL;
        delete[] m_pbdd_UnConVar;
        m_pbdd_UnConVar = NULL;

        delete[] m_pbdd_ConVarPrim;
        m_pbdd_ConVarPrim = NULL;
        delete[] m_pbdd_UnConVarPrim;
        m_pbdd_UnConVarPrim = NULL;

        delete[] m_pbdd_UnConPlantVar;
        m_pbdd_UnConPlantVar = NULL;
        delete[] m_pbdd_UnConPlantVarPrim;
        m_pbdd_UnConPlantVarPrim = NULL;

        delete[] m_pbdd_UnConSupVar;
        m_pbdd_UnConSupVar = NULL;
        delete[] m_pbdd_UnConSupVarPrim;
        m_pbdd_UnConSupVarPrim = NULL;

        delete[] m_pbdd_ConPhysicVar;
        m_pbdd_ConPhysicVar = NULL;
        delete[] m_pbdd_ConPhysicVarPrim;
        m_pbdd_ConPhysicVarPrim = NULL;

        delete[] m_pbdd_ConSupVar;
        m_pbdd_ConSupVar = NULL;
```

```
delete[] m_pbdd_ConSupVarPrim;
m_pbdd_ConSupVarPrim = NULL;

if (m_pPair_UnCon != NULL)
{
    for (int i = 0; i < m_usiMaxUnCon; i += 2)
    {
        if (m_pPair_UnCon[i/2] != NULL)
        {
            bdd_freepair(m_pPair_UnCon[i/2]);
            m_pPair_UnCon[i/2] = NULL;
        }
    }
    delete[] m_pPair_UnCon;
    m_pPair_UnCon = NULL;
}

if (m_pPair_Con != NULL)
{
    for (int i = 1; i < (unsigned short)(m_usiMaxCon + 1); i +=
2)
    {
        if (m_pPair_Con[(i - 1)/2] != NULL)
        {
            bdd_freepair(m_pPair_Con[(i - 1)/2]);
            m_pPair_Con[(i - 1)/2] = NULL;
        }
    }
    delete[] m_pPair_Con;
    m_pPair_Con = NULL;
}

if (m_pPair_UnConPrim != NULL)
{
    for (int i = 0; i < m_usiMaxUnCon; i += 2)
```

```
            {
                if (m_pPair_UnConPrim[i/2] != NULL)
                {
                    bdd_freepair(m_pPair_UnConPrim[i/2]);
                    m_pPair_UnConPrim[i/2] = NULL;
                }
            }
            delete[] m_pPair_UnConPrim;
            m_pPair_UnConPrim = NULL;
        }

        if (m_pPair_ConPrim != NULL)
        {
            for (int i = 1; i < (unsigned short)(m_usiMaxCon + 1); i += 2)
            {
                if (m_pPair_ConPrim[(i - 1)/2] != NULL)
                {
                    bdd_freepair(m_pPair_ConPrim[(i - 1)/2]);
                    m_pPair_ConPrim[(i - 1)/2] = NULL;
                }
            }
            delete[] m_pPair_ConPrim;
            m_pPair_ConPrim = NULL;
        }

    return 0;
}

/*
 * DESCR:   Generate a DES file name with path (*.hsc) from a sub file name
 *          with path (.sub) and a DES file name without path.
 *          ex: vsSubFile = "/home/roger/high.sub", vsDES = "AttchCase.hsc",
```

```
 *               will return "/home/roger/AttchCase.hsc"
 * PARA:      vsSubFile: sub file name with path
 *            vsDES: DES file name without path
 * RETURN:    Generated DES file name with path
 * ACCESS:    protected
 */
string CSub::GetDESFileFromSubFile(const string & vsSubFile,
                                   const string &vsDES)
{
    assert(vsSubFile.length() > 4);
    assert(vsSubFile.substr(vsSubFile.length() - 4) == ".sub");
    assert(vsDES.length() > 0);
    string sDES = vsDES;

    if (sDES.length() > 4)
    {
        if (sDES.substr(sDES.length() - 4) == ".hsc")
        {
            sDES = sDES.substr(0, sDES.length() - 4);
        }
    }
    sDES += ".hsc";

    unsigned int iPos = vsSubFile.find_last_of('/');

    if ( iPos == string::npos)
        return sDES;
    else
        return vsSubFile.substr(0, iPos + 1) + sDES;
}

/**
 * DESCR:   Add events to the event Map of this sub. If the event already
exits,
 *          return its index; Otherwise generate a new 16 bit unsigned
```

```
index
 *          and return the index.
 * PARA:    vsEventName:    Event name
 *          vEventType: Controllable? (CON_EVENT, UNCON_EVENT)
 * RETURN:  >0: event index (odd: controllable  even: uncontrollable)
 *          0:  error
 * ACCESS:  public
 */
unsigned short CSub::AddSubEvent(const string & vsEventName,
                                      const EVENTTYPE vEventType)
{
    const char * DEBUG = "CSub::AddSubEvent():";
    PRINT_DEBUG << "vsEventName = " << vsEventName << endl;

    LOCALEVENTS::const_iterator citer;

    citer = m_SubEventsMap.find(vsEventName);

    if (citer != m_SubEventsMap.end())  //the event exists, return its
index
        return citer->second;
    else  //the event does not exist, generate a new index.
    {
        if (vEventType == CON_EVENT)
        {
            m_usiMaxCon += 2;
            m_SubEventsMap[vsEventName] = m_usiMaxCon;
            m_InvSubEventsMap[m_usiMaxCon] = vsEventName;
            #ifdef DEBUG_TIME
            PRINT_DEBUG << "vEventType = CON_EVENT, m_usiMaxCon = "
<< m_usiMaxCon << endl;
            #endif

            return m_usiMaxCon;
        }
```

```
        else
        {
            m_usiMaxUnCon += 2;
            m_SubEventsMap[vsEventName] = m_usiMaxUnCon;
            m_InvSubEventsMap[m_usiMaxUnCon] = vsEventName;

            #ifdef DEBUG_TIME
            PRINT_DEBUG << "vEventType = UNCON_EVENT, m_usiMaxUnCon = " << m_usiMaxUnCon << endl;
            #endif

            return m_usiMaxUnCon;
        }
    }
    return 0;
}

/**
 * DESCR:   Set error msg and err code in this project
 * PARA:    vsvsErrMsg: Error message
 *          viErrCode: Error Code
 * RETURN:  None
 * ACCESS:  public
 */
void CSub::SetErr(const string & vsErrMsg, const int viErrCode)
{
    m_iErrCode = viErrCode;
    m_sErrMsg = vsErrMsg;
    return;
}

/**
 * DESCR:   Generate global event index from the event info in para
 * PARA:    viSubIndex(Sub index, highsub = 0, low sub start from 1.
 *                      Next 12 bits), (input)
```

```
 *          vusiLocalEventIndex(local event index, odd: controllable,
 *                                    even:uncontrollab. The rest 16 bits)
(input)
 * RETURN:  Generated global event index
 * ACCESS:  public
 */
int CSub::GenEventIndex(const unsigned short vusiLocalEventIndex)
{
    int iEventIndex = L_EVENT;
    iEventIndex = iEventIndex << 28;

    int iSubIndex = 1;
    iSubIndex = iSubIndex << 16;
    iEventIndex += iSubIndex;

    iEventIndex += vusiLocalEventIndex;

    return iEventIndex;
}

/*
 * DESCR:   Search an event by its name
 * PARA:    vsEventName: Event name(input)
 * RETURN:  >0: Gloable event index
 *          <0: not found
 * ACCESS:  public
 */
int CSub::SearchPrjEvent(const string & vsEventName)
{
    EVENTS::const_iterator citer;

    citer = m_AllEventsMap.find(vsEventName);

    if (citer != m_AllEventsMap.end())  //the event exists
        return citer->second;
```

```
    else //the event does not exist
        return -1;
}

/*
 * DESCR:   Search an event by its name
 * PARA:    vsEventName: Event name(input)
 * RETURN:  >0: Sub event index
 *          <0: not found
 * ACCESS:  public
 */
int CSub::SearchSubEvent(const string & vsEventName)
{
    LOCALEVENTS::const_iterator citer;

    citer = m_SubEventsMap.find(vsEventName);

    if (citer != m_SubEventsMap.end()) //the event exists
        return citer->second;
    else //the event does not exist
        return -1;
}

/**
 * DESCR:   Clear error msg and err code in this project
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
void CSub::ClearErr()
{
    m_iErrCode = 0;
    m_sErrMsg.empty();
    return;
}
```

```

/*
 * DESCR:   Add an event to CProject event map
 *          If the event exists already exists in the map, the  it
should have
 *          same global index;  Otherwise the event sets are not
disjoint
 * PARA:    vsEventName: Event name(input)
 *          viEventIndex: global event index (input)
 *          cEventSub: Event type ('H", 'L', 'R', 'A')
 *                     (output, only for new events)
 *          cControllable: Controllable? ('Y', 'N')(output)(only for
new events)
 * RETURN:  0: success
 *          <0 the event sets are not disjoint.
 * ACCESS:  public
 */
int CSub::AddPrjEvent(const string & vsEventName, const int
viEventIndex)
{
    EVENTS::const_iterator citer;

    citer = m_AllEventsMap.find(vsEventName);

    if (citer != m_AllEventsMap.end())  //the event exists, check if
the global
                                        //event index is same.
    {
        if (citer->second != viEventIndex)
        {
            return -1;
        }
    }
    else  //the event does not exist
    {
```

```
        m_AllEventsMap[vsEventName] = viEventIndex;
        m_InvAllEventsMap[viEventIndex] = vsEventName;
    }

    return 0;
}


```

Sub1.cpp

```
    //compute the marix storing number of shared events between every two DES
    m_piCrossMatrix = new int *[iNumofDES];
    for (int i = 0; i < iNumofDES; i++)
        m_piCrossMatrix[i] = new int[iNumofDES];
    for (int i = 0; i < iNumofDES; i++)
        for (int j = 0; j < iNumofDES; j++)
        {
            m_piCrossMatrix[i][j] =
                        NumofSharedEvents(m_pDESArr[i]->GetEventsArr(),
m_pDESArr[i]->GetNumofEvents(),
m_pDESArr[j]->GetEventsArr(),
m_pDESArr[j]->GetNumofEvents());
        }
    //Generate an initial order
    InitialDESOrder();
    UpdatePos();

    //Algorithm with force
    DESReorder_Force();
    UpdatePos();
```

```

    //sifting algorithm
    DESReorder_Sift();
    UpdatePos();

    //clear memory
    for (int i = 0; i < iNumofDES; i++)
    {
        delete[] m_piCrossMatrix[i];
        m_piCrossMatrix[i] = NULL;
    }
    delete[] m_piCrossMatrix;
    m_piCrossMatrix = NULL;

    //Order m_pDESArr according to the order of m_piDESOrderArr.
    CDES **pDESTmp = NULL;
    pDESTmp = new CDES *[this->GetNumofDES()];
    for (int i = 0; i < this->GetNumofDES(); i++)
    {
        pDESTmp[i] = m_pDESArr[m_piDESOrderArr[i]];
    }
    for (int i = 0; i < this->GetNumofDES(); i++)
    {
        m_pDESArr[i] = pDESTmp[i];
    }
    delete[] pDESTmp;
    pDESTmp = NULL;

    return 0;
}

/*
 * DESCR:   Using sifting algorithm to reorder DES
 * PARA:    None
 * RETURN:  0
```

```
 * ACCESS:  private
 */
int CSub::DESReorder_Sift()
{
    int iNumofDES = this->GetNumofDES();
    bool bChanged = false;
    double dMinCross = 0.0;
    double dCurCross = 0.0;
    int *piCurOpt = new int[iNumofDES];
    int *piInit = new int[iNumofDES];
    int iTemp = 0;
    int iCur = 0;
    int iCount = 0;
    double dOldCross = 0.0;
    double dInitCross = 0.0;
    double dSwapCross = 0.0;

    //initialize optimal des order and loop initial order;
    for (int j = 0; j < iNumofDES; j++)
    {
        piCurOpt[j] = m_piDESOrderArr[j];
        piInit[j] = m_piDESOrderArr[j];
    }

    //initialize cross over value
    dMinCross = TotalCross_Sift(0, 0, 0, 0);
    dOldCross = dMinCross;
    dInitCross = dMinCross;

    //Initialize m_piDESPosArr
    UpdatePos();

    //Optimize the DES order
    do
    {
```

```
iCount++;
bChanged = false;
for (int iDES = 0; iDES < iNumofDES; iDES++)
{
    iCur = m_piDESPosArr[iDES];

    //move backward
    for (int i = iCur; i < iNumofDES - 1; i++)
    {
        //compute dSwapCross
        dSwapCross = TotalCross_Sift(0, 0, i, 1);

        //swap i, i+1
        iTemp = m_piDESOrderArr[i + 1];
        m_piDESOrderArr[i + 1] = m_piDESOrderArr[i];
        m_piDESOrderArr[i] = iTemp;

        //test if current order is better
        dCurCross = TotalCross_Sift(dOldCross, dSwapCross, i,
2);
        dOldCross = dCurCross;
        if (dCurCross - dMinCross < 0)
        {
            bChanged = true;
            dMinCross = dCurCross;
            for (int j = 0; j < iNumofDES; j++)
                piCurOpt[j] = m_piDESOrderArr[j];
        }
    }

    //move forward
    for (int j = 0; j < iNumofDES; j++)
        m_piDESOrderArr[j] = piInit[j];
    dOldCross = dInitCross;
    for (int i = iCur; i > 0; i--)
```

```
        {
            //compute dSwapCross
            dSwapCross = TotalCross_Sift(0, 0, i - 1, 1);

            //swap i - 1, i
            iTemp = m_piDESOrderArr[i - 1];
            m_piDESOrderArr[i - 1] = m_piDESOrderArr[i];
            m_piDESOrderArr[i] = iTemp;

            //test if current order is better
            dCurCross = TotalCross_Sift(dOldCross, dSwapCross, i -
1, 2);
            dOldCross = dCurCross;
            if (dCurCross - dMinCross < 0)
            {
                bChanged = true;
                dMinCross = dCurCross;
                for (int j = 0; j < iNumofDES; j++)
                    piCurOpt[j] = m_piDESOrderArr[j];
            }
        }
        dInitCross = dMinCross;
        dOldCross = dMinCross;
        if (bChanged)
        {
            for (int j = 0; j < iNumofDES; j++)
            {
                m_piDESOrderArr[j] = piCurOpt[j];
                piInit[j] = m_piDESOrderArr[j];
            }
            UpdatePos();
        }
        else
        {
            for (int j = 0; j < iNumofDES; j++)
```

```
                    m_piDESOrderArr[j] = piInit[j];
                }
            }
    }while(bChanged == true );

    delete[] piCurOpt;
    piCurOpt = NULL;

    delete[] piInit;
    piInit = NULL;

    return 0;
}

/*
 * DESCR:   Compute total cross for sifting algorithm
 * PARA:    dOldCross:  old cross value
 *          dSwapCross: cross changed due to swapping
 *          iCur: current position
 *          iFlag: 0: completey compute total cross value
 *                 1: compute total cross based on the old cross and
swapped DES
 *                    (much faster)
 * RETURN:  new cross value
 * ACCESS:  private
 */
double CSub::TotalCross_Sift(double dOldCross, double dSwapCross,
                             int iCur, int iFlag)
{
    double dCross = 0;

    if (iFlag == 0) //completely compute the cross
    {
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
```

```
            for (int j = i + 2; j < this->GetNumofDES(); j++)
                dCross += cross(i, j);
        }
    }
    else if (iFlag == 1) //only compute iCur, iCur + 1
    {
        //iCur
        for (int i = 0; i < iCur - 1; i++)
            dCross += cross(i, iCur);
        for (int i = iCur + 2; i < this->GetNumofDES(); i++)
            dCross += cross(iCur, i);
        //iCur + 1
        for (int i = 0; i < (iCur + 1) - 1; i++)
            dCross += cross(i, iCur + 1);
        for (int i = (iCur + 1) + 2; i < this->GetNumofDES(); i++)
            dCross += cross(iCur + 1, i);
    }
    else //update
    {
        //iCur
        for (int i = 0; i < iCur - 1; i++)
            dCross += cross(i, iCur);
        for (int i = iCur + 2; i < this->GetNumofDES(); i++)
            dCross += cross(iCur, i);
        //iCur + 1
        for (int i = 0; i < (iCur + 1) - 1; i++)
            dCross += cross(i, iCur + 1);
        for (int i = (iCur + 1) + 2; i < this->GetNumofDES(); i++)
            dCross += cross(iCur + 1, i);

        dCross = dOldCross - dSwapCross + dCross;
    }
    return dCross;
}
```

```
/*
 * DESCR:   Compute the cross for DES i and DES j
 * PARA:    i,j: DES position index,
 * RETURN:  the cross for DES i and DES j
 * ACCESS:  private
 */
double CSub::cross(int i, int j)
{
    return sqrt((double)(m_piCrossMatrix[m_piDESOrderArr[i]]
                                        [m_piDESOrderArr[j]]) * (j - i
- 1));
}

