CONCURRENCY IN DES AND A CLASS OF HYBRID SYSTEMS: THEORY AND COMPUTATION

SEYED MEHDI FATEMI BOOSHEHRI

Concurrency in DES And A Class of Hybrid Systems: Theory And Computation

by

© Seyed Mehdi Fatemi Booshehri

B.Sc., Shiraz University, 2002

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Science

Computational Science Interdisciplinary Programme
Faculty of Science
MEMORIAL UNIVERSITY

June 2010

ST.JOHN'S NEWFOUNDLAND

Concurrency in DES And A Class of Hybrid Systems: Theory And Computation

by

Seyed Mehdi Fatemi Booshehri

Abstract

This thesis explains the general concurrency of discrete event systems (DES), and then extends it to hybrid systems. To this end, firstly, a theoretical extension with n-ary structure is considered for the concurrency of a group of DES in the presence of specifications. A new concept (called *map*) is then introduced to meet a particular class of specifications, which are based upon both events and states. The *map* will be used to develop the n-ary synchronous product composition to a new composition rule, called accommodating synchronous product (ASP), which can implement the mentioned class of specifications.

Moreover, the DES concepts will be extended to a class of hybrid dynamical systems, whose concurrency is allowed to happen exclusively in the logical part. In such systems, a continuous dynamics (a physical behaviour) generates an event which then is passed to a DES to cause a transition (if any).

In the last part, a MATLAB-based software has been developed as the testbed for the theory and algorithms described throughout the thesis. The software has been designed with the consideration of object-oriented design, vectorization, and

compatibility with standard DES software. Based on its structure and a variety of different methods, it can be used for manipulating and exploring both concurrent DES and concurrent hybrid systems. Finally, the concepts presented in the thesis will be demonstrated in an extensive computational example solved by the software.

To my parents,

Mahmood and Firoozeh.

Acknowledgements

Computational Science is an interdisciplinary programme, thus during the course of my studies, I have had the opportunity to be with a group of excellent people from different parties. I should say, being at Memorial and the National Research Council, I have greatly enjoyed the learning and the discovery that comes with the graduate work.

First of all, I would like to take this opportunity to thank my thesis supervisor, Dr. Jim Millan, who has been working with me closely every step toward this thesis. He has provided me with wonderful research opportunities and has been an enormous source of support. Dr. Millan is actually the person who introduced me to the world of discrete-event and hybrid systems and control. I am most grateful for his guidance, insights, and consistent encouragement through course work, research, presentations, and publications.

I have been honored to work with my programme supervisors: Dr. Tina Yu and Prof. Siu O'Young over the past more than two years. Dr. Yu was the first one at Memorial I had talked to and was instrumental in introducing me to my engineering co-supervisor Prof. O'Young, and then a telephone interview. I would like to thank her efforts which allowed me to start my programme at Memorial.

I would be grateful for all the helps and endless supports from Prof. O'Young. When you need a new idea, Siu is always the one you should have a talk with. His

greatly unconditional support, in fact, allowed me to concentrate solely on my research and come up with the current thesis.

I would also like to express my appreciation to the National Research Council Institute for Ocean Technology, the federal organization for which I have been working for just less than three years with lots of great memories and experiences.

Last but by no means least, I would like to thank my family: my father Mahmood, my mother Firoozeh, my brother Mohamad Reza, and my sister Elaheh for all their love, great support and understanding while I left them to continue my educations in Canada.

This thesis was carried out for the RAVEN (Remote Aerial Vehicles for ENvironmental monitoring) research program of Memorial University of Newfoundland (MUN). The project is jointly being supported by the Atlantic Canada Opportunities Agency (ACOA), the National Research Council of Canada (NRC), Institute for Ocean Technology (IOT) and the Institute for Aerospace Research (IAR), Provincial Aerospace Limited (PAL), the National Science and Engineering Research Council (NSERC) and Defence Research and Development Canada (DRDC).

