Implementation of Sampled-data Supervisory Control

By
Abubaker Hamid, M.Sc

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

Master of Science
Department of Computing and Software
McMaster University

© Copyright by Abubaker Hamid, June 17, 2014

MASTER OF SCIENCE(2014)
(Computer Science)

McMaster University
Hamilton, Ontario

TITLE: Implementation of Sampled-data Supervisory Control

AUTHOR: Abubaker Hamid, M.Sc(Sudan University of Science and Technology)

SUPERVISOR: Prof. Ryan J. Leduc

NUMBER OF PAGES: i, 183

Dedication

To my son, Abdul Rahman Abubaker Mahmoud.

Abstract

This thesis focuses on the issues related to the implementation of theoretical timed discrete-event systems (TDES) supervisors. In particular, we examine issues related to implementing TDES as sampled-data (SD) controllers, which were introduced by Wang and Leduc. [18], [21], and [22]. 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 state, and updates its outputs.

We first introduce the sampled-data setting from Wang, and then define the sampled-data properties he identified, including the SD controllability property. We then introduce Wang's formal representation of an SD controller as a Moore synchronous finite state machine (FSM). We then discuss Wang's modular and centralized translation method.

We next introduced new modular results for the SD controllability point 3.1, SD controllability point 3.2, SD controllability point 4, activity loop free and S-singular prohibitable behaviour that allow one to verify the properties using only a portion of the system, instead of having to construct the entire system model. This should allow faster verification times as well as allow larger systems to be verified. As a part of this work, we broke down into individual algorithms one of the large algorithms from Wang that checked multiple properties at once. We can now check each property individually. We then introduce for the first time algorithms to verify Wang's CS Deterministic and non self-loop ALF properties.

The remainder of the thesis focuses on developing algorithms and software to automatically convert a TDES first into an FSM, and then into a VERILOG module. VERILOG is a hardware description language which allows our FSM to be compiled and implemented on digital logic devices such as an FPGA.

We then tested our method by modelling a simple door locking system as TDES, checking that the system satisfies the required sampled-data properties, and then

translating the result into VERILOG. The above algorithms and methods have all been implemented as a part of the graphical DES research tool, DESpot.

Acknowledgements

Thank you God, in the beginning and in the end, for giving me the power to continue and finish this work. I would like to thank Professor Ryan J. Leduc, who patiently guided me through the dark corridors of the discrete-events systems area, enlightening these corridors and providing me with enough resources for the research. I would like to thank Yu Wang for the clarity and organization of his thesis, about the areas that needed to be extended and implemented.

Notation and Abbreviations

In this thesis we will use the following notations and abbreviations

DES: Discrete event systems.

TDES: Timed discrete event systems.

SD: Sampled Data.

FSM: Finite State Machine.

VERILOG: IEEE: 1364-2005 - Hardware definition language.

ALF: Activity Loop Free.

NSL ALF: Non-self-loop Activity Loop Free.

SPB: Singular prohibitable behaviour.

SSPB: S-Singular prohibitable behaviour.

CS Deterministic: Concurrent string deterministic.

PTB: Proper time behaviour.

Contents

Dedication **iii**

Abstract **v**

Acknowledgements **vii**

Notation and Abbreviations **ix**

Contents **xi**

List of Figures **xv**

1 Introduction **1**
- 1.1 Previous Work 1
- 1.2 Objectives 3

2 Introduction to Discrete-Event Systems **5**
- 2.1 Linguistic and Algebraic Preliminaries 5
  - 2.1.1 Alphabets and Strings 5
  - 2.1.2 Languages 6
  - 2.1.3 $\lambda - equivalence$ Relation 8
- 2.2 Timed Discrete Event Systems 8
  - 2.2.1 TDES 8
  - 2.2.2 Synchronization and Product TDES 12
  - 2.2.3 TDES Properties 14

3 Sampled-Data Controllers **19**
- 3.1 Introduction to SD Controllers 19
- 3.2 SD Controller's Input 21
- 3.3 SD Controllable Languages 25

4 Moore Finite State Machines (FSM) and VERILOG **31**
4.1 Moore FSM . . . 31
4.2 Formal Model . . . 31
4.3 Translation Method Introduction . . . 33
4.4 Event Mapping Functions . . . 37
4.5 Centralized Translation Method . . . 38
4.6 Output Equivalence . . . 39
4.7 Modular Translation Method . . . 40
4.8 FSM as VERILOG Module . . . 42
4.9 Field-Programmable Gate Array . . . 43

5 Modular Verification of Sampled-Data Properties **45**
5.1 Introduction to Previous Work . . . 45
5.2 ALF Modularity . . . 46
5.3 SPB and SSPB Modularity . . . 47
5.3.1 SPB implies SSPB . . . 47
5.3.2 SPB Modularity . . . 48
5.4 SD Controllability Modularity . . . 50
5.4.1 SD-Cont-iii.1 Modularity . . . 50
5.4.2 SD-Cont-iii.2 Modularity . . . 55
5.4.3 SD-Cont-iv Modularity . . . 55

6 Algorithmic Improvements **59**
6.1 Introduction . . . 59
6.2 Predicate Verification Preliminaries . . . 59
6.2.1 State Predicates . . . 60
6.2.2 Predicate Transformers . . . 61
6.3 Previous Approach . . . 62
6.4 Modular Verification Algorithm . . . 67
6.5 Modular Version of VerifySub . . . 78
6.6 System ALF Algorithm . . . 80
6.7 Non-selfloop ALF Algorithm . . . 83
6.8 SD Controllability, SSPB and CS Deterministic Algorithms . . . 84
6.8.1 CheckSDCont_SSPB_CSDet Algorithm . . . 86
6.8.2 AnalyseSampledState Algorithm . . . 88

7 VERILOG Translation **95**
7.1 Introduction . . . 95
7.2 TDES to FSM . . . 95
7.3 FSM to VERILOG . . . 100

7.4 Centeral FSM for SD Controllers 104
7.5 Central Module for SD Controllers 106
7.6 Translation Algorithms 109
7.6.1 writeFSM() Algorithm 115
7.6.2 writeHFSM() Algorithm 117
7.6.3 mainVERILOG() Algorithm 121
7.6.4 writeVERILOG() Algorithm 126
7.6.5 mainFSM() Algorithm 135
7.7 Removing Redundant Transitions 137

8 Lock System Example 139
8.1 Problem Description 139
8.1.1 System Components 141
8.2 Components Design 143
8.2.1 Signal Outputs and Transitions 146
8.2.2 Supervisor SupOpen 148
8.2.3 Supervisor SupChange 152
8.2.4 Supervisor SupAlarm 159
8.2.5 The Main Controller FSM 169
8.3 Performance Results 172
8.4 Other Projects Performance Results 174
8.4.1 Flexible Manufacturing System 174
8.4.2 Test Station of Manufacturing System 175

9 Conclusion 177
9.1 Conclusion 177
9.2 Future Work 179

Bibliography 181

List of Figures

2.1 ADAM Language Automata 7
2.2 An Example TDES 11
2.3 An Example Failing ALF Property, from Wang [21] 15
2.4 An Example Failing the Proper Time Behavior Property, from Wang [21] 17

3.1 Sampling Data Global *tick* 20
3.2 Nonminimal Example 24
3.3 An Example for **Point ii**, from Leduc et al [18] 27
3.4 An Example for **Point iii.1**, from Leduc et al [18] 29
3.5 An Example for **Point iii.2 and Point iv**, from Leduc et al [18] . . 30

4.1 FSM Translation Example 33
4.2 General Structure of FPGA, [9]. 44

7.1 Supervisor **SupOpen** 97
7.2 FSM of Supervisor **SupOpen** 98
7.3 FSMCarrier Class UML Diagram 111
7.4 Transition Struct UML [1] Diagram 112

8.1 Lock System Black Box Diagram 140
8.2 Lock System Block Diagram 142
8.3 FSM Detailed Sub-Modules 143
8.4 Lock System Plants Components 146
8.5 Supervisor **SupOpen** 148
8.6 FSM for supervisor **SupOpen** 149
8.7 Supervisor **SupChange** 153
8.8 FSM for supervisor **SupChange** 154
8.9 Supervisor **SupAlarm**. 160
8.10 FSM for Supervisor **SupAlarm** 161

Chapter 1

Introduction

1.1 Previous Work

Since ancient times such as 300 BC in Greece, man has been fond of controlling natural aspects around him. He started this control spree by inventing the clock (Time monitoring machine) with combined effort between Arabs and Greeks. The spree started by creating machines to track aspects around man, then continued to cover instruments for tracking the moon and other planet's motion, the direction of air flow, the prediction of seasons and most importantly, the compass.

Systems control theory had been there for a long time in the beginning of the steam engine era (18th century). However, implementing these theories in a practical methodology had only taken serious shape in the last few decades, with the introduction of a vast quantity of hardware controllers such as PLC [23], and the sophistication of Hardware Description Languages (HDL) such as VHDL [10], VERILOG [9].

The area of discrete event systems (DES) had been a target for serious theoretical studies, creating a base for a linguistic approach, that helps in analysing and applying properties, such as nonblocking, controllability. These serious studies can be found in [16] and [26]. The linguistic approach is inspired by looking at systems as languages and model them in terms of sentences and words.

A solid approach to implement supervisors as Sampled-data controllers was led by Leduc and Wang [18] [21] [22]. They introduced the concept of Sampled-data controllers as well as new properties such as SD controllability to make sure the behaviour of the controllers was closer to the theoretical model. This method focuses on timed DES [8] and [7]. We will be extending Wang's work.

An earlier attempt to implement untimed supervisors is the the implementation of Leduc in [14] and [15] of an finite state machine (FSM) version named Clocked Moore Synchronous State Machine (CMSSM) on a programmable logic controller (PLC). A test bed system was implemented which contains 29 DES supervisors. The system's synchronous product was order of 10^{16} states, worst case estimate. For additional examples of early works, see [18].

As for the modular verification we implemented in this thesis, it was inspired by the modular incremental verification for controllability by Malik et al [5]. This was a new approach, to verify systems and do synthesis, based on subsystem verification and combined use of counterexamples and heuristics to identify suitable subsystems incrementally. We also build upon the work by Wang [21] and Baloch [3].

There were plenty of good efforts in implementing system controls in variety of ways. Julien Provost, Jean-Marc Roussel and Jean-Marc Faure [13] had implemented a system that evaluates that the specifications of a system is testable, then implemented this using a Mealy FSM and PLC.

Aside from systems control, OMNET++ [20] is a huge work done in C++. OMNET++ is an open source library for C++ development that helps a lot in the area of discrete event systems' simulation. It has module building blocks for the system and uses messaging to enable the sub modules to communicate with each other. It is very useful in simulation, particularly network simulation and telecommunication systems simulation.

1.2 Objectives

Our objective in this work has two focuses. The first is to enhance the performance of the current SD controllability verification automation in DESpot, by enhancing the algorithms implemented so far. We do this by either splitting algorithms that are combined in function such as SSPB, SD controllability and CS deterministic, or by enhancing the code of the BDD-based algorithms themselves.

We also have to build new algorithms for the properties such as non-self loop activity free loop (NSL ALF), CS deterministic and translation algorithms that translates supervisors first into intermediate Moore FSM modules, and then then into VERILOG modules.

Our second focus is the implementation of sampled data supervisors. All the properties and theories of sampled data supervisory control is useless if not converted into a practical hardware or software implementation. In this work, we will translate traditional TDES supervisors into real VERILOG modules that will be programmed into digital hardware such as FPGA, to create concrete implementation of our supervisors. Please see Chapter 4 for more information about FPGA.

We are continuing the work done by Leduc et al [18] in developing properties that must be passed by a TDES project to be a good fit of a project ready to be implemented in hardware or software.

Chapter 2 in this thesis gives an introduction about the linguistic approach to system modelling. We also introduce TDES and its characteristics that enable us to produce SD controllers. TDES systems have the ability to be driven by a global central clock represented by the tick event that orchestrate the overall system to work in a synchronous manner.

Chapter 3 is about Sampled Data controllers; the concept and overview about how we are planning to implement them.

Chapter 4 is about the model of finite state machines (FSM) introduced by Edward F. Moore in the 1950(s). Worth mentioning here that most of the hardware description languages such as VHDL, and VERILOG include the ability to model the hardware in building blocks, where each block is an FSM.

Chapter 5 is about introducing new algorithms to the verification process. We do this by proposing new propositions and proving them. We then implement them as algorithms to be used in our software program, DESpot [12]. In this chapter, we introduce our main addition to the area by introducing modular verification concepts. Modular verification is an important contribution to this work as it allows one to avoid checking properties using the synchronous product of the entire system, and instead to check these properties on individual components. If the individual components all pass, then the synchronous product of the system also passes.

Chapter 6 describes our effort in enhancing the algorithms from Wang [21] as well as our newly built ones.

Chapter 7 is about the algorithms built to translate our supervisors, and consequently the TDES project, into a hardware module (VERILOG). The reason we split this chapter from the previous one is that it has many algorithms that are all focused on producing FSM machines in XML format, and VERILOG files.

Chapter 8 presents a lock system that we modelled as TDES. We verified that it passes all the properties we developed so far to assure its validity as a hardware SD controller. We used this example to show the files we generated and explain the translation process.

Chapter 9 contains our conclusions, and future work discussion.

Chapter 2

Introduction to Discrete-Event Systems

Discrete-Event Systems are systems that are driven by events, and these systems include but not limited to manufacturing systems, traffic lights systems, database management systems, security systems and production line systems [25].

2.1 Linguistic and Algebraic Preliminaries

In this thesis we consider discrete-event systems (DES) from a linguistic approach. We look at the events in the system as symbols from an alphabet, is a sequence of events as words from a language.

2.1.1 Alphabets and Strings

An alphabet is a collection of symbols we refer as Σ. An example of an alphabet is $\Sigma = \{\sigma, \beta, \gamma\}$. A string is an arbitrary non-empty sequence constructed from an alphabet. Example strings constructed from the previous mentioned alphabet are as follows: $\alpha\sigma\sigma, \gamma\beta\beta\sigma$ ...etc. We refer to the set of finite, non empty strings that can be constructed from this alphabet as Σ^+. The empty string ϵ is not in this set : $\epsilon \notin \Sigma$.

When we add the empty string to Σ^+ we get the set Σ^*.

Definition 2.1.1. The *length* of a string is a unary function that takes $s \in \Sigma^*$ and returns the number of symbols this string contains.

$$|\,| : \Sigma^* \to \mathbb{N}$$

.

Let $s = \sigma_1\sigma_2...\sigma_n, \in \Sigma^+$ such that $\sigma_i \in \Sigma$, for $i = \{1, 2..., n\}$. We thus have $|s| = n$

The empty string ϵ is the string that does not have any symbols: $|\epsilon| = 0$.

Definition 2.1.2. The *concatenation* function is a binary function that takes two strings s_1, s_2 from Σ^* and returns the string s_1s_2.

$$cat : \Sigma^* \times \Sigma^* \to \Sigma^*$$

Let $s_1 = \sigma_1\sigma_2...\sigma_m$ with $\sigma_i \in \Sigma$ and $i = \{1, 2..., m\}$ and $s_2 = \beta_1\beta_2...\beta_n$ with $\beta_j \in \Sigma$ and $j = \{1, 2..., n\}$. Then $cat(s_1, s_2) = s_1s_2 = \sigma_1\sigma_2...\sigma_m\beta_1\beta_2...\beta_n$, and $|s_1s_2| = m + n$

It follows from the above definitions that for any $s \in \Sigma^*$ $s = \epsilon s = s\epsilon$.

Definition 2.1.3. Let $s, t \in \Sigma^*$. We say s is a *prefix* of t, denoted as $s \leq t$, if $(\exists u \in \Sigma^*) su = t$

It is then obvious that any string is a prefix to itself, $s \leq s$ since $s\epsilon = s$, and $\epsilon \in \Sigma^*$.

Definition 2.1.4. For alphabet Σ_1 and alphabet Σ_2 , we define the *set subtraction* operation as $\Sigma_1 - \Sigma_2 := \{\sigma | \sigma \in \Sigma_1 \wedge \sigma \notin \Sigma_2\}$.

2.1.2 Languages

A language $L \subseteq \Sigma^*$ is any subset of Σ^*. We can represent a language using the concept of an Automata. For example the ADAM language from Figure 2.1 will be represented by the following tuple automata $ADAM = (Y, \Sigma, \eta, y_0, Y_m)$. We have y_0 is the initial state, Y is the set of all states in the automata, while $Y_m \subseteq Y$ are the marked

states. In this ADAM automata we have only one marked state that is also the initial state. The function $\eta : Y \times \Sigma* \rightarrow Y$ is the transition function and is a partial function.

We also partition Σ into controllable events, Σ_c, and uncontrollable events Σ_u. Controllable events are events that an external agent can disable and prevent from occurring. Uncontrollable events can not be disabled.

Definition 2.1.5. The *prefix closure* of a language, denoted as $\overline{L}$, is the set of all strings in Σ^* that can be continued to construct a string in L.

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

.

Since it is clear from the previous section that any string is a prefix of itself, then the prefix closure contains the language itself. ie $L \subseteq \overline{L}$.

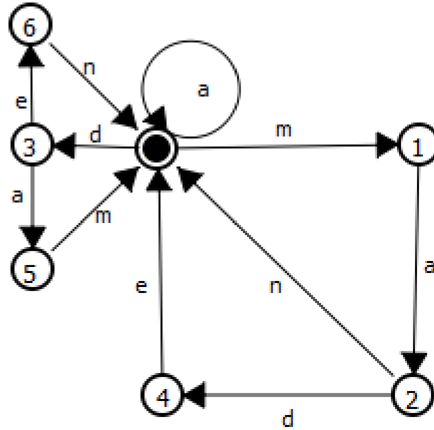


Figure 2.1: ADAM Language Automata

Definition 2.1.6. For a Language $L \subseteq \Sigma^*$ we define the *Nerode equivalence relation mod L* as:

$$(\forall s, t \in \Sigma^*) s \equiv_L t \ or \ s \equiv t (mod \ L) \iff [(\forall u \in \Sigma^*) su \in L \iff tu \in L]$$

2.1.3 $\lambda - equivalence$ Relation

Let X be a nonempty set, and let $E \subseteq X \times X$ be a binary relation on X.

Definition 2.1.7. The relation E is an *equivalence relation* on X if:

- $(\forall x \in X)xEx$ (E is reflexive)
- $(\forall x, x' \in X)xEx' \implies x'Ex$ (E is symmetric)
- $(\forall x, x', x'' \in X)xEx' \& x'Ex'' \implies xEx''$ (E is transitive).

For xEx' we may also write $x \equiv x'$ *(mod E)*.

We now introduce the $\lambda - equivalence$ relation.

Definition 2.1.8. 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 the closed loop behavior, namely $L(\mathbf{G})$. They also have the same future with respect to the marked behaviour, namely $L_m(\mathbf{G})$.

2.2 Timed Discrete Event Systems

The common way is to introduce untimed DES, then introduce timed discrete-event system (TDES) as a special case of DES. But since this thesis is all about TDES and modelling the TDES in a way to be easy to turn it into an SD controller, we will start with TDES.

2.2.1 TDES

A TDES has a *tick* event in the set of events, Σ, and all the other events are considered activity events (Σ_{act}). This makes $\Sigma = \Sigma_{act} \dot{\cup} \{tick\}$. We sometimes use τ to

stand for the tick when brevity is needed. The occurrence of the *tick* event correspond to the tick of global clock.

We also have the concept of *forcible events*, $\Sigma_{for} \subseteq \Sigma_{act}$. These are events we can ensure occur before the next *tick*. In TDES forcible events can be controllable or uncontrollable. However, for sampled-data systems we require that all forcible events be also controllable. This simplifies our definitions while still providing a flexible modelling approach.

We model forcing by having an external agent disables the *tick* event when a forcible event is required to occur before the *tick*. The agent is not preventing the *tick* from occurring, but instead removing the path of behaviour where a tick occurs before the event to be be forced, so that the TDES behaviour matches what will actually physically occur. Of course there must be a forcible event possible to occur, and preempt the *tick* event. We capture this requirement in the TDES controllability definition in the Section 2.2.19.

We also introduce the concept of *prohibitable events*, $\Sigma_{hib} \subseteq \Sigma_{act}$. These are activity events that are controllable. We thus have $\Sigma_c = \{\tau\} \cup \Sigma_{hib}$. In the sampled-data setting, we thus have $\Sigma_{hib} = \Sigma_{for}$.

We usually model a TDES formally as a generator $\mathbf{G}$, which is a five tuple defined as the following:

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

where

Q is the finite *state set* of the TDES.

Σ is the finite set of distinct symbols representing event labels. This set is partitioned into two sets:

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

$\delta : Q \times \Sigma \to Q$ 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. The transition function can be extended 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*.

Example 2.1. *Let* $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ *be the TDES shown in Figure 2.2. The convention we are following in this example and most of the examples in this thesis is that the initial state is the one that has two circles one outer and one inner. Marked states are shown as a solid circle. A controllable event is written in plain text, an uncontrollable event is written in italic and with a leading exclamation mark.*

$Q = \{0, 1, 2, ..., 7\}$;

$\Sigma = \Sigma_c \mathbin{\dot\cup} \Sigma_u$, *where* $\Sigma_c = \{tick, open\}$ *and* $\Sigma_u = \{enter, equal\}$;

$\delta = \{(0, tick, 0), (0, enter, 1), (0, equal, 2), (1, equal, 3), (1, tick, 0),$
$(2, tick, 0), (2, enter, 3), (3, tick, 4), (4, equal, 4), (4, open, 5), (4, enter, 7), (5, equal, 5),$
$(5, tick, 4), (5, equal, 5), (5, enter, 6), (6, equal, 6), (6, tick, 0), (7, equal, 7), (7, open, 6)\}$;

$q_0 = 0$; $Q_m = \{0\}$

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

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

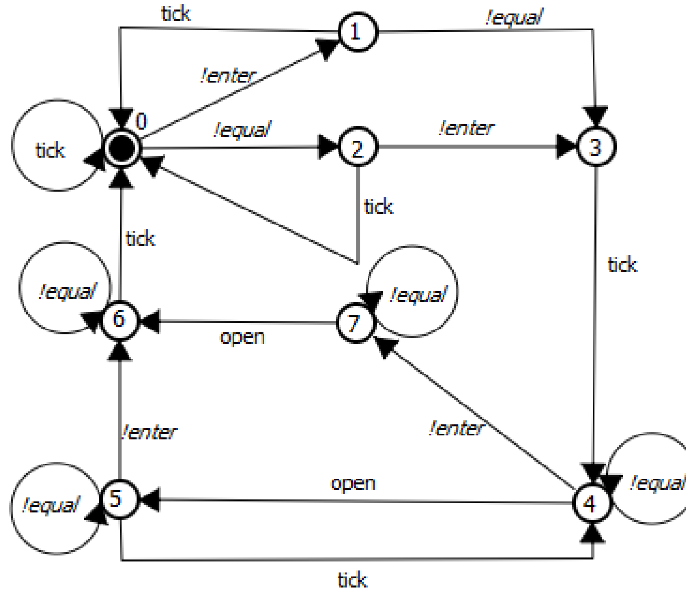


Figure 2.2: An Example TDES

In this thesis we will always assume that a TDES is reachable unless stated otherwise.

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

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

Definition 2.2.4. The *marked behavior* of TDES $\mathbf{G}$ 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 TDES $\mathbf{G}$ is defined to be a mapping $\zeta : Q \to \mathrm{Pwr}(\Sigma_{hib})$ that takes q and returns a set of prohibitable events enabled at q.

Example 2.2. *Example for control action from Figure 2.2 is* $\zeta(0) = \{\}, \zeta(3) = \{\}, \zeta(4) = \{open\}$.

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

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

Definition 2.2.7. TDES $\mathbf{G}$ is said to be *minimal*, if

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

TDES $\mathbf{G}$ is minimal if it does not have two distinct states in Q that are λ equivalent.

2.2.2 Synchronization and Product TDES

It is always recommended to model the system plant as individual components instead of one large DES. Each component describes a small activity in the system (e.g: if a person is sensed by the camera, the door opens). We then create a synchronous product of all these small TDES components to be the plant. We now define the *natural projection* function and its inverse function:

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a TDES. Take $\Sigma_o \subseteq \Sigma$ to be the set of *observable events.*

Definition 2.2.8. The *natural projection* $P : \Sigma^* \to \Sigma_o^*$ is defined as follows. For $s \in \Sigma^*$, $\sigma \in \Sigma$,

$$P(\epsilon) = \epsilon$$

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

$$P(s\sigma) = P(s)P(\sigma)$$

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

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

Definition 2.2.9. For a given set X, we define $Pwr(X)$ to be the set of all subsets of X:

$$Pwr(X) := \{X'|X' \in X\}$$

Example 2.4. *Let X be a set such that: $X = \{a, b, c\}$. We then have $Pwr(X) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.*

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 $H \subseteq \Sigma_o^*$

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

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

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

In this thesis when we speak about the synchronous product of two TDES automata, we always assume that both of them are over the same set of events, Σ.

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 TDES defined over event set Σ. The *product* of two TDES 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 *synchronous product of TDES* $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o_i}, Q_{m_i})$ $(i = 1, 2)$, denoted $\mathbf{G}_1||\mathbf{G}_2$, is defined to be a reachable TDES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ 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 2.2.13. Let $\mathbf{G}$ be a TDES 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 a TDES $\mathbf{G}'_i$ $(i = 1, 2)$ defined over event set Σ_i, the synchronous product operator of these two TDES components is created by first extending each TDES to be over Σ by adding self-loops of events that are not in the local TDES alphabet Σ_i, and then using the product operator. We take $\Sigma = \Sigma_1 \cup \Sigma_2$, and $\mathbf{G}_i = \mathbf{selfloop}(\mathbf{G}'_i, \Sigma - \Sigma_i)$, i=1,2. We thus have $G'_1||G'_2 = G_1 \times G_2$

2.2.3 TDES Properties

We now introduce several TDES properties that we will need in this thesis. Plant completeness in TDES is given by the following definition:

Definition 2.2.14. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a plant and TDES $\mathbf{S} = (Y, \Sigma, \zeta, y_0, Y_m)$ be a supervisor. $\mathbf{G}$ is *TDES complete* for $\mathbf{S}$ if

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

The above definition is the same as the untimed DES version from [2] but the concept of controllable events now has been changed to the concept of prohibitable events. Plant completeness says that in any state of the synchronous product of the

plant and the supervisor, if a prohibitable event σ is eligible in the supervisor then it must be eligible in the plant as well.

Now we would like to introduce the definition of *activity loop free*, which requires that a tick in a TDES will not be preempted indefinitely by activity events, as this situation is not realistic.

Definition 2.2.15. TDES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ has an *activity loop* if

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

Definition 2.2.16. TDES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ is *activity loop free* (ALF) if

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

where Q_r is the set of reachable states for G. We do not require the unreachable states to be ALF. Instead we only consider the reachable states, because the unreachable states do not contribute anything to the closed and marked behavior of the TDES. We see an example that violates the ALF property in Figure 2.3.

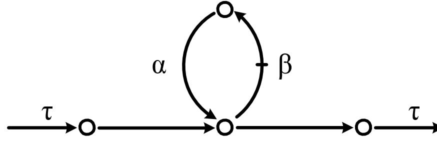


Figure 2.3: An Example Failing ALF Property, from Wang [21]

Another important thing is that the supervisors by their nature contains many self loops, and this will definitely violate the *activity loop free* property. Therefore, we will only require that the synchronous product of the entire system be ALF and not each TDES component.

Because supervisors often employ self loops, we will introduce a new condition on supervisors that they must be *non-self loop activity loop free*. This definition will be used later as a design guide to make it easier to translate TDES supervisors into SD controllers.

The non-self loop activity loop free concept is given by the following definition:

Definition 2.2.17. Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a TDES, and let $\mathbf{G}'$ be $\mathbf{G}$ with all activity event self loops removed. $\mathbf{G}$ is *non-selfloop activity loop free* iff $\mathbf{G}'$ is ALF.

This means that if we remove all the self activity events loops in the TDES, we must end up with a TDES that is ALF.

The proposition below was introduced by Wang [21]. It states that if two TDES automata are ALF then the synchronous product is also ALF.

Proposition 2.1. [21] 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.

We also have a more fruitful proposition by Wang that says that if we do not have two TDES passing the ALF property, then instead one of them needs to pass the property and the other must not introduce any new events that is not already in the first TDES. In this case, we know that the synchronous product will also be activity loop free. This proposition helps a lot as supervisors typically have activity loops. We will thus only require that the plant TDES components be ALF to assure that the synchronous product is ALF.

Proposition 2.2. [21] 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.

We will now introduce another property that helps us to prevent the unrealistic situation where we reach a state from which there is no path to a tick event. We now state the concept of *proper time behaviour* that was introduce by Kai Wong et al. [24].

Definition 2.2.18. TDES $\mathbf{G}$ has a *proper time behavior* if

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

Proper time behaviour is required to be satisfied by the system plant, and it says that at all states either a tick or an uncontrollable event is eligible. The reason is as follows: assume that a plant has a finite state space and at one of its states we have only a controllable event eligible, and no tick or uncontrollable event eligible. Now

assume that supervisor disables this controllable event. This means the system stops forever, with no ticks and no more uncontrollable event possible. This can happen even if the supervisor is controllable. Therefore, this property, namely proper time behaviour, will ensure that the system will not reach this dead state. An example of a plant that violates the proper time behaviour is shown in Figure 2.4. In the example, after the first tick we only have β eligible which is a controllable event, and thus it fails the PTB definition.

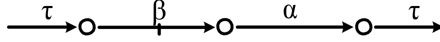


Figure 2.4: An Example Failing the Proper Time Behavior Property, from Wang [21]

Proposition 2.3 was proposed by Wang in [21]. It says that if the TDES G has a finite state space, is activity loop free and has proper time behaviour, then at any reachable state we can do a tick event after at most a finite number of activity events.

Proposition 2.3. [21] If a TDES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ has a finite state space, is activity loop free and has proper time behaviour, then

$$(\forall q \in Q_r)(\exists s \in \Sigma^*)\ \delta(q, s\tau)!$$

where Q_r is the set of reachable states.

TDES controllability is given in the following definition. In this thesis when we say a TDES is controllable we mean TDES controllability, as opposed to the untimed definition.

Definition 2.2.19. We define the arbitrary language $K \subseteq L(\mathbf{G})$ to be *TDES controllable* with respect to $\mathbf{G}$ if,

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

In other words definition 2.2.19 states that $\overline{K}$ must have an uncontrollable event eligible if it is eligible in the plant, and it must also have a tick event eligible if the tick event is eligible in the plant, and there is no forcible events eligible to preempt the tick event from occurring.

Chapter 3

Sampled-Data Controllers

In this chapter we will review the Sampled-Data systems and how we translate a TDES automaton to an SD controller (Sampled-Data Controller). This material is taken from Wang [21] and Leduc et al [18].

3.1 Introduction to SD Controllers

Sampled-data systems are systems that are driven by a global clock. As shown in Figure 3.1, a controller samples input on the rising edge of the clock (clock signal rises from zero to one), and internally only stores if a given input was true or false (high or low) at the clock edge.

The behaviour of the controller is to sample its inputs on the clock edge and then change state based on the current input values and its current state. It then sets its outputs to be true or false based on the current state. If an output is set to true, the controller will cause the correspondent event to occur before the next clock edge.

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

the clock edge that drives the SD controller with the occurrence of the TDES *tick* event.

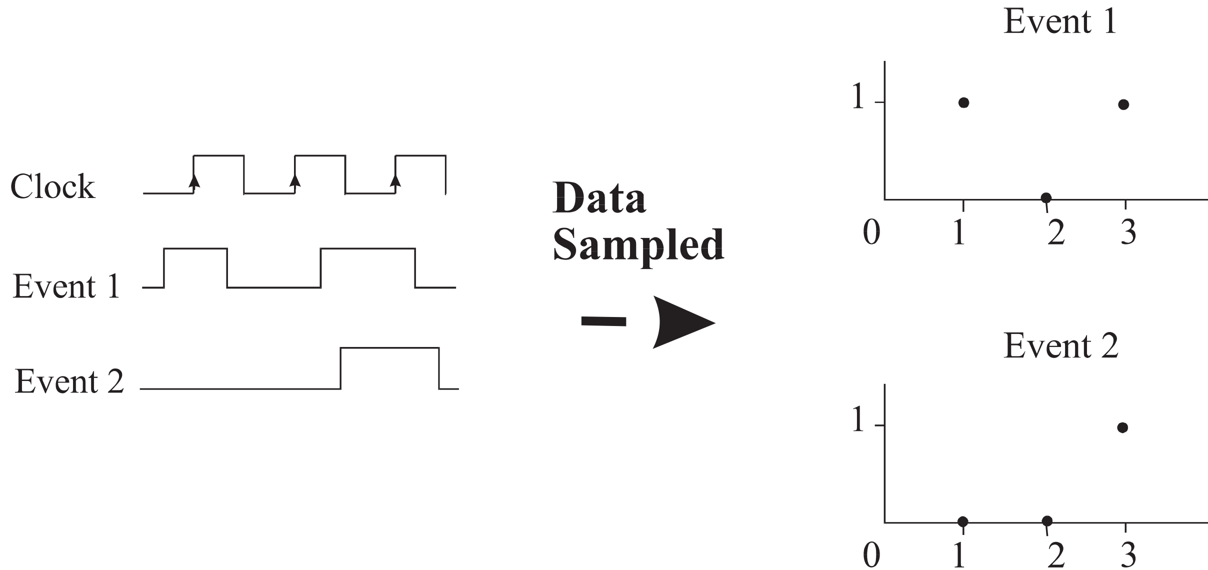


Figure 3.1: Sampling Data Global *tick*

We will assume the following facts while modelling SD controllers:

- We assume that the set of prohibitable events equals the set of forcible events i.e $\Sigma_{hib} = \Sigma_{for}$.
- We will consider that our SD controllers all are driven by the same global clock. This means they all sample inputs and change state at the same time, and they update their outputs at the same time.
- Prohibitable events will be allowed to occur at most once per sampling period.
- We will consider an event to have occurred if its input went *true* since the last clock edge. If an event occurs so close to the clock edge that it is missed and is seen in the next clock period, then it will not be considered to have happen in this sampling period, but in the next one.

- When we enable a prohibitable event in the sampling period, we force it to occur before the next clock edge.
- The input signal length for an event must be designed carefully. It must be short enough that it begins and ends before the rising edge of the clock (namely shorter than the clock period), but not too long that it spans more than one clock period, when the event actually only occurred once.

3.2 SD Controller's Input

To manipulate the TDES theory to work properly with the SD controllers theory we will consider that each TDES system has a shared event in its events set called *tick* (τ). This is the global clock event that indicates to all SD controllers to start sampling and processing its input. This means that an SD controller can observe only those strings that ends with a *tick* event, and the empty string. We consider the empty string as a sampled string because it represents the initial state of the controller which is known. Therefore, ϵ is a legitimate sampled string.

Definition 3.2.1. Given a event set Σ, the set of *sampled strings* is denoted by L_{samp} and is define as

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

The SD controller changes its state at a clock edge (*tick*). Its next state is determined by the strings that occur containing a single *tick* at the end since the last clock tick. We refer to these strings as *concurrent strings*. An SD controller starts at its initial state (reset) and then changes state according to the concurrent strings that occur.

Definition 3.2.2. Given an event set Σ, we denote the set of *concurrent strings* as L_{conc}, defined as

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

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

When a concurrent string occurs, the SD controller can not tell the order that the events in the string happened in. It also can not tell the difference whether the event occurred once or many times in the sampling period. Therefore, the following concurrent strings appear the same to an SD controller: $\alpha\beta\beta\beta\tau$, $\beta\alpha\tau$, and $\alpha\beta\alpha\beta\tau$. It means if two concurrent strings have the same *occurrence image* (set of events that the string contains) the SD controller will see them as the same *concurrent string*. We capture this concept with the following operator.