/*
 * DESCR:   Initialize a DES order for the sifting reorder algorithm
 *          (some ideas are from Zhonghua Zhong's STCT)
 * PARA:    None
 * RETURN:  0
 * ACCESS:  private
 */
int CSub::DESReorder_Force()
{
    int iNumofDES = this->GetNumofDES();
    int iCount = 0;

    //Optimize the DES order
    bool bChanged = false;
    double dMinCross = TotalCross_Force();
    double dCurCross = 0.0;
    do
    {
        iCount++;
        bChanged = false;
        int iOptPos = 0;
        int iDES = 0;
```

```
        for (iDES = 0; iDES < iNumofDES; iDES++)
        {
            int iPrePos = 0;
            int iNextPos = iNumofDES - 1;
            int iPos = m_piDESPosArr[iDES];
            iOptPos = iPos;
            int iNewPos = 0;
            while (true)
            {
                double dForce = Force(iPos);
                if (dForce < -0.05)
                {
                    iNextPos = iPos;
                    iNewPos = iPos - (((iPos - iPrePos) % 2 == 0)?
                        ((iPos - iPrePos) / 2):((iPos - iPrePos) / 2 +
1));
                    if (iNewPos <= iPrePos)
                        break;
                    InsertDES(iPos, iNewPos);
                    UpdatePos();

                    iPos = iNewPos;
                    dCurCross = TotalCross_Force();
                    if (dCurCross < dMinCross - 0.05)
                    {
                        iOptPos = iPos;
                        dMinCross = dCurCross;
                        bChanged = true;
                    }
                }
                else if (dForce > 0.05)
                {
                    iPrePos = iPos;
                    iNewPos = iPos + (((iNextPos - iPos) % 2 == 0)?
                        ((iNextPos - iPos) / 2):((iNextPos - iPos) / 2
```

```
+ 1));
                    if (iNextPos <= iNewPos)
                        break;
                    InsertDES(iPos, iNewPos);
                    UpdatePos();

                    iPos = iNewPos;
                    dCurCross = TotalCross_Force();
                    if (dCurCross < dMinCross - 0.05)
                    {
                        iOptPos = iPos;
                        dMinCross = dCurCross;
                        bChanged = true;
                    }
                }
                else
                    break;
            }
            InsertDES(m_piDESPosArr[iDES], iOptPos);
            UpdatePos();
        }
    }while(bChanged == true);

    return 0;
}

/*
 * DESCR:   Update DES position in array m_piDESPosArr according the new order
 * PARA:    None
 * RETURN:  None
 * ACCESS:  private
 */
void CSub::UpdatePos()
{
```

```
    for (int i = 0 ; i < this->GetNumofDES(); i++)
        m_piDESPosArr[m_piDESOrderArr[i]] = i;
    return;
}

/*
 * DESCR:   Swap variables in m_piDESOrderArr for DESReorder_Force()
 * PARA:    iCur: current variable position
 *          iPos: destinate variable position
 * RETURN:  None
 * ACCESS:  private
 */
void CSub::InsertDES(int iCur, int iPos)
{
    int iDES = m_piDESOrderArr[iCur];
    if (iCur < iPos)
    {
        for (int i = iCur + 1; i <= iPos; i++)
            m_piDESOrderArr[i - 1] = m_piDESOrderArr[i];
        m_piDESOrderArr[iPos] = iDES;
    }
    else if (iCur > iPos)
    {
        for (int i = iCur - 1; i >= iPos; i--)
            m_piDESOrderArr[i + 1] = m_piDESOrderArr[i];
        m_piDESOrderArr[iPos] = iDES;
    }
    return;
}

/*
 * DESCR:   Compute total cross for DESReorder_Force()
 * PARA:    None
 * RETURN:  total cross
 * ACCESS:  private
```

```
*/
double CSub::TotalCross_Force()
{
    double dCross = 0;
    for (int i = 0; i < this->GetNumofDES(); i++)
    {
        for (int j = i + 2; j < this->GetNumofDES(); j++)
            dCross += cross(i, j);
    }
    return dCross;
}

/*
 * DESCR:   Decide to move DES_i left or right. (< 0 : move left; >0 move right)
 *          for DESReorder_Force()
 * PARA:    i: position in m_piDESOrderArr
 * RETURN:  returned force
 * ACCESS:  private
 */
double CSub::Force(int i)
{
    double dForce = 0;
    for (int j = 0; j < i - 1; j++)
        dForce += sqrt((double)m_piCrossMatrix[m_piDESOrderArr[i]]
                                          [m_piDESOrderArr[j]] * (j - i + 1));
    for (int j = i + 2; j < this->GetNumofDES(); j++)
        dForce += sqrt((double)m_piCrossMatrix[m_piDESOrderArr[i]]
                                          [m_piDESOrderArr[j]] * (j - i - 1));
    return dForce;
}

/*
```

```
* DESCR:   Initialize a DES order
* PARA:    None
* RETURN:  0
* ACCESS:  private
*/
int CSub::InitialDESOrder()
{
    int i = 0;
    int j = 0;
    int k = 0;
    int iNumofDES = this->GetNumofDES();

    //There is no DES at all
    if (iNumofDES <= 0)
        return 0;

    //Only one DES
    m_piDESOrderArr[0] = 0;
    if (iNumofDES <= 1)
        return 0;

    int iPos = 0;
    double dLeftCross = 0;
    double dRightCross = 0;
    double dNewCross = 0;
    double dOldCross = 0;
    vector<int> vecDESOrder;
    vector<int> vecShared;

    //two or more DES
    vecDESOrder.push_back(0);
    vecDESOrder.push_back(1);

    for (i = 2; i < iNumofDES; i++)
    {
```

```
vecShared.clear();
for (j = 0; j < i; j++)
    vecShared.push_back(m_piCrossMatrix[i][vecDESOrder[j]]);

iPos = i;
dOldCross = MAX_DOUBLE;
for (j = i; j >= 0; j--)
{
    dLeftCross = 0;
    dRightCross = 0;

    for (k = 0; k < j; k++)
    {
        dLeftCross += vecShared[k] * (j - k - 1);
    }
    for (k = j; k < i; k++)
    {
        dRightCross += vecShared[k] * (k - j);
    }
    dNewCross = dLeftCross + dRightCross;

    if (dNewCross == 0)
    {
        iPos = j;
        break;
    }
    else
    {
        if (dNewCross < dOldCross - 0.05)
        {
            dOldCross = dNewCross;
            iPos = j;
        }
    }
}
```

```
        if (iPos == 0)
            vecDESOrder.insert(vecDESOrder.begin(), i);
        else if (iPos == i)
            vecDESOrder.push_back(i);
        else
        {
            vector<int>::iterator itr = vecDESOrder.begin();
            itr += iPos;
            vecDESOrder.insert(itr, i);
        }
    }

    assert((int)vecDESOrder.size() == this->GetNumofDES());

    for (i = 0; i < (int)vecDESOrder.size(); i++)
    {
        m_piDESOrderArr[i] = vecDESOrder[i];
    }
    return 0;
}
```

Sub2.cpp

```
    bdd bddStates = bddStateSet;
    int *piStateSet = fdd_scanallvar(bddStates);

    int count = 0;

    while (piStateSet != NULL && count < 3)
    {
        bdd bddVisitedState = bddtrue;

```

```
        cout << "¡";
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            int iState = piStateSet[m_piDESPosArr[i] * 2];
            cout << m_pDESArr[m_piDESPosArr[i]]->GetDESName() + ":";
            cout << m_pDESArr[m_piDESPosArr[i]]->GetStateName(iState);
            if (i < this->GetNumofDES() -1)
            {
                cout << ", ";
            }
            bddVisitedState &= fdd_ithvar(m_piDESPosArr[i] * 2, iState);
        }
        cout << "¿ ";
        free(piStateSet);

        bddStates = bddStates - bddVisitedState;
        piStateSet = fdd_scanallvar(bddStates);

        count++;
    }

    if (count == 3)
    {
        cout << "...";
    }
}

bdd CSub::GetOneState(const bdd & bddStates)
{
    int *piStateSet = fdd_scanallvar(bddStates);
    bdd bddState = bddtrue;

    if (piStateSet != NULL)
    {
```

```
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            int iState = piStateSet[m_piDESPosArr[i] * 2];
            bddState &= fdd_ithvar(m_piDESPosArr[i] * 2, iState);
        }
        free(piStateSet);
        return bddState;
    }
    return bddfalse;
}

int CSub::CountStates(const bdd & bddStateSet)
{
    int count = 0;
    bdd bddStates = bddStateSet;
    int *piStateSet = fdd_scanallvar(bddStates);

    while (piStateSet != NULL)
    {
        count++;

        bdd bddVisitedState = bddtrue;

        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            int iState = piStateSet[m_piDESPosArr[i] * 2];
            bddVisitedState &= fdd_ithvar(m_piDESPosArr[i] * 2,
iState);
        }
        free(piStateSet);

        bddStates = bddStates - bddVisitedState;
        piStateSet = fdd_scanallvar(bddStates);
    }
    return count;
```

```
}

/*
 * DESCR:   Print all the state vectors using state names
 * PARA:    bddStateSet: BDD respresentation of the state set (input)
 *          viSetFlat: 0: Initial state  1: All states 2: Marking
States (input)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  protected
 */
int CSub::PrintStateSet(const bdd & bddStateSet, int viSetFlag)
{
    int *statevec = NULL;
    int iStateIndex = 0;

    STATES statesMap;

    try
    {
        string sLine;
        bdd bddTemp = bddfalse;
        bdd bddNormStateSet = bddtrue;
        string sInitState;
        bool bInitState = false;

        //restrict the prime variable to 0
        for (int i = 0; i < this->GetNumofDES(); i++)
            bddNormStateSet &= fdd_ithvar(i * 2 + 1, 0);
        bddNormStateSet &= bddStateSet;

        //save number of states
        if (viSetFlag != 0)
            cout << bdd_satcount(bddNormStateSet) << endl;

        //Initial state
```

```
        STATES::const_iterator csmi = statesMap.begin();
        if (csmi != statesMap.end())
            sInitState = csmi->first;
        else
            sInitState.clear();
        //print all the vectors
        statevec = fdd_scanallvar(bddNormStateSet);
        while ( statevec!= NULL)
        {
            sLine.clear();
            sLine = "¡";
            for (int i = 0; i < this->GetNumofDES(); i++)
            {
                sLine += m_pDESArr[m_piDESPosArr[i]]->GetStateName(
                                        statevec[m_piDESPosArr[i] * 2]) +
",";
            }
            sLine = sLine.substr(0, sLine.length() - 1);
            sLine += "¿";
            iStateIndex++;

            //state index for initial state should be 0
            if (viSetFlag == 0)
            {
                iStateIndex = 0;
                statesMap[sLine] = iStateIndex;
            }
            else
            {
                //for marking states, should show the corresponding
state index
                if (viSetFlag == 2)
                    cout << statesMap[sLine] << " #” ¡¡ sLine ¡¡ endl;
          else //all the states
          {
```

```
            if (bInitState)  //initial state alredy been printed
            {
                statesMap[sLine] = iStateIndex;
                cout ¡¡ iStateIndex ¡¡ " #" << sLine << endl;
                    }
                    else
                    {
                        if (sLine != sInitState)
                        {
                            statesMap[sLine] = iStateIndex;
                            cout << iStateIndex << " #" ¡¡ sLine ¡¡ endl;
                }
                else
                {
                    iStateIndex--;
                    bInitState = true;
                    cout ¡¡ "0" ¡¡ " #" << sLine << endl;
                        }
                    }
                }
            }

            //remove the outputed state
            bddTemp = bddtrue;
            for (int i = 0; i < this->GetNumofDES(); i++)
                bddTemp &= fdd_ithvar(i * 2, statevec[i * 2]);
            bddNormStateSet = bddNormStateSet - bddTemp;
            free(statevec);
            statevec = NULL;

            statevec = fdd_scanallvar(bddNormStateSet);
        }
    }
    catch(...)
    {
```

```
        delete[] statevec;
        statevec = NULL;
        return -1;
    }
    return 0;
}

/*
 * DESCR:   Print all events from the pPrj->m_InvAllEventsMap
 * PARA:    fout: file stream (input)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  protected
 */
int CSub::PrintEvents(ofstream & fout)
{
    char cSub = '\0';
    char cCon = '\0';
    string sLine;

    try
    {
        INVEVENTS::const_iterator ci =
pSub->GetInvAllEventsMap().begin();
        for (; ci != pSub->GetInvAllEventsMap().end(); ++ci)
        {
            if ((ci->first & 0x0FFF0000) >> 16 == 1)
            {
                cCon = ci->first % 2 == 0 ? 'N':'Y';
                sLine = ci->second + "\t\t";
                sLine += cCon;
                sLine += "\t\t";
                sLine += cSub;
                fout << sLine << endl;
            }
        }
```

```
    }
    catch (...)
    {
        return -1;
    }
    return 0;
}

/*
 * DESCR:   Print all the transitions one by one
 * PARA:    fout: file stream (input)
 *          bddController: not simplified bdd control predicate for
sEventName
 *          EventSub: 'H'/'R'/'A'/L'
 *          usiLocalIndex: local index (in this sub)
 *          bddReach: BDD respresentation of reachable states in
 *                    synthesized automata-based supervisor or
syn-product of
 *                    the verified system.
 *          sEventName: Event Name
 *          statesMap: state name and index map (index is for the
output file)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  protected
 */
int CSub::PrintTextTrans(ofstream & fout, bdd & bddController,
                         unsigned short usiLocalIndex,
                          const bdd & bddReach, string sEventName,
                          STATES & statesMap)
{
    int *statevec1 = NULL;
    int *statevec2 = NULL;
    try
    {
        string sExit;
```

```
        string sEnt;
        bdd bddTemp = bddfalse;
        bdd bddNext = bddfalse;

        //extract each state from bddController
        statevec1 = fdd_scanallvar(bddController);
        while ( statevec1!= NULL)
        {
            sExit.clear();
            sExit = "¡";
            for (int i = 0; i < this->GetNumofDES(); i++)
                sExit += m_pDESArr[m_piDESPosArr[i]]->GetStateName(
                                        statevec1[m_piDESPosArr[i] * 2]) +
",";
            sExit = sExit.substr(0, sExit.length() - 1);
            sExit += "¿";

            bddTemp = bddtrue;
            for (int i = 0; i < this->GetNumofDES(); i++)
                bddTemp &= fdd_ithvar(i * 2, statevec1[i * 2]);
            bddController = bddController - bddTemp;
            free(statevec1);
            statevec1 = NULL;
            statevec1 = fdd_scanallvar(bddController);

            //Get the target state
            if (usiLocalIndex % 2 == 0)
                bddNext =
                    bdd_replace(
                        bdd_relprod(
                           m_pbdd_UnConTrans[(usiLocalIndex - 2) / 2],
                           bddTemp,
                           m_pbdd_UnConVar[(usiLocalIndex - 2) / 2]),
                        m_pPair_UnConPrim[(usiLocalIndex - 2) / 2]) &
                    bddReach;
```

```
            else
                bddNext =
                    bdd_replace(
                        bdd_relprod(
                            m_pbdd_ConTrans[(usiLocalIndex - 1) / 2],
                            bddTemp,
                            m_pbdd_ConVar[(usiLocalIndex - 1) / 2]),
                        m_pPair_ConPrim[(usiLocalIndex - 1) / 2]) &
                    bddReach;

            statevec2 = fdd_scanallvar(bddNext);
            if (statevec2 == NULL)
                throw -1;
            sEnt = "¡";
            for (int i = 0; i < this->GetNumofDES(); i++)
                sEnt += m_pDESArr[m_piDESPosArr[i]]->GetStateName(
                                        statevec2[m_piDESPosArr[i] *
2]) + ",";
            sEnt = sEnt.substr(0, sEnt.length() - 1);
            sEnt += "¿";
            free(statevec2);
            statevec2 = NULL;