Contents

List of Tables

List of Figures

List of Abbreviations

ASP Accommodating Synchronous Product

CSM Continuous System Model

DES Discrete Event System(s)

FSM Finite State Machine

FSA Finite State Automaton

IVP Initial Value Problem

ODE Ordinary Differential Equation

OOP Object-Oriented Programming

OOD Object-Oriented Design

OOE Object-Oriented Environment

TDS Time-Driven Systems

TSM Time-driven System Model

UAV Unmanned Aerial Vehicle

Chapter 1

Introduction

1.1 Background

In this research, a general class of systems involving more than one discrete event system (DES) is considered, in which each DES is accepted to be an independent entity. Roughly speaking, a DES is a level of abstraction in the modeling of a real-world system, in which the occurrence of an event is responsible for the evolution of the system's behaviour (instead of the progress of the time). As such, the behaviour of a DES is basically a finite or infinite number of *strings* of possible events, which is then called the *language* generated by the DES. However, in the modeling of real-world applications, a DES may be a model for a part of a larger system. More technically, in real-world applications, we have to deal with a set of concurrent DES. A computational problem emerges here is that to cope with the engineering issues such as information hiding and interface design, an n-ary framework is required, while the main body of the existing literature has been developed on the basis of binary operations (see for example (Wonham 2009) and (Cassandras and Lafortune 1999)). Moreover, when dealing with a set of DES, a specification for a DES can be introduced based not only upon the events generated by other DES, but upon the current state of the DES itself. As a result, a generalization to the current literature appears to be

required before introducing the desired object-oriented design.

As a natural extension, a set of continuous dynamics can be added to this n-ary view of DES, as the source of event generation. This will shape a simple class of hybrid systems which are allowed to have synchronization only in the logical level of abstraction. This class can be explained by the existing theory of hybrid automata, which shapes the last part of this work.

In this thesis, the basic theoretical concepts and required definitions are first considered. We then extend the theory to computation by introducing software which has been designed during this research.

1.2 Problem Statement

1.2.1 Motivation

Discrete event systems and supervisory control are well-developed areas, which have received considerable attention and research for more than two decades. However, when dealing with a group of DES, two issues are still remained to be addressed properly:

1. The DES theory, for the most part, has been constructed upon the binary formalism. It is still required to have standard definitions and methodologies dealing with a set of DES of the arbitrary cardinality $n \geq 2$. Note that, thanks to the commutativity and the associativity properties, the binary formalism can be generalized to an n-ary version; however, for the sake of software development, it is more convenient to have a complete n-ary formalism at the theory stage.

2. The existing theory and computational environments tackle only specifications which are solely event-based; it is required to meet the specifications based also on the states of each DES in the group.

3. Such a framework can also be extended to a limited class of hybrid systems, having synchronization only in the DES level of abstraction.

1.2.2 Problem Statement

1. Suppose that a number of systems working together, each modeled as a DES, and a set of specifications (in the most basic way can be stated in natural language words) which perfectly define both the individually and the concurrently desired behaviour, are given. An n-ary framework is required to capture the mentioned types of specifications, which can be modeled using both events and states, while the framework also allows for an object-oriented design.

2. It is desirable to have the framework extended to a limited class of "system of hybrid systems" (more than one), having synchronization only in the DES level of abstraction. Such a class of hybrid systems can be used to model a group of systems which are working concurrently, while only share their logical behaviours (their generated events).

1.3 Contributions

The very basic idea behind this research is the implementation. Thus, the work, in both the theoretical and the computational parts, is done with the idea of possibility of an object-oriented design. The contributions are as follows:

1. The theory of automata is extended with the goal of designing an n-ary concurrent system of DES:

   - to meet a particular class of specifications, which are based upon both events and states, a new concept (called *map*) is introduced,

- the *map* will then be used to extend the n-ary synchronous product composition to a new composition rule, called accommodating synchronous product (ASP), which can implement the mentioned class of specifications.

2. Software implementation:

   - Encompassing the concepts of the theory,
   - Object-oriented structure,
   - Importing from standard DES software,
   - Real-time addition/deletion of new system(s) to/from a given collection.

1.4 Scope

Although the theory for a hierarchical structure of a group of DES is described in this thesis, the implementation of the DES hierarchy in the software is beyond the scope of this work.

1.5 Organization

This thesis is organized as follows: Chapter 2 contains a general review of related concepts in the literature, and discusses primitive explanations, over which further concepts will be built up. Chapter 3 provides detailed explanations of the theory. All the main definitions will be explained through simple and extensive examples. Chapter 4 develops an object-oriented computational environment based on the theory and with the emphasis on software reliability and reusability. It also contains a complete example which demonstrates the capabilities of the proposed framework. Finally, Chapter 5 again summarizes the contributions of this thesis and offers directions for future work.

Chapter 2

Background and Related Work

In this chapter, a high level description of systems theory is provided. Since this thesis, for the most part, is about discrete event systems, the background provided in this chapter mostly focuses on the DES and supervisory control; however, continuous and hybrid systems are also explained. The writing is more descriptive than formal and is meant to provide a general background to a reader with little background in discrete-event and hybrid systems theory. Some further review will also be provided throughout Chapter 3 when discussing the definitions.