Definition 3.2.3. For $s \in \Sigma^*$, the *occurrence operator* is a function Occu : $\Sigma^* \rightarrow \mathrm{Pwr}(\Sigma)$ defined as below:

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

As the SD controllers only see strings that end with a *tick* (sampled strings), therefore only states in the SD controller entered by a *tick* are considered observable points in the SD controllers. We refer to theses states as sampled states.

Definition 3.2.4. A state $x \in X$ from TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, is a *sampled state* for $\mathbf{S}$ if

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

We call $X_{samp} \subseteq X$ the set of sampled states for $\mathbf{S}$. Since $\epsilon \in L_{samp}$, then by definition $x_o \in X_{samp}$. Namely the initial state in a TDES (SD controller) is always observable at least when the system is started up.

To convert TDES $\mathbf{S}$ into SD controller $\mathbf{C}$, we take the sampled states of $\mathbf{S}$ as the states of $\mathbf{C}$. The initial or reset state state of $\mathbf{C}$ would be the initial state of $\mathbf{S}$. We then determine which concurrent strings are possible from a given sampled state. The occurrence image of these concurrent strings would then define our next state conditions. i.e. the sampled state in $\mathbf{S}$ that the concurrent string takes us to.

For state x of $\mathbf{C}$, an output (enablement of some $\sigma \in \Sigma_{hib}$) is set to be true if the corresponding event is possible at state x in $\mathbf{S}$. We notice that outputs (enablement

information) are constant for the clock period. For a formal definition of SD controllers and the conversion process from a TDES, please see Section 4.2 of Chapter 4.

Now we see that a problem arises here: what if in the TDES **S** we have one sampled state x where two possible concurrent strings are defined, but these two concurrent strings have the same occurrence image, say $\alpha\beta\tau$ and $\beta\beta\alpha\tau$. Let one concurrent string lead to state x_i and the other one lead to x_j. We then face a situation with the possibilities:

- x_i, x_j are the same state: this is the desired situation and does not cause a problem.
- x_i, x_j are not the same state: since the concurrent strings appear the same to the controller, it will not know which state to choose.

Now we present a property from [18] and [21] that ensures the above problem does not happen. We call this property CS deterministic.

Definition 3.2.5. A TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ is *concurrent string deterministic* or *CS deterministic*, if

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

To see how a non-minimal TDES would cause problems, consider Figure 3.2.

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 3.2, part (a) shows a portion of **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 merge into a single state x. Clearly in (a), $\text{Occu}(s') \cap \Sigma_{act} = \text{Occu}(s'') \cap \Sigma_{act}$ but $\xi(x_n, s') \neq \xi(x_n, s'')$. This means the translation function of the SD controller will be non-deterministic. 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 we would

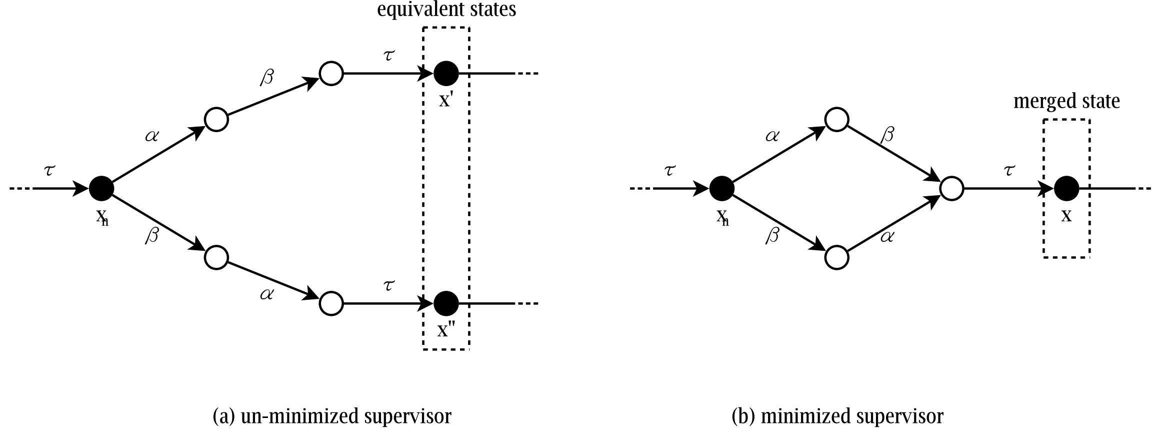


Figure 3.2: Nonminimal Example

not be currently able to convert this TDES into an SD controller.

To aid in defining the conversion from a TDES to an SD controller, we define the next sampling state function, it will be used in later chapters.

Definition 3.2.6. For CS deterministic TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, we define the *next-sampling-state* partial function: $\Delta : X_{samp} \times \mathrm{Pwr}(\Sigma_{act}) \to X_{samp}$ as follows. For $x \in X_{samp} \subseteq X$ and $\Sigma' \subseteq \Sigma_{act}$,

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

We want to introduce the *prohibited action* function, as defined in [18], that we will used later in our translation from TDES to SD controller, *prohibited action* function in short is the enablement that is done by the supervisor at any sampled states.

Definition 3.2.7. For TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ the *prohibited action* function: $\zeta : X_{samp} \to \mathrm{Pwr}(\Sigma_{hib})$ is defined for $x \in X_{samp} \subseteq X$ as follows:

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

We will assume that each prohibitable event occurs at most once per *concurrent string* because SD controllers can not tell how many times the event occurred.

Therefore, we will assume that our plant should reflect this concept by the following definition from [18] and [21].

Definition 3.2.8. (Old version) For TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, we say that $\mathbf{G}$ has *singular prohibitable behavior* if,

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

We now define the S-singular prohibitable behaviour.

Definition 3.2.9. [22] For plant $\mathbf{G}$ and supervisor $\mathbf{S}$, we say that G has *S-singular prohibitable behaviour* if:

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

3.3 SD Controllable Languages

Up to now we have required that our TDES system must satisfy the following properties: have a finite state space, be ALF and nonblocking, our plant must have proper time behaviour and be complete for our supervisor, and our supervisor be controllable for our plant. However, these conditions are not sufficient to translate a TDES supervisor into an SD controller and get the behaviour we expect. Our actual system behaviour under the control of the corresponding SD controller could block, violate our control law, or even exhibit behaviour not contained in our plant model. Therefore, we now need to also use the SD controllability property from Wang in [21] and [18].

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. Remember that we require that for a TDES system, $\Sigma_c = \Sigma_{hib} \cup \{\tau\}$ and that $\Sigma_{for} = \Sigma_{hib}$. As we will see later, this new condition implies our TDES controllability definition so that we do not have to test this condition separately.

Definition 3.3.1. 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) $\text{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq \text{Elig}_{L(\mathbf{S})}(s)$

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

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

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

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

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

Point i : This is the normal untimed controllability.

Point ii : The ($\Leftarrow$) direction and **point i** imply standard TDES controllability. The ($\Rightarrow$) part says if we enable a prohibitable event, we must disable *tick*. This captures notion that we only enable a prohibitable event when we want to force it. This makes it easier to translate supervisor **S** into an SD controller.

Point iii : When s is a sampled string then the following two conditions must be fulfilled:

iii.1) This condition states that prohibitable events that are enabled when the system changes states must remain enabled until the next sampling *tick* without addition or subtraction.

iii.2) This condition is saying that in any sampling period, if two concurrent strings are eligible, and these two concurrent strings have the same occurrence image, then they must have the same future with respect to the system's closed behaviour and marked behaviour.

Point iv : This point says that marked strings must be sampled strings (ϵ or end in a *tick*).

Let us now discuss some examples that illustrate some of the SD controllability properties to understand the above conditions. We explore these conditions without **Point i**, as it is the standard untimed controllability. The examples below are taken from [18].

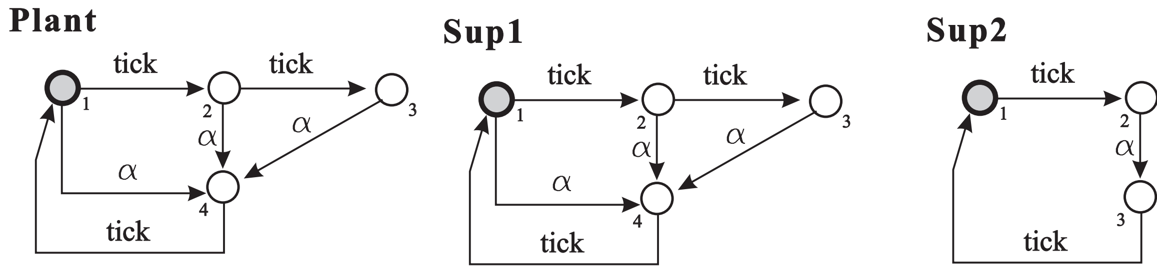


Figure 3.3: An Example for **Point ii**, from Leduc et al [18]

In Figure 3.3, supervisor **Sup1** does not satisfy ($\Longrightarrow$) of **Point ii** at states 1 and 2 because at these states we have both a *tick* and a prohibitable event enabled at the same state. This makes **Sup1** difficult to convert to an SD controller as we do not know whether we should force event α at state 1, 2, or state 3. Supervisor **Sup2** satisfies ($\Longrightarrow$) of **Point ii**. **Sup2** is essentially **Sup1** with the event α forced at state 2 and the unused behaviour removed.

In the example automata shown in Figure 3.4, we have a cannon set to fire through an open door. We note that supervisor **Sup1** is controllable and nonblocking with **Plant1**, but it fails **Point iii.1** at state 3. This is because prohibitable event *fire* was possible at state 2 immediately after the tick, but is not possible at state 3 although

it has not yet occurred. Essentially, **Sup1** says: fire the cannon in this clock cycle but only if the door has not yet closed. Because we have no information telling us when the cannon will fire and when the door will close during the clock cycle, it is possible they could occur close together meaning there is no way we can ensure that the cannon will not fire after the door closes, violating our control law. Worse, the SD controller will not even know the door has closed until the next tick, at which time it will be too late to react. It is important to note that although **Sup1** is controllable and nonblocking with **Plant1**, our implementation may not perform correctly.

Supervisor **Sup2** is also controllable and nonblocking with **Plant1**, but it fails **Point iii.1** at state 3. The problem is that prohibitable event *fire* was not possible at the start of the sampling period (namely state 2), but it is enabled and forced at state 3. In essence, supervisor **Sup2** says to wait until the door closes and then the cannon must be fired before the next *tick*. However, the SD controller will not know that the door has closed until the next *tick*. At this time it is too late to meet the required deadline, violating our control law.

Supervisor **Sup3** is controllable and nonblocking with **Plant1**, and it satisfies **Point iii.1**. It shows that if we are to ensure a prohibitable event occurring in a specific order relative to another event, then the two events must occur in two separate clock periods.

Finally, considering the combination of **Plant2** and **Sup1**. They are also controllable and nonblocking, but they fail **Point iii.1** at state 3. However, this time the plant model is specifying that prohibitable event *fire* cannot occur at state 3. Obviously the event could occur and our plant model is invalid and will give the designer a false idea of security.

In the example shown in Figure 3.5, supervisor **Sup1** is controllable and nonblocking with **Plant**, but it fails **Point iii.2** at states 4 and 7. This is because concurrent strings $\alpha\beta tick$ and $\beta\alpha tick$ have the same occurrence image, but they are not Nerode equivalent with respect to the closed-loop systems closed behaviour. Particularly, we see that **Sup1** enables ω after $\alpha\beta tock$, but enables γ after $\beta\alpha tick$. Because an SD

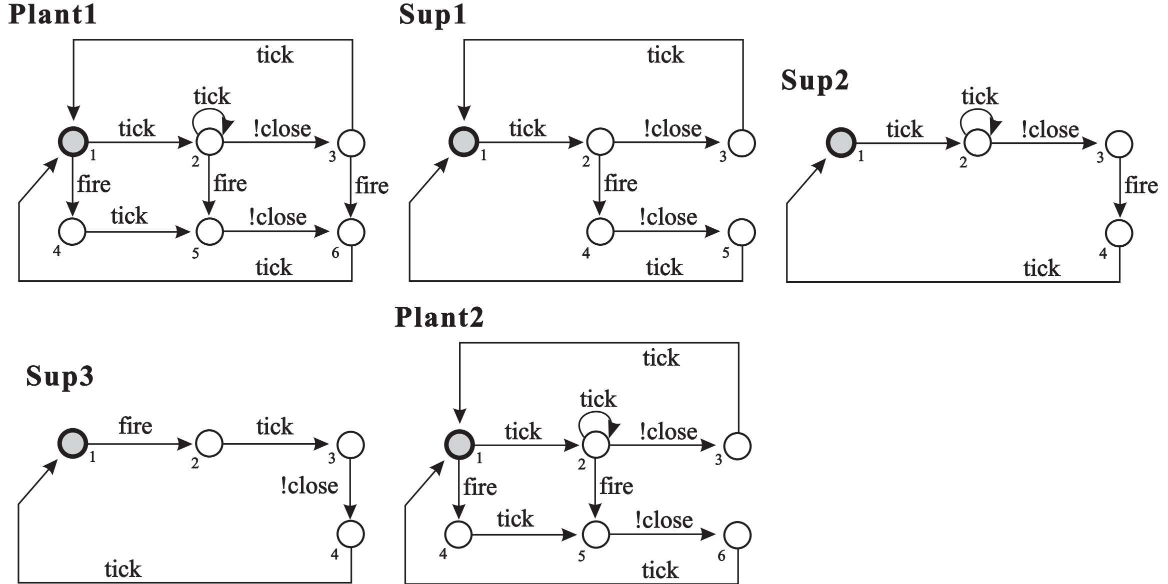


Figure 3.4: An Example for **Point iii.1**, from Leduc et al [18]

controller will not be able to tell the difference between these two strings, it will not know which action to take. **Sup1** can thus not be implemented as an SD controller.

We also note supervisor **Sup2** is controllable and nonblocking with **Plant**, but it fails **Point iii.2** at states 4 and 7. This is because concurrent string $\beta\alpha tick$ can be extended to a marked string by adding an ω while string $\alpha\beta tick$ can not.

Take into consideration that in the physical system we might only get string $\alpha\beta tick$ and never $\beta\alpha tick$ or vice versa. This might be due to a choice during the implementation of our controller or the fact that in our specific controller implementation, event α always takes less time to complete than event β. In situations like this, our physical system would block despite the fact our TDES model is nonblocking. If we moved the marker state of **Sup2** from state 8 to state 1, the system would now pass **Point iii.2** and our physical system would be nonblocking even if only one of strings $\alpha\beta tick$ and $\beta\alpha tick$ was possible.

We next note that in Figure 3.5, despite the fact that **Sup3** is controllable, non-

blocking with **Plant** and passes **Point iii.2**, its implementation would still block if only $\alpha\beta tick$ could occur. We note though that **Sup3** fails **Point iv** at state 5 as string β is marked and is not a sampled string. Moving the marking from state 5 to state 4 would solve the problem.

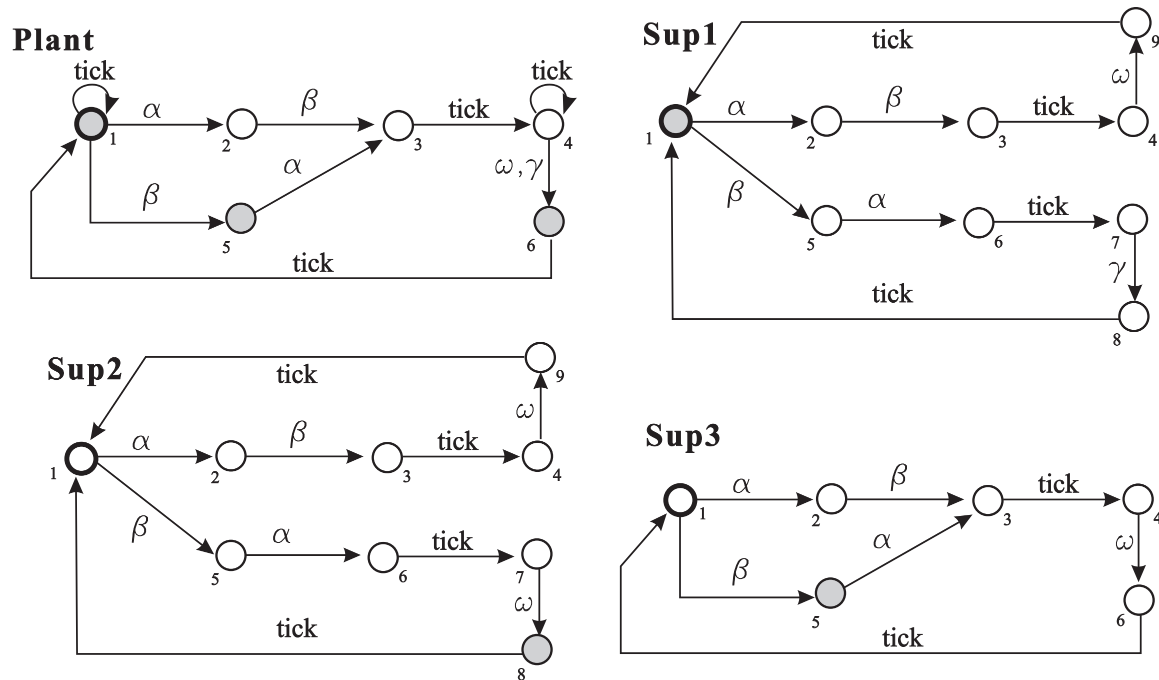


Figure 3.5: An Example for **Point iii.2 and Point iv**, from Leduc et al [18]

Chapter 4

Moore Finite State Machines (FSM) and VERILOG

The material in Sections 4.1-4.7 is largely taken from [18] and [22], with permission.

4.1 Moore FSM

In this chapter, we introduce a formal representation for SD controllers and a translation method for TDES supervisors. We will model SD controllers as Moore synchronous FSM. Please see [9]. Our choice to do so is based on the work of Leduc in [14] and [15] who implemented, in an *ad hoc* fashion, untimed DES as FSM. Using FSM to define SD controllers is a good choice as it provides a concrete definition of the controller, yet an FSM still allows a variety of physical implementations such as a PLC program as in Bolton [6], using digital logic [9], or as a software program on a computer.

4.2 Formal Model

Before we give a formal definition of an SD controller, we first need to discuss some notation. We will often be discussing Boolean vectors that change periodically with respect to some clock. A Boolean vector is a vector whose individual elements can only be assigned the values of *true (1)* or *false (0)*. We say "at time k" to indicate the

point of time at which k clock ticks have gone by since our starting reference point at $k = 0$. For any vector $\mathbf{v} = [v_1, v_2, ..., v_n]$, we write "$\mathbf{v}(k)$" and "$v_j(k)$" to denote the value of $\mathbf{v}$ and v_j ($j \in \{1, \ldots, n\}$) at time k. As our index k takes on new values, our vector $\mathbf{v}$ defines a sequence with respect to ticks of our clock, which we define to be $\{\mathbf{v}(k)|k = 0, 1, ...\}$, and is denoted as $\{\mathbf{v}(k)\}$. 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 or reset. For TDES systems, a "clock tick" corresponds to the occurrence of a *tick* event.

Definition 4.2.1. We define an *SD controller* $\mathbf{C}$ as the tuple

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

where:

- I is the set of possible Boolean vectors that the inputs to our controller can take on. Each vector $\mathbf{i} = [i_0, i_1, .., i_{v-1}] \in I$ has v input variables. Each element of $\mathbf{i}$ corresponds to a unique activity event in our system. When an element is set to 1, this indicates that the corresponding event has occurred at least once in the previous clock period, when an element is set to 0 this means it did not occur.

- Z is the set of possible Boolean vectors that the controller outputs can take on. Each vector $\mathbf{z} = [z_0, z_1, .., z_{r-1}] \in Z$ has r output variables. Each element of $\mathbf{z}$ corresponds to a unique prohibitable event in our system. When an element is set to 1, this means that corresponding event is enabled and that the controller will make the event occur before the next clock tick (force it), where 0 means it is disabled. Note that for the controller, both enabling and forcing a prohibitable event are indicated by setting the corresponding element of $\mathbf{z}$ to 1. Also, forcing and enablement of an event is constant for the controller for the given sampling period.

- Q is the set of possible Boolean vectors that the state of the controller can take on. Each vector $\mathbf{q} = [q_0, q_1, .., q_{l-1}] \in Q$ has l state variables.

- $\mathbf{q}_{res} \in Q$ is the initial (*reset*) state for the controller when it starts operating or is reset. We take $q(0) = q_{res}$.

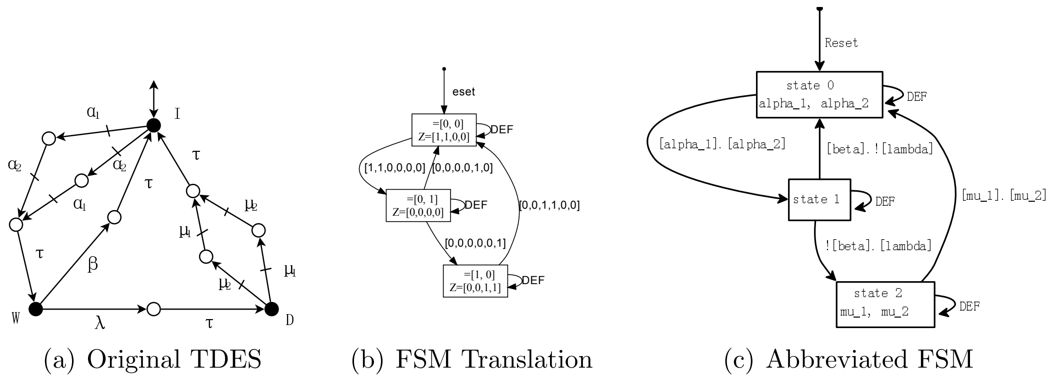
- $\Omega : Q \times I \to Q$ is the next-state function which takes the current state $\mathbf{q}(k) \in Q$ and an input vector $\mathbf{i}(k+1) \in I$, and returns the next state $\mathbf{q}(k+1) = \Omega(q(k)\mathbf{i}(k+1))$.

- $\Phi : Q \to Z$ is the state-to-output map.

For a specific run of our controller, we would receive a specific sequence of inputs $\{\mathbf{i}(k)\}$, starting at time $k = 0$. This sequence, combined with $\mathbf{q}_{res}$, and Ω, will uniquely define our current sequence of states, $\{\mathbf{q}(k)\}$. In turn, $\{\mathbf{q}(k)\}$ and Φ will uniquely define our current sequence of outputs $\{\mathbf{z}(k)\}$. If we wish to distinguish between two vector sequences, we will use different variables such as $\{\mathbf{i}(k)\}$ and $\{\mathbf{i}(k')\}$.

4.3 Translation Method Introduction

We will now present an informal translation method using the supervisor shown in Figure 4.1(a) to illustrate the translation steps.



(a) Original TDES (b) FSM Translation (c) Abbreviated FSM

Figure 4.1: FSM Translation Example

1. To convert TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ into SD controller $\mathbf{C}$, our first step is to define the input and output sets, I, and Z. As they represent all possible assign-

ments of Boolean vectors, we first need to define the size (number of elements/-variables) of each vector. The size of each input vector $\mathbf{i} \in I$ is defined to be $v = |\Sigma_{act}|$. The size of each output vector $\mathbf{z} \in Z$ is defined to be $r = |\Sigma_{hib}|$. The next step is to identify each element of vectors $\mathbf{i} = [i_0, i_1, .., i_{v-1}] \in I$ to a unique activity event, and to identify each element of vectors $\mathbf{z} = [z_0, z_1, .., z_{r-1}] \in Z$ to a unique prohibitable event.

For the example in Figure 4.1(a), we have $\Sigma_{act} = \{\alpha_1, \alpha_2, \mu_1, \mu_2, \beta, \lambda\}$, and $\Sigma_{hib} = \{\alpha_1, \alpha_2, \mu_1, \mu_2\}$. This gives $v = 6$ and $r = 4$. We identify the elements $i_0, i_1, .., i_5$ with $\alpha_1, \alpha_2, \mu_1, \mu_2, \beta, \lambda$ and the elements $z_0, z_1, .., z_3$ with $\alpha_1, \alpha_2, \mu_1, \mu_2$. For example, input vector $[1, 1, 0, 0, 0, 0] \in I$ means that only events α_1, and α_2 occurred since the last clock tick. Also, output vector $[0, 0, 1, 1] \in Z$ means that only prohibitable events μ_1, and μ_2 are enabled in the current clock period.

2. The next step we define the controller's state set, Q. We need to define the size of the state vectors, l, so that a state vector is large enough to encode a value for each sampled state $x \in X_{samp} \subseteq X$. We thus choose l to satisfy $2^{l-1} < |X_{samp}| \leq 2^l$. We then identify each sampled state $x \in X_{samp}$ of our supervisor with a unique state $\mathbf{q} \in Q$. We note that if $|X_{samp}| < 2^l$, then Q will contain some vector assignments that do not correspond to sampled states, but they will be unreachable. Next, we define the controller's initial state, $\mathbf{q}_{res}$, to be equal to the state $\mathbf{q} \in Q$ that we associated with the supervisor's initial state, x_o. Finally, we can define the *state set mapping* function, $\Lambda : X_{samp} \rightarrow Q$, which has $\Lambda(x_o) = \mathbf{q}_{res}$. For simplicity, we will typically assume that Q has no unused states and that Λ is bijective. If Q does have unused states, we can restrict Λ to the set of controller states we have associated with sampled states, and we get our desired bijective function.

We see that the sampled states of the supervisor in Figure 4.1(a) are states **I** (initial state), **W** and **D** (only other states with incoming *tick* event). As $2^1 < 3 \leq 2^2$, we take $l = 2$. We then equate the states of our SD controller (FSM) in Figure 4.1(b) as $\mathbf{q}_{res} = [0, 0]$ to **I**, $[0, 1]$ to **W** and $[1, 0]$ to **D**. We see

that the remaining fourth state, $[1,1] \in I$, is unused.

3. We next define our state-to-output map, Φ. For a given sampled state $x \in X_{samp}$ of our supervisor, let $\Sigma_{enb} := \{\sigma \in \Sigma_{hib} | \, \xi(x,\sigma)!\}$. If state $\mathbf{q} \in Q$ is the controller state that corresponds to state x, we would then define $\Phi(q) = \mathbf{z} = [z_0, z_1, .., z_{r-1}]$ such that each element z_i $(i = 0, \ldots, r-1)$ would be set to zero unless $\sigma_i \in \Sigma_{hib}$, where σ_i the the corresponding prohibitable event for z_i, in which case z_i would be set to one.

   In the Figure 4.1(a), the only prohibitable events defined at sampled state **I** are α_1, and α_2. We thus set $\Phi([0,0]) = [1,1,0,0]$. The remaining values of Φ for states $[0,1]$ and $[1,0]$ can be seen in Figure 4.1(b). We can set $\Phi([1,1])$ arbitrarily as state $[1,1]$ is unused and unreachable.

4. Finally, we define our next-state function, Ω. For each sampled state $x \in X_{samp}$ of our supervisor, let $\mathbf{q} \in Q$ be the unique controller state associated with x (i.e. $\Lambda(x) = \mathbf{q}$). Let $CB(x) := \{s \in L_{conc} | \, \xi(x,s)!\}$ be the set of concurrent strings defined at state x. For each string $s \in CB(x)$, let $\mathbf{i} = [i_0, i_1, .., i_{v-1}] \in I$ be the input vector that corresponds to the occurrence image of s, $\text{Occu}(s)$. This means that each element i_j $(j = 0, \ldots, v-1)$ of $\mathbf{i}$ would be set to zero unless $\sigma_j \in \text{Occu}(s)$, where σ_j the the corresponding activity event for i_j, in which case i_j would be set to one. We would then define $\Omega(\mathbf{q}, \mathbf{i}) = \Lambda(\xi(x,s))$. We note that any remaining combination of $\mathbf{q}' \in Q$ and $\mathbf{i}' \in I$ not defined in the above process represent either unused states in the controller, or combination of concurrent strings that can not occur at the corresponding sampled state. We can thus define $\Omega(\mathbf{q}', \mathbf{i}')$ arbitrarily. If $\mathbf{q}'$ is associated with a sampled state, we typically define $\Omega(\mathbf{q}', \mathbf{i}') = \mathbf{q}'$ to make the design more robust.

   Studying state **I** ($[0,0]$) in Figure 4.1(a), we see that the only concurrent strings leaving it are $\alpha_1\alpha_2\tau$ and $\alpha_2\alpha_1\tau$, which takes us to state **W** ($[0,1]$). As $\text{Occu}(\alpha_1 \alpha_2\tau) = \text{Occu}(\alpha_2\alpha_1\tau) = \{\alpha_2, \alpha_1, \tau\}$, the corresponding input vector is $[1,1,0,0,0,0]$. We would thus define $\Omega([0,0],[1,1,0,0,0,0]) = [0,1]$. The remaining transitions for defined sampled state and concurrent string pairs are shown in

Figure 4.1(b). The transitions leaving unused state $[1, 1]$ are defined arbitrarily. As Ω is a total function, we usually have to add to diagrams a **DEF**, or default transition, to cover input combination that we have not explicitly specified (i.e. it matches all remaining unspecified input combination), as shown in Figure 4.1(b). This is done solely as a shorthand to keep the diagram uncluttered.

We see that our translation method states that enablement and forcing information for a sampling period is completely defined by the sampled state reached by the last *tick* event. Therefore, every prohibitable event enabled at this state will remain enabled for the entire sampling period, and the controller must force every one of them to occur before the next *tick*. This makes it easy to extract this information from the supervisor and ensures that if multiple prohibitable events are available to be forced, there is no uncertainty on which one to generate; they all must occur.

In Figure 4.1(c), we present a more compact and human-readable version of the FSM. At each state in the FSM, we only list prohibitable events whose outputs are set to true. For next-state conditions, we use Boolean equations to express input conditions and we simplify them to only include input events possible at that state.

For example, at state 1 of FSM we see the next-state condition $[\beta]\cdot![\lambda]$ which means event β occurred in the last sampling period, but event λ did not. It will match any input vector for which this is true. Here we are representing logical AND as "$\cdot$," and logical negation as "!." We also represent logical OR as "+." In the FSM, we enclose event names by "[ ]" to distinguish between an event label and its corresponding input being set to 1.

The operation of our FSM is that at system reset, it starts operating at state $\mathbf{q}_{res} = [0, 0]$ with its outputs set to enable only α_1 and α_2. At each clock tick, it samples its inputs creating a new input vector, $\mathbf{i}$, which is matched to the current state's next-state conditions to determine its new state and output. It then changes to the new state, updates its outputs, and then waits for the next clock tick. For an example of how to implement an FSM on a programmable logic controller, see [14] and [15].

To be able to translate a TDES supervisor into an SD controller, we only require that it be CS deterministic to ensure that the resulting FSM will be deterministic. In practice, we also require that the TDES be non-selfloop ALF as a design aid.

This makes it more likely that the TDES be CS deterministic and typically makes the translation easier and more compact as we will see in Chapter 8. As a TDES that fails to be non-selfloop ALF can not model a physical system, this additional constraint is not a limitation.

4.4 Event Mapping Functions

As we will often be discussing vectors whose elements refer to specific events in Σ_{act}, we 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 a CS deterministic TDES supervisor for $\mathbf{G}$ with $\Sigma_{\mathbf{S}} \subseteq \Sigma$. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller for $\mathbf{S}$.

As we could have multiple controllers, each using their own event orderings, we first need define a default bijective map, γ_g, between our activity event set and a default index set that we will use for labeling the events. To simplify things, we will require the event mapping functions for each controller to respect the event ordering imposed by γ_g. This means that γ_g will induce a single way to define the various mapping functions for our controller.

Definition 4.4.1. Let bijective map $\gamma_g : \Sigma_{act} \rightarrow \{0, \ldots, |\Sigma_{act}| - 1\}$ be the *canonical event mapping function* for Σ_{act}.

We now define input and output mapping functions for our controller. Each function will map events to the index value for the corresponding variable in the specified vector (I or Z).

Definition 4.4.2. The *input event mapping function* for $\mathbf{C}$ is defined to be a bijective map $\gamma : \Sigma_{\mathbf{S}} \cap \Sigma_{act} \rightarrow \{0, 1, \ldots, v - 1\}$ with $v = |\Sigma_{\mathbf{S}} \cap \Sigma_{act}|$ 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 4.4.3. The *output event mapping function* for $\mathbf{C}$ is defined to be a bijective map $\eta : \Sigma_{\mathbf{S}} \cap \Sigma_{hib} \rightarrow \{0, 1, .., r - 1\}$ with $r = |\Sigma_{\mathbf{S}} \cap \Sigma_{hib}|$ such that

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

As these functions are bijective, their inverse functions always exist. We can thus use their inverse functions to map an index value to its corresponding activity event.

4.5 Centralized Translation Method

We will now discuss how to translate a TDES supervisor into a single centralized controller. Let TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be CS deterministic. To translate $\mathbf{S}$ into a controller $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$, we need to determine values for the members of the tuple.

We will start with I, Z, and Q. As they represent all possible assignments of Boolean vectors, we first need to define the size (number of elements/variables) of each vector. The size of each input vector $\mathbf{i} \in I$ is defined to be $v = |\Sigma_{act}|$. The size of each output vector $\mathbf{z} \in Z$ is defined to be $r = |\Sigma_{hib}|$.

We now specify Q. We need to define the size of the state vectors, l, so that a state vector is large enough to encode a value for each sampled state $x \in X_{samp} \subseteq X$. We thus choose l to satisfy $2^{l-1} < |X_{samp}| \leq 2^l$. We note that if $|X_{samp}| < 2^l$, then Q will contain some vector assignments that do not correspond to sampled states but they will be unreachable.

To complete our definition of I, Z, and Q, we now need to associate activity events with individual input variables, prohibitable events with individual output variables, and sampled states with state assignments of Q.

Definition 4.5.1. Let γ be the input event mapping function for controller $\mathbf{C}$. We define for $\mathbf{C}$ the *input set mapping* bijective function, $\Gamma_I : Pwr(\Sigma_{act}) \rightarrow I$, as follows. For $\Sigma' \subseteq \Sigma_{act}$, we have $\Gamma_I(\Sigma') = [i_0, i_1, .., i_{v-1}]$ such that for $j = 0, 1, .., v-1$,

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

Definition 4.5.2. Let η be the output event mapping function for controller $\mathbf{C}$. We define for $\mathbf{C}$ the *output set mapping* bijective function, $\Gamma_Z : \mathrm{Pwr}(\Sigma_{hib}) \rightarrow Z$, as follows. For $\Sigma' \subseteq \Sigma_{hib}$, we have $\Gamma_Z(\Sigma') = [z_0, z_1, .., z_{r-1}]$ such that for $j = 0, 1, .., r-1$,

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

Definition 4.5.3. We define for controller $\mathbf{C}$ the *state set mapping* function, $\Lambda : X_{samp} \to Q$, to be an arbitrary injective function of the designer's choice.

We can now define the reset state for $\mathbf{C}$ as $\mathbf{q}_{res} = \Lambda(x_o)$. Next, we define the next-state and state-to-output maps.

Definition 4.5.4. Let Δ be the next-sampling-state partial function for supervisor $\mathbf{S}$. For state $\mathbf{q} \in Q$ and input $\mathbf{i} \in I$, the *next-state* function, Ω, is defined to be

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

Definition 4.5.5. Let ζ be the *prohibited action* function for supervisor $\mathbf{S}$. For state $\mathbf{q} \in Q$, the *state-to-output map* Φ is defined to be

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

We now have completely defined our controller $\mathbf{C}$ for TDES supervisor $\mathbf{S}$. As long as $\mathbf{S}$ is CS deterministic, then its next-sampling-state partial function Δ will be well defined, allowing us to do the translation.

So far we have assumed our supervisor's event set is Σ. If instead our supervisor is defined over a subset $\Sigma_{\mathbf{S}} \subset \Sigma$, we would simply add to every state of our supervisor selfloops for each $\sigma \in \Sigma - \Sigma_{\mathbf{S}}$ and use this new supervisor for the translation.