            //print the transition
            fout << statesMap[sExit] << " ¡" << sEventName << "¿ " <<
                statesMap[sEnt] << endl;
        }
    }
    catch(...)
    {
        free(statevec1);
        statevec1 = NULL;
        free(statevec2);
        statevec2 = NULL;
        return -1;
```

```
    }
    return 0;
}

/*
 * DESCR:   Compute triple-prime simplified BDD control predicate for an event
 * PARA:    fout: file stream (input)
 *          bddController: BDD control predicate for event usiIndex
 *          EventSub: 'H'/'R'/'A'/L'
 *          usiIndex: local index (in this sub)
 * RETURN:  triple-prime simplified BDD control predicate
 * ACCESS:  protected
 */
bdd CSub::SimplifyController(const bdd & bddController,
                             const unsigned short usiIndex)
{
    //event should be controllable
    assert(usiIndex % 2 == 1);

    bdd bddElig = bddfalse;
    bdd bddSpecElig = bddfalse;

    //⋀dHs'
    bddElig = bdd_exist(m_pbdd_ConTrans[(usiIndex - 1) / 2],
                        m_pbdd_ConVarPrim[(usiIndex - 1) / 2]);
    //spec part
    bddSpecElig = bdd_exist(bddElig,
                        m_pbdd_ConPhysicVar[(usiIndex - 1) / 2]);

    return bddSpecElig & bdd_simplify(bddController, m_bddSuper & bddElig);
}


```

A.2.5 LowSub Class

LowSub.h

```
    virtual ~CLowSub();
    virtual int PrintSub(ofstream& fout);
    virtual int PrintSubAll(ofstream & fout);
    virtual string SearchEventName(unsigned short usiLocalIndex);
    virtual int LoadSub();
    virtual int VeriSub(const HISC_TRACETYPE showtrace,
                        HISC_SUPERINFO & superinfo);
private:
    virtual int MakeBdd();
    virtual int InitBddFields();
    virtual int ClearBddFields();
    int CheckIntf();
    int SynPartSuper(const HISC_COMPUTEMETHOD computemethod,
                                        bdd & bddReach, bdd & bddBad);
    int GenConBad(bdd &bddConBad);
    int VeriConBad(bdd &bddConBad, const bdd &bddReach, string & vsErr);
    int GenBalemiBad(bdd &bddBalemiBad);
    int VeriBalemiBad(bdd &bddBalemiBad, const bdd &bddReach, string & vsErr);
    int VeriALF(bdd &bddALFBad, bdd bddReach, string & vsErr);
    int VeriProperTimedBehavior(bdd &bddPTBBad, bdd bddReach, string & vsErr);
    int CheckSDControllability(bdd & bddSDBad, const bdd & bddreach, string & vsErr);
    int AnalyseSampledState(bdd & bddSSBad, const bdd & bddreach, const
```

```
bdd & bddSS,
        list< list<bdd> > & list_NerFail, bdd & bddSF, stack<bdd> & stack_bddSP, string & vsErr);

    int CheckTimedControllability(const EVENTSET & eventsDis, const EVENTSET & eventsPoss);
    int CheckTimedControllability(bdd & bddTCBad, const bdd & bddreach);

    bool RecheckNerodeCells(bdd & bddNCBad, const bdd & bddreach, list< list<bdd> > & list_NerFail);
    bool RecheckNerodeCell(bdd & bddNCBad, const bdd & bddreach, const list<bdd> & Zeqv, list< pair<bdd, bdd> > & listVisited);

    int DetermineNextState(bdd & bddLBBad, const EVENTSET & eventsPoss, const bdd & bddZ, const bdd & bddreach,
        const int & intB, int & intNextFreeLabel, map<int, bdd> & B_map, stack<int> & B_p,
        bdd & bddSF, stack<bdd> & stack_bddSP,
        map<int, EVENTSET> & B_occu, map<int, bdd> & B_conc, string & vsErr);

    void CheckNerodeCells(map<int, bdd> & B_conc, map<int, EVENTSET> & B_occu,
        list< list<bdd> > & list_NerFail);

    int CheckSDiv(bdd & bddSDivBad, const bdd & bddReach);

    EVENTSET GetTransitionEvents(const bdd & bddleave, const bdd & bddenter);

    int GenP4Bad(bdd &bddP4Bad);
    int VeriP4Bad(bdd &bddP4Bad, const bdd &bddReach, string &vsErr);
    int supcp(bdd & bddP);
    bdd cr(const bdd & bddPStart, const bdd & bddP, int & iErr);
```

```
    bdd r(const bdd &bddP, int &iErr);
    bdd p5(const bdd& bddP, int &iErr);
    bdd p6(const bdd& bddP, int &iErr);
    void BadStateInfo(const bdd& bddBad, const int viErrCode,
            const HISC_TRACETYPE showtrace, const string &vsExtraInfo = "");
};

#endif //_LSUB_H_


```

LowSub.cpp

```
 * PARA:    vsLowFile: subsystem file name with path (.sub)(input)
 *          viSubIndex: subsystem index (high: 0, low: 1,2,...)(input)
 * RETURN:  None
 * ACCESS:  public
 */
CLowSub::CLowSub(const string & vsLowFile):
CSub(vsLowFile)
{
    InitBddFields();
}

/**
 * DESCR:   Destructor
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
CLowSub::~CLowSub()
{
    // do nothing for now.
}
```

```

/*
 * DESCR:   Initialize BDD related data members (only those in
LowSub.h)
 * PARA:    None
 * RETURN:  0
 * ACCESS:  private
 */
int CLowSub::InitBddFields()
{
    return 0;
}

/*
 * DESCR:   Release memory for BDD related data members(only those in
Lowsub.h)
 * PARA:    None
 * RETURN:  0
 * ACCESS:  private
 */
int CLowSub::ClearBddFields()
{
    return 0;
}

/**
 * DESCR:   Load a low-level
 * PARA:    None
 * RETURN:  0 sucess <0 fail;
 * ACCESS:  public
 */
int CLowSub::LoadSub()
{
    ifstream fin;
    int iRet = 0;
```

```
    CDES *pDES = NULL;

    try
    {
        m_sSubFile = str_trim(m_sSubFile);

        if (m_sSubFile.length() <= 4)
        {
            pSub->SetErr("Invalid file name: " + m_sSubFile, HISC_BAD_LOW_FILE);
            throw -1;
        }

        if (m_sSubFile.substr(m_sSubFile.length() - 4) != ".sub")
        {
            pSub->SetErr("Invalid file name: " + m_sSubFile, HISC_BAD_LOW_FILE);
            throw -1;
        }

        fin.open(m_sSubFile.data(), ifstream::in);

        if (!fin) //unable to find low sub file
        {
            pSub->SetErr("Unable to open file: " + m_sSubFile,
                          HISC_BAD_LOW_FILE);
            throw -1;
        }

        m_sSubName = GetNameFromFile(m_sSubFile);

        char scBuf[MAX_LINE_LENGTH];
        string sLine;
        int iField = -1; //0: SYSTEM 1:PLANT 2:SPEC
        char *scFieldArr[] = {"SYSTEM", "PLANT", "SPEC"};
```

```
        string sDESFile;

        int iTmp = 0;

        int iNumofPlants = 0;
        int iNumofSpecs = 0;

        while (fin.getline(scBuf,  MAX_LINE_LENGTH))
        {
            sLine = str_nocomment(scBuf);
            sLine = str_trim(sLine);

            if (sLine.empty())
                continue;

            if (sLine[0] == '[' && sLine[sLine.length() - 1] == ']')
            {
                sLine = sLine.substr(1, sLine.length() - 1);
                sLine = sLine.substr(0, sLine.length() - 1);
                sLine = str_upper(str_trim(sLine));

                iField++;

                if (iField < 3)
                {
                    if (sLine != scFieldArr[iField])
                    {
                        pSub->SetErr(m_sSubName +
                                      ": Field name or order is wrong!",
                                        HISC_BAD_LOW_FORMAT);
                        throw -1;
                    }
                    if (iField == 1)
                    {
                        //Check number of Plants and apply for memory
```

space

```
                if (m_iNumofPlants + m_iNumofSpecs <= 0)
                {
                    pSub->SetErr(m_sSubName +
                            ": Must have at least one DES.",
                            HISC_BAD_LOW_FORMAT);
                    throw -1;
                }
                if (m_iNumofPlants < 0 || m_iNumofSpecs < 0)
                {
                    pSub->SetErr(m_sSubName +
                 ": Must specify the number of plant DES and spec DES.",
                            HISC_BAD_LOW_FORMAT);
                    throw -1;
                }
                m_pDESArr = new CDES *[this->GetNumofDES()];

                if(m_pDESArr == NULL) throw -1;

                for (int i = 0; i < this->GetNumofDES(); i++)
                    m_pDESArr[i] = NULL;

                //Initialize m_piDESOrderArr
                m_piDESOrderArr = new int[this->GetNumofDES()];
                for (int i = 0; i < this->GetNumofDES(); i++)
                    m_piDESOrderArr[i] = i;
                //Initialize m_piDESPosArr
                m_piDESPosArr = new int[this->GetNumofDES()];
                for (int i = 0; i < this->GetNumofDES(); i++)
                    m_piDESPosArr[i] = i;

            }
        }
        else
```

```
    {
        pSub->SetErr(m_sSubName + ": Too many fields!",
                        HISC_BAD_LOW_FORMAT);
        throw -1;
    }
}
else
{
    switch (iField)
    {
    case 0:  //[SYSTEM]
        if (!IsInteger(sLine))
        {
            pSub->SetErr(m_sSubName + ": Number of DES is absent!",
                                    HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        iTmp = atoi(sLine.data());
        if (iTmp < 1)
        {
            pSub->SetErr(m_sSubName +
                              ": Number of DES is less than 1!",
                            HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        if (m_iNumofPlants < 0)
            m_iNumofPlants = iTmp;
        else if (m_iNumofSpecs < 0)
            m_iNumofSpecs = iTmp;
        else
        {
            pSub->SetErr(m_sSubName +
                    ": Too many lines in SYSTEM field",
                    HISC_BAD_LOW_FORMAT);
```

```
            throw -1;
        }
        break;
    case 1: //[PLANT]
        sDESFile = GetDESFileFromSubFile(m_sSubFile, sLine);
        pDES = new CDES(this, sDESFile, PLANT_DES);
        if (pDES == NULL || pDES->LoadDES() < 0)
            throw -1;   //here LoadDES() will generate the err msg.
        else
        {
            iNumofPlants++;
            if (iNumofPlants > m_iNumofPlants)
            {
                pSub->SetErr(m_sSubName + ": Too many Plant DESs",
                        HISC_BAD_LOW_FORMAT);
                throw -1;
            }
            m_pDESArr[iNumofPlants - 1] = pDES;

            for (EVENTS::const_iterator ci = pDES->m_DESEventsMap.begin(); ci != pDES->m_DESEventsMap.end(); ++ci)
            {
                m_SubPlantEvents.insert(ci->second);
            }

            pDES = NULL;
        }
        break;
    case 2: //[SPEC]
        sDESFile = GetDESFileFromSubFile(m_sSubFile, sLine);
        pDES = new CDES(this, sDESFile, SPEC_DES);
```

```
                    if (pDES == NULL || pDES->LoadDES() < 0)
                        throw -1;   //here LoadDES() will generate the err msg.
                    else
                    {
                        iNumofSpecs++;
                        if (iNumofSpecs > m_iNumofSpecs)
                        {
                            pSub->SetErr(m_sSubName + ": Too many spec DESs",
                                         HISC_BAD_LOW_FORMAT);
                            throw -1;
                        }
                        m_pDESArr[m_iNumofPlants + iNumofSpecs - 1] = pDES;

                        for (EVENTS::const_iterator ci = pDES->m_DESEventsMap.begin(); ci != pDES->m_DESEventsMap.end(); ++ci)
                        {
                            m_SubSupervisorEvents.insert(ci->second);
                        }

                        pDES = NULL;
                    }
                    break;
                default:
                    pSub->SetErr(m_sSubName + ": Unknown error.",
                                 HISC_BAD_LOW_FORMAT);
                    throw -1;
                    break;
            }
        }
    } //while
    if (iNumofPlants < m_iNumofPlants)
    {
```

```
            pSub->SetErr(m_sSubName + ": Too few plant DESs",
                            HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        if (iNumofSpecs < m_iNumofSpecs)
        {
            pSub->SetErr(m_sSubName + ": Too few spec DESs",
                        HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        fin.close();

        this->DESReorder();
    }
    catch (int iError)
    {
        if (pDES != NULL)
        {
            delete pDES;
            pDES = NULL;
        }
        if (fin.is_open())
            fin.close();
        iRet = iError;
    }
    return iRet;
}

/*
 * DESCR:   Initialize BDD data memebers
 * PARA:    None
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::MakeBdd()
```

```
{
    const char * DEBUG = "CLowSub::MakeBdd():";

    try
    {
        //Initialize the bdd node table and cache size.
        long long lNumofStates = 1;

        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            lNumofStates *= m_pDESArr[i]->GetNumofStates();
            if (lNumofStates >= MAX_INT)
                break;
        }
        if (lNumofStates <= 10000)
            bdd_init(1000, 100);
        else if (lNumofStates <= 1000000)
            bdd_init(10000, 1000);
        else if (lNumofStates <= 10000000)
            bdd_init(100000, 10000);
        else
        {
            bdd_init(2000000, 1000000);
            bdd_setmaxincrease(1000000);
        }

        giNumofBddNodes = 0;
        bdd_gbc_hook(my_bdd_gbchandler);

        //define domain variables
        int *piDomainArr = new int[2];
        for (int i = 0; i < 2 * this->GetNumofDES(); i += 2)
        {
            VERBOSE(1) { PRINT_DEBUG << "Name of DES " << i << ": " << m_pDESArr[i/2]->GetDESName() << endl; }
```

```

            piDomainArr[0] = m_pDESArr[i/2]->GetNumofStates();
            piDomainArr[1] = piDomainArr[0];

            VERBOSE(1) { PRINT_DEBUG << "piDomainArr[0] (# of states): " << piDomainArr[0] << endl; }
            VERBOSE(1) { PRINT_DEBUG << "piDomainArr[1] (# of states): " << piDomainArr[1] << endl; }

            fdd_extdomain(piDomainArr, 2);
        }
        delete[] piDomainArr;
        piDomainArr = NULL;

        //compute the number of bdd variables (only for normal variables)
        m_iNumofBddNormVar = 0;
        for (int i = 0; i < 2 * (this->GetNumofDES()); i = i + 2)
        {
            m_iNumofBddNormVar += fdd_varnum(i);
        }

        //compute initial state predicate
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            m_bddInit &= fdd_ithvar(i * 2, m_pDESArr[i]->GetInitState());
        }

        //set the first level block
        int iNumofBddVar = 0;
        int iVarNum = 0;
        bdd bddBlock = bddtrue;
        for (int i = 0; i < 2 * (this->GetNumofDES()); i += 2)
        {
```

```
        iVarNum = fdd_varnum(i);
        bddBlock = bddtrue;

        for (int j = 0; j < 2 * iVarNum; j++)
        {
            bddBlock &= bdd_ithvar(iNumofBddVar + j);
        }
        bdd_addvarblock(bddBlock, BDD_REORDER_FREE);
        iNumofBddVar += 2 * iVarNum;
    }

    //compute marking states predicate
    bdd bddTmp = bddfalse;
    for (int i = 0; i < this->GetNumofDES(); i++)
    {
        bddTmp = bddfalse;
        MARKINGLIST::const_iterator ci =
             (m_pDESArr[i]->GetMarkingList()).begin();

        for (int j = 0; j < m_pDESArr[i]->GetNumofMarkingStates();
j++)

        {
            bddTmp |= fdd_ithvar(i * 2, *ci);
            ++ci;
        }
        m_bddMarking &= bddTmp;
    }

    //Compute transitions predicate
        if (m_usiMaxCon != 0xFFFF)
        {
            m_pbdd_ConTrans = new bdd[(m_usiMaxCon + 1) / 2];
            m_pbdd_ConVar = new bdd[(m_usiMaxCon + 1) / 2];
            m_pbdd_ConPlantTrans =  new bdd[(m_usiMaxCon + 1) / 2];
            m_pbdd_ConSupTrans =    new bdd[(m_usiMaxCon + 1) / 2];
```