2.1 Systems and Modeling

As Cassandras and Lafortune (1999) pointed out, the idea of *systems* is an intuitive concept rather than an accurately defined term. However, there exist a number of definitions in the standard encyclopedias which are all similar in their main concept:

Wikipedia: System (from Latin *systēma*, in turn from Greek *σύστημα* systema) is a set of interacting or interdependent entities, real or abstract, forming an integrated whole (Wikipedia.org 2009).

Princeton WordNet: Instrumentality that combines interrelated interacting artifacts designed to work as a coherent entity (Princeton University 2009)

IEEE Standard Dictionary of Electrical and Electronic Terms: A combination of components that act together to perform a function not possible with any of the individual parts (Radatz 1997).

There also exist definitions for specific types of systems or for the usage in a specific field of study, among which:

Biology (Wikipedia): A system is a group of organs that work together to perform a certain task. Common systems, such as those present in mammals and other animals, seen in human anatomy, are those such as the circulatory system, the respiratory system, the nervous system, etc. (Wikipedia.org 2009).

Biomedical and cognitive (Princeton WordNet): A group of physiologically or anatomically related organs or parts (Princeton University 2009).

Thermodynamics (Wikipedia): A thermodynamic system, originally called a working substance, is defined as that part of the universe that is under consideration. A real or imaginary boundary separates the system from the rest of the universe, which is referred to as the environment, surroundings, or reservoir (Wikipedia.org 2009).

In the fields of science and engineering, these definitions imply the need for: first, *modeling* of the part of the world under consideration (study and harness of natural phenomena governed by physical laws in general); and second, considering the input/output behaviour (which is necessary for concurrency and control). Both these constituents can be addressed by the concept of *state-space* modeling. However, a state of a system which is normally one (or a set of) variable(s) indicating the exact condition of the system uniquely–can evolve over time either as a direct or indirect function of time itself (time dependent), a function of some specific events (time independent), or a combination of those two (hybrid). As a result, a general high-level classification can be considered for all systems based on the natural behaviours and regardless of what the area of study is:

1. Time-driven Systems

2. Discrete Event Systems

3. Hybrid Systems

In the next sections, we describe these classes in more detail.

2.2 Time-driven Systems

Time-driven systems (TDS) are characterized by two important attributes:

1. state variables are continuous over $\mathbb{R}$, that is, accept any real number as their value.

2. state variables are functions of time.

Regarding the first property, time-driven systems are also known as *continuous systems* (Millan 2006). This type of system can be then distinguished (Luenberger 1979) in two sub-branches of discrete-time and continuous-time systems which normally lead into difference equations and differential equations respectively. However, to model real-world systems, computational techniques are sometimes necessary due to complexity issues (Cellier 1991), (Pichler and Moreno-Diaz 1990). Different control techniques have also been developed for such systems both analytically and computationally (Ogata 2001), (Khalil 2002).

A general state-space formulation for the continuous-time is of the form

$$\begin{cases} \dot{x} = f(x, u, t), \quad x(t_0) = x_0 \\ y = g(x, u) \end{cases} \tag{2.1}$$

Where, $x \in \mathcal{X} \subseteq \mathbb{R}^n$ is the state vector, x_0 is the initial state, $u \in \mathcal{U} \subseteq \mathbb{R}^m$ is the control (input) vector, $y \in \mathcal{Y} \subseteq \mathbb{R}^p$ is the output vector, and t denotes the continuous

time. Functions f and g are both considered to be Lipschitz continuous for the sake of existence and uniqueness (locally) of the solutions (Fonseca and Leoni 2007), (Freeman and Kokotovic 2008), and (Khalil 2002).

In discrete-time systems, modeling is quite similar, only in the form of difference equations instead (Luenberger 1979):

$$\begin{cases} x(k+1) = f(x(k), u(k), k), \quad x(0) = x_0 \\ y(k) = g(x(k), u(k)), \end{cases} \tag{2.2}$$

for $k \in \{0, 1, 2, ...\}$. This form, in general, is computationally more convenient for implementation, especially when dealing with stochastic properties (Kalman 1960).

Other classifications have also been widely applied (linear and non-linear systems for example); however, such classifications are not of concern in this thesis.