4.6 Output Equivalence

Before we define a modular translation method, we first need to consider how to determine if two different controllers would produce equivalent output (i.e. enablement information) for the same input sequence. The problem is that the controllers could be defined over different sets of input events and might use different event ordering for their vectors.

To handle different input requirements, we will assume that we have a system input vector, $\mathbf{i}_g$, containing all activity events and ordered with respect to the canonical

event mapping function, γ_g. We call $\{\mathbf{i}_g(k)\}$ a *canonical input sequence* and $\mathbf{i}_g \in \{\mathbf{i}_g(k)\}$ a *canonical input vector*. We can then map each $\mathbf{i}_g$ to a corresponding input vector for each controller using γ_g and the controllers own output event mapping function, γ. For example, we can map $\mathbf{i}_g = [i_{g,0}, i_{g,1}, .., i_{g,v_g-1}]$ $(v_g = |\Sigma_{act}|)$ to $\mathbf{i} = [i_0, i_1, .., i_{v-1}]$ by setting $i_l = i_{g,l'}$ for $l = 0, \ldots, v-1$, with $l' = \gamma_g((\gamma^{-1}(l)))$.

In particular, we are interested in comparing two controllers $\mathbf{C}_1$ and $\mathbf{C}_2$ that have been defined relative to the same CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$. We are thus only interested in determining if they generate the same output with respect to input sequences that represent valid input strings to $\mathbf{S}$ (i.e. $s \in L(\mathbf{S}) \cap L_{samp}$).

Definition 4.6.1. A canonical input sequence $\{i_g(k)\}$ is *input valid* for $\mathbf{S}$, if $(\forall k' \in \{1, 2, ...\})(\exists s_1, s_2, \ldots, s_{k'} \in L_{conc})\ [s_1 s_2 \ldots s_{k'} \in L(\mathbf{S})] \wedge [(\forall n \in \{1, 2, \ldots, k'\})(\forall \sigma \in \Sigma_{act})$

$i_{g,\gamma_g(\sigma)}(n) = 1$ iff $\sigma \in \text{Occu}(s_n)]$

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.

Definition 4.6.2. Let $\mathbf{C}_j = (I_j, Z_j, Q_j, \Omega_j, \Phi_j, \mathbf{q}_{res,j})$ be an SD controller with output mapping function η_j, $j = 1, 2$. 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 outputs $\mathbf{z}_j(k') = [z_{j,1}(k'), z_{j,2}(k'), .., z_{j,r_j}(k')] \in Z_j$ $(j = 1, 2)$ 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. $(\forall k' \in \{0, 1, ..\})\mathbf{z}_1(k') = \mathbf{z}_2(k')$

4.7 Modular Translation Method

For large systems, we typically define multiple modular supervisors rather than a single centralised supervisor. We would like to translate each individual supervisor

into their own modular controller, and then combine their outputs to produce the equivalent enablement information of a centralized controller.

Let TDES $\mathbf{S} = \mathbf{S}_1||\mathbf{S}_2||..||\mathbf{S}_n = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor where each modular supervisor $\mathbf{S}_j = (X_j, \Sigma_j, \xi_j, x_{o,j}, X_{m,j})$, $j = 1, 2, .., n$, is also CS deterministic. Let $\Sigma_j = \Sigma_{act,j} \dot{\cup} \{\tau\}$, $\Sigma_{hib,j} := \Sigma_{hib} \cap \Sigma_{act,j}$, and $\Sigma = \bigcup_{l \in \{1,2,..,n\}} \Sigma_l$.

For the following discussion, we will define the logical AND of vectors $\mathbf{u} = [u_1, u_2, .., u_m]$ and $\mathbf{v} = [v_1, v_2, .., v_m]$ as $\mathbf{u} \wedge \mathbf{v} = [u_1 \wedge v_1, u_2 \wedge v_2, .., u_m \wedge v_m]$. We define the concatenation of vectors $\mathbf{u} = [u_1, u_2, .., u_{m_1}]$ and $\mathbf{v} = [v_1, v_2, .., v_{m_2}]$ as $\mathbf{uv} = [u_1, u_2, .., u_{m_1}, v_1, v_2, .., v_{m_2}]$.

Definition 4.7.1. For $j = 1, 2, .., n$, the jth modular translation of CS deterministic supervisor $\mathbf{S}_j$ to modular controller $\mathbf{C}_j = (I_j, Z_j, Q_j, \Omega_j, \Phi_j, \mathbf{q}_{res,j})$ is performed using the method defined in Section 4.5 after replacing Σ_{act} by $\Sigma_{act,j}$, and Σ_{hib} with $\Sigma_{hib,j}$ in the definitions.

To define how we combine the individual outputs of the modular controllers together, we need to define the composite controller of our n modular controllers.

Definition 4.7.2. The composite controller for $\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_n$, $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res}) = \mathbf{comp}(\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_n)$, is defined as follows:

- The size of each input vector $\mathbf{i} \in I$ is defined to be $v = |\Sigma_{act}|$. The size of each output vector $\mathbf{z} \in Z$ is defined to be $r = |\Sigma_{hib}|$. Let γ to be the input event mapping function for $\mathbf{C}$ and η to be the output event mapping function.

- The number of state variables for vectors $\mathbf{q} \in Q$ is defined to be $l = \sum_{j=1}^{n} l_j$, where l_j is the number of state variables for Q_j. The reset state is defined to be $\mathbf{q}_{res} = \mathbf{q}_{res,1}\mathbf{q}_{res,2} \cdots \mathbf{q}_{res,n}$.

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

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

  where input vector $\mathbf{i}(k+1)$ in canonical form with respect to γ_g, was mapped to input vector $\mathbf{i}_j(k+1)$ $(j = 1, \ldots, n)$ as described in Section 4.6.

- The state-to-output map Φ is defined as follows. For $\mathbf{q} = \mathbf{q}_1\mathbf{q}_2..\mathbf{q}_n \in Q$, let $\mathbf{z}_j = \Phi_j(\mathbf{q}_j) = [z_{j,0}, z_{j,1}, \ldots, z_{j,r_j-1}] \in Z_j$, $j = 1, \ldots, n$. For each $\mathbf{z}_j$, we translate it to $\mathbf{z}'_j = [z'_{j,0}, z'_{j,1}, .., z'_{j,r-1}] \in Z$ such that,

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

  We can now define:

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

The following theorem essentially states that the enablement information of the composite controller is equivalent to that of the centralized controller.

Theorem 4.1. [22] Let CS deterministic supervisors $\mathbf{S} = \mathbf{S}_1||\mathbf{S}_2||\ldots||\mathbf{S}_n$ and $\mathbf{S}_j$ $(j = 1, 2, \ldots, n)$ be defined as above. Let $\mathbf{C}_j$ be the modular controller for $\mathbf{S}_j$, $\mathbf{C}' = \mathbf{comp}(\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}_n)$ be the composite controller, and $\mathbf{C}$ be the centralized controller translated from $\mathbf{S}$; then $\mathbf{C}$ and $\mathbf{C}'$ are output equivalent with respect to $\mathbf{S}$.

4.8 FSM as VERILOG Module

Our implementation is a two step process. The first step is to translate the TDES supervisors to FSM. The second step is to translate the FSM to VERILOG modules. VERILOG is a Hardware Description Language (HDL). See [9] for more details about VERILOG and digital logic design.

In this thesis we focus on implementing our SD controllers as FSM defined as VERILOG modules. This makes a good way to implement the supervisor in hardware using field programmable gate array (FPGA) discussed in the next section. Alternatively, the FSM could have been translated to a programming language such as C++ [4] and implemented in software. This has been left as future work.

4.9 Field-Programmable Gate Array

SD controllers can be implemented in hardware using a field programmable gate array (FPGA) device. An FPGA is a common type of programmable logic device that uses small units of logic called logic blocks. See [9] for more information.

The logic block employs one or more of the following:

- Transistor gates.
- Flip-flops.
- Multiplexers.
- Look up tables (LUT's).

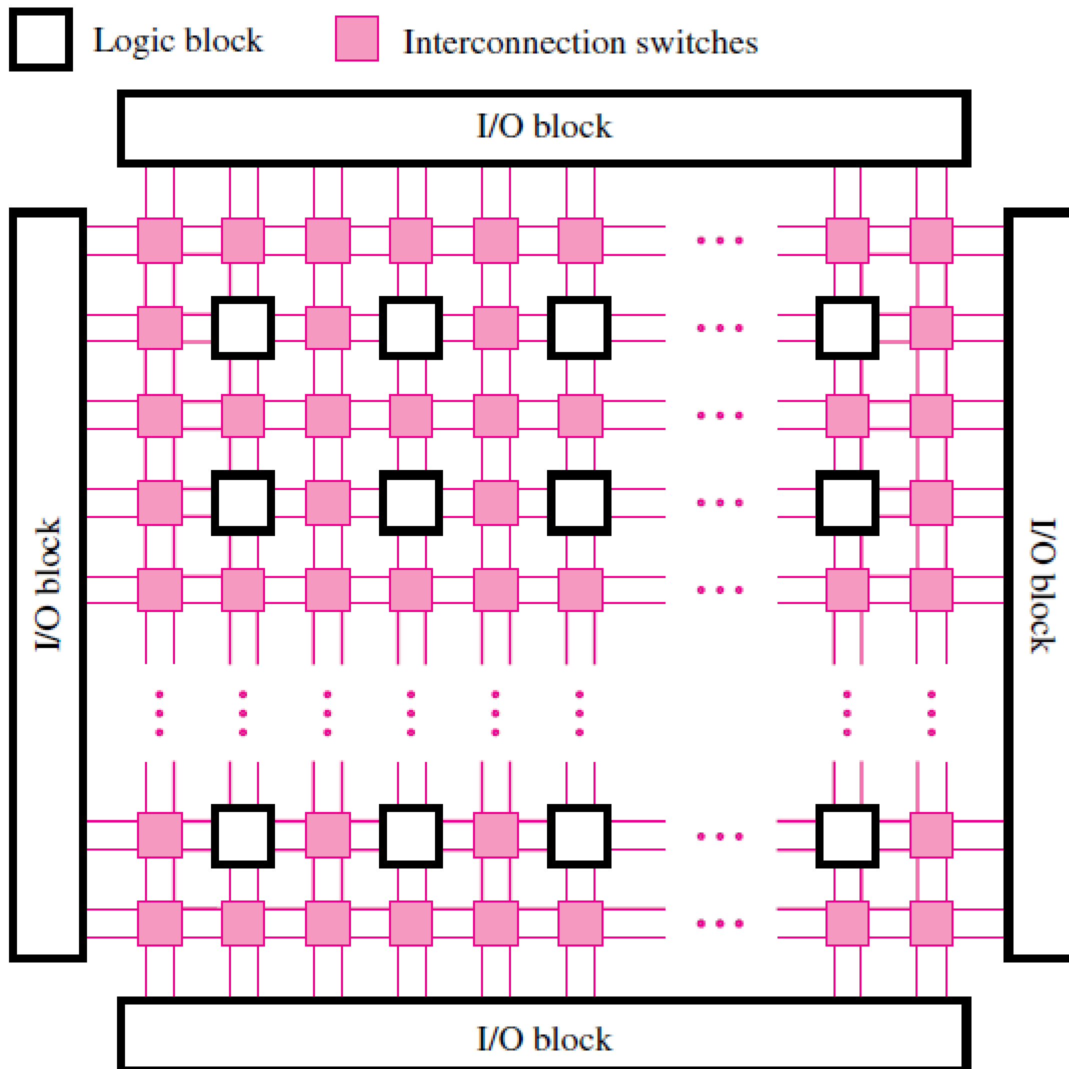


Figure 4.2: General Structure of FPGA, [9].

In Figure 4.2, we have logic blocks shown in thick black squares that are connected together with configurable wires and switches shown in tan lines and squares.

Chapter 5

Modular Verification of Sampled-Data Properties

5.1 Introduction to Previous Work

In [21] Wang introduced a number of predicate based algorithms to verify the majority of the sampled data conditions that we have discussed already. All of these algorithms were intended to work on the synchronous product of the system that contains plant $\mathbf{G}$ and supervisor $\mathbf{S}$. However, in large systems in may be the case that constructing and verifying $\mathbf{G}||\mathbf{S}$ may either take too much memory or time, as due to the large state space of $\mathbf{G}||\mathbf{S}$.

For large systems our plant is usually defined as a set of automata $\mathbf{G}_1, \ldots, \mathbf{G}_m$ and our supervisor as a set of automata $\mathbf{S}_1, \ldots, \mathbf{S}_n$. Our system plant would thus be $\mathbf{G} = \mathbf{G}_1||\mathbf{G}_2 \ldots ||\mathbf{G}_m$, and our system supervisor would be $\mathbf{S} = \mathbf{S}_1||\mathbf{S}_2|| \ldots ||\mathbf{S}_n$. It is easy to see that if we could check our property on the individual automata instead of on $\mathbf{G}||\mathbf{S}$, this would save a lot of time and memory.

In this chapter we investigate modular approach for verifying the following properties:

- **ALF**: Activity loop free.
- **SPB**: Singular prohibitable behaviour
- **SD-Cont-iii.1**: SD controllability point 3.1.

- **SD-Cont-iii.2**: SD controllability point 3.2.
- **SD-Cont-iv**: SD controllability point 4.

By modular approaches, we wish to develop results such that if automata $\mathbf{G}_1$ and $\mathbf{G}_2$ satisfy a property, then $\mathbf{G}_1||\mathbf{G}_2$ satisfies the property. Such a result can be easily extended to an arbitrary number of automata.

5.2 ALF Modularity

The first property we investigate is the ALF property. The ALF property requires a TDES to not contain any loops of activity events. This is to be sure that the occurrence of the *tick* event can not be infinitely preempted, a physically unrealistic fact.

In his masters thesis Wang has already started investigating modular methods for verifying ALF with the proposition below.

Proposition 5.1. [21] 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.

The problem is that supervisors often contain self loops and thus are not ALF themselves. Wang in [21] provided a further result to address this problem. If all the plant TDES components of the system are ALF and all the supervisor TDES components do not contribute any new event to the system, then the synchronous product of the overall components also is ALF. This is given in the proposition below. Where $\mathbf{G}_1$ would be the plant and $\mathbf{G}_2$ would be the supervisor.

Proposition 5.2. [21] 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.

.

Although Wang proved that the ALF property could be verified modularly he only implemented a monolithic ALF checker. Our contribution is to implement a modular ALF checker.

5.3 SPB and SSPB Modularity

Singular prohibitable behaviour (SPB) says that any prohibitable event is allowed to occur at most once in any concurrent string in an SD controller. This was expressed by Wang [21] as the following definition, which is applied to the system plant $\mathbf{G}$.

Definition 5.3.1. For TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, we say that $\mathbf{G}$ has *singular prohibitable behaviour* (old version) if,

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

Another way to capture this concept is to say that if $\sigma \in \Sigma_{hib}$ has already occurred since the last *tick* then σ must not be eligible again until after the next *tick*. This gives us the new definition below. It is easy to see that the two definitions are equivalent. From now and on, whenever we refer to the SPB definition we mean Definition 5.3.2.

Definition 5.3.2. For TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, we say that $\mathbf{G}$ has *singular prohibitable behavior* if:
$(\forall s \in L(\mathbf{G}) \cap L_{samp})(\forall s' \in \Sigma_{act}^*) ss' \in L(\mathbf{G}) \implies (\forall \sigma \in Occu(s') \cap \Sigma_{hib}) \sigma \notin Elig_{L(\mathbf{G})}(ss')$.

The reason we are introducing a new version of the SPB property is that we want to prove that SPB implies S-singular prohibitable behaviour(SSPB) that is defined in Definition 3.2.9 and this new form more clearly matches how the SSPB definition is expressed, as well as how our software evaluates the property. We will then show that the SPB property can be verified modularly, which then gives us a way to verify the SSPB property modularly. The reason we do not try to directly verify the SSPB property modularly is because it involves both the plant $\mathbf{G}$ and supervisor $\mathbf{S}$, where the SPB property only involves the plant. It also lends itself better to modular checks.

5.3.1 SPB implies SSPB

We now prove that singular prohibitable behaviour implies S-singular prohibitable behaviour. In Proposition 5.3 below, it is important that $\mathbf{S}$ is defined over the same

alphabet of $\mathbf{G}$. Otherwise, this proposition is not guaranteed to be correct.

Proposition 5.3. Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ and $\mathbf{S} = (X, \Sigma, \eta, x_0, X_m)$. If $\mathbf{G}$ has *singular prohibitable behaviour* then $\mathbf{G}$ has S-singular prohibitable behaviour.

Proof. Assume $\mathbf{G}$ has singular prohibitable behaviour. This implies:
$(\forall s \in L(\mathbf{G}) \cap L_{samp})(\forall s' \in \Sigma_{act}^*)ss' \in L(\mathbf{G}) \implies (\forall \sigma \in Occu(s') \cap \Sigma_{hib})\sigma \notin Elig_{L(\mathbf{G})}(ss')$. (1)

Must show that $\mathbf{G}$ has S-singular prohibitable behaviour.
Sufficient to show:
$(\forall s \in L(S) \cap L_{samp})(\forall s' \in \Sigma_{act}^*)ss' \in L(S) \cap L(G) \implies (\forall \sigma \in Occu(s') \cap \Sigma_{act})\sigma \notin Ellig_{L(G)}(ss')$
Let $s \in L(\mathbf{G}) \cap L(\mathbf{S}) \cap L_{samp}$. (2)
Let $s' \in \Sigma_{act}^*$.
Assume $ss' \in L(\mathbf{G}) \cap L(\mathbf{S})$. (3)
Let $\sigma \in Occu(s') \cap \Sigma_{hib}$. (4)
Must show implies $\sigma \notin Elig_{L(\mathbf{G})}(ss')$.
By (2), we have $s \in L(\mathbf{G}) \cap L_{samp}$. (5)
By (3), we have $ss' \in L(\mathbf{G})$. (6)
By (4), we have $\sigma \in Occu(s') \cap \Sigma_{hib}$.
$\implies \sigma \notin Elig_{L(\mathbf{G})}(ss')$. (by 1)

□

5.3.2 SPB Modularity

We now proof that if plant component $\mathbf{G}_1$ is SPB, and plant component $\mathbf{G}_2$ is SPB, then $\mathbf{G} = \mathbf{G}_1||\mathbf{G}_2$ is also SPB. In other words the property SPB is modular.

Proposition 5.4. Let $\mathbf{G}_1 = (Y_1, \Sigma_1, \delta_1, y_{0,1}, Y_{m,1})$ and $\mathbf{G}_2 = (Y_2, \Sigma_2, \delta_2, y_{0,2}, Y_{m,2})$ be two TDES.
If $\mathbf{G}_1, \mathbf{G}_2$ both have *singular prohibitable behaviour* then $\mathbf{G} = G_1||\mathbf{G}_2$ has *singular prohibitable behaviour.*

Proof. Let $\Sigma = \Sigma_1 \cup \Sigma_2$.
Let $P_i : \Sigma^* \mapsto \Sigma_i^*$, $i = 1, 2$, be natural projection.
Let $L_{samp} = \Sigma^*.\tau \cup \{\epsilon\}$
Assume that both plants $\mathbf{G}_1$ and $\mathbf{G}_2$ are SPB. (1).
Must show this implies that $\mathbf{G} = \mathbf{G}_1||\mathbf{G}_2$ is also SPB.
Sufficient to show that:
$(\forall s \in L(\mathbf{G}) \cap L_{samp})(\forall s' \in \Sigma_{act}^*) ss' \in L(\mathbf{G}) \implies (\forall \sigma \in Occu(s') \cap \Sigma_{hib})\sigma \notin Elig_{L(\mathbf{G})}(ss')$.
Let $s \in L(\mathbf{G}) \cap L_{samp}$ and $s' \in \Sigma_{act}^*$. (2)
Assume $ss' \in L(\mathbf{G})$. (3)
Let $\sigma \in Occu(s') \cap \Sigma_{hib}$. (4)
Must show implies $\sigma \notin Ellig_{L(\mathbf{G})}(ss')$.
Sufficient to show: $ss'\sigma \notin L(\mathbf{G})$.
From (2), we have $s \in L(\mathbf{G}) = P_1^{-1}(\mathbf{G}_1) \cap P_2^{-1}(\mathbf{G}_2)$.
$\implies P_i(s) \in L(\mathbf{G}_i)$, $i = 1, 2$.
$\implies P_i(s) \in L(\mathbf{G}_i) \cap L_{samp}$, by (2) and as $\tau \in \Sigma_i$, $i = 1, 2$, since $\mathbf{G}_i$ are TDES. (5)
By (2), we have: $P_i(s') \in \Sigma_{act}^*$, $i = 1, 2$. (6)
By (3), we have: $P_i(ss') \in L(\mathbf{G}_i)$, $i = 1, 2$.
$\implies P_i(s')P_i(s') \in L(\mathbf{G}_i)$, $i = 1, 2$, as P_i is cantenative. (7)
By (4), we have that $\sigma \in \Sigma_{hib}$.
As $\Sigma_{hib} \subseteq \Sigma = \Sigma_1 \cup \Sigma_2$ we have two cases: (1) $\sigma \in \Sigma_1$ or (2) $\sigma \in \Sigma_2$.
Case 1) $\sigma \in \Sigma_1$
By (4), we have that $\sigma \in Occu(s')$.
As $\sigma \in \Sigma_1$, we have $P_1(\sigma) = \sigma$. (8)
$\implies \sigma \in Occu(P_1(s'))$.
Combining (5),(6) and (7), we have: $P_1(s) \in L(\mathbf{G}_1) \cap L_{samp}) \wedge (P_1(s') \in \Sigma_{act}^*) \wedge (P_1(s)P_1(s') \in L(\mathbf{G}_1)$
As $\mathbf{G}_1$ is SPB we can then conclude: $\sigma \notin Elig_{L(\mathbf{G}_1)}(P_1(s)P_1(s'))$.
$\implies P_1(s)P_1(s')\sigma \notin L(\mathbf{G}_1)$.
$\implies P_1(ss'\sigma) \notin L(\mathbf{G}_1)$ by (8) and fact P_1 is catenative.
$\implies ss'\sigma \notin P_1^{-1}(\mathbf{G}_1)$
$\implies ss'\sigma \notin L(\mathbf{G}) = P_1^{-1}(\mathbf{G}_1) \cap P_2^{-1}(\mathbf{G}_2)$.

Case 2) $\sigma \in \Sigma_2$

Proof is identical to case (1) up to relabelling $\Sigma_1, P_1, \mathbf{G}_1$ to $\Sigma_2, P_2, \mathbf{G}_2$.

By case (1) and case (2), we have that: $ss'\sigma \notin L(\mathbf{G})$, thus $\sigma \notin Elig_{L(\mathbf{G})}(ss')$. We thus have that $\mathbf{G}$ is SPB, as required.

□

5.4 SD Controllability Modularity

SD controllability, as defined in Chapter 3, contains five properties since property (iii) contains two distinct sub-properties. The first property of SD controllability is essentially untimed controllability. This is a well known modular property and is examined in detail in [5]. As the second property is not modular, we will not discuss it here. We now prove the remaining properties, SD-Cont-iii.1, SD-Cont-iii.2, and SD-Cont-iv, are modular.

5.4.1 SD-Cont-iii.1 Modularity

In Chapter 3, the SD controllability property iii.1 was expressed in terms of a plant and a supervisor. For simplicity, we re-express it in terms of an arbitrary TDES $\mathbf{G}$.

Definition 5.4.1. We say that TDES $\mathbf{G}$ satisfies SD-Cont-iii.1 if and only if:
$(\forall s \in L(\mathbf{G}) \cap L_{samp})(\forall t \in \Sigma_{act}^*) st \in L(\mathbf{G})$
$\implies (Ellig_{L(\mathbf{G})}(st) \cup Occu(t)) \cap \Sigma_{hib} = Ellig_{L(\mathbf{G})}(\mathbf{S}) \cap \Sigma_{hib}.$

Theorem 5.1. Let $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{i,0}, Q_{i,m})$ be two arbitrary TDES, $i = 1, 2$. If $\mathbf{G}_1$ satisfies SD-Cont-iii.1, and $\mathbf{G}_2$ satisfies SD-Cont-iii.1, then $\mathbf{G} = \mathbf{G}_1 || \mathbf{G}_2$ satisfies SD-Cont-iii.1.

Proof. Let $\Sigma = \Sigma_1 \cup \Sigma_2$.
Let $P_i : \Sigma^* \to \Sigma_i^*$ for, $i = 1, 2$, be natural projections.
Let $L_{samp} = \Sigma^*.\tau \cup \{\epsilon\}$.
Assume $\mathbf{G}_1, \mathbf{G}_2$ both satisfy SD-Cont-iii.1
Must show implies $\mathbf{G} = \mathbf{G}_1 || \mathbf{G}_2$ satisfies SD-Cont-iii.1. (1)

Sufficient to show:
$(\forall s \in L(\mathbf{G}) \cap L_{samp})(\forall t \in \Sigma_{act}^*) st \in L(\mathbf{G}) \implies$
$(Elig_{L(\mathbf{G})}(st) \cup Occu(t)) \cap \Sigma_{hib} = Elig_{L(\mathbf{G})}(\mathbf{S}) \cap \Sigma_{hib}$

Let $s \in L(\mathbf{G}) \cap L_{samp}$ and $t \in \Sigma_{act}^*$. (2)
Assume $st \in L(\mathbf{G})$. (3)
Must show implies:
$(Elig_{L(\mathbf{G})}(st) \cup Occu(t)) \cap \Sigma_{hib} = Elig_{L(\mathbf{G})}(\mathbf{S}) \cap \Sigma_{hib}$
Sufficient to show: $\subseteq$ and $\supseteq$

Part A) Show: $(Elig_{L(\mathbf{G})}(st) \cup Occu(t)) \cap \Sigma_{hib} \subseteq Elig_{L(\mathbf{G})}(\mathbf{S}) \cap \Sigma_{hib}$.
Let $\sigma \in (Elig_{L(\mathbf{G})}(st) \cup Occu(t)) \cap \Sigma_{hib}$. (4)
Must show: $\sigma \in Elig_{L(\mathbf{G})}(\mathbf{S}) \cap \Sigma_{hib}$
Sufficient to show: $s\sigma \in L(\mathbf{G}) = P_1^{-1}(\mathbf{G}_1) \cap P_2^{-1}(\mathbf{G}_2)$.
It is thus sufficient to show: $(\forall i \in \{1, 2\}) P_i(s\sigma) \in L(\mathbf{G}_i)$.
Let $i \in \{1, 2\}$
From (4), we have $\sigma \in \Sigma_{hib}$ and either: (a) $\sigma \in Occu(t)$ or (b) $st\sigma \in L(\mathbf{G})$. (5)

We have two cases: (1) $\sigma \notin \Sigma_i$ or (2) $\sigma \in \Sigma_i$.

Case 1) $\sigma \notin \Sigma_i$.
From (2), we have $s \in L(\mathbf{G})$
$\implies P_i(s) \in L(\mathbf{G}_i)$
As $\sigma \notin \Sigma_i$ we have $P_i(\sigma) = \epsilon$, by definition of natural projection.
$\implies P_i(s\sigma) = P_i(s)P_i(\sigma) = P_i(s) \in L(\mathbf{G}_i)$, as P_i is catenative.
Case (1) complete.

Case 2) $\sigma \in \Sigma_i$ (6)
$\implies P_i(\sigma) = \sigma$
From (2), we have $s \in L_{samp}$ and $P_i(s) \in L(\mathbf{G}_i)$
As $\mathbf{G}_i$ is a TDES, $\tau \in \Sigma_i$, thus: $P_i(\tau) = \tau$
We thus have: $s \in L_{samp} \implies P_i(s) \in L_{samp}$
$\implies P_i(s) \in L(\mathbf{G}_i) \cap L_{samp}$

From (2), we also have $P_i(t) \in \Sigma_{act}^*$ as P_i removes events but does not add them.
From (3), we have $P_i(st) = P_i(s)P_i(t) \in L(\mathbf{G}_i)$

As $\mathbf{G}_i$ satisfies SD-Cont-iii.1, it thus follows:

$$(Elig_{L(\mathbf{G}_i)}(P_i(s)P_i(t)) \cup Occu(P_i(t))) \cap \Sigma_{hib} = Elig_{L(\mathbf{G}_i)}(P_i(s)) \cap \Sigma_{hib}. \tag{7}$$

By (5), we have $\sigma \in \Sigma_{hib}$ and either: a) $\sigma \in Occu(t)$ or b) $P_i(st\sigma) = P_i(s)P_i(t)P_i(\sigma) \in L(\mathbf{G}_i)$
We first note that as $\sigma \in \Sigma_i$ by (6), $P_i(\sigma) = \sigma$.

Case a) $\sigma \in Occu(t)$

$\implies t \in \Sigma^*.\sigma.\Sigma^*$
$\implies P_i(t) \in \Sigma^*.\sigma.\Sigma^*$
$\implies \sigma \in Occu(P_i(t))$
$\implies \sigma \in (Elig_{L(\mathbf{G}_i)}(P_i(s)P_i(t)) \cup Occu(P_i(t))) \cap \Sigma_{hib}$
$\implies \sigma \in Elig_{L(\mathbf{G}_i)}(P_i(s))$ by (7).
$\implies P_i(s)\sigma \in L(\mathbf{G}_i)$ by definition of $Elig$ operator.
$\implies P_i(s\sigma) \in L(\mathbf{G}_i)$, as $\sigma \in \Sigma_i$.
Case (a) complete.
Case b) $P_i(s)P_i(t)P_i(\sigma) \in L(\mathbf{G}_i)$.
$\implies P_i(s)P_i(t)\sigma \in L(\mathbf{G}_i)$, as $\sigma \in \Sigma_i$
$\implies \sigma \in Elig_{L(\mathbf{G}_i)}(P_i(s)P_i(t)) \cap \Sigma_{hib}$
$\implies \sigma \in Elig_{L(\mathbf{G}_i)}(P_i(s))$, by (7).
$\implies P_i(s)\sigma \in L(\mathbf{G}_i)$
$\implies P_i(s)P_i(\sigma) = P_i(s\sigma) \in L(\mathbf{G}_i)$, as $\sigma \in \Sigma_i$.
Case (b) complete. By case (a) and (b) we thus conclude $P_i(s\sigma) \in L(\mathbf{G}_i)$
Case (2) complete.
By case (1) and (2), we have $P_i(s\sigma) \in L(\mathbf{G}_i)$, for $i = 1, 2$.
We thus conclude that $\sigma \in Elig_{L(\mathbf{G})}(\mathbf{S}) \cap \Sigma_{hib}$

Part A complete.
Part B) Show: $Elig_{L(\mathbf{G})}(\mathbf{S}) \cap \Sigma_{hib} \subseteq (Elig_{L(\mathbf{G})}(st) \cup Occu(t)) \cap \Sigma_{hib}$

Let $\sigma \in Elig_{L(\mathbf{G})}(\mathbf{S}) \cap \Sigma_{hib}$ (8)

Must show implies: $\sigma \in (Elig_{L(\mathbf{G})}(st) \cup Occu(t)) \cap \Sigma_{hib}$.
Sufficient to show either (a) $\sigma \in Occu(t)$ or (b) $\sigma \in Elig_{L(\mathbf{G})}(st)$

We will assume $\sigma \notin Occu(t)$ and show this implies $\sigma \in Elig_{L(\mathbf{G})}(st)$
As $L(\mathbf{G}) = P_1^{-1}(L(\mathbf{G}_1)) \cap P_2^{-1}(L(\mathbf{G}_2))$ it is thus sufficient to show:

$$(\forall i \in \{1{,}2\})P_i(st\sigma) \in L(\mathbf{G}_i)$$

Let $i \in \{1, 2\}$
Must show implies $P_i(st\sigma) \in L(\mathbf{G}_i)$
We have two cases: (a) $\sigma \notin \Sigma_i$ or (b) $\sigma \in \Sigma_i$

Case a) $\sigma \notin \Sigma_i$.
$\Longrightarrow P_i(\sigma) = \epsilon$.
From (3), we have $st \in L(\mathbf{G})$.
$\Longrightarrow P_i(st) \in L(\mathbf{G}_i)$.
$\Longrightarrow P_i(st\sigma) = P_i(st)P_i(\sigma) = P_i(st) \in L(\mathbf{G}_i)$.

Case (a) complete.

Case b) $\sigma \in \Sigma_i$.
$\Longrightarrow P_i(\sigma) = \sigma$. (10)
From (8), we have $\sigma \in \Sigma_{hib}$ and $s\sigma \in L(\mathbf{G})$.
$\Longrightarrow P_i(s\sigma) = P_i(s)\sigma \in L(\mathbf{G}_i)$.
$\Longrightarrow \sigma \in Elig_{L(\mathbf{G}_i)}(P_i(s)) \cap \Sigma_{hib}$. (11)
From (2), we have $s \in L_{samp}$ and $P_i(s) \in L(\mathbf{G}_i)$.
As $\mathbf{G}_i$ is a TDES, $\tau \in \Sigma_i$, thus: $P_i(\tau) = \tau$.
We thus have $s \in L_{samp} \implies P_i(s) \in L_{samp}$
$\Longrightarrow P_i(s) \in L(\mathbf{G}_i) \cap L_{samp}$
From (2), we also have $P_i(t) \in \Sigma_{act}^*$ as P_i removes events, it does not add them.
From (3), we have $P_i(st) = P_i(s)P_i(t) \in L(\mathbf{G}_i)$
We thus have $P_i(s) \in L(\mathbf{G}_i) \cap L_{samp}$, $P_i(t) \in \Sigma_{act}^*$, and $P_i(s)P_i(t) \in L(\mathbf{G}_i)$.

As $\mathbf{G}_i$ satisfies SD-Cont-iii.1:

$$(Elig_{L(\mathbf{G}_i)}(P_i(s)P_i(t)) \cup Occu(P_i(t))) \cap \Sigma_{hib} = Elig_{L(\mathbf{G}_i)}(P_i(s)) \cap \Sigma_{hib}$$

By (11), we thus have: $\sigma \in (Elig_{L(\mathbf{G}_i)}(P_i(s)P_i(t)) \cup Occu(P_i(t)))$. (12).

By (9), we have $\sigma \notin Occu(t)$.
$\implies t \notin \Sigma^*.\sigma.\Sigma^*$.
$\implies P_i(t) \notin \Sigma^*.\sigma.\Sigma^*$, as $P_i(\sigma) = \sigma$.
$\implies \sigma \notin Occu(P_i(t))$.
$\implies \sigma \in Elig_{L(\mathbf{G}_i)}(P_i(s)P_i(t))$, by (12)
$\implies P_i(s)P_i(t)\sigma = P_i(s)P_i(t)P_i(\sigma) = P_i(st\sigma) \in L(\mathbf{G}_i)$.

Case (b) complete.

By case (a) and (b), we thus have: $P_i(st\sigma) \in L(\mathbf{G}_i)$.
We thus have $(\forall i \in \{1, 2\})P_i(st\sigma) \in L(\mathbf{G}_i)$.
$\implies st\sigma \in L(\mathbf{G})$
$\implies \sigma \in Elig_{L(\mathbf{G})}(st)$
We thus have: $\sigma \in Elig_{L(\mathbf{G})}(st) \cup Occu(t)$

Part (B) complete.

By Part (A) and Part (B), we thus conclude:

$$(Elig_{L(\mathbf{G})}(st) \cup Occu(t)) \cap \Sigma_{hib} = Elig_{L(\mathbf{G})}(\mathbf{S}) \cap \Sigma_{hib}$$

□

5.4.2 SD-Cont-iii.2 Modularity

Fortunately for us it has already been proven that SD-Cont-iii.2 is modular. Please see Baloch [3] for details.

5.4.3 SD-Cont-iv Modularity

We now want to prove that SD-Cont-iv is modular. As for SD-cont-iii.1, we re-express the property below in terms of a single arbitrary TDES $\mathbf{G}$.

Definition 5.4.2. For a TDES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$, we say that $\mathbf{G}$ satisfies SD-Cont-iv if:

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

.