```

                    m_pbdd_ConVarPrim =
                                    new bdd[(m_usiMaxCon + 1) / 2];
                    m_pbdd_ConPhysicVar =
                                    new bdd[(m_usiMaxCon + 1) / 2];
                    m_pbdd_ConSupVar =
                                    new bdd[(m_usiMaxCon + 1) / 2];
                    m_pbdd_ConPhysicVarPrim =
                                    new bdd[(m_usiMaxCon + 1) / 2];
                    m_pbdd_ConSupVarPrim =
                                    new bdd[(m_usiMaxCon + 1) / 2];

                    m_pPair_Con = new bddPair *[(m_usiMaxCon + 1) / 2];
                    for (int iPair = 0; iPair < (m_usiMaxCon + 1) / 2;
iPair++)
                    {
                        m_pPair_Con[iPair] = NULL;
                    }

                    m_pPair_ConPrim = new bddPair *[(m_usiMaxCon + 1) / 2];
                    for (int iPair = 0; iPair < (m_usiMaxCon + 1) / 2;
iPair++)
                    {
                        m_pPair_ConPrim[iPair] = NULL;
                    }
                }
                if (m_usiMaxUnCon != 0)
                {
                    m_pbdd_UnConTrans = new bdd[m_usiMaxUnCon/2];

                    m_pbdd_UnConVar = new bdd[m_usiMaxUnCon/2];
                    m_pbdd_UnConPlantTrans =
                                            new bdd[m_usiMaxUnCon/2];
                    m_pbdd_UnConSupTrans =
                                            new bdd[m_usiMaxUnCon/2];
```

```

                m_pbdd_UnConVarPrim = new bdd[m_usiMaxUnCon/2];
                m_pbdd_UnConPlantVar = new bdd[m_usiMaxUnCon/2];
                m_pbdd_UnConSupVar = new bdd[m_usiMaxUnCon/2];

                m_pbdd_UnConPlantVarPrim =
                                        new bdd[m_usiMaxUnCon/2];
                m_pbdd_UnConSupVarPrim =
                                        new bdd[m_usiMaxUnCon/2];

                m_pPair_UnCon = new bddPair *[m_usiMaxUnCon/2];
                for (int iPair = 0; iPair < m_usiMaxUnCon/2; iPair++)
                {
                    m_pPair_UnCon[iPair] = NULL;
                }
                m_pPair_UnConPrim = new bddPair *[m_usiMaxUnCon/2];
                for (int iPair = 0; iPair < m_usiMaxUnCon/2; iPair++)
                {
                    m_pPair_UnConPrim[iPair] = NULL;
                }
            }

        map<int, bdd> bddTmpTransMap;  //<event_index, transitions>
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            //before compute transition predicate for each DES, clear
it.
            bddTmpTransMap.clear();
            for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
            {
                bddTmpTransMap[(m_pDESArr[i]->GetEventsArr())[j]] =
bddfalse;
            }

            //compute transition predicate for each DES
```

```
        for (int j = 0; j < m_pDESArr[i]->GetNumofStates(); j++)
        {
            TRANS::const_iterator ci =
                                  (*(m_pDESArr[i]->GetTrans() + j)).begin();
            for (; ci != (*(m_pDESArr[i]->GetTrans() + j)).end(); ++ci)
            {
                bddTmpTransMap[ci->first] |= fdd_ithvar(i * 2, j) &
                    fdd_ithvar(i * 2 + 1, ci->second);
            }
        }

        //combine the current DES transition predicate to
        //subsystem transition predicate
        map<int, bdd>::const_iterator ciTmp = bddTmpTransMap.begin();
        for (; ciTmp != bddTmpTransMap.end(); ++ciTmp)
        {
            if (ciTmp->first % 2 == 0)  //uncontrollable, start from 2
            {
                int iIndex = (ciTmp->first & 0x0000FFFF) / 2 - 1;

                if (m_pbdd_UnConVar[iIndex] == bddfalse)
                {
                    m_pbdd_UnConTrans[iIndex] = bddtrue;
                    m_pbdd_UnConVar[iIndex] = bddtrue;
                    m_pbdd_UnConVarPrim[iIndex] = bddtrue;
                }

                m_pbdd_UnConTrans[iIndex] &= ciTmp->second;
                m_pbdd_UnConVar[iIndex] &= fdd_ithset(i * 2);
                m_pbdd_UnConVarPrim[iIndex] &= fdd_ithset(i * 2 + 1);
```

```

                //compute uncontrollable plant vars and varprimes
                if (m_pDESArr[i]->GetDESType() == PLANT_DES)
                {
                    if (m_pbdd_UnConPlantVar[iIndex] == bddfalse)
                    {
                        m_pbdd_UnConPlantTrans[iIndex] = bddtrue;
                        m_pbdd_UnConPlantVar[iIndex] = bddtrue;
                        m_pbdd_UnConPlantVarPrim[iIndex] = bddtrue;
                    }

                    m_pbdd_UnConPlantTrans[iIndex] &= ciTmp->second;
                    m_pbdd_UnConPlantVar[iIndex] &= fdd_ithset(i * 2);
                    m_pbdd_UnConPlantVarPrim[iIndex] &= fdd_ithset(i * 2 + 1);
                }
                else if (m_pDESArr[i]->GetDESType() == SPEC_DES)
                {
                    if (m_pbdd_UnConSupVar[iIndex] == bddfalse)
                    {
                        m_pbdd_UnConSupTrans[iIndex] = bddtrue;
                        m_pbdd_UnConSupVar[iIndex] = bddtrue;
                        m_pbdd_UnConSupVarPrim[iIndex] = bddtrue;
                    }

                    m_pbdd_UnConSupTrans[iIndex] &= ciTmp->second;
                    m_pbdd_UnConSupVar[iIndex] &= fdd_ithset(i * 2);
                    m_pbdd_UnConSupVarPrim[iIndex] &= fdd_ithset(i * 2 + 1);
                }
            }
            else  //controllable
```

```
{
    int iIndex = ((ciTmp->first & 0x0000FFFF) - 1)/ 2;

    if (m_pbdd_ConVar[iIndex] == bddfalse)
    {
        m_pbdd_ConTrans[iIndex] = bddtrue;
        m_pbdd_ConVar[iIndex] = bddtrue;
        m_pbdd_ConVarPrim[iIndex] = bddtrue;
    }
    m_pbdd_ConTrans[iIndex] &= ciTmp->second;
    m_pbdd_ConVar[iIndex] &= fdd_ithset(i * 2);
    m_pbdd_ConVarPrim[iIndex] &= fdd_ithset(i * 2 + 1);

    //compute controllable physical plant vars and varprimes
    if (m_pDESArr[i]->GetDESType() == PLANT_DES)
    {
        if (m_pbdd_ConPhysicVar[iIndex] == bddfalse)
        {
            m_pbdd_ConPlantTrans[iIndex] = bddtrue;
            m_pbdd_ConPhysicVar[iIndex] = bddtrue;
            m_pbdd_ConPhysicVarPrim[iIndex]= bddtrue;
        }

        m_pbdd_ConPlantTrans[iIndex] &= ciTmp->second;
        m_pbdd_ConPhysicVar[iIndex] &= fdd_ithset(i * 2);
        m_pbdd_ConPhysicVarPrim[iIndex] &= fdd_ithset(i * 2 + 1);
    }
    else if (m_pDESArr[i]->GetDESType() == SPEC_DES)
    {
        if (m_pbdd_ConSupVar[iIndex] == bddfalse)
        {
            m_pbdd_ConSupTrans[iIndex] = bddtrue;
```

```
                        m_pbdd_ConSupVar[iIndex] = bddtrue;
                        m_pbdd_ConSupVarPrim[iIndex]= bddtrue;
                    }

                    m_pbdd_ConSupTrans[iIndex] &= ciTmp->second;
                    m_pbdd_ConSupVar[iIndex] &= fdd_ithset(i * 2);
                    m_pbdd_ConSupVarPrim[iIndex] &= fdd_ithset(i * 2 + 1);
                }
            }
        }
    }

    // Add self loops of any event to plant (sup) trans predicate if the event
    // does not exist in the plants (sups), but exists in the sups (plants).
    int sig = 0;
    for (int iIndex = 0; iIndex < (m_usiMaxCon + 1) / 2; iIndex++)
    {
        sig = (iIndex * 2) + 1;
        if ((m_SubSupervisorEvents.find(sig) == m_SubSupervisorEvents.end())
            && (m_SubPlantEvents.find(sig) != m_SubPlantEvents.end()))
        {
            m_pbdd_ConSupTrans[iIndex] = bddtrue;
        }
        else if ((m_SubSupervisorEvents.find(sig) != m_SubSupervisorEvents.end())
            && (m_SubPlantEvents.find(sig) == m_SubPlantEvents.end()))
        {
            m_pbdd_ConPlantTrans[iIndex] = bddtrue;
        }
```

```
        }

        for (int iIndex = 0; iIndex < (m_usiMaxUnCon / 2); iIndex++)
        {
            sig = (iIndex + 1) * 2;
            if ((m_SubSupervisorEvents.find(sig) ==
m_SubSupervisorEvents.end())
                && (m_SubPlantEvents.find(sig) !=
m_SubPlantEvents.end()))
            {
                m_pbdd_UnConSupTrans[iIndex] = bddtrue;
            }
            else if ((m_SubSupervisorEvents.find(sig) !=
m_SubSupervisorEvents.end())
                && (m_SubPlantEvents.find(sig) ==
m_SubPlantEvents.end()))
            {
                m_pbdd_UnConPlantTrans[iIndex] = bddtrue;
            }
        }

        //compute m_pPair_UnCon, m_pPair_Con
        for (int j = 0; j < m_usiMaxUnCon; j += 2)
        {
            m_pPair_UnCon[j/2] = bdd_newpair();
            SetBddPairs(m_pPair_UnCon[j/2], m_pbdd_UnConVar[j/2],
                                m_pbdd_UnConVarPrim[j/2]);
            m_pPair_UnConPrim[j/2] = bdd_newpair();
            SetBddPairs(m_pPair_UnConPrim[j/2],
                                    m_pbdd_UnConVarPrim[j/2],
                                    m_pbdd_UnConVar[j/2]);
        }
        for (int j = 1; j < (unsigned short)(m_usiMaxCon + 1); j += 2)
        {
            m_pPair_Con[(j - 1) / 2] = bdd_newpair();
```

```
            SetBddPairs(m_pPair_Con[(j - 1) / 2],
                                  m_pbdd_ConVar[(j - 1) / 2],
                                  m_pbdd_ConVarPrim[(j - 1) / 2]);
            m_pPair_ConPrim[(j - 1) / 2] = bdd_newpair();
            SetBddPairs(m_pPair_ConPrim[(j - 1) / 2],
                                  m_pbdd_ConVarPrim[(j - 1) / 2],
                                  m_pbdd_ConVar[(j - 1) / 2]);
        }
    }
    catch(...)
    {
        string sErr;
        sErr = "Error happens when initializing low level ";
        sErr += " BDD!";
        pSub->SetErr(sErr, HISC_SYSTEM_INITBDD);
        return -1;
    }
    return 0;
}
```

LowSub1.cpp

```
 * DESCR:   Save DES list of low-levels in memory to a file (for
checking)
 * PARA:    fout: output file stream
 * RETURN:  0: sucess -1: fail
 * ACCESS:  public
 */
int CLowSub::PrintSub(ofstream& fout)
{
    try
    {
        fout << "#Sub system: " <<  m_sSubName << endl;
```

```
    fout << endl;

    fout << "[SYSTEM]" << endl;
    fout << m_iNumofPlants << endl;
    fout << m_iNumofSpecs << endl;
    fout << endl;

    fout << "[PLANT]" << endl;
    for (int i = 1; i < m_iNumofPlants; i++)
    {
        for (int j = 0; j < this->GetNumofDES(); j++)
        {
            if (m_piDESOrderArr[j] == i)
            {
                fout << m_pDESArr[j]->GetDESName() << endl;
                break;
            }
        }
    }

    fout << "[SPEC]" << endl;
    for (int i = m_iNumofPlants;
            i < this->GetNumofDES(); i++)
    {
        for (int j = 0; j < this->GetNumofDES(); j++)
        {
            if (m_piDESOrderArr[j] == i)
            {
                fout << m_pDESArr[j]->GetDESName() << endl;
                break;
            }
        }
    }

    fout <<
```

```
"#################################################"
<< endl;
    }
    catch(...)
    {
        return -1;
    }
    return 0;
}

/**
 * DESCR:   Save all the DES in low-levels to a text file for checking
 * PARA:    fout: output file stream
 * RETURN:  0: sucess -1: fail
 * ACCESS:  public
 */
int CLowSub::PrintSubAll(ofstream & fout)
{
    try
    {
        if (PrintSub(fout) < 0)
            throw -1;

        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            if (m_pDESArr[i]->PrintDES(fout) < 0)
                throw -1;
        }
    }
    catch(...)
    {
        return -1;
    }
    return 0;
}
```

```

/*
 * DESCR: Generate Bad state info during verfication
 *        Note: showtrace is not implemented, currently it is used for showing
 *              a blocking is a deadlock or livelock (very slow).
 * PARA:    bddBad: BDD for the set of bad states
 *          viErrCode: error code (see errmsg.h)
 *          showtrace: show a trace from the initial state to a bad state or not
 *                     (not implemented)
 *          vsExtraInfo: Extra errmsg.
 * RETURN:  None
 * ACCESS:  private
 */
void CLowSub::BadStateInfo(const bdd& bddBad, const int viErrCode,
                          const HISC_TRACETYPE showtrace, const string &vsExtraInfo)
{
    const char * DEBUG = "CLowSub::BadStateInfo():";
    if (bddfalse == bddBad)
    {
        VERBOSE(1) { PRINT_DEBUG << "bddBad = bddfalse" << endl; }
        return;
    }

    bdd bddBadTemp = bddBad;
    string sErr = GetSubName();

    if (viErrCode == HISC_VERI_LOW_UNCON)
        sErr += ": Untimed controllable checking failed at following state(s):";
    else if (viErrCode == HISC_VERI_LOW_CON)
        sErr += ": Proper timed behavior checking failed at following state(s):";
    else if (viErrCode == HISC_VERI_LOW_BLOCKING)
        sErr += ": Blocking state:";
```

```
    else if (viErrCode == HISC_VERI_LOW_P4FAILED)
        sErr += ": Interface consistent conditions Point 4 checking failed state:";
    else if (viErrCode == HISC_VERI_LOW_P5FAILED)
        sErr += ": Interface consistent conditions Point 5 checking failed state:";
    else if (viErrCode == HISC_VERI_LOW_P6FAILED)
        sErr += ": Interface consistent conditions Point 6 checking failed state:";
    else if (viErrCode == HISC_VERI_LOW_ALF)
        sErr += ": ALF checking failed state:";
    else if (viErrCode == HISC_VERI_LOW_PTB)
        sErr += ": Not proper timed behavior at state:";
    else if (viErrCode == HISC_VERI_LOW_SD_II)
        sErr += ": Failed SD Controllability condition II at state:";
    else if (viErrCode == HISC_VERI_LOW_SD_III_1)
        sErr += ": Failed SD Controllability condition III.1 at state:";
    else if (viErrCode == HISC_VERI_LOW_SD_III_2)
        sErr += ": Failed SD Controllability condition III.2 at state:";
    else if (viErrCode == HISC_VERI_LOW_SD_IV)
        sErr += ": Failed SD Controllability condition IV at state:";
    else if (viErrCode == HISC_VERI_LOW_ZERO_LB)
        sErr += ": There is some event has a lower bound less than 1 tick:";

    sErr += "\n";

    int count = 0;
    while (bddfalse != bddBadTemp && count < 10)
    {
        bdd bddstate = GetOneState(bddBadTemp);
        bddBadTemp -= bddstate;

        int *piBad = fdd_scanallvar(bddstate);

        if (NULL == piBad) break;

        //for blocking state, try to find the deadlock state
        //if there is no deadlock state, only show one of the live lock
```

```
states
        if (showtrace == HISC_SHOW_TRACE)
        {
            if (viErrCode == HISC_VERI_LOW_BLOCKING)
            {
                bdd bddBlock = bddBad;
                bdd bddNext = bddtrue;
                bdd bddTemp = bddtrue;
                do
                {
                    bddTemp = bddtrue;
                    for (int i = 0; i < this->GetNumofDES(); i++)
                        bddTemp &= fdd_ithvar(i * 2, piBad[i * 2]);

                    bddNext = bddfalse;
                    for (unsigned short usi = 2;
                            usi <= m_usiMaxUnCon && bddNext ==
bddfalse;
                            usi += 2)
                    {
                        bddNext |=
                            bdd_replace(
                                bdd_relprod(
                                    m_pbdd_UnConTrans[(usi - 2) / 2],
                                    bddTemp,
                                    m_pbdd_UnConVar[(usi - 2) / 2]),
                                    m_pPair_UnConPrim[(usi - 2) / 2]) &
                            bddBad;
                    }
                    for (unsigned short usi = 1;
                        usi < (unsigned short) (m_usiMaxCon + 1) &&
                        bddNext == bddfalse; usi += 2)
                    {
                        bddNext |=
                            bdd_replace(
```

```
                            bdd_relprod(
                                m_pbdd_ConTrans[(usi - 1) / 2],
                                bddTemp,
                                m_pbdd_ConVar[(usi - 1) / 2]),
                                m_pPair_ConPrim[(usi - 1) / 2]) &
                        bddBad;
                }

                if (bddNext == bddfalse)  //this is a deadlock state
                {
                    sErr += "[DeadLock]";
                    break;
                }
                else //not a deadlock state
                {
                    bddBlock = bddBlock - bddTemp;
                    free(piBad);
                    piBad = NULL;
                    piBad = fdd_scanallvar(bddBlock);
                }

                count++;
            } while (piBad != NULL);

            if (piBad == NULL) //live lock
            {
                sErr += "[LiveLock]";
                piBad = fdd_scanallvar(bddBad);
            }
        }
    }

    sErr += "\t¡";
```

```
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            sErr += m_pDESArr[m_piDESPosArr[i]]->GetDESName() + ":" +
                    m_pDESArr[m_piDESPosArr[i]]->GetStateName(
                                    piBad[m_piDESPosArr[i] *
2]);
            if (i < this->GetNumofDES() -1)
                sErr += ", ";
        }

        sErr += "¿\n";

        free(piBad);
        piBad = NULL;

        count++;
    }

    if (bddfalse != bddBadTemp)
    {
        sErr += "\t...";
    }

    sErr += "\n" + vsExtraInfo;

    pSub->SetErr(sErr, viErrCode);

    return;
}

/**
 * DESCR:   Search event name from this low-level local event index.
 * PARA:    k: R_EVENT/A_EVENT/H_EVENT/L_EVENT
 *          usiLocalIndex: this low-level local event index.
 * RETURN:  event name
```

```
 * ACCESS:  public
 */
string CLowSub::SearchEventName(unsigned short usiLocalIndex)
{
    int iEventIndex = 0;
    iEventIndex = pSub->GenEventIndex(usiLocalIndex);
    return (pSub->GetInvAllEventsMap())[iEventIndex];
}


```

LowSub3.cpp

```
int CLowSub::VeriSub(const HISC_TRACETYPE showtrace, HISC_SUPERINFO & superinfo)
{
  int iRet = 0;
  int iErr = 0;
  //Initialize the BDD data memebers
  CSub::InitBddFields();
  InitBddFields();
  bdd bddReach = bddfalse;
  string sErr;

  #ifdef DEBUG_TIME
  timeval tv1, tv2;
  #endif

  try
  {
    //Make transition bdds
    if (MakeBdd() < 0)
      throw -1;

    bdd bddConBad = bddfalse;
```

```
        bdd bddBalemiBad = bddfalse;
    bdd bddCoreach = bddfalse;
    bdd bddNBBad = bddfalse;
    bdd bddALFBad = bddfalse;
    bdd bddPTBBad = bddfalse;
    bdd bddSDBad = bddfalse;

    //compute bddReach
    #ifdef DEBUG_TIME
    cout << endl << "Computing reachable subpredicate..." << endl;
    gettimeofday(&tv1, NULL);
    #endif

    bddReach = r(bddtrue, iErr);
    if (iErr < 0)
    {
      throw -1;
    }

    #ifdef DEBUG_TIME
    gettimeofday(&tv2, NULL);
    cout << "R: " << (tv2.tv_sec - tv1.tv_sec) << "seconds." << endl;
    cout << "bddReach states:"
      << bdd_satcount(bddReach)/pow((double)2, double(m_iNumofBddNormVar))
      << endl;
    cout << "bddReach Nodes:" << bdd_nodecount(bddReach) << endl << endl;
    #endif

    m_bddMarking &= bddReach;


    #ifdef DEBUG_TIME
    cout << "Verifying controllablity..." << endl;
```

```
    gettimeofday(&tv1, NULL);
    #endif

    bddConBad = bddfalse;
    if (VeriConBad(bddConBad, bddReach, sErr) < 0)
      throw -1;

    #ifdef DEBUG_TIME
    gettimeofday(&tv2, NULL);
    cout << "VERI_CON: " << (tv2.tv_sec - tv1.tv_sec) << "seconds." << endl;
    #endif

    //check if any reachable states belong to bad states
    if (bddConBad != bddfalse)
    {
      BadStateInfo(bddConBad, HISC_VERI_LOW_UNCON, showtrace, sErr);
      throw -2;
    }

    #ifdef DEBUG_TIME
    cout << "Verifying Nonblocking..." << endl;
    gettimeofday(&tv1, NULL);
    #endif

    bddCoreach = cr(m_bddMarking, bddReach, iErr);
    if (iErr != 0)
      throw -1;

    #ifdef DEBUG_TIME
    gettimeofday(&tv2, NULL);
    cout << "VERI_NONBLOCKING: " << (tv2.tv_sec - tv1.tv_sec) << "seconds." << endl;
    #endif
```

```
        bddNBBad = bddReach & !bddCoreach;
    if (bddfalse != bddNBBad)
    {
      BadStateInfo(bddNBBad, HISC_VERI_LOW_BLOCKING, showtrace);
      throw -4;
    }

        #ifdef DEBUG_TIME
        cout << "Checking proper timed behavior..." << endl;
        gettimeofday(&tv1, NULL);
        #endif

        bddBalemiBad = bddfalse;
        if (VeriBalemiBad(bddBalemiBad, bddReach, sErr) < 0)
            throw -1;

        #ifdef DEBUG_TIME
        gettimeofday(&tv2, NULL);
        cout << "VERI_BALEMI: " << (tv2.tv_sec - tv1.tv_sec) << "seconds." << endl;
    #endif

        //check if any reachable states belong to Balemi bad states
        if (bddBalemiBad != bddfalse)
        {
            BadStateInfo(bddBalemiBad, HISC_VERI_LOW_CON, showtrace, sErr);
            throw -2;
        }

        // Checking if the system is ALF
    #ifdef DEBUG_TIME
    cout << "Verifying Activity Loop Free..." << endl;
    gettimeofday(&tv1, NULL);
    #endif
```

```

        bddALFBad = bddfalse;
        if (VeriALF(bddALFBad, bddReach, sErr) < 0)
            throw -1;

        #ifdef DEBUG_TIME
        gettimeofday(&tv2, NULL);
        cout << "VERI_ALF: " << (tv2.tv_sec - tv1.tv_sec) << "seconds."
<< endl;
    #endif

    if (bddALFBad != bddfalse)
    {
            BadStateInfo(bddALFBad, HISC_VERI_LOW_ALF, showtrace,
sErr);
            throw -2;
    }

        // Checking if the system has proper timed behavior
        #ifdef DEBUG_TIME
        cout << "Verifying Proper Timed Behavior..." << endl;
    gettimeofday(&tv1, NULL);
    #endif

        bddPTBBad = bddfalse;
        if (VeriProperTimedBehavior(bddPTBBad, bddReach, sErr) < 0)
            throw -1;

        #ifdef DEBUG_TIME
        gettimeofday(&tv2, NULL);
        cout << "VERI_PTB: " << (tv2.tv_sec - tv1.tv_sec) << "seconds."
<< endl;
    #endif

        if (bddPTBBad != bddfalse)
```

```
        {
            BadStateInfo(bddPTBBad, HISC_VERI_LOW_PTB, showtrace, sErr);
            throw -2;
        }

        // Checking SD Controllability
        #ifdef DEBUG_TIME
        cout << "Checking SD Controllability" << endl;
        gettimeofday(&tv1, NULL);
        #endif

        int ret = CheckSDControllability(bddSDBad, bddReach, sErr);
        if (-1 == ret)
            throw -1;

        #ifdef DEBUG_TIME
        gettimeofday(&tv2, NULL);
        cout << "VERI_SD: " << (tv2.tv_sec - tv1.tv_sec) << "seconds." << endl;
        #endif

        if (bddSDBad != bddfalse)
        {
            BadStateInfo(bddSDBad, ret, showtrace, sErr);
            throw -2;
        }

    //final synchronous product;
    m_bddSuper = bddReach;

    //save supervisor
    superinfo.statesize = bdd_satcount(m_bddSuper)/pow((double)2, double(m_iNumofBddNormVar));
    superinfo.nodesize =  bdd_nodecount(m_bddSuper);
```

```
  }
  catch (int iResult)
  {
    if (iResult < -1)
    {
      superinfo.statesize = bdd_satcount(bddReach)/pow((double)2, double(m_iNumofBddNormVar));
      superinfo.nodesize =  bdd_nodecount(bddReach);
    }

    iRet = -1;
  }
  ClearBddFields();
  CSub::ClearBddFields();
  bdd_done();

  return iRet;
}

/**
 * DESCR:   Does part of the sythesis work, i.e. controllable, p4, nonblocking
 * PARA:    computemethod: first compute reachable states or not (See BddHisc.h)
 *                         (input)
 *          bddReach: All the current reachable legal states
 *          bddBad: All the current bad states
 * RETURN:  0: sucess <0: fail
 * ACCESS:  private
 */
int CLowSub::SynPartSuper(const HISC_COMPUTEMETHOD computemethod,
                          bdd & bddReach, bdd & bddBad)
{
  bool bFirstLoop = true;
  bdd bddK = bddtrue;
```

```
int iErr = 0;

#ifdef DEBUG_TIME
int iCount = 0;
timeval tv1, tv2;
#endif

try
{
  if (computemethod == HISC_ONREACHABLE)
  {
    //compute controllable, p4, nonblocking fixpoint
    do
    {
      bddK = bddBad;

      //Computing [bddBad]
      #ifdef DEBUG_TIME
      cout << endl << "------------internal_loops:" << ++iCount <<
"---------------" << endl;
      cout << "Computing supremal controllable & P4 subpredicate..." <<
endl;
      gettimeofday(&tv1, NULL);
      #endif

      if (supcp(bddBad) < 0)
        throw -1;
      bddBad &= bddReach;

      #ifdef DEBUG_TIME
      gettimeofday(&tv2, NULL);
      cout << "supcp: " << (tv2.tv_sec - tv1.tv_sec) << "seconds." <<
endl;
      cout << "bddBad states:"
        << bdd_satcount(bddBad)/pow((double)2,
```

```
double(m_iNumofBddNormVar))
          << endl;
        cout << "bddBad Nodes:" << bdd_nodecount(bddBad) << endl;
        #endif

        if (bddK == bddBad && bFirstLoop == false)
          break;

        //Computing CR(not(bddBad))
        bdd bddTemp = bddReach - bddBad;

        #ifdef DEBUG_TIME
        cout << endl << "bddGood states:"
          << bdd_satcount(bddTemp)/pow((double)2,
double(m_iNumofBddNormVar))
          << endl;
        cout << "bddGood Nodes:" << bdd_nodecount(bddTemp) << endl;
        cout << endl << "Computing coreachable subpredicate..." << endl;
        gettimeofday(&tv1, NULL);
        #endif

        bddBad = bdd_not(cr(m_bddMarking, bddTemp, iErr));
        if (iErr != 0)
          throw -1;
        bddBad &= bddReach;
        bFirstLoop = false;

        #ifdef DEBUG_TIME
        gettimeofday(&tv2, NULL);
        cout << "cr: " << (tv2.tv_sec - tv1.tv_sec) << "seconds." <<
endl;
        cout << "bddBad states:"
          << bdd_satcount(bddBad)/pow((double)2,
double(m_iNumofBddNormVar))
          << endl;
```

```
        cout << "bddBad Nodes:" << bdd_nodecount(bddBad) << endl;
        #endif
      } while (bddBad != bddK);
    }
    else
    {
      //compute controllable, p4, nonblocking fixpoint
      do
      {
        bddK = bddBad;

        //Computing [bddBad]
        if (supcp(bddBad) < 0)
          throw -1;

        if (bddK == bddBad && bFirstLoop == false)
          break;

        //Computing CR(not(bddBad))
        bddBad = bdd_not(cr(m_bddMarking, bdd_not(bddBad), iErr));
        if (iErr != 0)
          throw -1;

        bFirstLoop = false;

      } while (bddBad != bddK);
    }
  }
  catch (int)
  {
    return -1;
  }
  return 0;
}

```

```
/**
 * DESCR:   Compute the initial bad states(Bad_{L_j})(uncontorlalble event part)
 * PARA:    bddConBad: BDD containing all the bad states (output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::GenConBad(bdd &bddConBad)
{
    try
    {
        bdd bddPlantTrans = bddfalse;

            for (int i = 0; i < m_usiMaxUnCon/ 2; i++)
            {
                //Compute illegal state predicate for each uncontrollable event
                bddConBad |= bdd_exist(m_pbdd_UnConPlantTrans[i],
                                       m_pbdd_UnConPlantVarPrim[i]) &
                             bdd_not(bdd_exist(m_pbdd_UnConSupTrans[i],
                             bdd_exist(m_pbdd_UnConVarPrim[i],
                                       m_pbdd_UnConPlantVarPrim[i])));
            }
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during generating controllable bad states.";
        pSub->SetErr(sErr, HISC_LOWERR_GENCONBAD);
        return -1;
    }
    return 0;
}
```

```
/**
 * DESCR:   Test if there are any bad states in the reachable states
 *          (Uncontorllable event part of Bad_{L_j})
 * PARA:    bddConBad: BDD containing tested bad states(output).
 *                     Initially, bddBad should be bddfalse.
 *          bddReach: BDD containing all reachable states
 *                    in this low-level(input)
 *          vsErr: returned errmsg(output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::VeriConBad(bdd &bddConBad, const bdd &bddReach, string & vsErr)
{
    try
    {
        int iErr = 0;

            for (int i = 0; i < m_usiMaxUnCon/ 2; i++)
            {
                //Compute illegal state predicate for each uncontrollable event
                bddConBad |= bdd_exist(m_pbdd_UnConPlantTrans[i],
                                       m_pbdd_UnConPlantVarPrim[i]) &
                             bdd_not(bdd_exist(m_pbdd_UnConSupTrans[i],
                                     bdd_exist(m_pbdd_UnConVarPrim[i],
                                     m_pbdd_UnConPlantVarPrim[i])));
                bddConBad &= bddReach;

                if (iErr < 0)
                {
                    throw -1;
                }

                if (bddConBad != bddfalse)
```

```
                {
                    vsErr = "Causing uncontrollable event: ";
                    vsErr += SearchEventName((i + 1) * 2);
                    throw -1;
                }
            }

    }
    catch(int)
    {
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during generating controllable bad states.";
        pSub->SetErr(sErr, HISC_LOWERR_GENCONBAD);
        return -1;
    }
    return 0;
}


/**
 * DESCR:   compute PLPC(P)
 * PARA:    bddP : BDD for predicate P. (input and output(=PHIC(P)))
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::supcp(bdd & bddP)
{
    bdd bddK1 = bddfalse;
    bdd bddK2 = bddfalse;
    int iEvent = 0;
    int iIndex = 0;

```

```
try
{
    while (bddP != bddK1)
    {
        bddK1 = bddP;
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            bddK2 = bddfalse;
            while (bddP != bddK2)
            {
                bddK2 = bddP;
                for (int j = 0; j < m_pDESArr[i]->GetNumofEvents();
j++)
                {
                    iEvent = (m_pDESArr[i]->GetEventsArr())[j];

                    iIndex = iEvent & 0x0000FFFF;
                    if ( iEvent % 2 == 0)
                    {
                        iIndex = (iIndex - 2) / 2;
                        bddP |=
                            bdd_appex(m_pbdd_UnConTrans[iIndex],
                                     bdd_replace(bddK2,
                                           m_pPair_UnCon[iIndex]),
                                     bddop_and,
                                     m_pbdd_UnConVarPrim[iIndex]);
                    }
                }
            }
        }
    }
}
catch (...)
{
    string sErr = this->GetSubName();
```

```
        sErr += ": Error during computing PLPC(P).";
        pSub->SetErr(sErr, HISC_LOWERR_SUPCP);
        return -1;
    }
    return 0;
}