Theorem 5.2. Let $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{0,i}, Q_{m,i})$, $i = 1, 2$, be two TDES. If $\mathbf{G}_1$ and $\mathbf{G}_2$ each satisfy SD-Cont-iv then $\mathbf{G} = \mathbf{G}_1 || \mathbf{G}_2$ satisfies SD-Cont-iv.

Proof. Let $\Sigma = \Sigma_1 \cup \Sigma_2$.
Let $P_i : \Sigma^* \mapsto \Sigma_i^*$, $i = 1, 2$, be natural projections.
Let $L_{samp} = \Sigma^*.\tau \cup \{\epsilon\}$.

Assume $\mathbf{G}_1$ and $\mathbf{G}_2$ satisfy SD-Cont-iv. (1)
Must show $\mathbf{G} = \mathbf{G}_1 || \mathbf{G}_2$ satisfy SD-Cont-iv.
Sufficient to show : $L_m(\mathbf{G}) = P_1^{-1}(L_m(\mathbf{G}_1)) \cap P_2^{-1}(L_m(\mathbf{G}_2)) \subseteq L_{samp}$.
Let $s \in P_1^{-1}(L_m(\mathbf{G}_1)) \cap P_2^{-1}(L_m(\mathbf{G}_2))$. (2)
Sufficient to show $s \in L_{samp}$
We have two cases: $s = \epsilon$ or $s \neq \epsilon$.
Case 1) $s = \epsilon$
We immediately have $s \in L_{samp}$
Case 2) $s \neq \epsilon$
This implies $P_1(s) \neq \epsilon$ or $P_2(s) \neq \epsilon$, as $\Sigma = \Sigma_1 \cup \Sigma_2$. As $\mathbf{G}_1$ and $\mathbf{G}_2$ are arbitrary TDES, we can assume $P_1(s) \neq \epsilon$ without any loss of generality.
From (2) we have: $s \in P_1^{-1}(L_m(\mathbf{G}_1))$

$\implies P_1(s) \in L_m(\mathbf{G}_1)$
From (1) we have: $L_m(\mathbf{G}_1) \subseteq L_{samp}$
$\implies P_1(s) \subseteq L_{samp}$
As $P_1(s) \neq \epsilon$ we have: $P_1(s) \in \Sigma^*.\tau$
We note that by definition of TDES, $\tau \in \Sigma_1$ and $\tau \in \Sigma_2$, thus:

$P_1(\tau) = P_2(\tau) = \tau$
$\implies (\exists s' \in \Sigma^*.\tau)(\exists s_1 \in (\Sigma - \Sigma_1)^*)$ such that $s's_1 = s$. (3)
$\implies P_2(s) \neq \epsilon$ and $P_2(s) \in \Sigma^*.\tau$ as $P_2(s) \in L_m(\mathbf{G}_2)$ (by 2) and by (1).
$\implies (\exists s_2 \in (\Sigma - \Sigma_2)^*) s's_2 = s$.
$\implies s_1 = s_2$.
$\implies s_1 = s_2 = \epsilon$, as $(\Sigma - \Sigma_2)^* \cap (\Sigma - \Sigma_1)^* = \epsilon$ since $\Sigma = \Sigma_1 \cup \Sigma_2$.
$\implies s = s'$
$\implies s \in \Sigma^*.\tau \subseteq L_{samp}$ by (3)
By case 1 and 2, we have $s \in L_{samp}$. □

When checking ALF, we saw that if $\mathbf{G}_1$ is ALF and $\mathbf{G}_2$ is under the same alphabet of $\mathbf{G}_1$ (i.e $\Sigma_2 \subseteq \Sigma_1$), then $\mathbf{G}_1||\mathbf{G}_2$ is also ALF. We will now prove that a similar result is true for SD-Cont-iv. If $\mathbf{G}_1$ satisfies SD-Cont-iv and $\Sigma_{\mathbf{G}_2} \subseteq \Sigma_{\mathbf{G}_1}$ then $\mathbf{G}_1||\mathbf{G}_2$ satisfies SD-Cont-iv. This will allow us to only check the TDES plant components as long as all supervisors have the same alphabet as the plants. This means we do not need to make each marked state in the supervisors a sampled state as well.

Proposition 5.5. Let $\mathbf{G}_1 = (Y_1, \Sigma_1, \delta_1, y_{0,1}, Y_{m,1})$ and $\mathbf{G}_2 = (Y_2, \Sigma_2, \delta_2, y_{0,2}, Y_{m,2})$ be two TDES. If $\mathbf{G}_1$ satisfies SD-Cont-iv and $\Sigma_2 \subseteq \Sigma_1$ then $\mathbf{G} = \mathbf{G}_1||\mathbf{G}_2$ satisfies SD-Cont-iv.

Proof. Let $\Sigma = \Sigma_1 \cup \Sigma_2$.
Define natural projection $P_2 : \Sigma^* \to \Sigma_2^*$.
Assume $\mathbf{G}_1$ satisfies SD-Cont-iv and $\Sigma_2 \subseteq \Sigma_1$. (1)
Sufficient to show that $L_m(\mathbf{G}_1||\mathbf{G}_2) \subseteq L_{samp} = \Sigma^*.\tau \cup \{\epsilon\}$.
Let $s \in L_m(\mathbf{G}_1||\mathbf{G}_2)$. (2)
Must show $s \in L_{samp}$.
From (1), we have:

$L_m(\mathbf{G}_1||\mathbf{G}_2) = L_m(\mathbf{G}_1) \cap P_2^{-1}(L_m(\mathbf{G}_2))$, as $\Sigma_2 \subseteq \Sigma_1$.
$\implies s \in L_m(\mathbf{G}_1)$ by (2).
We have $L_m(\mathbf{G}_1) \subseteq L_{samp}$ as $\mathbf{G}_1$ satisfies SD-Cont-iv.
$\implies s \in L_{samp}$, as required.

□

Chapter 6

Algorithmic Improvements

6.1 Introduction

The contribution we did on the algorithmic side is implementing the modular verification we introduced in Chapter 5. To implement this, we built upon the algorithms introduced by Wang [21]. We divided verification into two types: modular and non-modular. The modular properties where verified modularly, and the non-modular properties were verified on the synchronous product of the system.

In this chapter, we will briefly discuss the previous work, and then explain the algorithms for modular verification. We also introduce new algorithms for checking non-selfloop ALF and the CS deterministic properties.

6.2 Predicate Verification Preliminaries

In this section we will give a brief introduction about verification with predicates. Our work here is based on the work of Wang [21]. Please see first section of Chapter 6 of [21] for more details about predicate-based verification.

6.2.1 State Predicates

In this thesis we use the notation '$\equiv$' to mean logical equivalence between state predicates. We will use *"T"* and *"F"* for logical true and false.

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a TDES.

Definition 6.2.1. A *predicate* P defined on state set Q is a function

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

identified by the corresponding state subset

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

The state predicate *true* is identified by the set Q, while the state predicate *false* is identified by $\emptyset$, and the state predicate P_m is identified 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 $Pred(Q)$ to mean the set of all predicates defined on the set of states Q; therefore $Pred(Q)$ is identified by $\mathrm{Pwr}(Q)$. 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 6.2.2. For $P_1, P_2, P_3 \in Pred(Q)$ and $q \in Q$, we can build boolean expressions by using the following predicate operations.

$(\neg P)(q) = T \iff P(q) = F.$
$(P_1 \wedge P_2)(q) = T \iff P_1(q) = T$ and $P_2(q) = T.$
$(P_1 \vee P_2)(q) = T \iff P_1(q) = T$ or $P_2(q) = T.$
$(P_1 - P_2)(q) = T \iff P_1(q) = T$ and $P_2(q) = F.$

6.2.2 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 $f : Pred(Q) \rightarrow Pred(Q)$. Here we mention some essential predicate transformers from [21] which are required in this chapter:

- $R(\mathbf{G}, P)$: The *reachability predicate* $R(\mathbf{G}, P)$ is true for all the states in $\mathbf{G}$ that can be reached from q_o by states satisfying P. This means that $R(\mathbf{G}, true)$ represents all states that are reachable in $\mathbf{G}$.

- $CR(\mathbf{G}, P)$: 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. This means that $CR(\mathbf{G}, true)$ represents all the states that are coreachable in $\mathbf{G}$.

Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1||\mathbf{G}_2||..||\mathbf{G}_n$ be a TDES, where $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$, i=1,...,n. Let $q \in Q$ and $\sigma \in \Sigma$. We want to compute the transition $\delta(q, \sigma)$ using the symbolic representation introduced by Wang in [21]. 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 evaluate $P' := pr(Q'_P)$. However, computing $\delta(q, \sigma)$ one by one is not going to be efficient for large systems. Instead we want to compute P' directly from P. To do this, we introduce the following functions.

The first function $\hat{\delta} : Pred(Q) \times \Sigma \rightarrow Pred(Q)$, calculates a predicate that represents all states reachable by a $\sigma \in \Sigma$ transition from any state that satisfies a predicate $P \in Pred(Q)$.

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

The second function $\hat{\delta}^{-1} : Pred(Q) \times \Sigma \rightarrow Pred(Q)$, calculates a predicate that represents all states that can reach a state that satisfies $P \in Pred(Q)$ by a $\sigma \in \Sigma$ transition.

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

For more details on how these two predicates operators are implemented, please see Wang [21].

Given a plant $\mathbf{G} = (Y, \Sigma, \delta, y_0, Y_m)$ and a supervisor $\mathbf{S} = (X, \Sigma, \xi, x_0, X_m)$. Let $Q = Y \times X$ be state space of $\mathbf{G}||\mathbf{S}$, our closed loop system. Sometimes we need a transition function $\hat{\delta}$ and $\hat{\delta}^{-1}$ for $\mathbf{G}||\mathbf{S}$ and transition functions $\hat{\delta_{\mathbf{G}}}$ and $\hat{\delta_{\mathbf{G}}}^{-1}$ for the plant and $\hat{\xi}$ and $\hat{\xi}^{-1}$ for the supervisors, all defined to be for $Pred(Q) \times \Sigma \rightarrow Pred(Q)$. As $\mathbf{G}||\mathbf{S}$ has state set Q, $\hat{\delta}$ and $\hat{\delta}^{-1}$ are defined as above, but with δ replaced by $\delta \times \xi$.

For plant $\mathbf{G}$, let $\sigma \in \Sigma$, and $P \in Pred(Q)$. Then:

$$\hat{\delta_{\mathbf{G}}}(P,\sigma) := pr(\{q' = (y',x') \in Q | (\exists q = (y,x) \models P)\delta(y,\sigma) = y'\})$$

$$\hat{\delta_{\mathbf{G}}}^{-1}(P,\sigma) := pr(\{q = (y,x) \in Q | (\exists x' \in X)(\delta(y,\sigma),x') \models P\})$$

For supervisor $\mathbf{S}$, let $\sigma \in \Sigma$, and $P \in Pred(Q)$. Then:

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

$$\hat{\xi}^{-1}(P,\sigma) := pr(\{q = (y,x) \in Q | (\exists y' \in Y)(y', \xi(x,\sigma)) \models P\})$$

6.3 Previous Approach

Algorithm 6.1 shows the original algorithm approach from Wang [21] that verifies the system according to the synchronous product of its components. It is easy to see that the first thing we do in verifying the system is to build the synchronous product of the system from the plants $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$, i=1,...,m and supervisors $\mathbf{S}_i = (X_i, \Sigma_i, \xi_i, x_{o,i}, X_{m,i})$, i=1,...,n, and we pass it to Algorithm 6.2. The algorithm constructs the system plant $\mathbf{G}$, and supervisor $\mathbf{S}$ which is passed to Algorithm 6.2 (**VerifySub**). Algorithm 6.2 then performs BDD-based test on the desired properties. The algorithm fails if any property test fails.

In this algorithm, the notation "¬" means logical NOT. The variable *result* is a Boolean variable holds the value returned by algorithm **VerifySub**.

Algorithm 6.1 BddSDCheckProp()

1: $\mathbf{G} \leftarrow G_1||G_2||...||G_m$
2: $\mathbf{S} \leftarrow S_1||S_2||...||S_n$
3: $result \leftarrow VerifySub(\mathbf{G}, \mathbf{S})$
4: **if** $(\neg result)$ **then**
5:     **return** False
6: **else**
7:     **return** True
8: **end if**

The subalgorithm, **VerifySub**, is shown in Algorithm 6.2. This is the version from Wang [21]. The following variables are used in the algorithm, as described below:

- **bddConBad**: The predicate for untimed controllability check.
- **bddPLCOMPBad**: The predicate for plant completeness check.
- **bddNBBad**: The predicate for nonblocking check.
- **bddALFBad**: The predicate for ALF check.
- **bddPTBBad**: The predicate for proper time behaviour check.
- **bddSDBad**: The predicate for SD controllability check.
- **bddReach**: The reachability predicate for the system.
- **bddCoReach**: The co-reachability predicate for the reachable states in the system.

We also list the following sub algorithms that were created by Wang [21] to do the corresponding test. For more information on these algorithms see [21]:

- $R(\mathbf{G}, true)$: Find all reachable states in the system.

- $CR(\mathbf{G}, bddReach)$: Find all co-reachable states that are reachable.
- **VeriConBad**: Test if **S** is untimed controllable for **G**. This corresponds to Algorithm 6.3 of Wang [21].
- **VeriBalemiBad**: Test if the plant **G** is complete for **S**. This corresponds to Algorithm 6.4 of Wang [21].
- **Nonblocking**: Test that all states that satisfy $R(\mathbf{G}||\mathbf{S}, true)$, also satisfy $CR(\mathbf{G}||\mathbf{S}, R(\mathbf{G}||\mathbf{S}, true))$. This corresponds to Algorithm 6.5 of Wang [21]
- **VeriALF**: Test if the system **G**||**S** is activity free loop. This corresponds to Algorithm 6.6 of Wang [21]
- **VeriProperTimedBehavior**: Test if the plant **G** has proper time behaviour. This corresponds to Algorithm 6.7 of Wang [21]
- **CheckSDControllability**: Test if **S** is SD controllable for **G**. This corresponds to Algorithm 6.8 of Wang [21]

The old version of algorithm **VerifySub** tests all the properties together and fails if any one of the tests fail. Lines 1-8 of the algorithm initializes predicates to *false*. On line 9, the algorithm fails if there are no reachable states in the system. Line 12 returns False if the system is blocking. Line 15 tests untimed controllability. Lines 16-17 returns False if **BddConBad** contains any reachable states that fail the test.

On lines 19, 23, 27, and 31, the indicated algorithm is called. The algorithm returns False if the indicated predicate predicate is returned containing reachable states that fail the test. The algorithm returns with **True** if all tests succeeded.

Algorithm 6.2 VerifySub($\mathbf{G}, \mathbf{S}$) Part A

```
1: bddConBad ← false
2: bddPLCOMPBad ← false
3: bddNBBad ← false
4: bddALFBad ← false
5: bddPTBBad ← false
6: bddSDBad ← false
7: bddReach ← R(G||S, true)
8: bddCoReach ← CR(G||S,bddReach)
9: if (bddReach≡ false) then
10:    return False
11: end if
12: if (bddCoReach ≢ bddReach) then
13:    return False
14: end if
15: VeriConBad(bddConBad,bddReach)
16: if (bddConBad≢ false) then
17:    return False
18: end if
```

Algorithm 6.2 VerifySub(**G**,**S**) Part B

```
19: VeriBalemiBad(bddPLCOMPBad, bddReach)
20: if (bddPLCOMPBad≢ false) then
21:     return False
22: end if
23: VeriALF(bddALFBad, bddReach)
24: if (bddALFBad≢ false) then
25:     return False
26: end if
27: VeriProperTimedBehavior(bddPTBBad, bddReach)
28: if (bddPTBBad≢ false) then
29:     return False
30: end if
31: CheckSDControllability(bddSDBad, bddReach)
32: if (bddSDBad≢ false) then
33:     return False
34: end if
35: return True
```

6.4 Modular Verification Algorithm

We now introduce a new version of the **BddSDCheckProp** algorithm (Algorithm 6.3) that is designed to support modular tests. In section 6.5, we will introduce a new **VerifySub** algorithm that will also support modular tests.

We first need to introduce a new Boolean array named *testType*. The purpose of the array is to specify exactly which test should be performed. It is defined as follows:

- **testType[0]**: Do all the BDD properties test.
- **testType[1]**: Do nonblocking and untimed controllability test.
- **testType[2]**: Do untimed controllability test.
- **testType[3]**: Do nonblocking test.
- **testType[4]**: Do proper time behaviour test.
- **testType[5]**: Do plant completeness test.
- **testType[6]**: Do SD controllability test.
- **testType[7]**: Do SD controllability point i.
- **testType[8]**: Do SD controllability point ii.
- **testType[9]**: Do SD controllability point iii.1.
- **testType[10]**: Do SD controllability point iii.2.
- **testType[11]**: Do SD controllability point iv.
- **testType[12]**: Do activity loop free (ALF) test.
- **testType[13]**: Do S-singular prohibitable behaviour (SSPB) test.
- **testType[14]**: Do non-selfloop ALF test.

- **testType[15]**: Check that all supervisors are non-selfloop activity loop free.
- **testType[16]**: Check that all supervisors are CS deterministic.

In the following algorithms in this chapter, we will use the following indices names to specify positions in *testType*. For example, to specify that we wish to do the nonblocking test, we will set *testType[DoNB]=1*.

- **DoAll**: Index (0).
- **DoNBandCont**: Index (1).
- **DoCont**: Index (2).
- **DoNB**: Index (3).
- **DoPTB**: Index (4).
- **DoPCOMPL**: Index (5).
- **DoSDCont**: Index (6).
- **DoSDCont1**: Index (7).
- **DoSDCont2**: Index (8).
- **DoSDCont31**: Index (9).
- **DoSDCont32**: Index (10).
- **DoSDCont4**: Index (11).
- **DoALF**: Index (12).
- **DoSSPB**: Index (13).

- **DoNSLALF**: Index (14).
- **DoAllSupsNSLALF**: Index (15).

- **DoAllSupsCSDet**: Index (16).

We now presents a modular version of Algorithm 6.1, namely Algorithm 6.3. It is assumed that when Algorithm 6.3 is initially called by the system, only one position of the *testType* will be set to True.

For Algorithm 6.3, we will split it over several pages to show the various tests. In Algorithm 6.3, we use the following parameters and identifiers:

Plants : All system plant components, **Plants**=$\{\mathbf{G}_1, \mathbf{G}_2, ..., \mathbf{G}_m\}$.

$\mathbf{G}_i$: A single plant from the system plants, $i \in \{1, ..., m\}$

Sups : All system supervisor components, **Sups**=$\{\mathbf{S}_1, \mathbf{S}_2, ..., \mathbf{S}_n\}$

$\mathbf{S}_j$: A single supervisor from the system supervisors, $j \in \{1, ..., n\}$

$\mathbf{P_{GS}}$: All system components, plants and supervisors, $\mathbf{P_{GS}}$ =**Plants**$\cup$**Sups**

$\mathbf{A}_h$: A single component from the system, $h \in \{1, ..., m+n\}$. It could be a plant or a supervisor.

The *DoAllSupsNSLALF* check on lines 3-11, iterates over all the supervisors to check that they are non-selfloop ALF. It uses Algorithm 6.7, *CheckNSLALF*, to check that the supervisor is NSL ALF. If at least one test fails, the check returns false. The *DoAllSupsCSDet* check (lines 12-20), iterates over all the supervisors and checks that each TDES is CS deterministic using Algorithm 6.8, *CheckSDCont_SSPB_CSDet*. This check fails if at least one supervisor fails.

The *DoALF* check on lines 21-24, uses Algorithm 6.5 to determine if the system is *ALF*. This algorithm uses modular verifications first, and only checks the full system if the modular check fails. The *DoSSPB* check on lines 25-34, uses modular verification first by iterating over all the plants to verify that they have *singular prohibitable behaviour*. If this check fails, then *SSPB* property is checked on the entire system.

For the next part of Algorithm 6.3, we use the following new identifiers:

Algorithm 6.3 BddSDCheckProp(**Plants**, **Sups**, testType) Part A

```
1: G = G_1||....||G_m
2: S = S_1||....||S_n
3: if (testType[DoAllSupsNSLALF]) then
4:    for all (S_j ∈ Sups) do
5:       result ← CheckNSLALF(S_j)
6:       if (¬result) then
7:          return False
8:       end if
9:    end for
10:   return True
11: end if
12: if (testType[DoAllSupsCSDet]) then
13:    for all (S_j ∈ Sups ) do
14:       result ← CheckSDCont_SSPB_CSDet(S_j,S_j, testType)
15:       if (¬result) then
16:          return False
17:       end if
18:    end for
19:    return True
20: end if
```

rDoSDCont31 : We set this Boolean variable to true if the SD controllability point iii.1 modular check fails, and needs to be repeated for the entire system.

rDoSDCont32 : We set this Boolean variable to true if the SD controllability point iii.2 modular check fails, and needs to be repeated for the entire system.

rDoSDCont4 : We set this Boolean variable to true if the SD controllability point iv modular check fails, and needs to be repeated for the entire system.

The *DoSDCont* check starts on line 35 by checking the modular SD controllability properties first; namely,*SD controllability point iii.1*, *SD controllability point iii.2* and *SD controllability point iv.* It checks each one by iterating over all automata in the

Algorithm 6.3 BddSDCheckProp(**Plants**,**Sups**, testType) Part B

21: **if** (testType[DoALF]) **then**
22: result ← CheckSystemIsALF(**Plants**,**Sups**)
23: **return** result
24: **end if**
25: **if** (testType[DoSSPB]) **then**
26: **for all** ($\mathbf{G}_i \in$ **Plants**) **do**
27: result ← CheckSDCont_SSPB_CSDet($\mathbf{G}_i$, $\mathbf{G}_i$, testType)
28: **if** ($\neg result$) **then**
29: result ← CheckSDCont_SSPB_CSDet(**G**, **S**, testType)
30: **return** result
31: **end if**
32: **end for**
33: **return** True
34: **end if**

system. As the subalgorithm that does the check expects two automata which it will internally synchronize together, we can reuse the algorithm to test a single TDES, $\mathbf{A}_i$, by passing in two copies of $\mathbf{A}_i$, as $\mathbf{A}_i||\mathbf{A}_i = \mathbf{A}_i$. However, for *SD controllability point iv*, we have two strategies for the modular check. The first strategy is to check the property against only the plants automata if the supervisors do not introduce new events. If the supervisors do introduce new events, we then test the property against each automata $\mathbf{A}_i \in P_{GS}$. We note that if *DoSDCont* is set in *testType* at line 35, we can assume that all other positions of *testType* have been set to False.

When the algorithm finishes the modular checks, it then calls the subalgorithem to build the system synchronous product to test SD controllability points i, ii and the points that failed the modular verification.

On line 36, the algorithm first sets **testType[DoSDCont31]** to 1 indicating to only do this part of the test. We set **rDoSDCont31** to false to indicate that we do not have to repeat the **DoSDCont31** test. Variable **rDoSDCont31** will be set to true if the modular test fails. For lines 38-44, we test each automata in the system

Algorithm 6.3 BddSDCheckProp(**Plants**,**Sups** , testType) Part C

```
35: if (testType[DoSDCont]) then
36:   testType[DoSDCont31] ← 1
37:   rDoSDCont31 ← False
38:   for all (A_i ∈ P_GS) do
39:     result ← CheckSDCont_SSPB_CSDet(A_i,A_i, testType)
40:     if (¬result) then
41:       rDoSDCont31 ← True
42:       exit for
43:     end if
44:   end for
45:   testType[DoSDCont31] ← 0
46:   testType[DoSDCont32] ← 1
47:   rDoSDCont32 ← False
48:   for all (A_i ∈ P_GS) do
49:     result ← CheckSDCont_SSPB_CSDet(A_i,A_i, testType)
50:     if (¬result) then
51:       rDoSDCont32 ← True
52:       exit for
53:     end if
54:   end for
```

for SD controllability point iii.1 only. On lines 45-54, we disable the iii.1 test, and enable the iii.2 test. Lines 48-54, then test each automata for point iii.2.

For the next part of the algorithm (lines 55-74), we also use the following new variables:

- Σ_{pln}: The set of events contained in the event set of the system's plants.
- Σ_{sup}: The set of events contained in the event set of the system's supervisors.

On lines 55-74, we start the modular check of the SD controllability point iv. First, if the supervisors do not introduce any new events, then we can check just the plants, as per Theorem 5.5 in Chapter 5. If the supervisor introduces new events, we

then need to iterate over all of the system components to check for SD controllability point iv. If the two previous steps fail, then we need to set SD controllability identifier *rDoSDCont4* to true to repeat this test using the entire system.

Algorithm 6.3 BddSDCheckProp(**Plants**,**Sups**, testType) Part D

```
55:  testType[DoSDCont32] ← 0
56:  testType[DoSDCont4] ← 1
57:  rDoSDCont4 ← False
58:  if (Σ_sup ⊆ Σ_pln) then
59:     for all (G_i ∈ Plants) do
60:        result ← CheckSDCont_SSPB_CSDet(G_i,G_i, testType)
61:        if (¬result) then
62:           rDoSDCont4 ← True
63:           exit for
64:        end if
65:     end for
66:  else
67:     for all (A_i ∈ P_GS) do
68:        result ← CheckSDCont_SSPB_CSDet(A_i,A_i, testType)
69:        if (¬result) then
70:           rDoSDCont4 ← True
71:           exit for
72:        end if
73:     end for
74:  end if
```

When we have finished all the modular checks for SD controllability, we then know which modular checks failed as their corresponding identifiers, *rDoSDcont31, rDoSDcont32, rDoSDcont4* will be set to True. Our next step is to test the properties that failed the modular tests on the entire system, as well as test the non-modular properties, namely SD controllability points i and ii. This occurs on lines 75-93.

On line 75 we clear the flag for point iv. We then start by setting the *testType*

flags for points i and ii (Lines 76-77). We then set the flags for points iii.1, iii.2, and iv to 1 only if their corresponding re-test flags have been set (lines 78, 81, 84). Line 87 performs all of the required tests on the entire system.

Algorithm 6.3 BddSDCheckProp(**Plants**, **Sups**, testType) Part E

```
75:     testType[DoSDCont4] ← 0
76:     testType[DoSDCont1] ← 1
77:     testType[DoSDCont2] ← 1
78:     if (rDoSDCont31) then
79:        testType[DoSDCont31] ← 1
80:     end if
81:     if (rDoSDCont32) then
82:        testType[DoSDCont32] ← 1
83:     end if
84:     if (rDoSDCont4) then
85:        testType[DoSDCont4] ← 1
86:     end if
87:     result ← CheckSDCont_SSPB_CSDet(G,S, testType)
88:     if (¬result) then
89:        return False
90:     else
91:        return True
92:     end if
93:  end if
```

The remaining part of Algorithm 6.3, (lines 94-133) perform the *DoNB*, *DoCont*, *DoNBandCont*, *DoPTB*, *DoPCOMPL*, and the *DoAll* checks. The first five just use the existing methods from Wang [21], via calls to **VerifySub** (Algorithm 6.4).

We will now discuss how we perform the **DoAll** test, (lines 134-147). We use the following identifiers:

- **i**: An integer variable.

Algorithm 6.3 BddSDCheckProp(**Plants**, **Sups**, testType) Part F

```
94:  if (testType[DoNB]) then
95:    result ← VerifySub(G,S, testType)
96:    if (¬result) then
97:      return False
98:    else
99:      return True
100:   end if
101: end if
102: if (testType[DoCont]) then
103:   result ← VerifySub(G,S, testType)
104:   if (¬result) then
105:     return False
106:   else
107:     return True
108:   end if
109: end if
```

- **testIndex**: An integer index representing the current test to be done.
- **resTemp**: A temporary Boolean variable.
- **result**: A Boolean variable.

The *DoAll* check iterates over all the checks and executes them. However, checking that all the supervisors are CS deterministic or NSL ALF is not part of the *DoAll* check so they are not performed here. For each check, we start by clearing all the flags for *testType* as they may have been modified by the previous call to **BDDSD-Checkprop** (lines 137-139). We then set the flag for the desired check and execute it (lines 140-141). If any checks fails, we will return False. In the actual **DESpot** [12] code, we keep track of which tests pass and fail and return that information to the calling program.

Algorithm 6.3 BddSDCheckProp(**Plants**, **Sups**, testType) Part G

```
110: if (testType[DoNBandCont]) then
111:   result ← VerifySub(G, S, testType)
112:   if (¬result) then
113:     return False
114:   else
115:     return True
116:   end if
117: end if
118: if (testType[DoPTB]) then
119:   result ← VerifySub(G, S, testType)
120:   if (¬result) then
121:     return False
122:   else
123:     return True
124:   end if
125: end if
126: if (testType[DoPCOMPL]) then
127:   result ← VerifySub(G, S, testType)
128:   if (¬result) then
129:     return False
130:   else
131:     return True
132:   end if
133: end if
```

Algorithm 6.3 BddSDCheckProp(**Plants**, **Sups**, testType) Part H

```
134: if (testType[DoAll]) then
135:   result ← True
136:   for all (testIndex ∈ {DoALF, DoSSPB, DoSDCont, DoNB, DoPTB, DoPCMPL}) do
137:     for all (i ∈ {0, .., 16}) do
138:       testType[i]← 0
139:     end for
140:     testType[testIndex] ← 1
141:     resTemp ← BddSDCheckProp(Plants, Sups, testType)
142:     if (¬ resTemp) then
143:       result ← False
144:     end if
145:   end for
146:   return result
147: end if
```

6.5 Modular Version of VerifySub

We now present a new version of **VerifySub**, Algorithm 6.4, designed to be called by Algorithm 6.3.

Algorithm 6.4 VerifySub($\mathbf{G}, \mathbf{S}$, testType) Part A

```
1: bddConBad ← false
2: bddPLCOMPBad ← false
3: bddNBBad ← false
4: bddALFBad ← false
5: bddPTBBad ← false
6: bddSDBad ← false
7: bddReach ← R(G||S, true)
8: bddCoReach ← CR(G||S,bddReach)
9: if (bddReach≡ false) then
10:    return False
11: end if
12: if (testType[DoAll] OR testType[DoNBandCont] OR testType[DoNB]) then
13:    if (bddCoReach ≢ bddReach) then
14:       return False
15:    end if
16: end if
17: if (testType[DoAll] OR testType[DoNBandCont] OR testType[DoCont] OR test-
    Type[DoSDCont]) then
18:    VeriConBad(bddConBad,bddReach)
19:    if (bddConBad≢ false) then
20:       return False
21:    end if
22: end if
```

Algorithm 6.4 starts by building the same variables as the old version, Algorithm 6.2. It also returns False if the system has no reachable states as per line 9. On lines 12-16, it checks if the system is blocking. On lines 17-22, the algorithm test for untimed controllability.

In the final par the Algorithm 6.4 uses the following variable:

- ret: This is a temporary Boolean variable.

Algorithm 6.4 VerifySub($\mathbf{G}, \mathbf{S}$, testType) Part B

```
23: if (testType[DoAll] OR testType[DoPCOMPL]) then
24:   VeriBalemiBad(bddPLCOMPBad, bddReach)
25:   if (bddPLCOMPBad≢ false) then
26:     return False
27:   end if
28: end if
29: if (testType[DoAll] OR testType[DoALF]) then
30:   VeriALF(bddALFBad, bddReach)
31:   if (bddALFBad≢ false) then
32:     return False
33:   end if
34: end if
35: if (testType[DoAll] OR testType[DoPTB]) then
36:   VeriProperTimedBehavior(bddPTBBad, bddReach)
37:   if (bddPTBBad≢ false) then
38:     return False
39:   end if
40: end if
41: if (testType[DoAll] OR testType[DoSDCont]) then
42:   ret←CheckSDCont_SSPB_CSDet(G, S, testType)
43:   if (¬ret) then
44:     return False
45:   end if
46: end if
47: return True
```

On lines 23-28, the algorithm tests the system for plant completeness. On lines 29-34, the algorithm tests if the system is activity loop free. On lines 35-40, the

algorithm tests if the system has proper time behavior. On lines 41-47, the algorithm tests if the system is SD controllable.

6.6 System ALF Algorithm

It was proven by Wang in [21] that if all automatons in the system are *ALF* then the system's synchronous product is also *ALF*. It was also proven that if the system plant **G** is ALF and the system supervisor **S** does not introduce any new events, then the synchronous product is also ALF.

We implemented the modular ALF check as Algorithm 6.5, *CheckSystemIsALF*. First we check if the supervisors introduce any new events. If they do not, then we check each plant for ALF separately (lines 4-14).

If the supervisors do introduce new events, we then do the same as above but we iterate over all the plants and supervisors in the project (lines 17-27). If each one is *ALF*, then the system is *ALF*. Otherwise we build the synchronous product of the entire system and check to see if it is *ALF* (line 20). If it passes, then the system is *ALF*, otherwise it is not *ALF*.

We use the following variables in Algorithm 6.5 and Algorithm 6.6:

- Σ_{pln}: The set of events contained in the event set of the system's plants.
- Σ_{sup}: The set of events contained in the event set of the system's supervisors.
- $\mathbf{P_{GS}}$: All system components, plants and supervisors, $\mathbf{P_{GS}} = \mathbf{Plants} \cup \mathbf{Sups}$.
- Σ_{act}: The set of activity events in the system.
- result: Temporary Boolean variable.

Algorithm 6.6, *CheckALF* is taken verbatim from Wang [21] and is given here for the convenience of the reader. Please see [21] for a discussion of the algorithm.

Algorithm 6.5 CheckSystemIsALF(**Plants**, **Sups**)

```
1:  G = G_1||....||G_m
2:  S = S_1||....||S_n
3:  if (Σ_sup ⊆ Σ_pln) then
4:     for all (G_i ∈ Plants) do
5:        result ← CheckALF(G_i)
6:        if (¬result) then
7:           result ← CheckALF(G||S)
8:           if (¬result) then
9:              return False
10:          else
11:             return True
12:          end if
13:       end if
14:    end for
15:    return True
16: else
17:    for all (A_i ∈P_GS) do
18:       result ← CheckALF(A_i)
19:       if (¬result) then
20:          result ← CheckALF(G||S)
21:          if (¬result) then
22:             return False
23:          else
24:             return True
25:          end if
26:       end if
27:    end for
28:    return True
29: end if
```

Algorithm 6.6 CheckALF($\mathbf{G}$), From [21]

1: $P_{chk} \leftarrow R(\mathbf{G}, true)$
2: $P_{tmp} \leftarrow false$
3: **for** $(q \models P_{chk})$ **do**
4: $\quad P_{visit} \leftarrow \left(\bigvee_{\sigma \in \Sigma_{act}} \hat{\delta}(pr(\{q\}), \sigma)\right) \wedge P_{chk}$
5: $\quad overlap \leftarrow False$
6: $\quad P_{next} \leftarrow P_{visit}$
7: $\quad$ **repeat**
8: $\quad\quad P_{next} \leftarrow \left(\bigvee_{\sigma \in \Sigma_{act}} \hat{\delta}(P_{next}, \sigma)\right) \wedge P_{chk}$
9: $\quad\quad P_{tmp} \leftarrow P_{visit}$
10: $\quad\quad$ **if** $(P_{visit} \wedge P_{next} \not\equiv false)$ **then**
11: $\quad\quad\quad overlap \leftarrow True$
12: $\quad\quad$ **end if**
13: $\quad\quad P_{visit} \leftarrow P_{visit} \vee P_{next}$
14: $\quad\quad$ **if** $(q \models P_{visit})$ **then**
15: $\quad\quad\quad$ **return** false
16: $\quad\quad$ **end if**
17: $\quad$ **until** $(P_{visit} \equiv P_{tmp})$
18: $\quad P_{chk} \leftarrow P_{chk} - pr(\{q\})$
19: $\quad$ **if** $(\neg overlap)$ **then**
20: $\quad\quad P_{chk} \leftarrow P_{chk} - P_{visit}$
21: $\quad$ **end if**
22: **end for**
23: **return** true

6.7 Non-selfloop ALF Algorithm

Algorithm 6.7, *CheckNSLALF*, is essentially the ALF algorithm designed by Wang [21] with only one line modified, line 4. On line 4 we added this part "$-pr(\{q\})$". This was a small change, but it was very non-obvious. On line 3, a state q is chosen from the set of reachable states that have not yet been shown to have no activity loops leaving the state. Line 4 of Algorithm 6.6 calculated all states reachable from q by a single activity event transition. The only way q would be included here is if it has activity event selfloops. In Algorithm 6.7, we thus add to line 4 the ending "$-pr(\{q\})$", which removes state q from the predicate P_{visit}. As the rest of algorithm is unchanged, it should still detect if an activity loop longer than a selfloop returns us to q.