/**
 * DESCR:   compute CR(G_{L_j}, P', \Sigma', P)
  * PARA:    bddPStart: P' (input)
  *          bddP: P (input)
  *          viEventSub: \Sigma' (input) (0,1,2,3) <-> (H,R,A,L)
ALL_EVENT<->All
  *          iErr: returned Errcode (0: success <0: fail)(output)
  * RETURN:  BDD for CR(G_{L_j}, P', \Sigma', P)
  * ACCESS:  private
  */
bdd CLowSub::cr(const bdd & bddPStart, const bdd & bddP, int & iErr)
{
  try
  {
    bdd bddK = bddP & bddPStart;
    bdd bddK1 = bddfalse;
    bdd bddK2 = bddfalse;
    bdd bddKNew = bddfalse;
    int iEvent = 0;
    int iIndex = 0;

    #ifdef DEBUG_TIME
    int iLoopCount = 0;
    timeval tv1, tv2;
    #endif

    while (bddK != bddK1)
    {
```

```
#ifdef DEBUG_TIME
gettimeofday(&tv1, NULL);
#endif

bddK1 = bddK;

for (int i = 0; i < this->GetNumofDES(); i++)
{
  bddK2 = bddfalse;
  while (bddK != bddK2)
  {
    bddKNew = bddK - bddK2;
    bddK2 = bddK;
    for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
    {
      iEvent = (m_pDESArr[i]->GetEventsArr())[j];

        iIndex = iEvent & 0x0000FFFF;
        if (iEvent % 2 == 0)
        {
          iIndex = (iIndex - 2) / 2;
          bddK |= bdd_appex(m_pbdd_UnConTrans[iIndex],
            bdd_replace(bddKNew, m_pPair_UnCon[iIndex]),
            bddop_and, m_pbdd_UnConVarPrim[iIndex])
            & bddP;
        }
        else
        {
          iIndex = (iIndex - 1) / 2;
          bddK |= bdd_appex(m_pbdd_ConTrans[iIndex],
            bdd_replace(bddKNew, m_pPair_Con[iIndex]),
            bddop_and, m_pbdd_ConVarPrim[iIndex])
            & bddP;
        }

```

```
          }
        }
      }
      #ifdef DEBUG_TIME
      gettimeofday(&tv2, NULL);
      cout << "CR: Iteration_" << ++iLoopCount << " nodes: " << bdd_nodecount(bddK);
      cout << "\t time: " << ((tv2.tv_sec - tv1.tv_sec) * 1000000.0 + (tv2.tv_usec - tv1.tv_usec))/1000000.0 << " s";
      cout << "\t states: " << bdd_satcount(bddK)/pow((double)2, double(m_iNumofBddNormVar)) << endl;
      #endif
    }
    return bddK;
  }
  catch (...)
  {
    string sErr = this->GetSubName();
    sErr += ": Error during computing coreachable.";
    pSub->SetErr(sErr, HISC_LOWERR_COREACH);
    iErr = -1;
    return bddfalse;
  }
}


/**
 * DESCR:   compute R(G_{L_j}, P)
 * PARA:    bddP: P (input)
 *          iErr: returned Errcode (0: success <0: fail)(output)
 * RETURN:  BDD for R(G_{L_j}, P)
 * ACCESS:  private
 */
bdd CLowSub::r(const bdd &bddP, int &iErr)
{
```

```
try
{
  bdd bddK = bddP & m_bddInit;
  bdd bddK1 = bddfalse;
  bdd bddK2 = bddfalse;
  bdd bddKNew = bddfalse;
  int iEvent = 0;
  int iIndex = 0;

  #ifdef DEBUG_TIME
  int iLoopCount = 0;
  timeval tv1, tv2;
  #endif

  while (bddK != bddK1)
  {
    #ifdef DEBUG_TIME
    gettimeofday(&tv1, NULL);
    #endif

    bddK1 = bddK;


    for (int i = 0; i < this->GetNumofDES(); i++)
    {
      bddK2 = bddfalse;
      while (bddK != bddK2)
      {
        bddKNew = bddK - bddK2;
        bddK2 = bddK;

        for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
        {
          iEvent = (m_pDESArr[i]->GetEventsArr())[j];

```

```
              iIndex = iEvent & 0x0000FFFF;
              if (iEvent % 2 == 0)
              {
                iIndex = (iIndex - 2) / 2;
                bddK |= bdd_replace(
                  bdd_appex(m_pbdd_UnConTrans[iIndex], bddKNew,
bddop_and,
                                      m_pbdd_UnConVar[iIndex]),
                  m_pPair_UnConPrim[iIndex]) & bddP;
              }
              else
              {
                iIndex = (iIndex - 1) / 2;
                bddK |= bdd_replace(
                  bdd_appex(m_pbdd_ConTrans[iIndex], bddKNew, bddop_and,
                                      m_pbdd_ConVar[iIndex]),
                  m_pPair_ConPrim[iIndex]) & bddP;
              }
            }
          }
        }
        #ifdef DEBUG_TIME
        gettimeofday(&tv2, NULL);
        cout << "R: Iteration_" << ++iLoopCount << " nodes: " <<
bdd_nodecount(bddK);
        cout << "\t time: " << ((tv2.tv_sec - tv1.tv_sec) * 1000000.0 +
(tv2.tv_usec - tv1.tv_usec))/1000000.0 << " s";
        cout << "\t states: " << bdd_satcount(bddK)/pow((double)2,
double(m_iNumofBddNormVar)) << endl;
        #endif
      }
      return bddK;
    }
    catch (...)
    {
```

```
        string sErr = this->GetSubName();
        sErr += ": Error during computing coreachable.";
        pSub->SetErr(sErr, HISC_LOWERR_REACH);
        iErr = -1;
        return bddfalse;
    }
}


```

LowSub4.cpp

```

        //If tick does not exist
        if (iTick < 0)
        {
            string sErr = this->GetSubName();
            sErr += ": Tick event is not found.";
            pSub->SetErr(sErr, HISC_TICK_NOT_FOUND);

            cout << "Tick not found." << endl;
            return 0;
        }

        for (int i = 0; i < m_usiMaxUnCon / 2; i++)
        {
            // Get all the states left by uncontrollable event i.
            bddTemp = bdd_exist(m_pbdd_UnConPlantTrans[i],
m_pbdd_UnConPlantVarPrim[i]);
            bddP1 |= bddTemp;
        }

        // Get all states left by tick event
        bddTemp = bdd_exist(m_pbdd_ConPlantTrans[iTick],
m_pbdd_ConPhysicVarPrim[iTick]);
```

```
        bddP1 |= bddTemp;

        VERBOSE(2)
        {
            PRINT_DEBUG << "bddReach: ";
            PrintStateSet2(bddReach);
            cout << endl;
        }

        bddPTBBad = bddReach - bddP1;

        if(bddPTBBad != bddfalse)
        {
            VERBOSE(2)
            {
                PRINT_DEBUG << "bddPTBBad: ";
                PrintStateSet2(bddPTBBad);
                cout << endl;
            }

            vsErr = "Not proper timed behavior.";
            throw -1;
        }
    }
    catch(int)
    {
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error when checking proper timed behavior.";
        pSub->SetErr(sErr, HISC_LOWERR_PTB);
        return -1;
    }
```

```
    return 0;
}

int CLowSub::VeriALF(bdd &bddALFBad, bdd bddReach, string & vsErr)
{
    const char * DEBUG = "CLowSub::VeriALF():";

    int iTick = (SearchSubEvent(sTick) - 1) / 2;
    VERBOSE(1) { PRINT_DEBUG << "iTick = " << iTick << endl; }

    //If tick does not exist
    if (iTick < 0)
    {
        string sErr = this->GetSubName();
        sErr += ": Tick event is not found.";
        pSub->SetErr(sErr, HISC_TICK_NOT_FOUND);

        cout << "Tick not found." << endl;
        return 0;
    }

    bdd bddChk = bddReach;
    bdd bddTemp = bddfalse;

    try
    {
        while (bddfalse != bddChk)
        {
            VERBOSE(2)
            {
                PRINT_DEBUG << "bddChk: ";
                PrintStateSet2(bddChk);
                cout << endl;
            }

```

```
        bdd bddQ = GetOneState(bddChk);

        VERBOSE(2)
        {
            PRINT_DEBUG << "bddQ: ";
            PrintStateSet2(bddQ);
            cout << endl;
        }

        bdd bddVisit = bddfalse;

        for (int i = 0; i < (m_usiMaxCon + 1) / 2; i++)
        {
            if (i == iTick) continue;

            bddTemp = bdd_relprod(m_pbdd_ConTrans[i], bddQ, m_pbdd_ConVar[i]);
            bddVisit |= bdd_replace(bddTemp, m_pPair_ConPrim[i]);
        }

        for (int i = 0; i < (m_usiMaxUnCon / 2); i++)
        {
            bddTemp = bdd_relprod(m_pbdd_UnConTrans[i], bddQ, m_pbdd_UnConVar[i]);
            bddVisit |= bdd_replace(bddTemp, m_pPair_UnConPrim[i]);
        }

        bddVisit &= bddChk;

        VERBOSE(2)
        {
            PRINT_DEBUG << "bddVisit: ";
            PrintStateSet2(bddVisit);
            cout << endl;
        }
```

```

bool overlap = false;

bdd bddNext = bddfalse;

for (int i = 0; i < (m_usiMaxCon + 1) / 2; i++)
{
    if (i == iTick) continue;

    bddTemp = bdd_relprod(m_pbdd_ConTrans[i], bddVisit, m_pbdd_ConVar[i]);
    bddNext |= bdd_replace(bddTemp, m_pPair_ConPrim[i]);
}

for (int i = 0; i < (m_usiMaxUnCon / 2); i++)
{
    bddTemp = bdd_relprod(m_pbdd_UnConTrans[i], bddVisit, m_pbdd_UnConVar[i]);
    bddNext |= bdd_replace(bddTemp, m_pPair_UnConPrim[i]);
}

bddNext &= bddChk;

VERBOSE(2)
{
    PRINT_DEBUG << "bddNext: ";
    PrintStateSet2(bddNext);
    cout << endl;
}

bdd bddOldVisit = bddfalse;
do
{
    bddOldVisit = bddVisit;

```

```
        if (bddfalse != (bddVisit & bddNext))
        {
            overlap = true;
        }

        bddVisit |= bddNext;

        VERBOSE(2)
        {
            PRINT_DEBUG << "bddVisit: ";
            PrintStateSet2(bddVisit);
            cout << endl;
        }

        bddALFBad = bddQ & bddVisit;
        if (bddfalse != bddALFBad)
        {
            VERBOSE(2)
            {
                PRINT_DEBUG << "bddALFBad: ";
                PrintStateSet2(bddALFBad);
                cout << endl;
            }

            vsErr = "Not ALF.";
            throw -1;
        }

        bdd bddNewNext = bddfalse;

        for (int i = 0; i < (m_usiMaxCon + 1) / 2; i++)
        {
            if (i == iTick) continue;

            bddTemp = bdd_relprod(m_pbdd_ConTrans[i], bddNext,
```

```
m_pbdd_ConVar[i]);
                    bddNewNext |= bdd_replace(bddTemp, m_pPair_ConPrim[i]);
                }

                for (int i = 0; i < (m_usiMaxUnCon / 2); i++)
                {
                    bddTemp = bdd_relprod(m_pbdd_UnConTrans[i], bddNext, m_pbdd_UnConVar[i]);
                    bddNewNext |= bdd_replace(bddTemp, m_pPair_UnConPrim[i]);
                }

                bddNext = bddNewNext & bddChk;

                VERBOSE(2)
                {
                    PRINT_DEBUG << "bddNext: ";
                    PrintStateSet2(bddNext);
                    cout << endl;
                }
            }
            while (bddVisit != bddOldVisit);

            VERBOSE(1) { PRINT_DEBUG << "overlap: " << (overlap ? "true" : "false") << endl; }

            bddChk -= bddQ;
            if (!overlap)
            {
                bddChk -= bddVisit;
            }
        }
    }
    catch(int)
```

```
    {
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error when checking ALF.";
        pSub->SetErr(sErr, HISC_LOWERR_ALF);
        return -1;
    }
    return 0;
}

/**
 * DESCR:   Compute the Balemi bad states
 * PARA:    bddBalemiBad: BDD containing all the bad states (output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::GenBalemiBad(bdd &bddBalemiBad)
{
    const char * DEBUG = "CLowSub::VeriBalemiBad():";

    int iTick = (SearchSubEvent(sTick) - 1) / 2;
    VERBOSE(1) { PRINT_DEBUG << "iTick = " << iTick << endl; }

    try
    {
        bdd bddPlantTrans = bddfalse;

        for (int i = 0; i < (m_usiMaxCon + 1) / 2; i++)
        {
            if (i == iTick) continue;

            //Compute illegal state predicate for each uncontrollable
event
```

```
                bddBalemiBad |= bdd_not(bdd_exist(m_pbdd_ConPlantTrans[i],
                                        m_pbdd_ConPhysicVarPrim[i])) &
                                        bdd_exist(m_pbdd_ConSupTrans[i],
                                        bdd_exist(m_pbdd_ConVarPrim[i],
                                        m_pbdd_ConPhysicVarPrim[i]));
            }
        }
        catch(...)
        {
            string sErr = this->GetSubName();
            sErr += ": Error during generating bad states for proper timed behavior.";
            pSub->SetErr(sErr, HISC_LOWERR_GENBALEMIBAD);
            return -1;
        }
        return 0;
}

/**
 * DESCR:   Test if there are any Balemi bad states in the reachable states
 * PARA:    bddBalemiBad: BDD containing tested bad states(output).
 *                         Initially, bddBad should be bddfalse.
 *          bddReach: BDD containing all reachable states
 *                         in this low-level(input)
 *          vsErr: returned errmsg(output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::VeriBalemiBad(bdd &bddBalemiBad, const bdd &bddReach, string & vsErr)
{
        const char * DEBUG = "CLowSub::VeriBalemiBad():";

        int iTick = (SearchSubEvent(sTick) - 1) / 2;
```

```
    VERBOSE(1) { PRINT_DEBUG << "iTick = " << iTick << endl; }

    //If tick does not exist
    if (iTick < 0)
    {
        string sErr = this->GetSubName();
        sErr += ": Tick event is not found.";
        pSub->SetErr(sErr, HISC_TICK_NOT_FOUND);

        cout << "Tick not found." << endl;
        return 0;
    }

    try
    {
        int iErr = 0;

            for (int i = 0; i < (m_usiMaxCon + 1) / 2; i++)
            {
                if (i == iTick) continue;

                //Compute illegal state predicate for each uncontrollable event
                bddBalemiBad |= bdd_not(bdd_exist(m_pbdd_ConPlantTrans[i],
                                                  m_pbdd_ConPhysicVarPrim[i])) &
                                bdd_exist(m_pbdd_ConSupTrans[i],
                                          bdd_exist(m_pbdd_ConVarPrim[i],
                                          m_pbdd_ConPhysicVarPrim[i]));
                bddBalemiBad &= bddReach;
                //bddBalemiBad = r(bddBalemiBad, iErr);
                if (iErr < 0)
                {
                    throw -1;
                }
```

```
                    if (bddBalemiBad != bddfalse)
                    {
                        vsErr = "Causing controllable event:";
                        vsErr += SearchEventName((i * 2) + 1);
                        throw -1;
                    }
                }
        }
        catch(int)
        {
        }
        catch(...)
        {
            string sErr = this->GetSubName();
            sErr += ": Error during generating bad states for proper timed behavior.";
            pSub->SetErr(sErr, HISC_LOWERR_GENBALEMIBAD);
            return -1;
        }
        return 0;
}
```

LowSub5.cpp

```
            {
                VERBOSE(1) { PRINT_DEBUG << "CheckTimedControllability()\t= " << ret << endl; }

                throw HISC_VERI_LOW_SD_II;
            }

            bdd bddSF = m_bddInit;

```

```
        stack<bdd> stack_bddSP;
        stack_bddSP.push(m_bddInit);

        list< list<bdd> > list_NerFail;

        int iSubTick = SearchSubEvent(sTick);
        VERBOSE(1) { PRINT_DEBUG << "iSubTick = " << iSubTick << endl; }

        //If tick does not exist
        if (iSubTick < 0)
        {
            string sErr = this->GetSubName();
            sErr += ": Tick event is not found.";
            pSub->SetErr(sErr, HISC_TICK_NOT_FOUND);