Algorithm 6.7 CheckNSLALF($\mathbf{G}$)

1: $P_{chk} \leftarrow R(\mathbf{G}, true)$
2: $P_{tmp} \leftarrow false$
3: **for** $(q \models P_{chk})$ **do**
4: $\quad P_{visit} \leftarrow \left(\bigvee_{\sigma \in \Sigma_{act}} \hat{\delta}(pr(\{q\}), \sigma)\right) \wedge P_{chk} - pr(\{q\})$
5: $\quad overlap \leftarrow False$
6: $\quad P_{next} \leftarrow P_{visit}$
7: $\quad$ **repeat**
8: $\quad\quad P_{next} \leftarrow \left(\bigvee_{\sigma \in \Sigma_{act}} \hat{\delta}(P_{next}, \sigma)\right) \wedge P_{chk}$
9: $\quad\quad P_{tmp} \leftarrow P_{visit}$
10: $\quad\quad$ **if** $(P_{visit} \wedge P_{next} \not\equiv false)$ **then**
11: $\quad\quad\quad overlap \leftarrow True$
12: $\quad\quad$ **end if**
13: $\quad\quad P_{visit} \leftarrow P_{visit} \vee P_{next}$
14: $\quad\quad$ **if** $(q \models P_{visit})$ **then**
15: $\quad\quad\quad$ **return** false
16: $\quad\quad$ **end if**
17: $\quad$ **until** $(P_{visit} \equiv P_{tmp})$
18: $\quad P_{chk} \leftarrow P_{chk} - pr(\{q\})$
19: $\quad$ **if** $(\neg overlap)$ **then**
20: $\quad\quad P_{chk} \leftarrow P_{chk} - P_{visit}$
21: $\quad$ **end if**
22: **end for**
23: **return** true

6.8 SD Controllability, SSPB and CS Deterministic Algorithms

The Algorithms in this section are used to check the SD controllability, S-singular Prohibitable Behaviour and CS Deterministic properties. All the algorithms in this section are based on the versions created by Wang in [21]. We recommend the reader first familiarize themselves with Wang's algorithms as we will focus on explaining

only the parts that we modified.

The algorithms in this section make the following assumptions:

- The set of prohibitable events Σ_{hib} equals the set of forcible events Σ_{for}.
- The plant has proper time behaviour.
- The System is ALF.
- All TDES are finite and deterministic.

The algorithms use certain variables as they executes. They are as follows:

$\mathbf{G}_{cl}$: The synchronous product of the system, $\mathbf{G}_{cl} = \mathbf{G}||\mathbf{S}$.

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 analysed by the algorithm.

$\hat{\delta}$: Transition function for state predicates for $\mathbf{G}_{cl}$.

$\hat{\delta}_{\mathbf{G}}$: Transition function for $\mathbf{G}$ only, as defined in Section 6.2.2.

$\hat{\xi}$: Transition function for $\mathbf{S}$ only, as defined in Section 6.2.2.

$pNerFail$: This set $pNerFail \subseteq \text{Pwr}(Pred(Q))$ is a set of sets of predicates that stores information where **Point iii.2** in definition of SD controllability in Chapter 3 may have failed.

result: This flag asserts if $\mathbf{S}$ is SD controllable with respect to $\mathbf{G}$.

testType: This is a Boolean array which defines which tests to perform.

In Algorithm 6.8 and 6.9, we also use a C++ data structure *FSMCarrier* which is explained in Chapter 7. This data structure is used only when we are checking supervisors for CS deterministic. Whenever we check a supervisor TDES for the CS deterministic property, we store the resulting information in this data structure.

- *Fsm*: This variable is assigned an instance of the *FSMCarrier* data structure, initialized for supervisor **S**, and emptied from all data when initialized.

6.8.1 CheckSDCont_SSPB_CSDet Algorithm

Algorithm 6.8 is the entry algorithm for checking the SD controllability, S-singular prohibitable behaviour and CS deterministic properties. We note that if Algorithm 6.8 is called with both **G** and **S** set to the plant, it can be used to check the singular prohibitable behaviour. We also note that if the CS deterministic property is being checked, then both **G** and **S** should be set to the desired supervisor component by the calling function.

When the *testType* is set for **DoSDCont1** (untimed controllability), we use algorithm **CheckUntimedControllability** from Wang. For details on this algorithm, please see Wang [21].

Algorithm 6.8 is based on the *CheckSDControllability* algorithm from Wang [21]. Our algorithm is essentially the same, but we add the ability to check the CS deterministic property and we pass in the *testType* Boolean array. In the original algorithm, all SD controllability properties and S-singular prohibitable behaviour property would always be checked. In the new version, only the properties whose flags are set to True in the *testType* are checked. In particular this allows the existing algorithms to be reused for modular checks of a specific property.

As part of the CS deterministic check, we set variable *Fsm* to be an instantiation of the data structure **FSMCarrier** (line 2), initialized for supervisor **S**. As part of the CS deterministic check, Algorithm 6.9, *AnalyseSampledState*, collects information that will be used in the TDES supervisor to FSM translation algorithms in Chapter 7.

In this algorithm we also use the following sub algorithms from Wang [21]. As we did not need to do any changes on them, we are not discussing them here:

- CheckSamplingMarkingStates: See Wang [21], Algorithm 6.15.

Algorithm 6.8 CheckSDCont_SSPB_CSDet($\mathbf{G}, \mathbf{S}$, testType) Part A

1: **if** (testType[DoCSDet]) **then**
2: Fsm ← FSMCarrier($\mathbf{S}$)
3: **end if**
4: $P_{reach} \leftarrow R(\mathbf{G}||\mathbf{S}, true)$
5: **if** (testType[DoSDCont1] AND CheckUntimedControllability($\mathbf{G}, \mathbf{S}, P_{reach}$) ≡ False) **then**
6: **return** False
7: **end if**
8: **if** (testType[DoSDCont2] AND CheckSDContii($\mathbf{G}, \mathbf{S}, P_{reach}$) ≡ False) **then**
9: **return** False
10: **end if**
11: result ← True
12: $P_{SF} \leftarrow pr\{z_0\}$
13: $Z_{SP} \leftarrow \{pr\{z_0\}\}$
14: $pNerFail \leftarrow \emptyset$
15: **while** ($Z_{SP} \neq \emptyset$) **do**
16: P_{ss} ←Pop(Z_{SP})
17: $result \leftarrow$ AnalyseSampledState($\mathbf{G}, \mathbf{S}, P_{SF}, Z_{SP}, P_{reach}, P_{ss}, pNerFail, testType, Fsm$)
18: **if** ($\neg result$ AND (testType[DoSSPB] OR testType[DoSDCont31])) **then**
19: **return** False
20: **end if**
21: **end while**

- CheckUntimedControllability: See Wang [21], Algorithm 6.3.
- CheckSDContii: See Wang [21], Algorithm 6.9.
- RecheckNerodeCells: See Wang [21], algorithm 6.13.

Algorithm 6.8 CheckSDCont_SSPB_CSDet($\mathbf{G}, \mathbf{S}$, testType) Part B

```
22: if (pNerFail ≠ ∅) then
23:    result ← RecheckNerodeCells(pNerFail)
24:    if (¬result AND (testType[DoSDCont32] OR testType[DoCSDet])) then
25:       return False
26:    end if
27: else
28:    if (testType[DoCSDet]) then
29:       Fsm.setCSDet(True)
30:       return True
31:    end if
32: end if
33: if (testType[DoSDCont4] AND ¬ CheckSamplingMarkingStates(P_reach)) then
34:    return False
35: end if
36: return True
```

6.8.2 AnalyseSampledState Algorithm

Algorithm 6.9 is based on the *AnalyseSampledState* algorithm from Wang [21]. We modified the algorithm to take the additional parameter *testType*. This parameter allows us to specify which SD controllability or S-singular prohibitable behaviour should be tested. Before, all properties were always tested.

We have also added support for checking the CS deterministic property. As part of this check, we pass the new parameter *Fsm* which is an instance of the **FSMCarrier** data type, defined in Chapter 7. This variable gathers information about the supervisor for use in the TDES to FSM translation algorithms of Chapter 7.

During execution, Algorithm 6.9 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 representing the current state q 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 $B_{map} : \mathcal{N} \to Pred(Q)$ 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.

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 $\text{Occu}_B \subseteq \mathcal{N} \times \text{Pwr}(\Sigma)$.

P_{ss}: A predicate that represents the sampled state that is being analysed inside the algorithm.

As this is mostly the same as Wang's algorithm, we will only discuss the modifications we have made. Please refer to [21] for a detailed explanation of the algorithm.

Algorithm 6.9 evaluates the concurrent behaviour of sampled state q_{ss} (represented by predicate P_{ss}). Basically, the algorithm builds a reachability tree starting at state q_{ss}, until all nodes of the tree terminates at a tick event, or a check fails. To do this, it makes use of the algorithms below. Please refer to Wang [21] for details of these algorithms.

- NextState: See Wang [21], Algorithm 6.11.
- CheckNerodeCells: See Wang [21], Algorithm 6.12.

For lines 1-37 of Algorithm 6.9, we explain the changes that we made to the original *AnalyseSampledState* algorithm from Wang [21]. On line 1, we check if we are doing *DoCSDet* and if so, we then add P_{ss} to the string array named **sampledstates** of *Fsm*. On the lines 19 and 21 we add controllable and uncontrollable events to the appropriate sets of *Fsm* object.

The **if** statement on line 29 will be True only once for every call of the algorithm, and it will be the first pass of the while loop started on line 9. The reason is that only on the first pass will we have $P_q \equiv P_{ss}$. As for the following passes, P_q will be changed by algorithm *NextState*. Please see [21] for details about this algorithm. Therefore, on line 30 we add an item to the *Fsm* object, namely, **poss**. It represents a sampled state P_q and all events with transitions defined at this state. Please see Chapter 7 for more discussion about *FSMCarrier* members.

The **if** statement on line 32 will also be True only one time for the above reason. However, here we are adding data to another member of *Fsm*, namely, **ellig**. This data represents a sampled state and all the prohibitable events with transitions leaving that state.

In the remaining portion of Algorithm 6.9, we are using another data structure, called **Transition**. In the algorithm we will use a temporary variable of type **Transition** called *tr*.

Transition is a data structure that has three member variables:

exit: Is the source sampled state.

enter: Is the target sampled state.

occu: Is the occurrence image (the set of the events in the concurrent string that takes us from the *exit* sampled state to the *enter* sampled state).

In the remaining part, we define the following variable:

- *ret*: This is a temporary Boolean variable.

Algorithm 6.9 AnalyseSampledState($\mathbf{G}, \mathbf{S}, P_{SF}, Z_{SP}, P_{reach}, P_{ss}, pNerFail, testType, Fsm$)
Part A

```
1: if (testType[DoCSDet]) then
2:    Fsm.sampledstates ← Fsm.sampledstates ∪{P_ss}
3: end if
4: B_map ← {(0, P_ss)}
5: B_conc ← ∅
6: B_p ← {0}
7: nextLabel ← 1
8: Occu_B ← {(0, ∅)}
9: while B_p ≠ ∅ do
10:    b ←Pop(B_p)
11:    P_q ← B_map(b)
12:    Σ_poss ← ∅
13:    Σ_Gposs ← ∅
14:    for all σ ∈ Σ do
15:       if (δ̂(P_q, σ) ≢ false) then
16:          Σ_poss ← Σ_poss ∪ {σ}
17:          if (testType[DoCSDet]) then
18:             if (σ ∈ Σ_hib) then
19:                Fsm.c_events ← Fsm.c_events ∪{σ}
20:             else if (σ ≠ tick) then
21:                Fsm.u_events ← Fsm.u_events ∪{σ}
22:             end if
23:          end if
24:       end if
25:       if (δ̂_G(P_q, σ) ≢ false) then
26:          Σ_Gposs ← Σ_Gposs ∪ ({σ} ∩ Σ_hib)
27:       end if
28:    end for
29:    if (testType[DoCSDet] AND P_q ≡ P_ss) then
30:       Fsm.poss ← Fsm.poss ∪{P_q, Σ_poss}
31:    end if
```

Algorithm 6.9 AnalyseSampledState($\mathbf{G}, \mathbf{S}, P_{SF}, Z_{SP}, P_{reach}, P_{ss}, pNerFail, testType, Fsm$)
Part B

```
32:     if (P_q ≡ P_ss) then
33:         Σ_Elig ← Σ_poss ∩ Σ_hib
34:         if (testType[DoCSDet]) then
35:             Fsm.ellig ← Fsm.ellig ∪{P_q, Σ_Elig}
36:         end if
37:     end if
38:     if (testType[DoSDCont31] AND (Σ_poss ∪ Occu_B(b)) ∩ Σ_hib ≠ Σ_Elig) then
39:         return False
40:     end if
41:     ret=NextState(b, Σ_poss, Σ_Gposs, P_q, nextLabel, B_map, B_p, B_conc, P_SF, Z_SP, Occu_B(b))
42:     if (testType[DoCSDet]) then
43:         for all ((b', P_q') ∈ B_conc do
44:             tr.exit ← P_ss
45:             tr.occu ← Occu(b')
46:             tr.enter ← P_q'
47:             Fsm.alltrans ← Fsm.alltrans ∪ {tr}
48:         end for
49:     end if
50:     if (¬ret AND testType[DoSSPB]) then
51:         return False
52:     end if
53: end while
54: CheckNerodeCells(B_conc, Occu_B, pNerFail)
55: return True
```

For lines 38-57, we now explain the changes that we made to the original *AnalyseSampledState* algorithm from Wang [21]. On line 38, if we are testing for SD controllability point iii.1. We return False if the property does not pass. We only return False if testing this point as it does not affect the other properties.

For lines 42-49, we record transition data to *Fsm* if we are testing CS deterministic.Lines 43-48 loops through all the tupples $(b', P_{q'})$ of B_{conc} to construct the transitions from the sampled state P_{ss} to the sampled states $P_{q'}$. For $(b', P_{q'})$, b' is the index of the node in the reachability tree at which a tick is possible and $P_{q'}$ represents the sampled state that *tick* takes us to.

Line 44 assigns P_{ss} to be the **exit** state of the transition data structure. Line 45, assigns all the events of the concurrent string that leads to $P_{q'}$. Line 46 assigns $P_{q'}$ to be the **enter** state of the transition. And finally, line 47 adds the transition data structure to the **alltrans** member of the **Fsm** object. We would like to point out that *tr* does not store a single event transition for a TDES, but stores the occurrence image of how the FSM will transition from one sampled state to the next.

On line 50, if the previous call to *NextState*, on line 41, failed and we are testing S-singular prohibitable behaviour, we then return False.

Chapter 7

VERILOG Translation

7.1 Introduction

In this chapter we will describe how to convert a system's supervisors to SD controllers. First we will present algorithms to convert TDES supervisors to finite state machines (FSM). From this format, we could then translate the FSM into a "C++" software program [4] or into a VERILOG module for implementation in digital logic [9]. In this thesis, we will focus on generating VERILOG code and leave other implementation for future works.

7.2 TDES to FSM

We now will explain how we can take a single supervisor from our system and turn it into a Moore FSM. In Section 7.3, we will show how to convert from an **FSM** to a **VERILOG** module. We will examine a supervisor from the *Lock System* example described in detail in Chapter 8. The supervisor we have chosen is called **SupOpen**, shown in Figure 7.1. This supervisor is responsible for opening the door when the correct combination of a four digit code has been entered.

In Figure 7.1, we show the supervisor **SupOpen**. The initial state is identified by concentric circles. Marked states are identified by a filled circle. This makes state 0 an initial and marked state. The *tick* event is the global clock tick and we associate

its occurrence with the sampling instance of the FSM. All events prefixed with the exclamation mark "!" in a TDES diagram are uncontrollable events, and those without it are prohibitable events.

Listing 7.2 shows the generated code corresponding to the FSM that represents **SupOpen**. The listing is in XML format. Please see [17] for more on XML. The FSM listing starts with the root element <*FSM*> that identifies the corresponding FSM name. The first element is <*ResetState*> which is always the initial state of the TDES. The FSM then lists all the possible activity events as <*Signals*>, where the type can be *IO* for input-output signals or *I* for input only. Prohibitable events are of type *IO* and uncontrollable events will be of type *I*.

The FSM listing then lists the states of the FSM. They correspond to the sampled states of the TDES supervisor. They are included in the element <*States*>. Each state in the FSM has the same name as the state in the TDES. Each state in the FSM has an *outputvector* that lists the prohibitable events that are enabled when the FSM is in this state. This represents the output signals of the FSM. The outputs corresponding to the events listed will be set to true at this state, while the remaining outputs will be set to false. It is common to find that the states of the FSM are less than those of the TDES as TDES typically contain both sampled and non-sampled states, while the FSM contains only states corresponding to sampled states.

The transition section of the FSM listing contains each state followed by the transitions that leave that state. The *inputvector* portion represents the input pattern that must be met when the FSM is at <*StartState*>, in order to follow the transition to the indicated <*EndState*>.

As this FSM has three defined signals, an *inputvector* would be a *3-bit* Boolean vector, where the first bit corresponds to event *open*, the second to event *equal* and the third to event *enter*. If the corresponding bit is true, then that means the event has occurred since the last tick, otherwise the event has not occurred. (i.e the bit is false).

In Listing 7.2, the *inputvector* is listed as a Boolean equation. The interpretation

is that it matches any input vector that makes the equation true. For example, at state 4 we see the input vector "open.!equal.enter". This matches the input vector with the equal bit set to false and the open and enter bits set to true. Here we use the exclamation mark "!" to mean logical negation, the dot "." to mean logical AND, and the plus sign "+" to mean logical OR. The transition label DEF represents the default transition. It is used as shorthand to match all remaining unspecified input combinations.

To determine the input vectors for the transitions leaving a given state, we examine the corresponding state in the TDES, and identify all the concurrent strings leaving that state. The occurrence image of a given string determines the required input vector, and the state the string takes you to determines the <*EndState*> for the transition. For example, the first transition in start state 4 was defined by the concurrent string *open enter tick* that takes us to state 0 of the TDES.

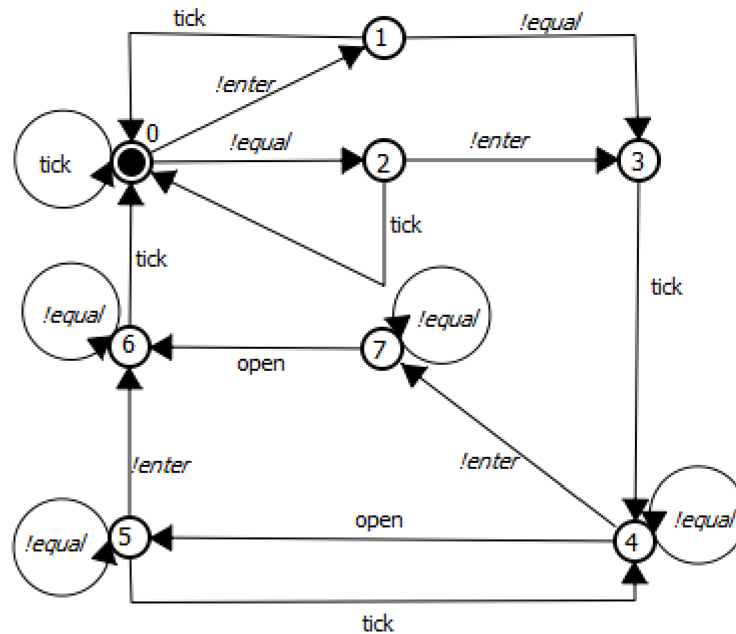


Figure 7.1: Supervisor **SupOpen**

Figure 7.2 shows a graphical representation of the FSM described in Listing 7.2.

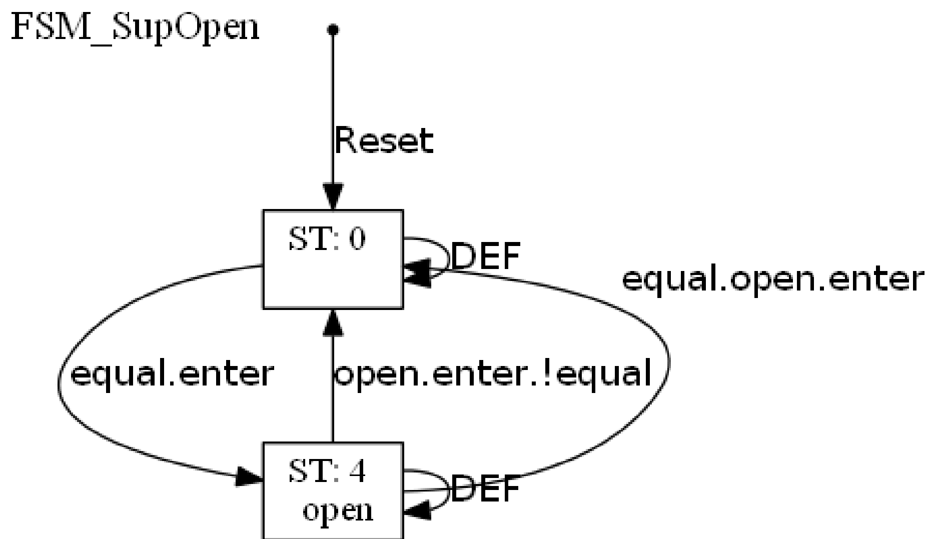


Figure 7.2: FSM of Supervisor **SupOpen**

The initial (Reset) state of the FSM is indicated by the ”Reset” arrow (i.e state 0). For a given state, only the prohibitable names enabled at a state are listed. The remaining arrows show the transitions for the FSM, using the notation introduced in Listing 7.2.

Listing 7.1: The Generated HFSM for Supervisor **SupOpen**

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <FSM Name="SupOpen">
3          <ResetState Name="0"> </ResetState>
4          <Signals>
5                  <Signal Name="open" order="0" type="IO"/>
6                  <Signal Name="equal" order="1" type="I"/>
7                  <Signal Name="enter" order="2" type="I"/>
8          </Signals>
9          <States>
10                 <State Name="0" outputvector=""/>
11                 <State Name="4" outputvector="open"/>
12         </States>
13         <Transitions>
14                 <StartState Name="0">
```

```
15                      <Transition inputvector="!open.equal.enter" endstate="4"/>
16                      <Transition inputvector="DEF" endstate="0"/>
17              </State>
18              <StartState Name="4">
19                      <Transition inputvector="open.!equal.enter" endstate="0"/>
20                      <Transition inputvector="open.equal.enter" endstate="0"/>
21                      <Transition inputvector="DEF" endstate="4"/>
22              </State>
23      </Transitions>
24 </FSM>
```

Listing 7.2 shows another version of Listing 7.1. The difference is how the *inputvector* is created. Here we simply have a comma separated list of the activity events that the concurrent string contains.

Listing 7.2: The Generated FSM for Supervisor **SupOpen**

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <FSM Name="SupOpen">
3          <ResetState Name="0"> </ResetState>
4          <Signals>
5                  <Signal Name="open" order="0" type="IO"/>
6                  <Signal Name="equal" order="1" type="I"/>
7                  <Signal Name="enter" order="2" type="I"/>
8          </Signals>
9          <States>
10                 <State Name="0" outputvector=""/>
11                 <State Name="4" outputvector="open"/>
12         </States>
13         <Transitions>
14                 <StartState Name="0">
15                         <Transition inputvector="equal,enter" endstate="4"/>
16                         <Transition inputvector="DEF" endstate="0"/>
17                 </State>
18                 <StartState Name="4">
19                         <Transition inputvector="open,enter" endstate="0"/>
```

```
20                  <Transition inputvector="equal,open,enter" endstate="0"/>
21                  <Transition inputvector="DEF" endstate="4"/>
22              </State>
23      </Transitions>
24  </FSM>
```

7.3 FSM to VERILOG

In our approach we translate each FSM into a single VERILOG module. The module for supervisor **SupOpen** is shown in Listing 7.3. The module first lists the activity events as two types of arrays: $z[n]$ where n is the number of prohibitable events in the supervisor, and $i[m]$ where m is the number of activity events in the supervisor. Each event in the z array is also in the same place in the i array. For example see event *open* in Listing 7.3. This is because we keep the same order as in the FSM where we list prohibitable events first and then the uncontrollable events.

The module then starts by defining the signals $i, clock, resetn$, z, $z, Next$ and $Current$:

- *clock*: This input is the clock that causes the FSM to change state. In TDES, we equate the rising edge of this signal to the occurrence of the *tick* event.
- i: This input to the controller is a bit vector of size m where each place represents an activity event. A 1 at location $j \in \{0, ..., m-1\}$ means that the corresponding event occurred since the last clock edge, 0 means it did not.
- $resetn$: This asynchronous signal forces the FSM to immediately go to its reset state when the signal is equal to 0.
- z: This registered output stores which prohibitable events are enabled at a given state.
- $Current$: This is the register that saves the current state of the FSM.
- $Next$: This is the register that saves the next state that the FSM will switch to on the next rising edge of the clock as long as input *resetn* is not equal to 0.

We also define several bit patterns as constants to represent needed input, output and state bit vectors:

- IN_h: These are Boolean vectors (i.e contain only 0 and 1), where h corresponds to an *inputvector* from Listing 7.1. In Listing 7.1, there are three such vectors, so we have $h \in \{0, 1, 2\}$. The *inputvector* DEF is handled by simply leaving the current state unchanged if the input does not match one of the **IN_h** parameters. These parameters are the possible input values that cause the SD controller to change from one state to another. For example, in this module we have IN_0 takes the SD controller from ST_0 to ST_4 while both IN_1 and IN_2 takes ST_4 to ST_0.

- ST_j: These are Boolean vectors that represent the states where j is the name of the state in the TDES. It is kept in order to improve readability and to determine by eyes only which TDES states became which states in the SD controller.

- OT_j: These are Boolean vectors that represents the prohibitable events that are enabled (forced) at each state of the FSM. Since we have only one prohibitable event in this SD controller, the length of the vector is only one. Also, the event is enabled only at state 4 (i.e: ST_4).

The logic of the FSM is implemented in two VERILOG **always** blocks (lines 16-45 and lines 46-52). To understand the behaviour of the **always** blocks, it is important to note that variables of type **reg** (i.e: *z, Current*, and *Next*) retain their current value until they are assigned a new value.

An **always** block gives a behavioural representation of a circuit. It does not describe how the physical circuit would execute as sequential code describes the operation of a program, but instead it describes how variables will be assigned in response to changes of other variables.

An **always** block is evaluated when one of the variables in its sensitivity list changes value. For the block at line 46, it will be evaluated when either variable *Current* or i changes. The statements in the block (lines 17-45) are then evaluated

sequentially, and if a variable is assigned a new value during the evaluation, it will take on the value it was last assigned (if assigned more than once) at the end of the block's evaluation.

The purpose of the **always** block at lines 46-52 is to update the current state of the FSM. If *resetn* changes to 0 then *Current* is assigned the reset state irrespective of the *clock* signal. When *resetn* is 1, then a positive edge of the *clock* signal (goes from 0 to 1) causes the current state of the FSM (register *Current*) to be assigned the value of the *Next* register.

The purpose of the **always** block at lines 16-45 is to assign the value of output z based on the current state (register *Current*) of the FSM, and to calculate the state the FSM should switch to on the next clock edge (register *Next*). If the **always** block is activated by variable *Current* changing, this means the FSM has just changed state. If variable i has changed, then our input information (i.e: information about which activity events have occurred) has changed.

This block consists of a case statement (lines 18-44), which switches on the current state (register *Current*) of the FSM. Based on the value of *Current*, output z is assigned, and the expected next state, register *Next*, is assigned based on register *Current* and which input pattern matches. If i does not match any of the defined patterns, then *Next* is not changed. This corresponds to the **DEF** transition discussed earlier.

Listing 7.3: The Generated SD VERILOG Module of Supervisor **SupOpen**

```
1 /*********************************
2 z[0],i[0]:open
3 i[1]: equal
4 i[2]: enter
5 *********************************/
6 module SupOpen(clock,resetn,i,z);
7       input clock,i,resetn;
8       output [0:0] z;
```

```
 9         reg [0:0] z;
10         reg [0:0] Current;
11         reg [0:0] Next;
12         wire [2:0] i;
13         parameter IN_0=3'b110, IN_1=3'b101, IN_2=3'b111;
14         parameter ST_0=1'b0, ST_4=1'b1;
15         parameter OT_0=1'b0, OT_4=1'b1;
16         always @(Current or i)
17         begin
18                 case (Current)
19                         ST_0:
20                         begin
21                                 z=OT_0;
22                                 if(i==IN_0)
23                                 begin
24                                         Next<=ST_4;
25                                 end
26                         end
27                         ST_4:
28                         begin
29                                 z=OT_4;
30                                 if(i==IN_1)
31                                 begin
32                                         Next<=ST_0;
33                                 end
34                                 else
35                                 if(i==IN_2)
36                                 begin
37                                         Next<=ST_0;
38                                 end
39                         end
40                         default:
41                         begin
42                                 Next<=ST_0;
43                         end
```

```
44                  endcase
45          end
46          always @(posedge clock or negedge resetn)
47          begin
48                  if (resetn==0)
49                          Current<=ST_0;
50                  else
51                          Current<=Next;
52          end
53  endmodule
```

7.4 Centeral FSM for SD Controllers

In Chapter 4, we presented two ways to convert a TDES supervisor to an SD controller. When we have only a single supervisor, we would use the monolithic translation method. This would correspond to our discussion in Section 7.3. If we have multiple TDES supervisor in our system we would use the second approach called modular translation. In this approach, we would convert each supervisor as we did in Section 7.3, and then we would add a new item to specify how to combine all if the individual behaviour together. In this section we will present this as an XML description. In the next section, we will express this as a new VERILOG module.

The central FSM for a system is an XML description that defines the global input and output signals for the system as well as the individual FSM for all TDES supervisors in the system. The purpose of this description is to be used in further translations tasks such as converting to VERILOG modules or software source code. Listing 7.4 shows the central FSM description for the Lock System example that is described in Chapter 8. We show it here so we can explain its formal file.

The central FSM file starts by defining the name of the central FSM which is the name of the project (system) being verified. On lines (4-12), the listing identifies the signals in the project. It starts by listing the output signals (**type="O"**), then listing the input signals (**type="I"**).

Starting at line 13, the listing starts to identify each individual FSM in the system, and for each FSM it lists the input/output signals (type="IO"), and then the input only signals.