            VERBOSE(1) { PRINT_DEBUG << "Tick not found." << endl; }
            return 0;
        }

        int r;
        while (!stack_bddSP.empty())
        {
            bdd bddSS = stack_bddSP.top();
            stack_bddSP.pop();

            r = AnalyseSampledState(bddSDBad, bddreach, bddSS,
list_NerFail, bddSF, stack_bddSP, vsErr);
            if (r < 0)
            {
                VERBOSE(1) { PRINT_DEBUG << "AnalyseSampledState() ¡ 0" <<
endl; }
                vsErr = "AnalyseSampledState() Failed: " + vsErr;
                throw r;
            }
        }
```

```

        if (!list_NerFail.empty())
        {
            VERBOSE(1) { PRINT_DEBUG << "list_NerFail is not empty." << endl; }
            if (!RecheckNerodeCells(bddSDBad, bddreach, list_NerFail))
            {
                VERBOSE(1) { PRINT_DEBUG << "RecheckNerodeCells() ¡ 0" << endl; }
                vsErr = "list_NerFail is not empty and RecheckNerodeCells() Failed.";
                throw HISC_VERI_LOW_SD_III_2;
            }
        }

        CheckSDiv(bddSDBad, bddreach);
        if (bddSDBad != bddfalse)
        {
            VERBOSE(1) { PRINT_DEBUG << "(m_bddMarking - bddTemp) != m_bddInit" << endl; }
            vsErr = "There is a reachable marking state reached by a non-tick event.";
            throw HISC_VERI_LOW_SD_IV;
        }
    }
    catch(int failureCode)
    {
        ret = failureCode;
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error when checking SD Controllability.";
        pSub->SetErr(sErr, HISC_LOWERR_SD);
        return -1;
```

```
    }
    return ret;
}

// Algorithm 6.12
int CLowSub::AnalyseSampledState(bdd & bddSSBad, const bdd & bddreach, const bdd & bddSS,
    list< list<bdd> > & list_NerFail, bdd & bddSF, stack<bdd> & stack_bddSP, string & vsErr)
{
    const char * DEBUG = "CLowSub::AnalyseSampledState():";

    VERBOSE(2)
    {
        cout << endl;
        PRINT_DEBUG << "Analysing Sample State: ";
        PrintStateSet2(bddSS);
        cout << endl;
    }

    map<int, bdd> B_map;

    // Line 1
    B_map[0] = bddSS;

    // Line 2
    map<int, bdd> B_conc;

    stack<int> B_p;

    // Line 3
    B_p.push(0);

    // Line 4
    int intNextFreeLabel = 1;
```

```
    // Line 5
    map<int, EVENTSET> B_occu;

    // Line 6
    EVENTSET eventsElig;

    int iSubTick = SearchSubEvent(sTick);
    int iTick = (iSubTick - 1) / 2;
    VERBOSE(1) { PRINT_DEBUG << "iTick\t= " << iTick << endl; }

    // Line 7
    while (!B_p.empty())
    {
        VERBOSE(1)
        {
            cout << endl;
            PRINT_DEBUG << "B_p.size()\t= " << B_p.size() << endl;
        }

        // Line 8
        int b = B_p.top();
        B_p.pop();

        // Line 9
        bdd bddZ = B_map[b];

        VERBOSE(2) { PRINT_DEBUG << "bddZ : "; PrintStateSet2(bddZ); cout << endl; }

        bdd bddtemp = bddfalse;

        /* *************************************** */

        // Line 10
```

```
        EVENTSET eventsA;

        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP START : m_SubPlantEvents" << endl; }

        // Line 11
        for (EVENTSET::iterator i = m_SubPlantEvents.begin(); i != m_SubPlantEvents.end(); i++)
        {
            int iIndex, event = *i;
            if (event < 1)
            {
                VERBOSE(1) { PRINT_DEBUG << "ERROR - Found a Sub-level event index lower than 1" << endl; }
                return HISC_INTERNAL_ERR_SUBEVENT;
            }

            VERBOSE(1) { PRINT_DEBUG << "event\t= " << m_InvSubEventsMap[event] << endl; }
            if (1 == event % 2) //Controllable
            {
                iIndex = (event - 1) / 2;
                bddtemp = bdd_relprod(m_pbdd_ConPlantTrans[iIndex], bddZ, m_pbdd_ConVar[iIndex]);
                bddtemp = bdd_replace(bddtemp, m_pPair_ConPrim[iIndex]);
            }
            else //Uncontrollable
            {
                iIndex = (event / 2) - 1;
                bddtemp = bdd_relprod(m_pbdd_UnConPlantTrans[iIndex], bddZ, m_pbdd_UnConVar[iIndex]);
                bddtemp = bdd_replace(bddtemp, m_pPair_UnConPrim[iIndex]);
            }
```

```

            bddtemp &= bddreach;
            VERBOSE(2) { PRINT_DEBUG << "bddtemp\t= "; PrintStateSet2(bddtemp); cout << endl; }

            // Line 12
            if (bddtemp != bddfalse)
            {
                VERBOSE(1) { PRINT_DEBUG << "bddtemp != bddfalse" << endl; }

                // Line 13
                eventsA.insert(event);
                VERBOSE(1) { PRINT_DEBUG << "eventsA.size()\t= " << eventsA.size() << endl; }

            // Line 14
            }

        // Line 15
        }
        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP END   : m_SubPlantEvents" << endl; }

        // Line 16
        EVENTSET eventsD;

        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP START : m_SubSupervisorEvents" << endl; }

        // Line 17
        for (EVENTSET::iterator i = m_SubSupervisorEvents.begin(); i != m_SubSupervisorEvents.end(); i++)
        {
            bdd bddSupervisorTrans = bddfalse;
```

```
        int iIndex, event = *i;
        if (event < 1)
        {
            VERBOSE(1) { PRINT_DEBUG << "ERROR - Found a Sub-level event index lower than 1" << endl; }
            return HISC_INTERNAL_ERR_SUBEVENT;
        }

        if (1 == event % 2) //Controllable
        {
            iIndex = (event - 1) / 2;
            //Get supervisor transition predicate
            bddSupervisorTrans = bdd_exist(m_pbdd_ConTrans[iIndex], m_pbdd_ConPhysicVar[iIndex]);
            bddSupervisorTrans = bdd_exist(bddSupervisorTrans, m_pbdd_ConPhysicVarPrim[iIndex]);

            bddtemp = bdd_relprod(bddSupervisorTrans, bddZ, m_pbdd_ConVar[iIndex]);
            bddtemp = bdd_replace(bddtemp, m_pPair_ConPrim[iIndex]);
        }
        else //Uncontrollable
        {
            iIndex = (event / 2) - 1;
            //Get supervisor transition predicate
            bddSupervisorTrans = bdd_exist(m_pbdd_UnConTrans[iIndex], m_pbdd_UnConPlantVar[iIndex]);
            bddSupervisorTrans = bdd_exist(bddSupervisorTrans, m_pbdd_UnConPlantVarPrim[iIndex]);

            bddtemp = bdd_relprod(bddSupervisorTrans, bddZ, m_pbdd_UnConVar[iIndex]);
            bddtemp = bdd_replace(bddtemp, m_pPair_UnConPrim[iIndex]);
```

```
            }

            bddtemp &= bddreach;

            // Line 18
            if (bddtemp != bddfalse)
            {
                VERBOSE(1) { PRINT_DEBUG << "bddtemp != bddfalse" << endl; }

                // Line 19
                eventsD.insert(event);
                VERBOSE(1) { PRINT_DEBUG << "eventsD.size()\t= " << eventsD.size() << endl; }

            // Line 20
            }

        // Line 21
        }
        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP END  : m_SubSupervisorEvents" << endl; }

        EVENTSET eventsPoss;
        EVENTSET eventsDis = eventsA;
        for (EVENTSET::iterator i = eventsA.begin(); i != eventsA.end(); i++)
        {
            if (eventsD.end() != eventsD.find(*i))
            {
                // Line 22
                eventsPoss.insert(*i);
                eventsDis.erase(*i);
            }
        }
```

```

/* **************************************** */

        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP START : eventsPoss" << endl; }
        for (EVENTSET::iterator i = eventsPoss.begin(); i != eventsPoss.end(); i++)
        {
            if ((*i) < 1)
            {
                VERBOSE(1) { PRINT_DEBUG << "ERROR - Found a Sub-level event index lower than 1" << endl; }
                return HISC_INTERNAL_ERR_SUBEVENT;
            }
            VERBOSE(1) { PRINT_DEBUG << "eventsPoss : " << m_InvSubEventsMap[(*i)] << endl; }
        }
        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP END   : eventsPoss" << endl; }

        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP START : eventsDis" << endl; }
        for (EVENTSET::iterator i = eventsDis.begin(); i != eventsDis.end(); i++)
        {

            if ((*i) < 1)
            {
                VERBOSE(1) { PRINT_DEBUG << "ERROR - Found a Sub-level event index lower than 1" << endl; }
                return HISC_INTERNAL_ERR_SUBEVENT;
            }
            VERBOSE(1) { PRINT_DEBUG << "eventsDis : " << m_InvSubEventsMap[(*i)] << endl; }
        }
```

```
        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP END   : eventsDis" << endl; }

        // Line 23
        if (bddZ == bddSS)
        {
            VERBOSE(1) { PRINT_DEBUG << "bddZ == bddSS" << endl; }
            eventsElig = eventsPoss;

            // Line 24
            // Remove uncontrollable events
            for (int i = 0; i < m_usiMaxUnCon / 2; i++)
            {
                eventsElig.erase((i + 1) * 2);
            }
            // Remove tick event
            eventsElig.erase(iSubTick);

        // Line 25
        }

        VERBOSE(1) { PRINT_DEBUG << "eventsElig.size() :" << eventsElig.size() << endl; }
        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP START : eventsElig" << endl; }
        for (EVENTSET::iterator i = eventsElig.begin(); i != eventsElig.end(); i++)
        {

            if ((*i) < 1)
            {
                VERBOSE(1) { PRINT_DEBUG << "ERROR - Found a Sub-level event index lower than 1" << endl; }
                return HISC_INTERNAL_ERR_SUBEVENT;
            }
```

```
            VERBOSE(1) { PRINT_DEBUG << "eventsElig : " << m_InvSubEventsMap[(*i)] << endl; }
        }
        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP END   : eventsElig" << endl; }

        EVENTSET eventsTemp = eventsPoss;
        eventsTemp.insert(B_occu[b].begin(), B_occu[b].end());

        // Remove uncontrollable events
        for (int i = 0; i < m_usiMaxUnCon / 2; i++)
        {
            eventsTemp.erase((i + 1) * 2);
        }
        // Remove tick event
        eventsTemp.erase(iSubTick);

        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP START : eventsTemp" << endl; }
        for (EVENTSET::iterator i = eventsTemp.begin(); i != eventsTemp.end(); i++)
        {
            if ((*i) < 1)
            {
                VERBOSE(1) { PRINT_DEBUG << "ERROR - Found a Sub-level event index lower than 1" << endl; }
                return HISC_INTERNAL_ERR_SUBEVENT;
            }
            VERBOSE(1) { PRINT_DEBUG << "eventsTemp = (eventsPoss V B_occu[" << b << "]) ^ ¡P_hib¿ : " << m_InvSubEventsMap[(*i)] << endl; }
        }
        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP END   : eventsTemp" << endl; }

        // Line 26
```

```
        if ((eventsTemp < eventsElig) || (eventsTemp > eventsElig) )
        {
            bddSSBad = bddZ;
            VERBOSE(1) { PRINT_DEBUG << "eventsTemp ¡¿ eventsElig" << endl; }

            VERBOSE(1) { PRINT_DEBUG << "eventsTemp.size() :" << eventsTemp.size() << endl; }

            // Line 27
            return HISC_VERI_LOW_SD_III_1;

        // Line 28
        }

        // Line 29
        if (-1 == DetermineNextState(bddSSBad, eventsPoss, bddZ, bddreach, b, intNextFreeLabel, B_map, B_p,
            bddSF, stack_bddSP, B_occu, B_conc, vsErr))
        {
            // Line 30
            return HISC_VERI_LOW_ZERO_LB;
        // Line 31
        }

    // Line 32
    }

    // Line 33
    CheckNerodeCells(B_conc, B_occu, list_NerFail);
    return 0;
}

void CLowSub::CheckNerodeCells(map<int, bdd> & B_conc, map<int, EVENTSET> & B_occu,
```

```
    list< list<bdd> > & list_NerFail)
{
    const char * DEBUG = "CLowSub::CheckNerodeCells():";

    VERBOSE(1) { PRINT_DEBUG << "WHILE-LOOP START : !B_conc.empty()" << endl; }

    // Line 2
    while (!B_conc.empty())
    {
        map<int, bdd>::iterator i = B_conc.begin();

        // Line 3
        int b = (*i).first;
        bdd bddZ = (*i).second;
        B_conc.erase(i);

        VERBOSE(2)
        {
            PRINT_DEBUG << "(b, bddZ) = (" << b << ", ";
            PrintStateSet2(bddZ);
            cout << ")" << endl;
        }

        // Line 3
        list<bdd> Zeqv;

        // Line 4
        Zeqv.push_back(bddZ);

        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP START : B_conc" << endl; }

        // Work around: C++ doesn't allow a map collection (i.e. B_Conc) to be modified in a loop
```

```
//                 by collection iterator. Need to first save B_Conc iterators in a list,
//                 and then read the iterators from the list in the loop from // Line 5.
list<map<int, bdd>::iterator> iteratorList_B_Conc;
for (map<int, bdd>::iterator k = B_conc.begin(); B_conc.end() != k; k++)
{
    iteratorList_B_Conc.push_back(k);
}

// Line 5
bool sameCell = true;

// Line 6
for (list<map<int, bdd>::iterator>::iterator j = iteratorList_B_Conc.begin(); iteratorList_B_Conc.end() != j; j++)
{
    int bprime = (*(*j)).first;
    VERBOSE(1) { PRINT_DEBUG << "bprime : " << bprime << endl; }

    bdd bddZprime = (*(*j)).second;

    VERBOSE(2)
    {
        PRINT_DEBUG << "bddZprime : ";
        PrintStateSet2(bddZprime);
        cout << endl;
    }
    // Line 7
    if (B_occu[b] == B_occu[bprime])
    {
        VERBOSE(1) { PRINT_DEBUG << "B_occu[b:" << b << "] == B_occu[bprime:" << bprime << "]" << endl; }

```

```
                // Line 8
                Zeqv.push_back(bddZprime);

                // Line 9
                B_conc.erase(*j);

                // Line 10
                if (bddZ != bddZprime)
                {
                    VERBOSE(1) { PRINT_DEBUG << "bddZ != bddZprime" << endl; }

                    // Line 11
                    sameCell = false;

                // Line 12
                }
            // Line 13
            }
        // Line 14
        }
        VERBOSE(1) { PRINT_DEBUG << "FOR-LOOP END : B_conc" << endl; }

        // Line 15
        if (!sameCell)
        {
            VERBOSE(1) { PRINT_DEBUG << "sameCell : false" << endl; }
            // Line 16
            list_NerFail.push_back(Zeqv);

        // Line 17
        }

    // Line 18
```

```
    }
    VERBOSE(1) { PRINT_DEBUG << "WHILE-LOOP END   : !B_conc.empty()" << endl; }

    // Line 19
    return;
}

int CLowSub::DetermineNextState(bdd & bddLBBad, const EVENTSET & eventsPoss, const bdd & bddZ, const bdd & bddreach,
    const int & intB, int & intNextFreeLabel, map<int, bdd> & B_map, stack<int> & B_p,
    bdd & bddSF, stack<bdd> & stack_bddSP,
    map<int, EVENTSET> & B_occu, map<int, bdd> & B_conc, string & vsErr)
{
    const char * DEBUG = "CLowSub::DetermineNextState():";

    // Line 1
    if (eventsPoss.empty())
    {
        VERBOSE(1) { PRINT_DEBUG << "eventsPoss is empty" << endl; }

        // Line 2
        return 0;
    } //Line 3

    int iSubTick = SearchSubEvent(sTick);
    int iTick = (iSubTick - 1) / 2;

    VERBOSE(1) { PRINT_DEBUG << "iSubTick = " << iSubTick << endl; }
    VERBOSE(1) { PRINT_DEBUG << "iTick = " << iTick << endl; }

    // Line 4
    if (eventsPoss.end() != eventsPoss.find(iSubTick))
```

```
    {
        VERBOSE(1) { PRINT_DEBUG << "Found tick in eventsPoss." << endl; }

        // Line 5
        bdd bddZprime = bdd_relprod(m_pbdd_ConTrans[iTick], bddZ, m_pbdd_ConVar[iTick]);
        bddZprime = bdd_replace(bddZprime, m_pPair_ConPrim[iTick]);
        bddZprime &= bddreach;

        VERBOSE(2)
        {
            PRINT_DEBUG << "bddZprime = ";
            PrintStateSet2(bddZprime);
            cout << endl;
        }

            // Line 7
            B_conc.insert(make_pair(intB, bddZprime));

            // Line 8
            if ((bddZprime & bddSF) == bddfalse)
            {
                // Line 9
                bddSF |= bddZprime;

                // Line 10
                stack_bddSP.push(bddZprime);

            // Line 11
            }

            VERBOSE(1) { PRINT_DEBUG << "eventsPoss.size() = " << eventsPoss.size() << endl; }

            // If tick is the only event in eventsPoss, then no need to
```

```
run anything after Line 14.
                if (1 == eventsPoss.size())
                {
                    VERBOSE(1) { PRINT_DEBUG << "eventsPoss only has a tick." << endl; }
                    return 0;
                }

        // Line 13
        }

        // Line 14
        for (EVENTSET::iterator i = eventsPoss.begin(); i != eventsPoss.end(); i++)
        {
            int event, iSubEvent = *i;

            if (iSubEvent < 1)
            {
                VERBOSE(1) { PRINT_DEBUG << "ERROR - Found a Sub-level event index lower than 1" << endl; }
                return HISC_INTERNAL_ERR_SUBEVENT;
            }

            VERBOSE(1) { PRINT_DEBUG << "iSubEvent = " << m_InvSubEventsMap[iSubEvent] << endl; }

            if (iSubEvent == iSubTick)
            {
                continue;
            }

            // Line 15
            bdd bddZprime;

```

```
if (1 == iSubEvent % 2) //Controllable
{
    event = (iSubEvent - 1) / 2;
    VERBOSE(1) { PRINT_DEBUG << "Controllable event = " << m_InvSubEventsMap[iSubEvent] << endl; }
    bddZprime = bdd_relprod(m_pbdd_ConTrans[event], bddZ, m_pbdd_ConVar[event]);
    bddZprime = bdd_replace(bddZprime, m_pPair_ConPrim[event]);
}
else //Uncontrollable
{
    event = (iSubEvent / 2) - 1;
    VERBOSE(1) { PRINT_DEBUG << "Uncontrollable event = " << m_InvSubEventsMap[iSubEvent] << endl; }
    bddZprime = bdd_relprod(m_pbdd_UnConTrans[event], bddZ, m_pbdd_UnConVar[event]);
    bddZprime = bdd_replace(bddZprime, m_pPair_UnConPrim[event]);
}

bddZprime &= bddreach;

VERBOSE(2)
{
    PRINT_DEBUG << "bddZprime = "; PrintStateSet2(bddZprime); cout << endl;
}

EVENTSET eventsTemp = B_occu[intB];

// Line 17
if ((1 == iSubEvent % 2) && (eventsTemp.end() != eventsTemp.find(iSubEvent)))
{
    bddLBBad = B_map[intB];
```

```
            vsErr = "Event " + SearchEventName(iSubEvent) + " is found to occur more than 1 times in this sampling period.";

            // Line 18
            return -1;

        // Line 19
        }

        // Line 20
        int intBprime = intNextFreeLabel;

        VERBOSE(1) { PRINT_DEBUG << "intBprime = " << intBprime << endl; }

        // Line 21
        intNextFreeLabel++;

        // Line 22
        B_map.insert(make_pair(intBprime, bddZprime));

        VERBOSE(1) { PRINT_DEBUG << "B_map.size() = " << B_map.size() << endl; }

        // Line 23
        B_p.push(intBprime);

        VERBOSE(1) { PRINT_DEBUG << "B_p.size() = " << B_p.size() << endl; }

        eventsTemp.insert(iSubEvent);

        VERBOSE(1) { PRINT_DEBUG << "eventsTemp.size() = " << eventsTemp.size() << endl; }

```

```
        // Line 24
        B_occu.insert(make_pair(intBprime, eventsTemp));

        VERBOSE(1) { PRINT_DEBUG << "B_occu.size() = " << B_occu.size() << endl; }

        for (EVENTSET::iterator i = B_occu[intB].begin(); i != B_occu[intB].end(); i++)
        {
            VERBOSE(1) { PRINT_DEBUG << "B_occu[intB = " << intB << "]: " << m_InvSubEventsMap[(*i)] << endl; }
        }

        for (EVENTSET::iterator i = B_occu[intBprime].begin(); i != B_occu[intBprime].end(); i++)
        {
            VERBOSE(1) { PRINT_DEBUG << "B_occu[intBprime = " << intBprime << "]: " << m_InvSubEventsMap[(*i)] << endl; }
        }
    // Line 26
    }

    // Line 27
    return 0;
}

int CLowSub::CheckTimedControllability(bdd & bddTCBad, const bdd & bddreach)
{
    bdd bddZhib = bddfalse;

    int iTick = (SearchSubEvent(sTick) - 1) / 2;

    for (int i = 0; i < (m_usiMaxCon + 1) / 2; i++)
    {
```

```
        if (iTick == i) continue;

        bddZhib |= bdd_exist(m_pbdd_ConTrans[i], m_pbdd_ConVarPrim[i]);
    }

    bddTCBad = bdd_exist(m_pbdd_ConTrans[iTick], m_pbdd_ConVarPrim[iTick]) & bddZhib & bddreach;

    if (bddfalse != bddTCBad)
    {
        return -3;
    }

    bddTCBad = bdd_exist(m_pbdd_ConPlantTrans[iTick], m_pbdd_ConPhysicVarPrim[iTick])
        & (!bdd_exist(m_pbdd_ConSupTrans[iTick], m_pbdd_ConSupVarPrim[iTick]))
        & (!bddZhib) & bddreach;

    if (bddfalse != bddTCBad)
    {
        return -2;
    }

    return 0;
}


int CLowSub::CheckTimedControllability(const EVENTSET & eventsDis, const EVENTSET & eventsPoss)
{
    //Uncontrollable events
    cout << "CLowSub::CheckTimedControllability() : FOR-LOOP START : eventsDis" << endl;
    for (EVENTSET::iterator i = eventsDis.begin(); i !=
```

```
eventsDis.end(); i++)
    {
        if (0 == (*i) % 2)
        {
            cout << "CLowSub::CheckTimedControllability() : Uncontrollable event found in eventsDis : " << m_InvSubEventsMap[(*i)] << endl;
            return -1;
        }
    }
    cout << "CLowSub::CheckTimedControllability() : FOR-LOOP END   : eventsDis" << endl;

    int iSubTick = SearchSubEvent(sTick);

    // Prohibitable events intersect with Poss events
    bool bool_Poss_and_Hib = false;

    cout << "CLowSub::CheckTimedControllability() : FOR-LOOP START : eventsPoss" << endl;
    for (EVENTSET::iterator i = eventsPoss.begin(); i != eventsPoss.end(); i++)
    {
        if (iSubTick == (*i)) continue;
        if (1 == (*i) % 2)
        {
            cout << "CLowSub::CheckTimedControllability() : Prohibitable event found in eventsPoss : " << m_InvSubEventsMap[(*i)] << endl;
            bool_Poss_and_Hib = true;
        }
    }
    cout << "CLowSub::CheckTimedControllability() : FOR-LOOP END   : eventsPoss" << endl;

    if (!bool_Poss_and_Hib && (eventsDis.end() != eventsDis.find(iSubTick)))
```

```
    {
        return -2;
    }

    if (bool_Poss_and_Hib && (eventsPoss.end() != eventsPoss.find(iSubTick)))
    {
        return -3;
    }

    return 0;
}

bool CLowSub::RecheckNerodeCells(bdd & bddNCBad, const bdd & bddreach, list< list<bdd> > & list_NerFail)
{
    // Line 1
    if (list_NerFail.empty())
    {
        // Line 2
        return true;
    // Line 3
    }

    // Line 4
    list< pair<bdd, bdd> > listVisited;

    // Line 5
    for (list< list<bdd> >::iterator i = list_NerFail.begin(); i != list_NerFail.end(); i++)
    {
        // Line 6
        list<bdd> Zeqv = *i;

        // Line 7
```

```
        if (!RecheckNerodeCell(bddNCBad, bddreach, Zeqv, listVisited))
        {
            if (bddfalse == bddNCBad)
            {
                for (list<bdd>::iterator j = Zeqv.begin(); j != Zeqv.end(); j++)
                {
                    bddNCBad |= *j;
                }
            }
            // Line 8;
            return false;
        // Line 9
        }
    // Line 10
    }

    // Line 11
    return true;
}

bool CLowSub::RecheckNerodeCell(bdd & bddNCBad, const bdd & bddreach, const list<bdd> & Zeqv, list< pair<bdd, bdd> > & listVisited)
{
    const char * DEBUG = "CLowSub::RecheckNerodeCell():";

    // Line 1
    list<bdd>::const_iterator z1 = Zeqv.begin();

    if (Zeqv.end() == z1)
    {
        return true;
    }

    // Line 2
```

```
    list < pair<bdd, bdd> > listPending;

    list<bdd>::const_iterator z2 = Zeqv.begin();
    z2++;

    // Line 3, 4
    while(Zeqv.end() != z2)
    {
        // Line 5
        listPending.push_back(make_pair(*z1, *z2));
        z2++;
    // Line 6
    }

    // Line 7
    while (!listPending.empty())
    {
        // Line 8
        list< pair<bdd, bdd> >::iterator itr_Pending = listPending.begin();
        bdd bddz1 = itr_Pending->first;
        bdd bddz2 = itr_Pending->second;
        listPending.erase(itr_Pending);

        // Line 9
        bdd bddP = bddz1 | bddz2;

        // Line 10
        if ((bddfalse != (bddP & m_bddMarking)) && (bddP != (bddP & m_bddMarking)))
        {
            bddNCBad = bddP;
            VERBOSE(1) { PRINT_DEBUG << "Neither all states in Zeqv are marked nor non of them are marked." << endl; }

```

```
            // Line 11
            return false;

        // Line 12
        }

        // Line 13
        for (EVENTSET::iterator itr_event = m_SubPlantEvents.begin(); itr_event != m_SubPlantEvents.end(); itr_event++)
        {
            int event, iSubEvent = *itr_event;
            VERBOSE(1) { PRINT_DEBUG << "iSubEvent : " << m_InvSubEventsMap[iSubEvent] << " (index: " << iSubEvent << ")" << endl; }

            if (iSubEvent < 1)
            {
                VERBOSE(1) { PRINT_DEBUG << "ERROR - Found a Sub-level event index lower than 1" << endl; }
                return HISC_INTERNAL_ERR_SUBEVENT;
            }

            bdd bddPprime = bddfalse;
            bdd bddz1prime = bddfalse;
            bdd bddz2prime = bddfalse;

            bdd bddTemp = bddfalse;

            if (1 == iSubEvent % 2) //Controllable
            {
                event = (iSubEvent - 1) / 2;

                bddTemp = bdd_relprod(m_pbdd_ConTrans[event], bddP, m_pbdd_ConVar[event]);
                bddPprime |= bdd_replace(bddTemp, m_pPair_ConPrim[event]);
```

```
            bddTemp = bdd_relprod(m_pbdd_ConTrans[event], bddz1,
m_pbdd_ConVar[event]);
            bddz1prime |= bdd_replace(bddTemp,
m_pPair_ConPrim[event]);
            bddTemp = bdd_relprod(m_pbdd_ConTrans[event], bddz2,
m_pbdd_ConVar[event]);
            bddz2prime |= bdd_replace(bddTemp,
m_pPair_ConPrim[event]);
        }
        else //Uncontrollable
        {
            event = (iSubEvent / 2) - 1;
            bddTemp = bdd_relprod(m_pbdd_UnConTrans[event], bddP,
m_pbdd_UnConVar[event]);
            bddPprime |= bdd_replace(bddTemp,
m_pPair_UnConPrim[event]);
            bddTemp = bdd_relprod(m_pbdd_UnConTrans[event], bddz1,
m_pbdd_UnConVar[event]);
            bddz1prime |= bdd_replace(bddTemp,
m_pPair_UnConPrim[event]);
            bddTemp = bdd_relprod(m_pbdd_UnConTrans[event], bddz2,
m_pbdd_UnConVar[event]);
            bddz2prime |= bdd_replace(bddTemp,
m_pPair_UnConPrim[event]);
        }
        // Line 14
        bddPprime &= bddreach;
        // Line 15
```

```
        bddz1prime &= bddreach;

        // Line 16
        bddz2prime &= bddreach;

        VERBOSE(2)
        {
            PRINT_DEBUG << "bddPprime : ";
            PrintStateSet2(bddPprime);
            cout << endl;
            PRINT_DEBUG << "bddz1prime : ";
            PrintStateSet2(bddz1prime);
            cout << endl;
            PRINT_DEBUG << "bddz2prime : ";
            PrintStateSet2(bddz2prime);
            cout << endl;
        }

        // Line 17
        if (bddfalse != bddPprime)
        {
            // Line 18
            if ((bddfalse != (bddz1prime & bddPprime)) && (bddfalse != (bddz2prime & bddPprime)))
            {
                // Line 19
                if (bddz1prime != bddz2prime)
                {
                    // Need to manually search for the pair, since bdd::operator< returns bdd
                    // instead of bool, which makes all STL containers with ability to search
                    // malfunctional.
                    bool found = false;
                    for (list< pair<bdd, bdd> >::iterator itr =
```

```
listVisited.begin();
                    itr != listVisited.end(); itr++)
                {
                    if ((itr->first == bddz1prime) &&
(itr->second == bddz2prime))
                    {
                        found = true;
                    }
                }

                if (!found)
                {
                    // Line 20
                    listVisited.push_back(make_pair(bddz1prime,
bddz2prime));
                    // Line 21
                    listVisited.push_back(make_pair(bddz2prime,
bddz1prime));
                    // Line 22
                    listPending.push_back(make_pair(bddz1prime,
bddz2prime));
                }
            // Line 23
            }
        }
        // Line 24
        else
        {
            bddNCBad = bddP;
            // Line 25
            return false;
        // Line 26
        }
    // Line 27
    }
```

```
            // Line 28
            }
        // Line 29
        }

        // Line 30
        return true;
}

int CLowSub::CheckSDiv(bdd & bddSDivBad, const bdd & bddReach)
{
        int iTick = (SearchSubEvent(sTick) - 1) / 2;

        // Line 1: Get all states entered by non-tick event from a reachable state.
        bdd bddTemp = bddfalse;

        for (int i = 0; i < (m_usiMaxCon + 1) / 2; i++)
        {
            if (iTick == i) continue;

            bddTemp |= bdd_replace(
                                bdd_exist(m_pbdd_ConTrans[i] & bddReach, m_pbdd_ConVar[i]),
                            m_pPair_ConPrim[i]);
        }

        for (int i = 0; i < m_usiMaxUnCon / 2; i++)
        {
            bddTemp = bdd_replace(
                                bdd_exist(m_pbdd_UnConTrans[i] & bddReach, m_pbdd_UnConVar[i]),
                            m_pPair_UnConPrim[i]);
        }

```

```
    // Line 2 - 4: Each reachable marking states must not be reached by a non-tick event from a reachable state.
    bddSDivBad = (m_bddMarking & bddReach) & bddTemp;

    return 0;
}

EVENTSET CLowSub::GetTransitionEvents(const bdd & bddleave, const bdd & bddenter)
{
    EVENTSET events;
    events.clear();

    if ((bddleave == bddfalse) || (bddenter == bddfalse))
    {
        cout << "CLowSub::GetTransitionEvents() : bddleave is empty or bddfalse is empty." << endl;
        return events;
    }

    //Controllable events
    for (int i = 0; i < (m_usiMaxCon + 1) / 2; i++)
    {
        bdd bddtrans = bddleave & bdd_replace(bddenter, m_pPair_Con[i]);
        if ((bddtrans & m_pbdd_ConTrans[i]) != bddfalse)
        {
            events.insert((i * 2) + 1);
        }
    }

    //Uncontrollable events
    for (int i = 0; i < m_usiMaxUnCon / 2; i++)
    {
        bdd bddtrans = bddleave & bdd_replace(bddenter,
```

```
m_pPair_UnCon[i]);
        if ((bddtrans & m_pbdd_UnConTrans[i]) != bddfalse)
        {
            events.insert((i + 1) * 2);
        }
    }

    return events;
}
```