Listing 7.4: The Generated Central FSM Module

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<FSMMain Name="LockSystem">
  <Signals>
    <Signal Name ="alarm" type="O" order="0"/>
    <Signal Name ="do_change" type="O" order="1"/>
    <Signal Name ="new" type="O" order="2"/>
    <Signal Name ="open" type="O" order="3"/>
    <Signal Name= "change" type="I" order="4" />
    <Signal Name= "enter" type="I" order="5" />
    <Signal Name= "equal" type="I" order="6" />
    <Signal Name= "not_equal" type="I" order="7" />
  </Signals>
  <FSMS>
    <FSM Name="SupAlarm">
      <Signals>
        <Signal Name="alarm"" type="IO"/>
        <Signal Name="not_equal"" type="I"/>
        <Signal Name="enter"" type="I"/>
        <Signal Name="equal"" type="I"/>
        <Signal Name="change"" type="I"/>
      </Signals>
    </FSM>
    <FSM Name="SupChange">
      <Signals>
        <Signal Name="new"" type="IO"/>
        <Signal Name="do_change"" type="IO"/>
        <Signal Name="equal"" type="I"/>
        <Signal Name="change"" type="I"/>
```

```
29        <Signal Name="enter"" type="I"/>
30      </Signals>
31    </FSM>
32    <FSM Name="SupOpen">
33      <Signals>
34        <Signal Name="open"" type="IO"/>
35        <Signal Name="equal"" type="I"/>
36        <Signal Name="enter"" type="I"/>
37      </Signals>
38    </FSM>
39  </FSMS>
40 </FSMMain>
```

7.5 Central Module for SD Controllers

The central module for the SD controller is the orchestrator of the individual FSM such as the FSM for supervisor **SupOpen**. See Listing 7.5 for details. The central SD controller is called the orchestrator purposely as it does not contain any logic other than connecting the system inputs to the modular SD controller sub modules, and combine output from them to form the system's output.

The rules to produce the system output is as follows. The system has one output for every prohibitable event in the system. The SD controller for a given TDES supervisor has one output for each prohibitable event in the supervisor's event set. If no supervisor cares about a given prohibitable event, then it is always enabled. The system output is created by taking the logical AND of all the outputs for the corresponding prohibitable event from every SD controller that has an output for that event.

A very important point here is that there is no overlap between the arrays z and i of the central controller as was the case in the individual modular SD controller. The reason is a bit non-obvious and needs to be understood carefully here.

The system's activity event set consists of prohibitable events and uncontrollable events. For SD controllers, prohibitable events are generated by the controllers, and thus are not inputs to the system. Only uncontrollable events come from outside the central controller. This is why in Listing 7.5 only uncontrollable events are listed as part of input vector i. In the VERILOG implementation of SD controllers, a prohibitable event is assumed to occur when its corresponding output is set to 1. In the central controller, each prohibitable event is assigned to a bit in the output vector z. This vector is then fed to the individual FSM for each supervisor as input for the corresponding prohibitable event. When this input is set to 1, it means the event has been globally enabled, and thus occurred, while the local output for the FSM only represents whether the event is locally enabled by the FSM.

This is why vector i contains four places, from 0 to 3, to accommodate the four uncontrollable events *enter, change, equal,* and **not_equal**. The output vector z contains four places, from 0 to 3, to accommodate the four prohibitable events: *alarm, do_change, new,* and *open.*

It is now clear how the role of the main SD controller is simply to connect and combine the individual FSM for each supervisor so that the result behaves like a monolithic SD controller as per Section 4.5.

Listing 7.5 is based on the example described in Chapter 8. This example contains three supervisors: **SupAlarms**, **SupChange** and **SupOpen**. In Listing 7.5, lines 15-21 define input and output vectors for the FSM for each supervisor.

Lines 23 - 30 instantiate an FSM instance for the FSM SupAlarms. Lines 24 - 28 map system inputs and outputs to the local inputs needed for the supervisor. Line 29 instantiates the FSM and defines the inputs and the outputs connection mapping for the FSM based on the input and output port connections of the VERILOG module that defines the FSM. Listing 7.3 shows the module definition for supervisor FSM **SupOpen**, as an example.

Similarly lines 32-37 instantiate the FSM for **SupChange**, and lines 40-44 instan-

tiate the FSM for supervisor **SupOpen**. For full details of these supervisor FSM, see Chapter 8.

Lines 47-50 define the main controller's outputs in terms of the local outputs of the modular FSM. Normally the output for a given prohibitable event is the logical AND of the corresponding local inputs from each FSM. However for this example, each output has only a local output from a single supervisor, so we only have to map that local output to the corresponding system output.

Listing 7.5: The Generated VERILOG for Central Controller

```
1  /**********************************
2  z[0]: alarm
3  z[1]: do_change
4  z[2]: new
5  z[3]: open
6  i[0]: change
7  i[1]: enter
8  i[2]: equal
9  i[3]: not_equal
10 **********************************/
11 module LockSystem(clock,resetn,i,z);
12   input clock,i,resetn;
13   output [3:0] z;
14   wire [3:0] i;
15   //Submodules input/output Wires:
16   wire [4:0] i_SupAlarms;
17   wire [0:0] z_SupAlarms;
18   wire [4:0] i_SupChange;
19   wire [1:0] z_SupChange;
20   wire [2:0] i_SupOpen;
21   wire [0:0] z_SupOpen;
22
23   //Constructing submodules:
24   assign i_SupAlarms[0]= z[0];
```

```
25    assign i_SupAlarms[1]= i[3];
26    assign i_SupAlarms[2]= i[1];
27    assign i_SupAlarms[3]= i[2];
28    assign i_SupAlarms[4]= i[0];
29    SupAlarms m_SupAlarms(clock,resetn,i_SupAlarms,z_SupAlarms);
30    //-------------------------------------------------------------
31
32    assign i_SupChange[0]= z[2];
33    assign i_SupChange[1]= z[1];
34    assign i_SupChange[2]= i[2];
35    assign i_SupChange[3]= i[0];
36    assign i_SupChange[4]= i[1];
37    SupChange m_SupChange(clock,resetn,i_SupChange,z_SupChange);
38    //-------------------------------------------------------------
39
40    assign i_SupOpen[0]= z[3];
41    assign i_SupOpen[1]= i[2];
42    assign i_SupOpen[2]= i[1];
43    SupOpen m_SupOpen(clock,resetn,i_SupOpen,z_SupOpen);
44    //-------------------------------------------------------------
45
46    //Converging output from submodules:
47    assign z[0]=z_SupAlarms[0];
48    assign z[1]=z_SupChange[1];
49    assign z[2]=z_SupChange[0];
50    assign z[3]=z_SupOpen[0];
51  endmodule
```

7.6 Translation Algorithms

In the algorithms in this section, we use the following identifiers:

- **write**: This is a method that writes to a file on disk. This method is programming language depended. For simplicity, we assume that whenever a program variable appears in the string we pass to **write**, it will automatically be replaced

with a formatted string corresponding to the contents of the variable. Usually when **write** is used, it creates a line of the output file with a newline character at the end. However, sometimes we need to pass the first part of a line to **write**, then algorithmically generate the next part, then we pass the remaining of the line via another call to **write**, (e.g lines 25-35 of Algorithm 7.2). We tell **write** to not generate a new line by passing in parameter that contains an opening ”<” but not a closing ”>”, as we did on line 25 of Algorithm 7.2. When we provided the closing ”>” on line 35, we told write that it should now end the line.

- **FC**: This is a shorthand notation for *FSMCarrier* object (see below) that is associated with the corresponding supervisor TDES.
- **idx**: This is an integer variable.

For each TDES supervisor, we create an SD controller. As was described in Chapter 6, as part of the algorithms to verify the SD controllable properties iii.1 and iii.2, we can use these algorithms to gather information about the desired supervisor and store the information in an **FSMCarrier** object for each supervisor. The **FSMCarrier** object is a C++ class that we have developed specially for this purpose, and you can view its details as a UML [1] diagram in Figure 7.3.

We save an array of objects for this class, one for each modular TDES supervisor in our system. We construct each object when we verify that each supervisor is CS deterministic (see Algorithm 6.8 in Chapter 6). We will use the information in these object to construct the FSM for each supervisor. We can then use the FSM information to implement the FSM as a VERILOG module.

In Figure 7.3, we define the following members:

- **name**: A string that is the name of the SD controller which matches the name of the TDES supervisor.
- **mCSDet**: A boolean variable that indicates if the TDES is CS deterministic or not.

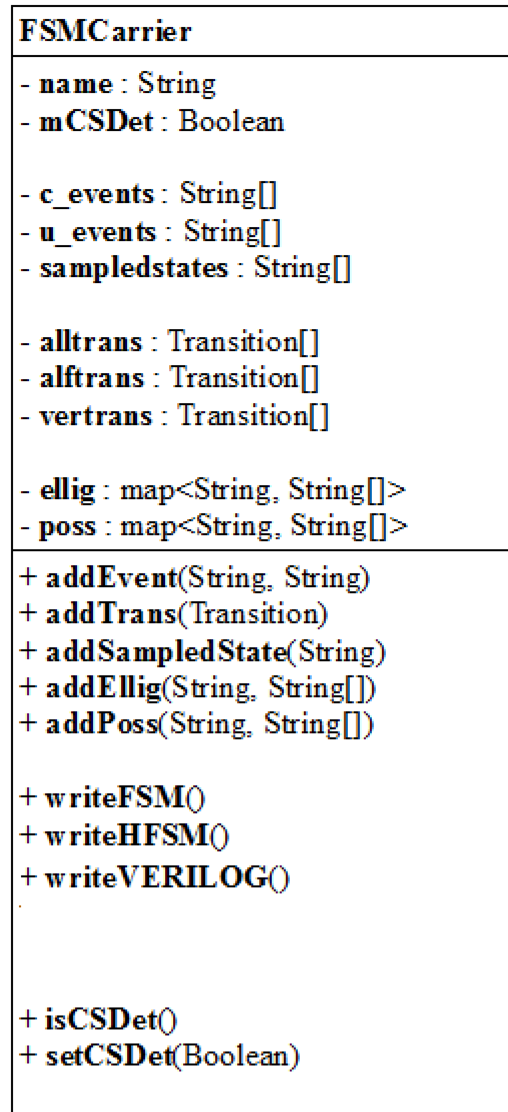


Figure 7.3: FSMCarrier Class UML Diagram

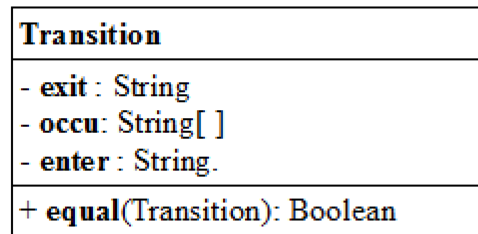


Figure 7.4: Transition Struct UML [1] Diagram

- **c_events**: This is an array of strings that lists all the prohibitable events in the TDES supervisor.
- **u_events**: This is an array of strings that lists all the uncontrollable events in the TDES supervisor.
- **sampledstates**: This is an array of strings that lists all the sampled states in the TDES supervisor. These are states in the TDES that are reached by a *tick* event, plus the initial state. The first state, **sampledstates[0]**, is always the initial state of the TDES supervisor.
- **alltrans**: This is an array of transitions for the SD controller. As discussed in Chapter 4, SD controller state changes are defined by concurrent strings, not individual events. We thus define our transitions as shown in the Transition UML diagram shown in Figure 7.4. For our purpose, we define a transition using the indicated three variables and the one method, **equal**. The first variable is called **exit**. This is the sampled state that the transition starts from. The next variable is called **enter**. This is the sampled state that the transition goes to. The third variable, **occu**, is an array of strings that contains the occurrence image of the concurrent string that takes our corresponding TDES supervisor from our **exit** state, to our **enter** state. As defined in Chapter 3, the occurrence image of a string is the set of events that the string is composed of. We do not include the *tick* event in our occurrence image as it is understood to be present. The **equal** method is used to compare the current transition with another transition. It returns true when **exit**, **enter**, and **occu** is the same in both transitions. All transitions in **alltrans** are FSM transitions and each one

is represented by a concurrent string, not a single event as in the TDES transitions. All the transitions in **alftrans** and **vertrans** are also FSM transitions.

- **alftrans**: This is an array of transitions that represents activity loops (see Definition 2.2.15 in Chapter 2). These are transitions containing no *tick* event, and when the **exit** state and **enter** state are the same state. These states are almost always self loops transitions only, since when creating TDES supervisors, the designer should make sure the TDES is non-self loop ALF. This is not a necessity but it is recommended. This array has no big impact on our algorithms, we are using **alltrans** instead.

- **vertrans**: This is the set of all transitions in **alltrans**, that are not in **alftrans**. We call them **vertrans** to indicate that they are the transitions that will be used when generating the VERILOG modules.

- **ellig**: This item maps the name of sampled states to an array of strings which represents the names of the prohibitable events that are enabled at this state. This means the SD controller must force and enable these events at the indicated state. We use this mapping to determine the enablement of outputs at each state in the FSM code.

- **poss**: This item maps the name of sampled states to an array of strings which represents the names of the activity events that are possible at this state.

- **addEvent**: This is a method to add an event to either **c_events** or **u_events**, depending on the event type. The first string is the name of the event and the second is the event type (uncontrollable or prohibitable).

- **addTrans**: This is a method to add a transition. It takes a Transition struct as a parameter. The transition is added to **alltrans** array, then if it is an activity

loop, it is added to **alftrans**, otherwise it is added to **vertrans**.

- **addSampledState**: This is a method to add a sampled state to the **sampled-states** array. It uses the same name as given in the TDES automaton for clarity and readability.

- **addEllig**: This method adds one item to the **ellig** map. It takes a string representing the name of a sampled state and an array of strings representing the set of prohibitable events eligible at this state.

- **addPoss**: This method adds one item to the **poss** map. It takes a string that represents the name of a sampled state and an array of strings representing the set of events that can occur at this state in the TDES automaton.

- **writeFSM**: This method implements the algorithm for creating the FSM description, and writes it to a file.

- **writeHFSM**: This method is another version of the above **writeFSM** method. The only difference is that this method formats the **inputvector** as a Boolean expression. It negates all events that must not happen at a given state. It uses a notation that is easier for humans to read.

- **writeVERILOG**: A method that generates the VERILOG module code for the SD controller.

- **isCSDet**: This is a boolean method returns the value of member *mCSDet*.

- **setCSDet**: A method to set the value of the mCSDet boolean variable.

In the following sections, we will present algorithm algorithms for members: writeFSM, writeHFSM, mainVERILOG, writeVERILOG, and mainFSM.

7.6.1 writeFSM() Algorithm

Algorithm 7.1 is responsible for creating FSM files such as Listing 7.2. It uses the **FsmCarrier** object associated with each supervisor TDES to translate the TDES supervisor to an FSM.

Algorithm 7.1 writeFSM() Part A

```
1: write(<?xml version=1.0 encoding=UTF-8 standalone=yes?>)
2: write(<FSM Name= FC.name >)
3: write(<ResetState Name= FC.sampledstates[0] ></ResetState>)
4: write(<Signals>)
5: idx:=0
6: for all (e ∈ FC.c_events) do
7:    write(<Signal Name= e order= idx type=IO/>)
8:    idx:= idx+1
9: end for
10: for all (e ∈ FC.u_events) do
11:    write(<Signal Name= e order= idx type=I/>)
12:    idx:= idx+1
13: end for
14: write(</Signals>)
```

Lines 1-4 are for writing the first 4 lines of Listing 7.2. Line 2 uses **FC.name** to write the name of the TDES supervisor as the name of the FSM. On line 3, we use **FC.sampledstates[0]** to write the reset state of the FSM, which is the initial state of the TDES. On line 4, we write the beginning tag of the signals element. From lines 6-9, we loop through all the prohibitable events of the TDES, namely **FC.c_events**, and write them out with type **"IO"**. We do the same from lines 10-13, but we write the uncontrollable events in the TDES, namely **FC.u_events**.

Lines 15-19 are responsible for writing the states of Listing 7.2. We loop through each sampled state, use the mapping **FC.ellig** to get the eligible prohibitable events at each state, and then write it as on **outputvector** formatted as a comma separated list.

Algorithm 7.1 writeFSM() Part B

```
15: write(<States>)
16: for all (q ∈ FC.sampledstates) do
17:   write(<State Name= q outputvector= FC.ellig[q] />)
18: end for
19: write(</States>)
20: write(<Transitions>)
21: for all (q ∈ FC.sampledstates) do
22:   write(<StartState Name= q >)
23:   for all (t ∈ FC.alltrans) do
24:     if (t.exit = q AND t.enter ≠ q) then
25:       write(<Transition inputvector= t.occu endstate= t.enter />)
26:     end if
27:   end for
28:   write(<Transition inputvector=DEF endstate= q />)
29: end for
30: write(</Transitions>)
31: write(</FSM>)
```

Lines 20-30 are responsible for creating the *Transitions* block in the FSM description, as in Listing 7.2. We loop through the sampled states, and for each one we build a <*StartState* > block and then fill it with all the transitions from this state to other states. For each start state, we loop through **FC.alltrans** and find those transitions whose *exit* state is this state, and their *enter* state is different to avoid self loops. When we find one, we write it to the file as on line 25 and we use **Transition.occu** as the *inputvector* for this transition, and **Transition.enter** as the *endstate*.

This algorithm creates an FSM file such as in Listing 7.2. However, when it writes

the transition it does not write the transition as it is shown on Listing 7.1, line 19; namely as Boolean functions like: *!equal.open.enter*. Instead it writes a comma separated list of the events that must happen in the transition, ignoring those that must not happen; thus would output: *open,enter*.

When we finish writing all the actual transitions from this state to other possible states, we go to line 28 and we write the default (DEF) transition as a self loop. This is a shorthand to match any input vector not already specified.

Line 31 closes the root tag of the FSM description.

7.6.2 writeHFSM() Algorithm

Algorithm 7.2 is responsible for creating FSM files with input vectors formatted as logical functions, as shown in Listing 7.1. It uses the **FsmCarrier** object associated with each supervisor TDES.

In this algorithm we use the following identifiers, as well as the previous ones mentioned in Section 7.6.

- **in_vector**: This is a string variable.
- **trimlast**: This is a string processing function that deletes the last letter from the provided string.

This algorithm is identical to the previous **writeFSM**, with the exception of how we write the *Transitions* block to the file. Therefore, we will only explain how the algorithm writes this block.

For lines 20-24, it does the same things as in Algorithm 7.1. Lines 25-35 are responsible for writing only one line in the FSM description, and this is the *Transition* line that corresponds to line 19 of Algorithm 7.2. The *outputvector* in this line is *!equal.open.enter*. It means the *equal* event must not happen, the *open* event must

Algorithm 7.2 writeHFSM() Part A

1: write(<?xml version=1.0 encoding=UTF-8 standalone=yes?>)
2: write(<FSM Name= FC.name >)
3: write(<ResetState Name= FC.sampledstates[0] ></ResetState>)
4: write(<Signals>)
5: idx:=0
6: **for all** ($e \in FC.c_events$) **do**
7: write(<Signal Name= e order= idx type=IO/>)
8: idx:= idx + 1
9: **end for**
10: **for all** ($e \in FC.u_events$) **do**
11: write(<Signal Name= e order= idx type=I/>)
12: idx:= idx + 1
13: **end for**
14: write(</Signals>)
15: write(<States>)
16: **for all** ($q \in FC.sampledstates$) **do**
17: write(<State Name= q outputvector= FC.ellig[q] />)
18: **end for**
19: write(</States>)

happen and the *enter* event must happen in order to match this input pattern.

To generate an *inputvector* so formatted, the algorithm fetches each transition from **alltrans**. It makes sure that the **exit** state is the current one, and the **enter** state is not the same. Then on line 26, it creates an empty string named **in_vector**. From lines 27-33, we loop through all the activity events in the FSM. If the event is not in **t.occu** then this event must not happen in this particular transition, so we put the negation identifier **"!"** before it. We then put the event, followed by the **AND** identifier, **"."**. If the event is in **t.occu**, then this event must happen in this particular transition, therefore, we list the event and follow it by the **AND** operator, ".".

Line 34 trims the last **AND** operator from the string and then we write the remainder of the string to the file (line 38).

We note that on lines 29 and 31, we specify string concatenation by combining strings using the "+" operator.

Algorithm 7.2 writeHFSM() Part B

```
20: write(<Transitions>)
21: for all (q ∈ FC.sampledstates) do
22:    write(<StartState Name= q >)
23:    for all (t ∈ FC.alltrans) do
24:       if (t.exit = q AND t.enter ≠ q) then
25:          write(<Transition inputvector=)
26:          in_vector:=""
27:          for all (e ∈ FC.c_events ∪ FC.u_events) do
28:             if (e ∈ t.occu) then
29:                in_vector := in_vector + e + "."
30:             else
31:                in_vector := in_vector + "!" + e + "."
32:             end if
33:          end for
34:          trimlast (in_vector)
35:          write(in_vector endstate= t.enter />)
36:       end if
37:    end for
38:    write(<Transition inputvector=DEF endstate= q />)
39: end for
40: write(</Transitions>)
41: write(</FSM>)
```

7.6.3 mainVERILOG() Algorithm

Algorithm 7.3 is called **mainVERILOG**. It is responsible for creating the VERILOG modules for the FSM for each supervisor, and the main VERILOG module that uses these submodules. For example, if this algorithm was applied to the Lock System example in Chapter 8, it would generate a VERILOG module for each supervisor (**SupOpen**, **SupChange**, and **SupAlarm**) and a central controller module to combine the individual FSM together. Listing 7.3 shows the module for **SupOpen** while Listing 7.5 shows the central controller module.

This algorithm uses the following identifiers. These identifiers may also be used in the algorithms that follow:

- **fsmcarriers**[]: This is an array of *FSMCarrier* objects. It is created when testing whether each supervisor in the project is CS deterministic. We will access is as if it is a set.

- Σ_{hib}: This is the set of all prohibitable events in the project.
- Σ_u: This is the set of all uncontrollable events in the project.
- Σ_{free}: This is the set of all prohibitable events in the project that are not part of the event set of any TDES supervisor in the project.
- **idx, ord and counter**: These are miscellaneous integer variables used in the algorithm.
- **projName**: This is a string containing the name of the project that is begin verified.
- **indexOf**: This is a utility function that takes a set as the first parameter, and an set item as the second parameter. It then returns the index of the item in the set. It returns -1 if the item is not in the set.
- $|object|$: This is a size function that returns the number of items in the list, array or set.

- **write**: This is a method that writes to a file on disk. This method is programming language depended. For simplicity, we assume that whenever a program variable appears in the string we pass to **write** it will automatically be replaced with a formatted string corresponding to the contents of the variable.

In VERILOG any line that starts with "//" is a comment that is not compiled by VERILOG. Also, any block of text between "/*" and "*/" is also comment text.

We now describe how lines 1-24 of the **mainVERILOG()** algorithm work. Lines 1-5 check that all the TDES supervisors are CS deterministic. If any are not, then the algorithm exits. Lines 6-8 loop through all the *FSMCarrier* objects in array **fsmcarriers**[], and call the method **writeVERILOG** for each one. This writes the VERILOG module for the corresponding TDES supervisor to a separate file. See Allgorithm 7.4 for more details on **writeVERILOG**.

On lines 9-24, the algorithm generates the comments for the beginning of the SD controller central module shown on lines 1-10 of Listing 7.5. These comments are not compiled by ALTERA [11] or any other VERILOG compiler. They are only for clarity and readability. They show the event ordering by listing the prohibitable event positions in bit vector z, then the uncontrollable event positions in bit vector i.

We now describe the behaviour of lines 25-50 of Algorithm 7.3. Lines 25-30 in the algorithm generate lines 11-14 in Listing 7.5. For lines 31-36, the algorithm loops through the objects in **fsmcarriers**[] array to define the input and output vectors for each one. See lines 16-21 of the central module from Listing 7.5. It uses $|fc.c_events| + |fc.u_events|$ which returns the number of prohibitable events plus the number of uncontrollable events for the TDES. We subtract 1 from each size because in our VERILOG code bit vector indices start at 0.

For lines 37-50, the algorithm writes lines 24-43 of Listing 7.5. In these lines, we contstruct the input bit vector for each FSM from the z and i bit vectors of the central module. For each supervisor object in the **fsmcarriers**[] array, we first loop through all the prohibitable events in the TDES supervisor. We find the index of the event in the bit vector z and save it in **idx**. We then use **idx** to assign the prohibitable

Algorithm 7.3 mainVERILOG() Part A

1: **for all** (fc $\in$ fsmcarriers[]) **do**
2:     **if** ($\neg$fc.isCSDet()) **then**
3:         **return**
4:     **end if**
5: **end for**
6: **for all** (fc $\in$ fsmcarriers[]) **do**
7:     fc.writeVERILOG()
8: **end for**
9: write (/************************)
10: idx:=0
11: **for all** (e $\in \Sigma_{hib}$) **do**
12:     **if** (e $\in \Sigma_{free}$) **then**
13:         write (z[idx]: e /FREE EVENT)
14:     **else**
15:         write (z[idx]: e)
16:     **end if**
17:     idx:= idx+ 1
18: **end for**
19: idx:= 0
20: **for all** (e $\in \Sigma_u$) **do**
21:     write (i[idx]: e)
22:     idx:= idx+ 1
23: **end for**
24: write (************************/)

event in bit vector z to the corresponding one in the TDES. We do the same for the uncontrollable events, however we use the correct array for the uncontrollable events in the central module, namely i.

When all the inputs of the TDES supervisor have been assigned, we then define an instantiation of the individual FSM module as seen on lines 29, 37 and 43 of Listing 7.5. We do this by giving the variable the same name as the individual TDES supervisor, but adding the prefix **"m_"**. This occurs on line 49 of our algorithm.

We use the following variable in the next part of the algorithm:

- **size**: This is an integer variable.

We now discuss the last part of Algorithm 7.3, shown on lines 51-68. This is where we assign each event in the output bit vector z of the central module a Boolean value. It is assigned the conjunction of the corresponding local output for each FSM that cares about the event. If the event is not part of the event set of any supervisor, then this output is set to 1.

In this part of the algorithm we use the following identifier as well as the previously mentioned ones:

- **tmp_assign**: This is a temporary string variable to hold the output assignment before writing it to the file.

On lines 52-67, we loop through the prohibitable events of the system. For each event, we check if it is in Σ_{free}. If it is, we then assign 1 to that output. If this is not the case, we then do the following: we create the left hand portion of the assignment string, namely "assign z[ord]=", as shown on line 56 in the algorithm. We then loop through all the FSMs and get the index of the event in them and save it in **idx**. If **idx** is greater than or equal to 0 (means the FSM contains the event), we add the event from the SD controller and then add **"&"** which is the bitwise logical AND operator in VERILOG.

Algorithm 7.3 mainVERILOG() Part B

```
25: write (module projName(clock,resetn,i,z);)
26: write (input clock,i,resetn;)
27: size := |Σ_hib| − 1;
28: write (output [size:0] z;)
29: size := |Σ_u| − 1;
30: write (wire [size:0] i;)
31: for all (fc ∈ fsmcarriers[]) do
32:     size := |fc.c_events| + |fc.u_events| − 1
33:     write (wire [size:0] i_fc.name;)
34:     size := |fc.c_events| − 1
35:     write (wire [size:0] z_fc.name;)
36: end for
37: for all (fc ∈ fsmcarriers[]) do
38:     ord := 0
39:     for all (e ∈ fc.c_events) do
40:         idx := indexOf(Σ_hib, e)
41:         write (assign i_fc.name[ord]= z[idx];)
42:         ord := ord + 1
43:     end for
44:     for all (e ∈ fc.u_events) do
45:         idx := indexOf(Σ_u, e)
46:         write (assign i_fc.name[ord]= i[idx];)
47:         ord := ord + 1
48:     end for
49:     write (fc.name m_fc.name(clock,resetn,i_fc.name,z_fc.name);)
50: end for
```

Finally, we write the ”end module” keyword on line 68, to close the VERILOG module.

Algorithm 7.3 mainVERILOG() Part C

```
51: ord := 0
52: for all (e ∈ Σ_hib) do
53:    if (e ∈ Σ_free) then
54:       write (assign z[ord]= 1;)
55:    else
56:       tmp_assign := ”assign z[ord]=”
57:       for all (FC ∈ fsmcarriers[]) do
58:          idx := indexOf(FC.c_events, e)
59:          if (idx≥ 0) then
60:             tmp_assign := tmp_assign + ”z_” + FC.name + ”[”+ idx +”]” + ”&”
61:          end if
62:       end for
63:       trimlast (tmp_assign)
64:       write (tmp_assign ;)
65:       ord := ord + 1
66:    end if
67: end for
68: write (endmodule)
```

7.6.4 writeVERILOG() Algorithm

Algorithm 7.4 generates the VERILOG FSM module from an *FsmCarrier* for each TDES supervisor. As an example, Listing 7.3 shows the module generated for the supervisor **SupOpen**.

The VERILOG modules that were generated by this algorithm for the supervisors in Chapter 8 have been tested on ALTERA [11] web edition. All the modules generated by this algorithm were compiled without any errors and without any manual

modifications.

In this algorithm we use the following utility functions:

- **binaryInputVector()**:

  This is a utility function that takes a set of activity events, and returns a binary bit vector. Its size is the number of activity events in the event set of the TDES supervisor. All the bits corresponding to the events that are passed as a parameter, namely **Transition.occu**, are given the value "1". Each of the remaining bits are given the value "0" indicating the event must not occur in this transition.

- **binaryOutputVector()**:

  This is a utility function that takes a set of prohibitable events, and returns a binary bit vector. Its size is the number of prohibitable events in the event set of the TDES supervisor. All the bits corresponding to the events that are passed as a parameter, namely eligible events in *FC.ellig[q]*, are given the value 1,. Each of the remaining bits are given the value zero.

- **calcStateSize()**:

  Takes a positive integer as input and calculates the minimum number of bits needed to represent the inputted number of states.

- **binaryStateVector()**:

  This is a utility function that returns a binary bit vector. The bit vector is a binary representation to a state in the SD controller. It takes the state order in the TDES and returns its binary equivalent. The number of bits in the created bit vector is equal to to **calcStateSize(|FC.sampledstates|)**.

- **parameterizedTransition()**:

  This utility function takes a **Transition** structure as input and converts the transition's **exit, enter,** and **occu** members to strings that match the bit vector parameters used in the VERILOG FSM module code (i.e see lines 13-15 of Listing 7.3). For example, say we have **Transition tr**, with **tr.exit=***"2"*,

tr.enter="*3*", and **tr.occu**[]={"*event1*", "*event3*"}. If we applied this function to $tr \in FC.alltrans$, we would get a new transition **tr1**, with **tr1.exit=**"*ST_2*", **tr1.enter=**"*ST_3*", and **tr1.occu[0]=**"*IN_x*" where **x** would corresponds to the value of **idx** on line 30 of Algorithm 7.4 when the correspondent parameter for our occurrence image was defined for **tr**.

- **Current**:

  This is the current state in the SD controller.

- **Next**:

  This is the next state that will be assigned to *Current* on the next positive edge of the clock.

- **tmpString**: A temporary string variable.

- **size**: An integer number used temporarily.

Algorithm 7.4 writeVERILOG() Part A

```
1: idx:= 0
2: write(/********************)
3: for all (e ∈ FC.c_events) do
4:    write (z[idx],i[idx]: e)
5:    idx:= idx + 1
6: end for
7: for all (e ∈ FC.u_events) do
8:    write (i[idx]: e)
9:    idx:= idx + 1
10: end for
11: write(***************/)
```

The first part of Algorithm 7.4 (lines 1-11), is responsible for creating the first set of comment lines in Listing 7.3, namely lines 1-5 of the listing. These comments are ignored by the VERILOG compiler and they are only for readability and clarity.

Lines 3-6 in the algorithm, loop through the prohibitable events. Each one is allocated two places, one place in the output vector z, and the other place is in the input vector i. On lines 7-10, we do the same for the uncontrollable events, however these events are allocated places in the input vector only, namely i.

Algorithm 7.4 writeVERILOG() Part B

12: write(module FC.name(clock,resetn,i,z);)
13: write(input clock,i,resetn;)
14: size := $|FC.c_events| - 1$
15: write(output [size:0] z;)
16: size := $|FC.c_events| - 1$
17: write(reg [size:0] z;)
18: size := calcStateSize($|FC.sampledstates|$)
19: write(reg [size:0] Current;)
20: write(reg [size:0] Next;)
21: size := $|FC.c_events| + |FC.u_events| - 1$
22: write(wire [size:0] i;)

Lines 12-22 of Algorithm 7.4, is responsible for creating lines 6-12 of Listing 7.3. We use the size function $|FC.c_events| - 1$ to determine the size of the output vector z. The "-1" is because the vector base starts at 0. As i is the input bit vector, we determine its size by $|FC.c_events| + |FC.u_events| - 1$, namely the number of activity events in the event set of the TDES supervisor. On line 18, we pass the number of states to the function **calcStateSize** to calculate how many bits we need to represent a state.

Lines 23-39 of Algorithm 7.3, are responsible for creating line 13 of Listing 7.3. Namely parameters *IN_idx*, such that **idx** is a sequence number for each transition in the FSM.

On lines 26-38, we loop through the array **FC.alltrans** and give each non-selfloop transition a sequence number **idx**. On line 27, we check if we have a self loop transition, namely **tr.exit=tr.enter**. If this is the case, we jump to the beginning of the

Algorithm 7.4 writeVERILOG() Part C

```
23: tmpString := "parameter "
24: idx:= 0
25: counter:= 0
26: for all (tr ∈ FC.alltrans) do
27:    if (tr.exit = tr.enter) then
28:       continue
29:    end if
30:    tmpString := tmpString + "IN_" + idx + "=" + binaryInputVector(tr.occu)
31:    counter:= counter + 1
32:    if (counter <|FC.alltrans|) then
33:       tmpString := tmpString + ","
34:    else
35:       tmpString := tmpString + ";"
36:    end if
37:    idx:= idx + 1
38: end for
39: write (tmpString)
```

for loop using *continue*. For each non-selfloop transition, we call the utility function **binaryInputVector** with the **tr.occu** parameter. See the description of the utility function **binaryInputVector** above. We then write "," if we have more transitions or ";" if this is the last transition. On line 39, we write the constructed string to the file.

Algorithm 7.4 writeVERILOG() Part D

```
40: tmpString := "parameter "
41: idx:= 0
42: counter:= 0
43: for all (q ∈ FC.sampledstates) do
44:    tmpString := tmpString + "ST_" + q +"=" + binaryStateVector(idx))
45:    counter:= counter + 1
46:    if (counter <|FC.sampledstates|) then
47:       tmpString := tmpString + ","
48:    else
49:       tmpString := tmpString + ";"
50:    end if
51:    idx:= idx + 1
52: end for
53: write (tmpString)
```

Lines 40-53 of Algorithm 7.4, are responsible for creating line 14 of Listing 7.3. Namely ST_q, such that **idx** is a sequence number for each state in the SD controller.

For lines 43-52, we loop over the array **FC.sampledstates**, and we give each state a sequence number **idx**. For each state, we call the utility function **binaryStateVector** with the **idx** parameter. See description of **binaryStateVector** above. We then write "," if we have more states or ";" if this is the last state. We use **idx** as a parameter and not the state itself **(q)** because states can be identified by string names and not numbers. On line 53, we write the constructed string, *tmpString*, variable to the file.

Algorithm 7.4 writeVERILOG() Part E

```
54: tmpString :=”parameter ”
55: counter := 0
56: for all (q ∈ FC.sampledstates) do
57:    tmpString := tmpString + ”OT_” + q + ”=” + binaryOutputVector(ellig[q]))
58:    counter := counter + 1
59:    if (counter <|FC.sampledstates|) then
60:       tmpString := tmpString + ”,”
61:    else
62:       tmpString := tmpString + ”;”
63:    end if
64: end for
65: write (tmpString)
```

Lines 54-65 of Algorithm 7.4, are responsible for creating line 15 of Listing 7.3. Namely OT_q, which will represent the output bit vector for state **q**.

For lines 56-64, we loop through the array **FC.sampledstates** and for each state we call the utility function **binaryOutputVector** with **FC.ellig[q]** as the parameter. See function description above. We then write ”,” if we have more states or ”;” if this is the last state. On line 65, we write the constructed string, *tmpString*, to the file.

Lines 66-96 of Algorithm 7.4, are responsible for creating the first **always** block of Listing 7.3 (lines 16-45).

Line 66 writes the **always** block with *Current* and i as its sensitivity parameters. Lines 68-95 define a case statement that switches on the *Current* variable. From lines 69-89, we loop through all the sampled states of the SD controller. For each state, we write the state's binary representation as the value of the above case statement, namely **ST_q** as in line 70. Line 72 changes the output of the SD controller to be the corresponding state output. For lines 74-86, we loop through all the transitions in **FC.alltrans**. We use function **parameterizedTransition** to first construct transitions to match the parameters we created earlier for the corresponding Binary vectors.

Algorithm 7.4 writeVERILOG() Part F

```
66: write(always @(Current or i))
67: write(begin)
68: write(case Current)
69: for all (q ∈ FC.sampledstates) do
70:    write(ST_q:)
71:    write (begin)
72:    tmpString := ”z= OT_” + q +”;”
73:    write (tmpString)
74:    idx:= 0
75:    for all (tr ∈ FC.alltrans) do
76:       if (tr.exit = q) then
77:          tr= parameterizedTransition(tr)
78:          idx:= idx + 1
79:          if (idx >1) then
80:             write (else)
81:          end if
82:          write (if (i == tr.occu[0]))
83:          write (begin)
84:          write (Next <= tr.enter;)
85:          write (end)
86:       end if
87:    end for
88:    write (end)
89: end for
90: write (default:)
91: write (begin)
92: tmpString := ”Next <= ST_” + FC.sampledstates[0] +”;”
93: write (tmpString)
94: write (end)
95: write (endcase)
96: write(end)
```

See the function definition for more information.

Line 77 makes sure that the **exit** state is equal to the sampled state we are currently using in this iteration, namely **q**. If **idx** is greater than 1, then this means that this is not the first condition of the **if** statement, and this means we have to put an **else** part.

For lines 81-84, we write the VERILOG if statement that says, if the input i is equal to the current transition's concurrent string **occu[0]**, then we have to assign the variable **Next** the value of the transition's **enter** state, which is the state we are going to. The reason we are using the first item in **occu[0]** is because when the function **parameterizedTransition** converts the transition, it puts the parameter string in this location.

Lines 90-94 are very important and this might not be obvious at first glance. These lines are evaluated by VERILOG compiler when *Current* has no value similar to any one in the **FC.sampledstates**. This condition should never be reached but we put it here as a fail safe.

Algorithm 7.4 writeVERILOG() Part G

```
97:  write(always @(posedge clock or negedge resetn))
98:  write(begin)
99:  write (if (resetn==0))
100: write (Current <= ST_FC.sampledstates[0])
101: write (else)
102: write (Current <= Next;)
103: write(end)
104: write(endmodule)
```

The final part of Algorithm 7.4 (lines 97-104) is responsible for writing the second **always** block of Listing 7.3. Line 97 shows that the **always** block is evaluated when the positive edge of *clock* occurs, or the negative edge of *resetn* occurs.

Line 99 says if **"resetn==0"**, we then assign **FC.sampledstates[0]** to *Current*, namely we are resetting the system. If not then, this evaluation of the **always** block is evaluated because of a positive edge of the clock has occurred, thus we have to set *Current* equal to *Next*.

7.6.5 mainFSM() Algorithm

Algorithm 7.5 is responsible for creating the central FSM for the system. For an example, Listing 7.4 shows the results of running this algorithm on the example system of Chapter 8.

On lines 1-13 of Algorithm 7.5, it writes to a file the header part of the XML code. This corresponds to lines 1 of Listing 7.4. On line 2, it uses the project name as the name of the central FSM. On lines 5-8, it loops through the prohibitable events in the system and writes them as signals. The name of the signal is the same as the event, while the type of the signal is **O** meaning these signals are output signals.

Then the algorithm does the same for the uncontrollable events (lines 9-12), however it identifies these events as type **I**, input only events.

The remaining part of the algorithm starts a new XML element, namely **FSMS** to define all the individual FSM of the project, one for each supervisor in the project. It uses array *fsmcarriers[]* to access data structure, *FC*, for each supervisor. For each object *(FC)* the algorithm starts a new element with *"FSM Name= FC.name"*. It then executes two sequential loops, one to write all the input/output signals of the FSM from the array **FC.c_events**, and one to to write all the input signals from the array **FC.u_events**.

Algorithm 7.5 mainFSM() Part A

1: write(<?xml version=1.0 encoding=UTF-8 standalone=yes?>)
2: write(<FSMMain Name= projName >)
3: write(<Signals>)
4: idx:=0
5: **for all** ($e \in \Sigma_{hib}$) **do**
6: write(<Signal Name= e type=O order= idx />)
7: idx:= idx+1
8: **end for**
9: **for all** ($e \in \Sigma_u$) **do**
10: write(<Signal Name= e type=I order= idx />)
11: idx:= idx+1
12: **end for**
13: write(</Signals>)

Algorithm 7.5 mainFSM() Part B

14: write(<FSMS >)
15: **for all** (FC $\in$ fsmcarriers[]) **do**
16: write(<FSM Name= FC.name>)
17: write(<Signals>)
18: **for all** (e $\in$ FC.c_events) **do**
19: write(<Signal Name=e type=IO />)
20: **end for**
21: **for all** (e $\in$ FC.u_events) **do**
22: write(<Signal Name=e type=I />)
23: **end for**
24: write(</Signals>)
25: write(</FSM >)
26: **end for**
27: write(</FSMS >)
28: write(</FSMMain >)

7.7 Removing Redundant Transitions

Figure 7.2 shows a shortcoming of our algorithms. We see two transitions from state 4 to state 0 that differ only by *equal* and *!equal*. This implies the value of input *equal* is not important here and can be removed. We can thus replace both transitions with a single transition labelled *open.enter*. There are likely other simplifications that can be made.

A useful future improvement would be to remove such redundancies in the list of transitions while keeping equivalent behaviour. This would produce more compact FSM listings and VERILOG modules. This was left as future work due to time constraints and the fact that the VERILOG compiler should make these simplifications automatically so this should not make a difference in the final results.

Chapter 8

Lock System Example

In this chapter, we will review and explore an example of designing systems for SD supervisory control. We will describe the system problem, the components of the system and then we will show the translation results to SD controllers. We will show timing results for the verification software comparing modular methods to monolithic algorithm.

The Lock System example is an automatic door system that is locked by a secret code, composed of four binary digits. The external agent here is a human that interacts with the lock system. The external agent can unlock the door, or change the saved code, but only if they enter the correct code first.

The purpose of this example is to design an FSM for this system using TDES. We will then use our algorithms to generate the VERILOG code for the FSM.

8.1 Problem Description

The purpose of this example is to design an FSM to control the 4-bit combination lock shown in Figure 8.1. The user of the system can either open the lock, or change the 4-bit saved password. To do either, the user must first correctly enter the current password. Entering two incorrect passwords in a row will set off an alarm. Once the

alarm is activated, it will stay activated until the system is reset.

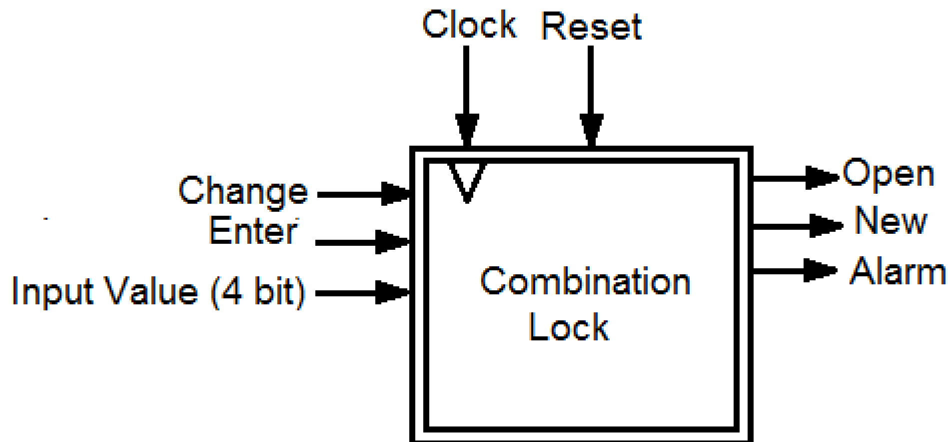


Figure 8.1: Lock System Black Box Diagram

We will now define the specifications of the problem in detail.

Opening the lock: To open the lock, the user must set the 4-bit combination (labelled as "Input value" in Figure 8.1), and then set signal *Enter* to 1 for one clock pulse. If the 4 bits match the combination stored in a 4-bit register, the door is unlocked and opens. The door opening is represented by output signal *Open* being set to 1. The door should stay open until user again sets *Enter* to 1 for one clock pulse, at which point the door closes (output *Open* is set to 0) and locks again. If two incorrect combinations are entered in a row, the alarm goes off (output *Alarm* is set to 1). The alarm can only be turned off (output *Alarm* set to 0) by resetting the system. When the system is reset, the stored combination is set to "0000".

Changing the combination: To change the combination, the user enters the current combination and then sets input *Change* to 1 for one clock pulse. If the entered 4-bits match the saved combination, then output signal *New* is set to 1 to indicate to the user to enter a new 4-bit combination. The user then enters a new

4-bit combination and sets input *Enter* or input *Change* to 1 for one clock period. The new combination is saved, and output *New* is set back to 0. Again, entering the wrong combination twice in a row will set off the alarm.

For inputs *Enter* and *Change*, we will assume the user has a button labelled *Enter* and one labelled *Change* that when pressed, will cause the corresponding input to be set to 1 for one clock cycle.

We note that signals *Change* and *Enter* will be external inputs beyond the control of the FSM, while signals *Open*, *New* and *Alarm* represent outputs that the FSM must provide.

8.1.1 System Components

Figure 8.2 shows a block diagram of the components needed to implement our combination lock. The system consists of the FSM that we will design as a TDES supervisor, as well as a 4-bit register to store the 4-bit combination and a 4-bit comparator. We assume we are given the register and comparator and will focus on designing the FSM to make Figure 8.2 work.

The 4-bit register is a standard digital login item from [10]. When its clear input is set to 0, it will store the 4-bit number "0000", which will become the value of the register's **Q** output. As long as the clear input is set to 1, the register will store the 4-bit value at its D input whenever its clock input (labeled as ">") changes from 0 to 1. In Figure 8.2, we have set this to the *Do_Change* output of the FSM.

The 4-bit comparator is a standard digital logic item from [10]. It compares the 4-bit input **X** to the 4-bit input **Y**, and sets **EQ** to 1 and **NEQ** to 0 when **X**=**Y**. Otherwise, **EQ** is set to 0 and **NEQ** to 1. **EQ** and **NEQ** are input signals to our FSM.

Figure 8.3 shows a break down of our FSM design. Rather than design one FSM, we will instead design three labelled *Open*, *Alarm* and *Change*, that will work to-

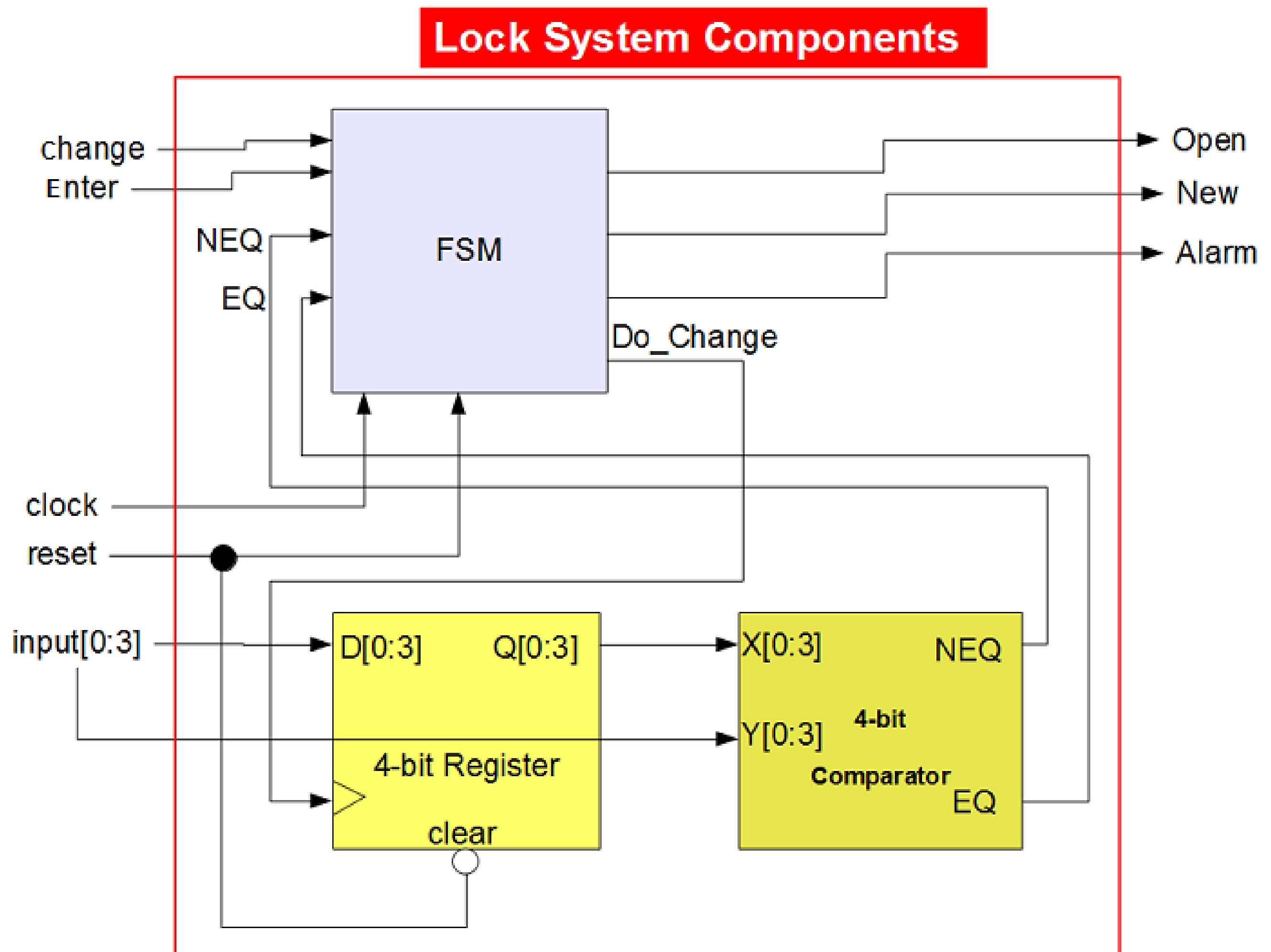


Figure 8.2: Lock System Block Diagram

gether to implement the FSM for Figure 8.2. The *Open* FSM will handle the logic to generate the Open output, while the *Change* FSM will handle the *Change* output. The *Alarm* FSM will determine when to sound the alarm, and will control the *Alarm* output.

Instead of designing the FSM directly as would normally be done, we will instead model the system as plant TDES, and then design a TDES supervisor each to im-

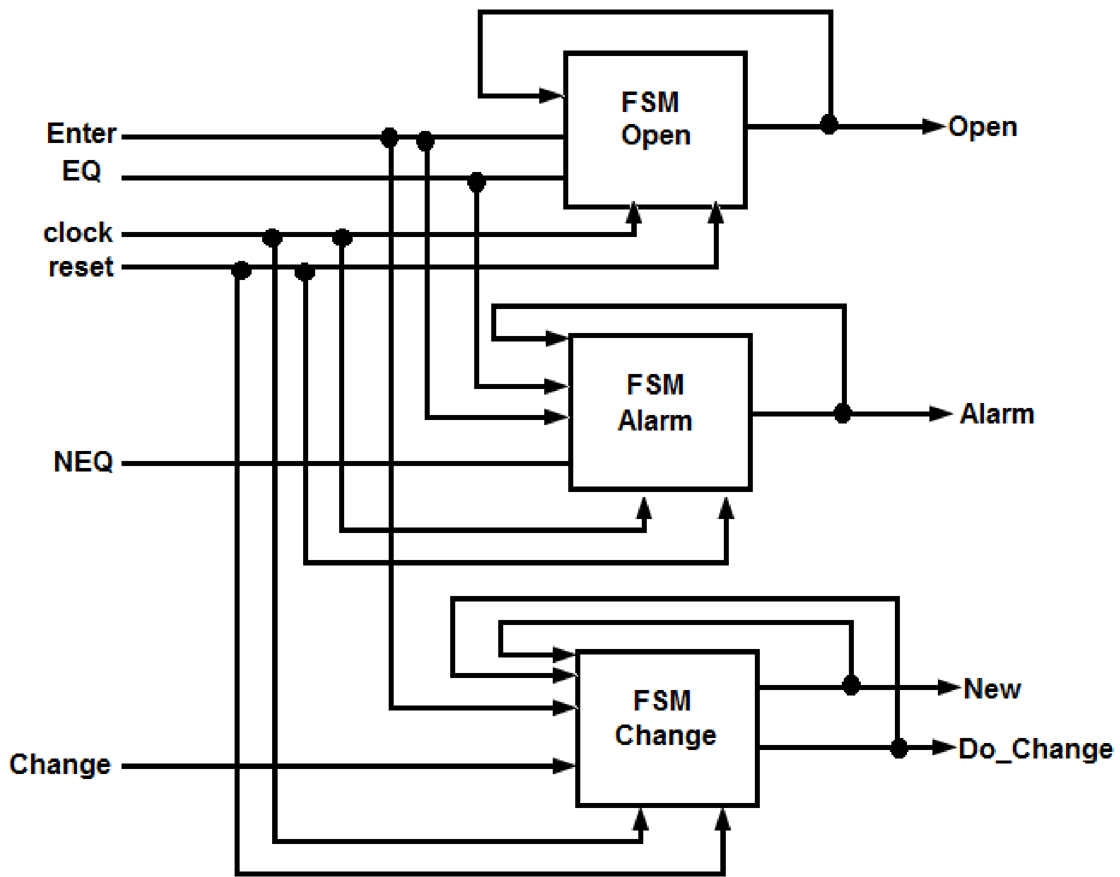


Figure 8.3: FSM Detailed Sub-Modules

plement the *Open*, *Change* and *Alarm* FSM. We will then verify that our plant has proper time behaviour, is complete for our supervisor, and has S-singular prohibitable behaviour. We will also verify that our supervisor is SD controllable for our plant, and that our closed-loop system is nonbocking and ALF.

8.2 Components Design

In Figure 8.4, we see the seven different plants that comprise the system. These plant components describe the behaviour of the following events:

open: This is a controllable (prohibitable) event, that enables the system to open the door.

do_change: This is a controllable (prohibitable) event, that enables the system to replace the four digits saved in the memory with the newly entered ones. It will correspond to the clock signal for the register in Figure 8.2.

new: This is a controllable (prohibitable) event, that enables the system to signal that it started the process to change the saved 4-bit security code. This signals the user to enter the new 4-bit code and press either *Enter* or *Change* to complete the operation.

enter: This is the uncontrollable event that means that the external agent either had entered the four digit that matches the saved ones and he/she wants the door to open, or that the external agent entered the new four bits and he/she wants them to be saved when the *New* signal is HIGH (set to 1).

change: This is the uncontrollable event that means that the external agent either had entered the four digit and he wants the door to open, or that the external agent entered the four bits and he/she wants them to be saved when the *New* signal is HIGH.

is_equal: This uncontrollable event means that the recently entered 4-bit code matches the one saved in the register. This event corresponds to signal **EQ** in Figure 8.2. In a given clock cycle, either event *is_equal* or *is_not_equal* will occur, but never both.

is_not_equal: This uncontrollable event means that the recently entered 4-bit code does not match the one saved in the register. This event corresponds to signal

NEQ in Figure 8.2.

alarm: This event is controllable (prohibitable) and represents that the system is in the alarm state.

However in the FSM shown in Figures: 8.6, 8.8 and 8.10, we are using shorthand names for these events because the diagram will not fit in the page if we use the full names. The shortened versions are given below:

- **OP**: open.
- **EN**: enter.
- **CH**: change.
- **DC**: do_change.
- **AL**: Alarm.
- **NW**: new.
- **EQ**: is_equal.
- **NQ**: is_not_equal.

We note that the first six plants in Figure 8.4 simply show that their corresponding event have no upper time bound and that each event can occur at most once per clock cycle. This is because the input events can occur whenever, and the output events are unrestricted and their behaviour will be determined by the supervisors.

The TDES diagrams shown in this chapter used the notations that a filled circle with a white circle ring (see state 0 of plant *Alarm* as an example) represents a marked initial state, a filled circle a marked state, and an unfilled circle an unmarked state. All uncontrollable events have an "!" preceding their name.

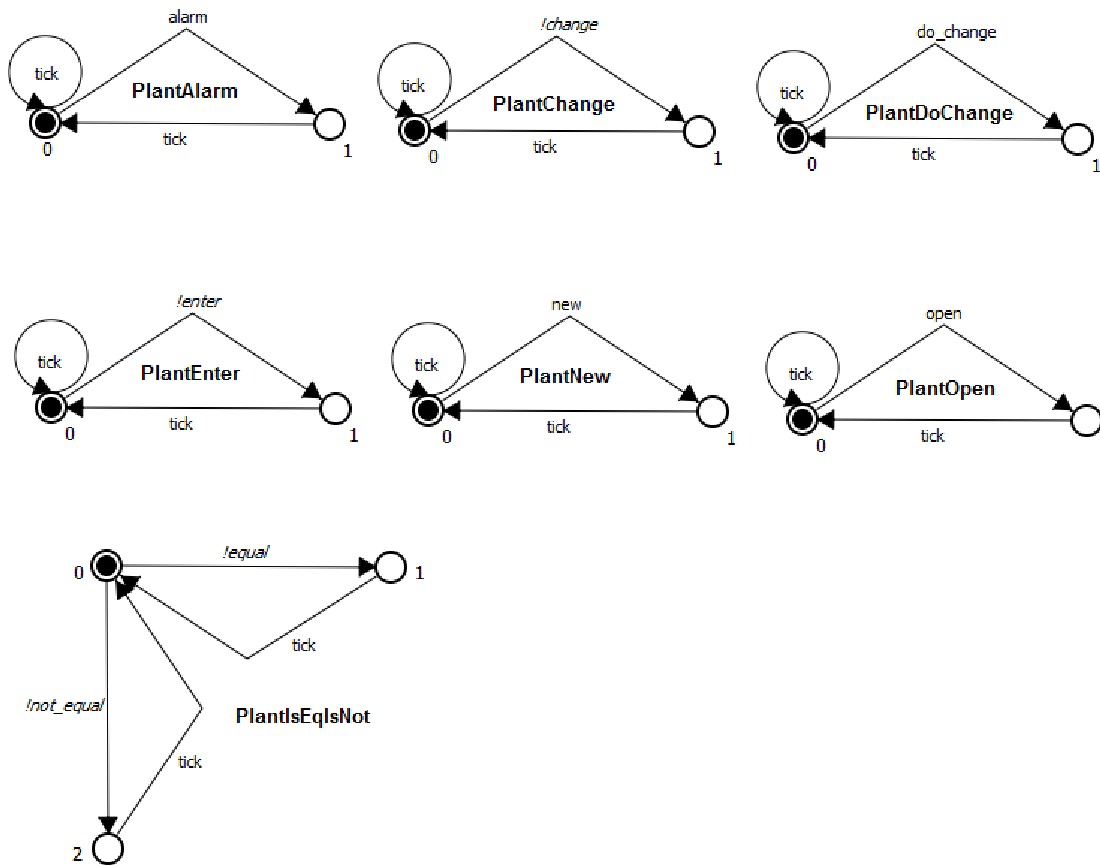


Figure 8.4: Lock System Plants Components

8.2.1 Signal Outputs and Transitions

When designing FSM for VERILOG implementation, there are two common tasks that the designer might be asked to do. The first is to have a signal output generate a pulse. By this we mean that the output be set to 1 for one clock period, and then set back to 0 for at least one clock period. An example is the *Do_Change* output of the FSM shown in Figure 8.2. When it is time to store a new 4-bit password, the FSM needs to generate a pulse on this output so that the 4-bit register will store the new combination.

Supervisor **SupChange**, shown in Figure 8.7, shows how to specify using TDES

that an output should generate a pulse. To specify output *Do_Change* should be true for one clock cycle, TDES **SupChange** enables the corresponding TDES event *do_change* immediately after the *tick* (state 9) and keep the event enabled (and the tick even disabled) until the event has occurred once. The event *do_change* is then kept disabled until the next *tick*. After the *tick* event occurs, event *do_change* must stay disabled until the next *tick* event occurs in order to complete the pulse.

The second common task is to set an output to 1 when a given condition has been met, and keep it at 1 until a new condition has been met. An example of this is output signal *Open* from Figure 8.2. Supervisor **SupOpen** shows how this is specified using a TDES supervisor. Reaching state 4 of the supervisor matches the condition to set output *Open* to 1. Output *Open* should stay at 1 until event *enter* occurs. To specify that *Open* should be set to 1 and stay at 1 till event *enter* occurs (represented by reaching state 0 again of the TDES), we enable event open at state 4, and keep it enabled until it has occurred. If the end condition (*enter* occurring) has not been met, the next *tick* should take us to a state where *open* is still enabled. We repeat the above logic until the end condition is met. In Figure 8.5, this corresponds to reaching state 5 and then the *tick* returning us to state 4.

If the end condition is met, the next *tick* takes us to a state where event *open* is disabled. In Figure 8.5, this corresponds to reaching state 6, and then the *tick* event takes us to state 0. Note that if event *enter* occurs at state 4 before event *open* occurs, we still need to keep *open* enabled until it occurs. That clock period, as signal *open* is not suppose to be set to 0 until after the next *tick* event, so event *open* must occur once this clock period. The above behaviour is specifically required to satisfy point iii.i of the SD controllability definition.

It is worth noting here that the TDES to FSM translation process described in Chapter 4 specifies that the states of the FSM correspond to the reachable sampled states of the TDES. Further, it specifies that if a prohibitable event is enabled at a sampled state, then the corresponding output is set to 1 at that state in the FSM. This means setting an output to 1 or 0, corresponds to enabling the corresponding prohibitable event or not at the sampled state in the TDES.

8.2.2 Supervisor SupOpen

Supervisor **SupOpen**, shown in Figure 8.5, is responsible for unlocking the door if a correct code is entered, and keeps the door open until the next *enter* event.

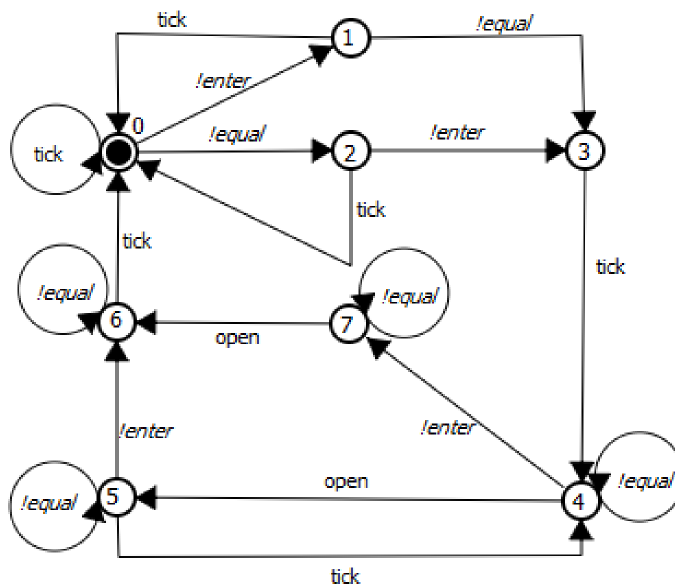


Figure 8.5: Supervisor **SupOpen**

At state **0** of the TDES, we wait for both events *enter* and *equal* to occur in the same clock cycle, representing that the entered combination matched the saved combination when the enter button was pressed. This takes us to state 4 where output signal *Open* should be set to 1 and stay there until the enter button is pressed. See discussion in Section 8.2.1 for more details on how to express this using TDES.

Figure 8.6 shows the FSM that corresponds to TDES **SupOpen**, and matches the FSM in Listing 8.1. Please see Chapter 7 for the details of how TDES **SupOpen** is converted into Listing 8.1, as well as an explanation of the file format. The notation used in FSM diagram is as follows. The arrow labelled "Reset" indicates the reset (initial) state of the FSM. The rectangles represent states, and labels starting with

"ST:" are the state names. For example, label "ST:0" represents state "0". For the FSM, we use the same state names as the TDES so we can see how the two compare.

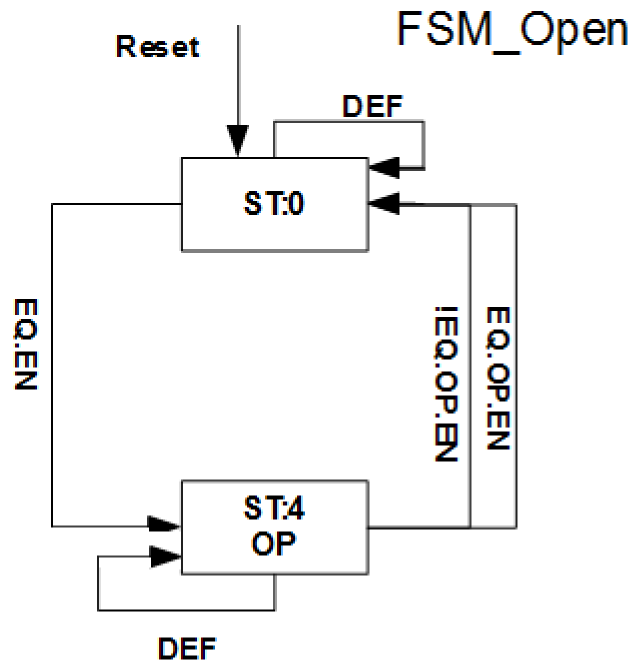


Figure 8.6: FSM for supervisor **SupOpen**

In the FSM diagrams, we specify which outputs should be set to 1 at each state by only listing the ones that are set to 1 at that state. For Figure 8.6, all outputs are set to 0 at state 0, and only output **OP** is set to 1 at state 4. In the FSM diagrams, we label translations as a Boolean algebra equation that represents all input combinations that would cause the Boolean equation to evaluate to true. For example, if both inputs **EQ** and **EN** are true at state 0, we switch to state 4, otherwise we stay at state 0. We use the translation label "DEF" as a shorthand to match any input condition that doesnt match one of the other transitions leaving that state.

For Boolean equations, we are using "!" to represent logical negation, "+" to represent logical OR, and "·" to represent logical AND. In the diagrams, we actually use "." instead of "·" as it is easier to produce.

Listing 8.1: The Generated FSM for Supervisor **SupOpen**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<FSM Name="SupOpen">
        <ResetState Name="0"> </ResetState>
        <Signals>
                <Signal Name="open" order="0" type="IO"/>
                <Signal Name="equal" order="1" type="I"/>
                <Signal Name="enter" order="2" type="I"/>
        </Signals>
        <States>
                <State Name="0" outputvector=""/>
                <State Name="4" outputvector="open"/>
        </States>
        <Transitions>
                <StartState Name="0">
                        <Transition inputvector="equal.enter" endstate="4"/>
                        <Transition inputvector="DEF" endstate="0"/>
                </State>
                <StartState Name="4">
                        <Transition inputvector="!equal.open.enter" endstate="0"/>
                        <Transition inputvector="equal.open.enter" endstate="0"/>
                        <Transition inputvector="DEF" endstate="4"/>
                </State>
        </Transitions>
</FSM>
```

The FSM for **SupOpen** is then translated to the VERILOG module shown in listing 8.2. For a description of the VERILOG code and the translation method, see Chapter 7.

Listing 8.2: The Generated VERILOG Module for Supervisor **SupOpen**

```
/*********************************
z[0],i[0]:open
i[1]: equal
i[2]: enter
*********************************/
```

```
6  module SupOpen(clock,resetn,i,z);
7          input clock,i,resetn;
8          output [0:0] z;
9          reg [0:0] z;
10         reg [0:0] Current;
11         reg [0:0] Next;
12         wire [2:0] i;
13         parameter IN_0=3'b110, IN_1=3'b101, IN_2=3'b111;
14         parameter ST_0=1'b0, ST_4=1'b1;
15         parameter OT_0=1'b0, OT_4=1'b1;
16         always @(Current or i)
17         begin
18                 case (Current)
19                         ST_0:
20                         begin
21                                 z=OT_0;
22                                 if(i==IN_0)
23                                 begin
24                                         Next<=ST_4;
25                                 end
26                         end
27                         ST_4:
28                         begin
29                                 z=OT_4;
30                                 if(i==IN_1)
31                                 begin
32                                         Next<=ST_0;
33                                 end
34                                 else
35                                 if(i==IN_2)
36                                 begin
37                                         Next<=ST_0;
38                                 end
39                         end
40                         default:
```

```
41                  begin
42                         Next<=ST_0;
43                  end
44           endcase
45    end
46    always @(posedge clock or negedge resetn)
47    begin
48           if (resetn==0)
49                  Current<=ST_0;
50           else
51                  Current<=Next;
52    end
53 endmodule
```

8.2.3 Supervisor SupChange

Supervisor **SupChange**, shown in Figure 8.7, allows the user to change the 4-bit combination lock. If the user enters a 4-bit number that matches the saved combination and then presses the change button, the supervisor will set output *New* to 1, and hold it there until either the change or enter button is pressed. **SupChange** then generates a pulse for output *Do_Change* which causes the current 4-bit input value to be saved to the register shown in Figure 8.2.

In Figure 8.7, the user entering the correct combination is signified by the *equal* and *change* events occurring in the same clock period, taking us from state 0 to state 4. At state 4, **SupChange** maintains output *New* at 1 until the *enter* or *change* event occurs. At state 9, **SupChange** generates a pulse for output *Do_Change*. Both are performed as described in Section 8.2.1.

Figure 8.8 contains the FSM that corresponds to TDES **SupChange** and matches the FSM in Listing 8.3. The corresponding VERILOG module for the FSM is shown in Listing 8.4. Please see Chapter 7 for a description of the FSM listing and VERILOG code, as well as for the translation algorithms.

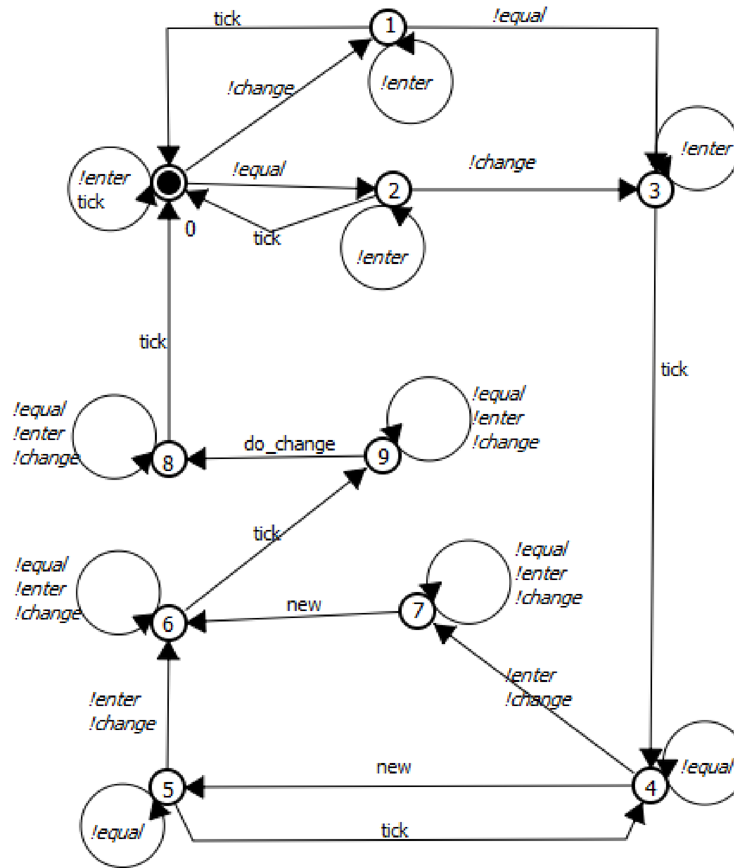


Figure 8.7: Supervisor **SupChange**

Listing 8.3: The Generated FSM for Supervisor **SupChange**

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <FSM Name="SupChange">
3          <ResetState Name="0"> </ResetState>
4          <Signals>
5                  <Signal Name="new" order="0" type="IO"/>
6                  <Signal Name="do_change" order="1" type="IO"/>
7                  <Signal Name="equal" order="2" type="I"/>
8                  <Signal Name="change" order="3" type="I"/>
9                  <Signal Name="enter" order="4" type="I"/>
10         </Signals>
11         <States>
```

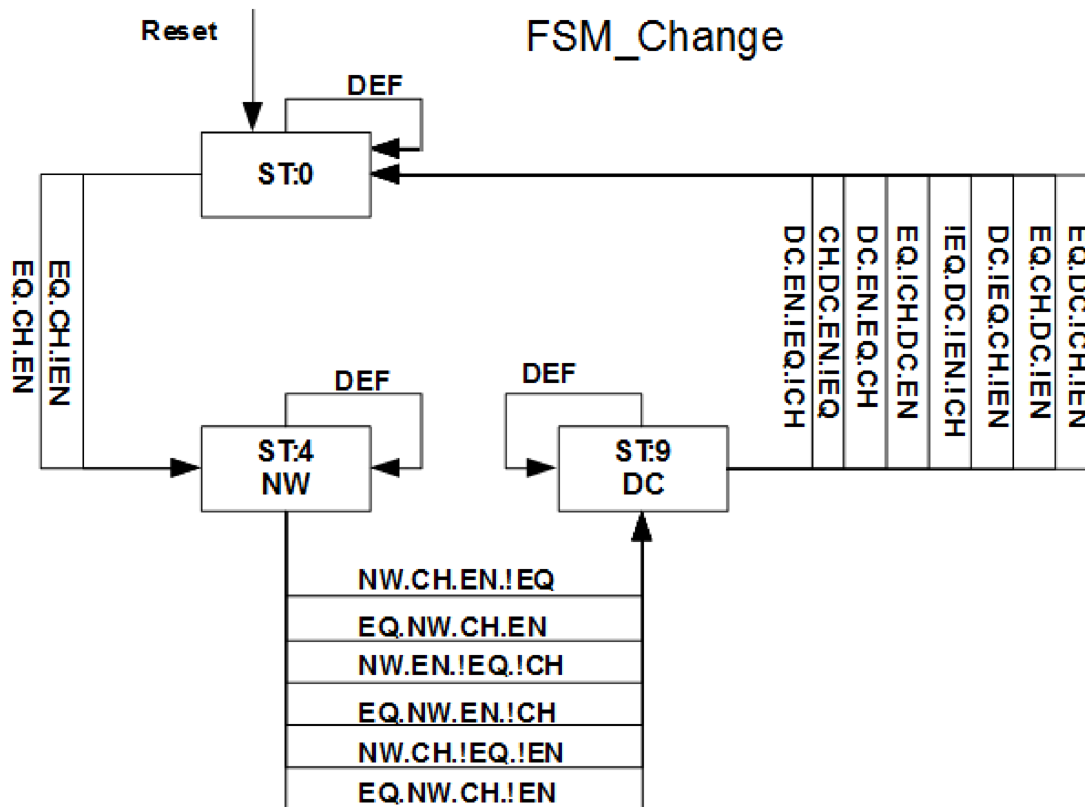


Figure 8.8: FSM for supervisor **SupChange**

```
12              <State Name="0" outputvector=""/>
13              <State Name="4" outputvector="new"/>
14              <State Name="9" outputvector="do_change"/>
15       </States>
16       <Transitions>
17              <StartState Name="0">
18                     <Transition inputvector="equal.change.enter" endstate="4"/>
19                     <Transition inputvector="equal.change.!enter" endstate="4"/>
20                     <Transition inputvector="DEF" endstate="0"/>
21              </State>
22              <StartState Name="4">
```

```
23                  <Transition inputvector="!equal.new.change.enter"
24                  endstate="9"/>
25                  <Transition inputvector="equal.new.change.enter"
26                  endstate="9"/>
27                  <Transition inputvector="!equal.new.!change.enter"
28                  endstate="9"/>
29                  <Transition inputvector="equal.new.!change.enter"
30                  endstate="9"/>
31                  <Transition inputvector="!equal.new.change.!enter"
32                  endstate="9"/>
33                  <Transition inputvector="equal.new.change.!enter"
34                  endstate="9"/>
35                  <Transition inputvector="DEF" endstate="4"/>
36              </State>
37              <StartState Name="9">
38                  <Transition inputvector="!equal.!change.do_change.enter"
39                  endstate="0"/>
40                  <Transition inputvector="!equal.change.do_change.enter"
41                  endstate="0"/>
42                  <Transition inputvector="equal.change.do_change.enter"
43                  endstate="0"/>
44                  <Transition inputvector="equal.!change.do_change.enter"
45                  endstate="0"/>
46                  <Transition inputvector="!equal.!change.do_change.!enter"
47                  endstate="0"/>
48                  <Transition inputvector="!equal.change.do_change.!enter"
49                  endstate="0"/>
50                  <Transition inputvector="equal.change.do_change.!enter"
51                  endstate="0"/>
52                  <Transition inputvector="equal.!change.do_change.!enter"
53                  endstate="0"/>
54                  <Transition inputvector="DEF" endstate="9"/>
55              </State>
56          </Transitions>
57  </FSM>
```

Listing 8.4: The Generated VERILOG Module for Supervisor **SupChange**

```
/********************************
z[0],i[0]:new
z[1],i[1]:do_change
i[2]: equal
i[3]: change
i[4]: enter
********************************/
module SupChange(clock,resetn,i,z);
       input clock,i,resetn;
       output [1:0] z;
       reg [1:0] z;
       reg [1:0] Current;
       reg [1:0] Next;
       wire [4:0] i;
       parameter IN_0=5'b11100, IN_1=5'b01100, IN_2=5'b11001,
       IN_3=5'b11101, IN_4=5'b10001, IN_5=5'b10101, IN_6=5'b01001,
       IN_7=5'b01101, IN_8=5'b10010, IN_9=5'b11010, IN_10=5'b11110,
       IN_11=5'b10110, IN_12=5'b00010, IN_13=5'b01010, IN_14=5'b01110,
       IN_15=5'b00110;
       parameter ST_0=2'b00, ST_4=2'b01, ST_9=2'b10;
       parameter OT_0=2'b00, OT_4=2'b01, OT_9=2'b10;
       always @(Current or i)
       begin
              case (Current)
                     ST_0:
                     begin
                            z=OT_0;
                            if(i==IN_0)
                            begin
                                   Next<=ST_4;
                            end
                            else
                            if(i==IN_1)
                            begin
```

```
35                          Next<=ST_4;
36                  end
37          end
38          ST_4:
39          begin
40                  z=OT_4;
41                  if(i==IN_2)
42                  begin
43                          Next<=ST_9;
44                  end
45                  else
46                  if(i==IN_3)
47                  begin
48                          Next<=ST_9;
49                  end
50                  else
51                  if(i==IN_4)
52                  begin
53                          Next<=ST_9;
54                  end
55                  else
56                  if(i==IN_5)
57                  begin
58                          Next<=ST_9;
59                  end
60                  else
61                  if(i==IN_6)
62                  begin
63                          Next<=ST_9;
64                  end
65                  else
66                  if(i==IN_7)
67                  begin
68                          Next<=ST_9;
69                  end
```

```
70                  end
71                  ST_9:
72                  begin
73                          z=OT_9;
74                          if(i==IN_8)
75                          begin
76                                  Next<=ST_0;
77                          end
78                          else
79                          if(i==IN_9)
80                          begin
81                                  Next<=ST_0;
82                          end
83                          else
84                          if(i==IN_10)
85                          begin
86                                  Next<=ST_0;
87                          end
88                          else
89                          if(i==IN_11)
90                          begin
91                                  Next<=ST_0;
92                          end
93                          else
94                          if(i==IN_12)
95                          begin
96                                  Next<=ST_0;
97                          end
98                          else
99                          if(i==IN_13)
100                         begin
101                                 Next<=ST_0;
102                         end
103                         else
104                         if(i==IN_14)
```

```
105                             begin
106                                    Next<=ST_0;
107                             end
108                             else
109                             if(i==IN_15)
110                             begin
111                                    Next<=ST_0;
112                             end
113                      end
114                      default:
115                      begin
116                             Next<=ST_0;
117                      end
118               endcase
119        end
120        always @(posedge clock or negedge resetn)
121        begin
122               if (resetn==0)
123                      Current<=ST_0;
124               else
125                      Current<=Next;
126        end
127 endmodule
```

8.2.4 Supervisor SupAlarm

Supervisor **SupAlarm**, shown in Figure 8.9, is the last of the three supervisors, as well as the most complex. Its job is to detect when two wrong lock combinations have been entered in a row. When this happens, the supervisor sets output *alarm* to 1 and holds it there. The only way to turn the alarm off is to reset the FSM. This is not apparent in the TDES as resetting is built into an FSM implemented with VERILOG, so it is not modelled at the TDES level.

In Figure 8.9, the user entering the wrong combination is signified by events

not_equal and *enter* or *change* occurring at the same time (ie. see state sequence 0-2-3-4). However, this is complicated by the fact that whenever the correct combination and *change* or *enter* occur, supervisors **SupOpen** and **SupChange** require an additional *change* or *enter* event to close their operation. As these *enter*/*change* events are not related to the correctness of the lock combination, they should not count toward triggering the alarm. This is captured by the state sequence 12-13-14-0. When Supervisor **SupAlarm** has reached state 8, two wrong combinations in a row have occurred and the system is in the alarm state. Output *Alarm* is set to 1, and maintained at 1.

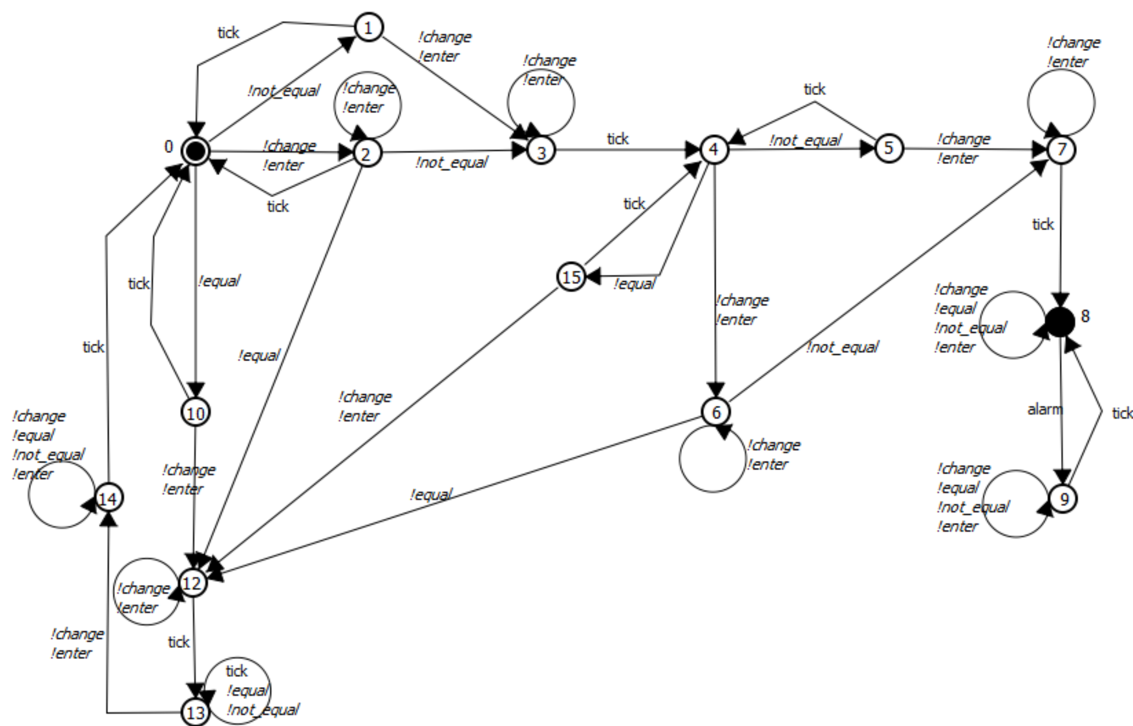


Figure 8.9: Supervisor **SupAlarm**.

Figure 8.10 contains the FSM that corresponds to TDES **SupAlarm** and matches the FSM in Listing 8.5. The corresponding VERILOG module for the FSM is shown in Listing 8.6. Please see Chapter 7 for a description of the FSM listing and VERILOG code, as well as the translation algorithms.

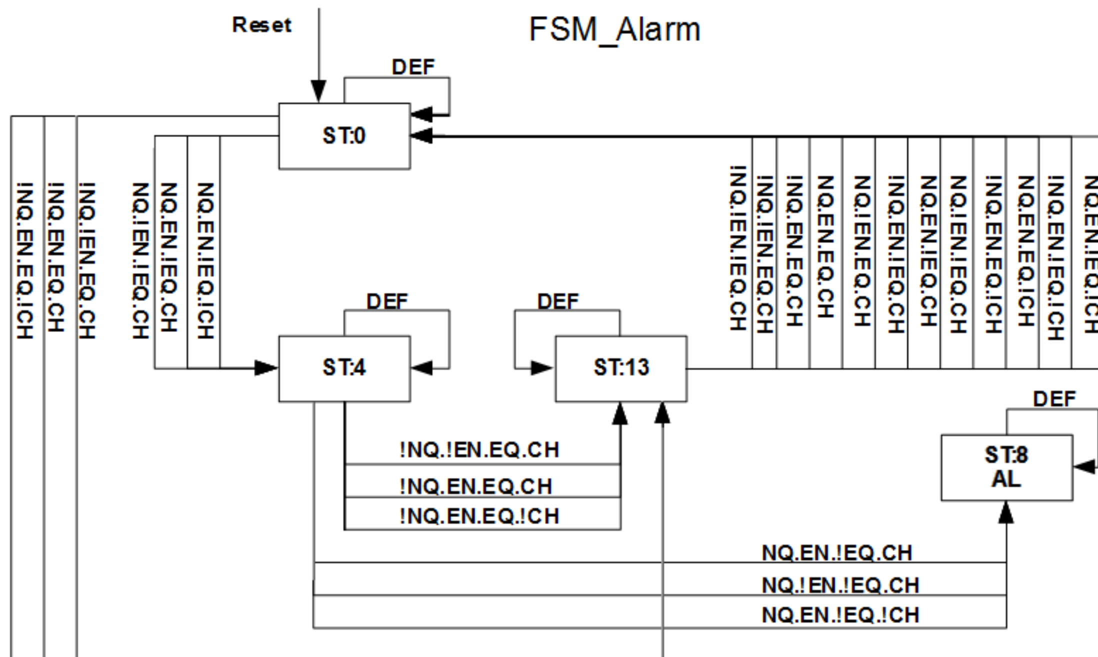


Figure 8.10: FSM for Supervisor **SupAlarm**

Listing 8.5: The Generated FSM for Supervisor **SupAlarm**

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <FSM Name="SupAlarm">
3         <ResetState Name="0"> </ResetState>
4         <Signals>
5                <Signal Name="alarm" order="0" type="IO"/>
6                <Signal Name="not_equal" order="1" type="I"/>
7                <Signal Name="enter" order="2" type="I"/>
8                <Signal Name="equal" order="3" type="I"/>
9                <Signal Name="change" order="4" type="I"/>
10        </Signals>
11        <States>
12               <State Name="0" outputvector=""/>
```

```
13              <State Name="4" outputvector=""/>
14              <State Name="8" outputvector="alarm"/>
15              <State Name="13" outputvector=""/>
16         </States>
17         <Transitions>
18              <StartState Name="0">
19                   <Transition inputvector="!not_equal.!enter.equal.change"
20                    endstate="13"/>
21                   <Transition inputvector="!not_equal.enter.equal.change"
22                    endstate="13"/>
23                   <Transition inputvector="not_equal.enter.!equal.change"
24                    endstate="4"/>
25                   <Transition inputvector="not_equal.!enter.!equal.change"
26                    endstate="4"/>
27                   <Transition inputvector="!not_equal.enter.equal.!change"
28                    endstate="13"/>
29                   <Transition inputvector="not_equal.enter.!equal.!change"
30                    endstate="4"/>
31                   <Transition inputvector="DEF" endstate="0"/>
32              </State>
33              <StartState Name="4">
34                   <Transition inputvector="!not_equal.!enter.equal.change"
35                    endstate="13"/>
36                   <Transition inputvector="!not_equal.enter.equal.change"
37                    endstate="13"/>
38                   <Transition inputvector="not_equal.enter.!equal.change"
39                    endstate="8"/>
40                   <Transition inputvector="not_equal.!enter.!equal.change"
41                   endstate="8"/>
42                   <Transition inputvector="!not_equal.enter.equal.!change"
43                   endstate="13"/>
44                   <Transition inputvector="not_equal.enter.!equal.!change"
45                   endstate="8"/>
46                   <Transition inputvector="DEF" endstate="4"/>
47              </State>
```

```
48                <StartState Name="8">
49                        <Transition inputvector="DEF" endstate="8"/>
50                </State>
51                <StartState Name="13">
52                        <Transition inputvector="!not_equal.!enter.!equal.change"
53                        endstate="0"/>
54                        <Transition inputvector="!not_equal.!enter.equal.change"
55                        endstate="0"/>
56                        <Transition inputvector="!not_equal.enter.equal.change"
57                        endstate="0"/>
58                        <Transition inputvector="not_equal.enter.equal.change"
59                        endstate="0"/>
60                        <Transition inputvector="not_equal.!enter.equal.change"
61                        endstate="0"/>
62                        <Transition inputvector="!not_equal.enter.!equal.change"
63                        endstate="0"/>
64                        <Transition inputvector="not_equal.enter.!equal.change"
65                        endstate="0"/>
66                        <Transition inputvector="not_equal.!enter.!equal.change"
67                        endstate="0"/>
68                        <Transition inputvector="!not_equal.enter.equal.!change"
69                        endstate="0"/>
70                        <Transition inputvector="not_equal.enter.equal.!change"
71                        endstate="0"/>
72                        <Transition inputvector="!not_equal.enter.!equal.!change"
73                        endstate="0"/>
74                        <Transition inputvector="not_equal.enter.!equal.!change"
75                        endstate="0"/>
76                        <Transition inputvector="DEF" endstate="13"/>
77                </State>
78        </Transitions>
79 </FSM>
```

Listing 8.6: The Generated VERILOG Module for Supervisor **SupAlarm**

```
1 /**********************************
```

```
2  z[0],i[0]:alarm
3  i[1]: not_equal
4  i[2]: enter
5  i[3]: equal
6  i[4]: change
7  *********************************/
8  module SupAlarm(clock,resetn,i,z);
9         input clock,i,resetn;
10        output [0:0] z;
11        reg [0:0] z;
12        reg [1:0] Current;
13        reg [1:0] Next;
14        wire [4:0] i;
15        parameter IN_0=5'b11000, IN_1=5'b11100, IN_2=5'b10110,
16        IN_3=5'b10010, IN_4=5'b01100, IN_5=5'b00110, IN_6=5'b11000,
17        IN_7=5'b11100, IN_8=5'b10110, IN_9=5'b10010, IN_10=5'b01100,
18        IN_11=5'b00110, IN_12=5'b10000, IN_13=5'b11000, IN_14=5'b11100,
19        IN_15=5'b11110, IN_16=5'b11010, IN_17=5'b10100, IN_18=5'b10110,
20        IN_19=5'b10010, IN_20=5'b01100, IN_21=5'b01110, IN_22=5'b00100,
21        IN_23=5'b00110;
22        parameter ST_0=2'b00, ST_4=2'b01, ST_8=2'b10, ST_13=2'b11;
23        parameter OT_0=1'b0, OT_4=1'b0, OT_8=1'b1, OT_13=1'b0;
24        always @(Current or i)
25        begin
26               case (Current)
27                      ST_0:
28                      begin
29                             z=OT_0;
30                             if(i==IN_0)
31                             begin
32                                    Next<=ST_13;
33                             end
34                             else
35                             if(i==IN_1)
36                             begin
```

```
37                          Next<=ST_13;
38                   end
39                   else
40                   if(i==IN_2)
41                   begin
42                          Next<=ST_4;
43                   end
44                   else
45                   if(i==IN_3)
46                   begin
47                          Next<=ST_4;
48                   end
49                   else
50                   if(i==IN_4)
51                   begin
52                          Next<=ST_13;
53                   end
54                   else
55                   if(i==IN_5)
56                   begin
57                          Next<=ST_4;
58                   end
59            end
60            ST_4:
61            begin
62                   z=OT_4;
63                   if(i==IN_6)
64                   begin
65                          Next<=ST_13;
66                   end
67                   else
68                   if(i==IN_7)
69                   begin
70                          Next<=ST_13;
71                   end
```

```
72                      else
73                      if(i==IN_8)
74                      begin
75                              Next<=ST_8;
76                      end
77                      else
78                      if(i==IN_9)
79                      begin
80                              Next<=ST_8;
81                      end
82                      else
83                      if(i==IN_10)
84                      begin
85                              Next<=ST_13;
86                      end
87                      else
88                      if(i==IN_11)
89                      begin
90                              Next<=ST_8;
91                      end
92              end
93              ST_8:
94              begin
95                      z=OT_8;
96              end
97              ST_13:
98              begin
99                      z=OT_13;
100                     if(i==IN_12)
101                     begin
102                             Next<=ST_0;
103                     end
104                     else
105                     if(i==IN_13)
106                     begin
```

```
107                         Next<=ST_0;
108                 end
109                 else
110                 if(i==IN_14)
111                 begin
112                         Next<=ST_0;
113                 end
114                 else
115                 if(i==IN_15)
116                 begin
117                         Next<=ST_0;
118                 end
119                 else
120                 if(i==IN_16)
121                 begin
122                         Next<=ST_0;
123                 end
124                 else
125                 if(i==IN_17)
126                 begin
127                         Next<=ST_0;
128                 end
129                 else
130                 if(i==IN_18)
131                 begin
132                         Next<=ST_0;
133                 end
134                 else
135                 if(i==IN_19)
136                 begin
137                         Next<=ST_0;
138                 end
139                 else
140                 if(i==IN_20)
141                 begin
```

```
142                         Next<=ST_0;
143                     end
144                     else
145                     if(i==IN_21)
146                     begin
147                         Next<=ST_0;
148                     end
149                     else
150                     if(i==IN_22)
151                     begin
152                         Next<=ST_0;
153                     end
154                     else
155                     if(i==IN_23)
156                     begin
157                         Next<=ST_0;
158                     end
159                 end
160                 default:
161                 begin
162                     Next<=ST_0;
163                 end
164             endcase
165         end
166         always @(posedge clock or negedge resetn)
167         begin
168             if (resetn==0)
169                 Current<=ST_0;
170             else
171                 Current<=Next;
172         end
173 endmodule
```

8.2.5 The Main Controller FSM

As described in Chapter 7, a system is converted to FSM in a modular fashion. We first create an FSM for each TDES supervisor in the system and then a main VERILOG module that ties them together into the equivalence of a single monolithic FSM. So far, we have specified the FSM for supervisors **SupOpen**, **SupChange**, and **SupAlarm**. We will now show how we tie them together to create the FSM specified in Figure 8.2. We first present the XML based version in Listing 8.7 and then the VERILOG module in Listing 8.8. Please refer to Chapter 7 for a description of the various elements as well as the translation algorithms.

Listing 8.7, which is the XML code for the central FSM, lists all the signals in the system. While listing the signals, we specify the *type* of the signal. Type **O** means the signal represents a prohibitable event, meaning it is used in the output. Type **I** means the signal represents an uncontrollable event, meaning it is used only in input vectors.

We then list all the individual FSM that make up the system. For each FSM we specify the name of the FSM (i.e. **Name="SupAlarm"**). We then specify which system signals each FSM uses, and for each signal, what type of signal it is. As an example, the first signal for the FSM **SupAlarm** is: **Name="alarm" type="IO"**. The fact that its type is **"IO"**, means this signal is both an input and output for the FSM.

Listing 8.7: The Generated Main FSM Module

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <FSMMain Name="LockSystem">
3    <Signals>
4      <Signal Name ="alarm" type="O" order="0"/>
5      <Signal Name ="do_change" type="O" order="1"/>
6      <Signal Name ="new" type="O" order="2"/>
7      <Signal Name ="open" type="O" order="3"/>
8      <Signal Name= "change" type="I" order="4" />
9      <Signal Name= "enter" type="I" order="5" />
10     <Signal Name= "equal" type="I" order="6" />
```

```
11      <Signal Name= "not_equal" type="I" order="7" />
12    </Signals>
13    <FSMS>
14      <FSM Name="SupAlarm">
15        <Signals>
16          <Signal Name="alarm" type="IO"/>
17          <Signal Name="not_equal" type="I"/>
18          <Signal Name="enter" type="I"/>
19          <Signal Name="equal" type="I"/>
20          <Signal Name="change" type="I"/>
21        </Signals>
22      </FSM>
23      <FSM Name="SupChange">
24        <Signals>
25          <Signal Name="new" type="IO"/>
26          <Signal Name="do_change" type="IO"/>
27          <Signal Name="equal" type="I"/>
28          <Signal Name="change" type="I"/>
29          <Signal Name="enter" type="I"/>
30        </Signals>
31      </FSM>
32      <FSM Name="SupOpen">
33        <Signals>
34          <Signal Name="open" type="IO"/>
35          <Signal Name="equal" type="I"/>
36          <Signal Name="enter" type="I"/>
37        </Signals>
38      </FSM>
39    </FSMS>
40  </FSMMain>
```

Listing 8.8: The Generated VERILOG Module for Main FSM

```
1
2  /*********************************
3  z[0]: alarm
```

```
4  z[1]: do_change
5  z[2]: new
6  z[3]: open
7  i[0]: change
8  i[1]: enter
9  i[2]: equal
10 i[3]: not_equal
11 ********************************/
12 module LockSystem(clock,resetn,i,z);
13   input clock,i,resetn;
14   output [3:0] z;
15   wire [3:0] i;
16   //Submodules input/output Wires:
17   wire [4:0] i_SupAlarm;
18   wire [0:0] z_SupAlarm;
19   wire [4:0] i_SupChange;
20   wire [1:0] z_SupChange;
21   wire [2:0] i_SupOpen;
22   wire [0:0] z_SupOpen;
23
24   //Constructing submodules:
25   assign i_SupAlarm[0]= z[0];
26   assign i_SupAlarm[1]= i[3];
27   assign i_SupAlarm[2]= i[1];
28   assign i_SupAlarm[3]= i[2];
29   assign i_SupAlarm[4]= i[0];
30   SupAlarm m_SupAlarm(clock,resetn,i_SupAlarm,z_SupAlarm);
31   //------------------------------------------------------------
32
33   assign i_SupChange[0]= z[2];
34   assign i_SupChange[1]= z[1];
35   assign i_SupChange[2]= i[2];
36   assign i_SupChange[3]= i[0];
37   assign i_SupChange[4]= i[1];
38   SupChange m_SupChange(clock,resetn,i_SupChange,z_SupChange);
```

```
39   //---------------------------------------------------------------
40
41   assign i_SupOpen[0]= z[3];
42   assign i_SupOpen[1]= i[2];
43   assign i_SupOpen[2]= i[1];
44   SupOpen m_SupOpen(clock,resetn,i_SupOpen,z_SupOpen);
45   //---------------------------------------------------------------
46
47   //Converging output from submodules:
48   assign z[0]=z_SupAlarm[0];
49   assign z[1]=z_SupChange[1];
50   assign z[2]=z_SupChange[0];
51   assign z[3]=z_SupOpen[0];
52 endmodule
```

8.3 Performance Results

Table 8.1 shows the performance results of the Lock System verification. The table, and the following performance tables too, have 7 columns. The first column lists the BDD properties that we are checking. The next three columns are for the modular testing, while the last three columns are for the monolithic testing. The second column gives the number of seconds needed to finish each BDD property check. Its header is therefore **Modular**. The first **Result** column shows the modular test result, while the second **Result** column shows the monolithic test result. Results are given as **PASS**, **FAIL** or **N/A**. This means either the algorithm passed the test, failed the test, or the test was not run for this category. The **AVG Size** column shows the average number of BDD states of the tested TDES components. The column **Monolithic** gives the number of seconds needed to finish the monolithic test of the BDD property. The last column, **Max Size**, shows the state size of the system in the monolithic test.

In each table cell where we say **N/A**, this means "not applicable". This is when there does not exist a monolithic or modular test for the given property. For example, the **SD1 + SD2** do not have modular tests so we see **N/A** in the modular check

column for these properties.

To test the performance of the algorithms on the following projects, the following machine configuration was used:

- Intel Core 2 Duo CPU T6600 (Computer specification)
- 4GB of Dual channel DDR2 RAM
- Windows 7 Ultimate

Table 8.1: Lock System Checking Performance Results

Property	Modular	Result	AVG Size	Monolithic	Result	Max Size
ALF	1	PASS	15	34	PASS	801
SSPB	0	PASS	15	5	PASS	801
SD 1 + SD 2	N/A	N/A	N/A	1	PASS	801
SD 3.1	1	PASS	48	7	PASS	801
SD 3.2	1	PASS	48	1	PASS	801
SD 4	0	PASS	15	1	PASS	801

We now discuss the performance results in Table 8.1. The ALF test passes modularly in 1 second. ALF also passes the monolithic test in 34 seconds. The SSPB property passes the modular test in less than a second (0), while it passes the monolithic test in 5 seconds. Properties SD 1 and SD 2 are not applicable for the modular test. However, both properties pass the monolithic test in 1 second. Property SD 3.1 passes the modular test in about 1 second. Property SD 3.2 passes the modular test also in 1 second. Property SD 4 passes the modular test in less than a second (0).

In Table 8.1, we see that when testing SD controllability points 3.1, 3.2 and 4 on the entire system the time spent was not very different from the modular tests. The time spent was 7 seconds for 3.1, 1 second for 3.2, and 1 second for 4. The reason is that the system is small. However, the state space was 801 and this consumes memory more than the modular tests for these properties. We note an order of magnitude reduction in some cases between modular and monolithic methods for state size and run time.

8.4 Other Projects Performance Results

We now show performance results for other projects that we tested our verification algorithms on.

8.4.1 Flexible Manufacturing System

We also applied an algorithm to the manufacturing example in Wang [22]. Table 8.2 shows the results of the various tests we performed on the project.

Table 8.2: Flexible Manufacturing System Checking Performance Results

Property	Modular	Result	AVG Size	Monolithic	Result	Max Size
ALF	1	PASS	49	9	PASS	82608
SSPB	1	PASS	46	143	PASS	82608
SD 1 + SD 2	N/A	N/A	N/A	32	PASS	82608
SD 3.1	1	FAIL	149	31	PASS	82608
SD 3.2	1	PASS	149	32	PASS	82608
SD 4	0	PASS	46	32	PASS	82608

We now discuss the performance results in Table 8.2. The ALF test passes modularly in 1 second. ALF also passes the monolithic test in 9 seconds. The SSPB property passes the modular test in 1 second, while it passes the monolithic test in 143 second. Property SD 3.1 passes the modular test in about 1 second. Property SD 3.2 passes the modular test also in 1 second. Property SD 4 passes the modular test in less than a second (0). Finally the monolithic test for SD controllability 1 and 2 took around 32 seconds.

In Table 8.2, we see that when testing SD controllability points 3.1, 3.2 and 4 on the entire system the time spent was very different from the modular tests. The time spent was 31 seconds for 3.1, 32 seconds for 3.2, and 32 seconds for 4. The reason is that the system's synchronous product is bigger than the average TDES in the system. Furthermore, the state space was 82608 and this consumes memory more than the modular tests for these properties.

8.4.2 Test Station of Manufacturing System

We also applied our algorithm to the test station (TSMS) modelled by Vleuten [19]. It is a model for a manufacturing system that was originally untimed DES and then modified to TDES to implement SD supervisory control for it. When the system was completed it passes only SSPB modularly and SD-Cont-iii.2 modularly. However it does not pass the other modular properties.

In this project some tests failed to complete after sixteen hours, so we put the word **"LONG"** to express this, as we then stopped the test. As we can see, when the modular test succeeded, we saw significant speed up.

Table 8.3: TSMS Checking Performance Results

Property	Modular	Result	AVG Size	Monolithic	Result	MAX Size
ALF	1	FAIL	88	1223	PASS	7.4×10^7
SSPB	1	PASS	88	LONG	LONG	7.4×10^7
SD 1 + SD 2	N/A	N/A	N/A	LONG	LONG	7.4×10^7
SD 3.1	1	FAIL	196	LONG	LONG	7.4×10^7
SD 3.2	9	PASS	196	LONG	LONG	7.4×10^7
SD 4	1	FAIL	88	LONG	LONG	7.4×10^7

We now discuss the performance results in Table 8.3. The ALF test fails modularly in 1 second. However, it passes the monolithic test in 1223 seconds (20 min). The SSPB property passes the modular test in 1 second, while it took long time for the monolithic test ("LONG"), so we stopped the program. We see that SD 1 and SD 2 properties are not applicable for modular tests. However, both properties test took long time for the monolithic test, so we stopped the program ("LONG"). Property SD 3.1 fails the modular test in about 1 second. Property SD 3.2 passes the modular test in 9 seconds. Property SD 4 fails the modular test in 1 second.

Chapter 9

Conclusion

9.1 Conclusion

The core work in this thesis is setting up the tools to verify systems modularly. We created algorithms of all properties that we were able to prove to be modular, namely ALF, SSPB, SD controllability point iii.1, SD controllability point iii.2 and SD controllability point iv. We implemented in software simple algorithms that verify these properties modularly. We have also created algorithms to test that all supervisors in the system are CS deterministic, and that all of them are non-self loop ALF.

We proved that several properties are modular. The modular verification that we introduced in this thesis is important in terms of computation time. It is often faster to test a project modularly than to do a monolithic verification. It is also important in terms of memory, as monolithic systems normally have very large state space. It is also useful as now designers can verify systems by eye only, since they can see that once this single TDES does not pass the property, then the modular check for this property will fail. Unlike the monolithic system verification, the only way to know is to run the test, and this will cost a lot of time and memory.

Another major aspect of this thesis was creating algorithms to generate finite state machines (FSM) descriptions of supervisors in the verified system. We also have created algorithms to translate these FSM into VERILOG modules as hardware

implementation of sampled-data controllers. We also tested these VERILOG mdoules using the ALTERA [11] simulation system.

We also designed a lock system consisting of three FSM as timed DES (TDES) supervisors. We then verified the sampled-data properties, and used our new translation software to translate the TDES supervisors first to FSM and then to VERILOG modules.

Finally, we gathered test models for our new example as well as two others, showing that when the modular approach succeeds, it can offer significant savings.

9.2 Future Work

As we focused on the VERLOG translation for these FSM as SD controllers, we recommend trying other implementation targets such as VHDL [10], or Programmable Logic Controllers (PLC) [23]. This will make our results more widely applicable.

Just as we targeted VERILOG, it looks useful to implement FSM as a C++ [4] program to give more flexibility. This would allow SD controllers to be implemented in software on regular computers.

We recommend future work in generating code for GUI (graphical user interface). One could program a GUI as finite state machines, and all the GUI component operations as events in the FSM. This is a good approach as most of the GUI programs are either procedural or event driven OOP (object-oriented programming) programs, and (GUI)s are now very common.

In this manner we can design our supervisors as specifications that changes the state of the screens from one state to another, then model these supervisors and verify them using the algorithms developed so far and generating the code that controls the GUI enablements and disablements accordingly.

We recommend expanding the simple modular approach we developed to a full incremental approach, such as Malik et al in [5]. Namely, choose a modular property to be verified and if individual components do not satisfy the property then we combine the individual components into groups and perform verification process on these groups. This can be done in such a way so that the modular test will fail if and only if the monolithic test would fail.

Bibliography

[1] Sinan Si Alhir. "Learning UML". *O'Reilly Media, Inc.", 2003.*

[2] S. Balemi. Input/output discrete event processes and communication delays. *Discrete Event Dynamic Systems*, 4(1):41–85, 1994.

[3] Mahvash Baloch. "A Compositional Approach for Verifying Sampled-Data Supervisory Control.". *M.Sc. Thesis, Dept. of Computing and Software, McMaster University, March 2012.*

[4] Stanley B. Lippman Barbara E. Moo, Jose Lajoie. C++ Primer, Fifth Edition. *Pearson Education, 2012.*

[5] Bertil A. Brandin, Robi Malik, and Petra Malik. "Incremental Verification and Synthesis of Discrete-Event Systems Guided by Counter-Examples". *IEEE J CST, volume 12, may 2004, doi = 10.1109/TCST.2004.824795.*

[6] W. Bolton. "Programmable Logic Controllers, 4th edn.". *Elsevier (2006).*

[7] B. A. Brandin. Real-time supervisory control of automated manufacturing systems,. *PhD thesis, University of Toronto. Graduate Department of Electrical and Computer Engineering, 1992.*

[8] Bertil Brandin and W. Murray Wonham. Supervisory control of timed discrete-event systems. *IEEE Trans. on Automatic Control*, 39(2):329–342, Feb 1994.

[9] Stephen Brown and Zvonko Vranesic. "Fundamentals of Digital Logic with Verlilog Design". *McGraw-Hill Higher Education, Second Education, May 14 2007.*

[10] Vranesic Z. Brown, S. "Fundamentals of Digital Logic with VHDL Design, 3rd edn.". *Mc-Graw Hill Higher Education (2008).*

[11] Altera Corporation. "Quartus ii Software". *http://www.altera.com/.*

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

[13] Julien Provost, Jean-Marc Roussel, and Jean-Marc Faure. "Testing Programmable Logic Controllers from Finite State Machines specification". *International Workshop on Dependable Control of Discrete Systems, Saarbrcken: Germany*, DCDS(3rd):Pages 1 – 6, 2011.

[14] R.J. Leduc and W.M. Wonham. "PLC implementation of a des supervisor for a manufacturing testbed". *In Proc. of Thirty-third Annual Allerton Conference on Communication, Control, and Computing, pp. 519-528, Oct 4-6, 1995.*

[15] Ryan Leduc. PLC implementation of a DES supervisor for a manufacturing testbed: An implementation perspective. Master's thesis, Dept. of Elec and Comp Eng, University of Toronto, Toronto, Ont, 1996.

[16] P. Ramadge and W. Murray Wonham. Supervisory control of a class of discrete-event processes. *SIAM J. Control Optim*, 25(1):206–230, 1987.

[17] Erik T. Ray. "Learning XML". *O'Reilly Media, Inc.", Sep 22, 2003.*

[18] Ryan J. Leduc, Yu Wang, and Fahim Ahmed. "Sampled-data supervisory control". *Discrete Event Dynamic Systems, November, 2013, DOI: 10.1007/s10626-013-0172-4.*

[19] J.W.C.M. van der Vleuten. "Application of Sampled-Data Supervisory Control to Physical Systems". *Eindhoven University of Technology, Department of Mechanical Engineering (2013).*

[20] Andrs Varga and Rudolf Hornig. "An Overview of the OMNet++ Simulation Environment". *Budapest University of Technology and Economics, Department of Telecommunications (1993).*

[21] Yu Wang. "Sampled-Data supervisory Control". *M.A.Sc. Thesis, Dept. of Computing and Software, McMaster University, January 2009.*

[22] Yu Wang and R.J. Leduc. "Sampled-data controller implementation". *International Journal of Control, vol. 85, no. 9, pp. 1343 - 1360, Sept, 2012.*

[23] John W. Webb and Ronald A. Reis. "Programmable Logic Controllers: Principles and Applications". *Prentice Hall PTR, Upper Saddle River, NJ, USA 1998.*

[24] K. C. Wong and W. M. Wonham. Hierarchical control of timed discrete-event systems. *Discrete Event Dynamic Systems*, 6(3):Pages 275 – 306, July 1996.

[25] W. M. Wonham. *Supervisory Control of Discrete-Event Systems.* Department of Electrical and Computer Engineering, University of Toronto, July 2008. Monograph and TCT software can be downloaded at http://www.control.toronto.edu/DES/.

[26] W. M. Wonham and P. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM J. Control Optim*, 25(3):637–659, May 1987.