2.3 Discrete Event Systems

Although most of the work done in the systems control area is about time-driven systems, there still exist systems which cannot be described by the theories and framework presented in the previous section. The reason is that these systems intrinsically show different behaviours regarding both their state space and state evolution (Cassandras and Lafortune 1999). This class of systems, as opposed to time-driven systems, is characterized by two attributes of

1. state variables accept discrete values,

2. state variables are no longer functions of time, rather, they jump through their different possible values only as a result of the occurrence of an event.

Tangibly, this class of systems is called Discrete Event Systems (DES). Mathematical definitions for such systems are quite different from those of time-driven systems, regarding the fact that continuous mathematics is no longer applicable in this area.

Indeed, diverse approaches have been developed to address the modeling of DES (Cassandras and Lafortune 1999), among which the theory of automata[1] is considered in this thesis because of its explicitness of events and states, and computational privileges. Exploration of other modeling theories (Petri Nets, for example) is beyond the scope of this thesis.

Example 2.1

Consider a machine including four states as the following (Millan 2006):

1. Working
2. Down
3. Scrap
4. Idle

Also consider the following six events for such a systems:

- a: the machine starts working, Idle to Working,
- b: completes its work and returns to the Idle state, Working to Idle,
- c: breaks down, going to the Down state, Working to Down,
- d: gets repaired, returning to the Idle state, Down to Idle,
- e: gets repaired, returning to the Working state, Down to Working,
- f: is scrapped, moving to the Scrap state, Down to Scrap.

The DES abstraction for this machine is illustrated in Figure 2-1 using IDES software[2]. The machine is initially at the *Idle* state (illustrated by a small arrow

[1] Also called *state machines*.

[2] IDES (Integrated Discrete-Event Systems) is a Java-based software developed by The Open-Symphony Group at Queen's university and under the supervision of K. Rudie (Rudie 2008).

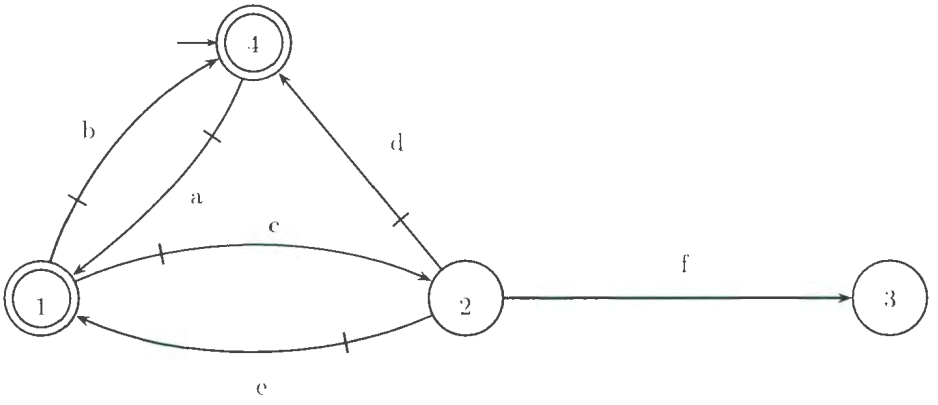

Figure 2-1: An automaton model for example 2.1 using IDES software.

next to state 4). It starts, working at the occurrence of event a. Then, by event b it goes back to *Idle* or by event c it goes to the *Down* state, and so forth. States 1 and 4 are marked (desired) states, illustrated by double circling, and all the events except event f are controllable[3], illustrated by a short dash on the corresponding arrows. One may observe that if this system goes to state 3, it will then completely stop running. Such states are called deadlock (Cassandras and Lafortune 1999); one goal of control synthesis for DES's can be avoiding the deadlocks.

Mathematical details of the automata theory will be provided in Chapter 3.

2.3.1 Supervisory Control

A specification is typically a set of logical policies to be applied to a machine, a mechanism, or a behaviour which can in turn be modeled at a logical (un-timed) level of abstraction (Cassandras and Lafortune 1999). To achieve the desired behaviour meeting the provided specifications, a closed-loop controller can be defined to disable specific controllable events when required. A control synthesis paradigm known as *supervisory control theory* (SCT) was developed by Ramadge and Wonham (1987) for DES. More detail about the implementation of supervisory control can be found in

[3]In the context of DES, an event is called controllable if it can be prevented from happening, or it can be disabled by a supervisor (Cassandras and Lafortune 1999).

(Wonham 2009). However, it should be noted that the standard supervisory control only deals with the specifications defined on the events. In this thesis, the model for specifications are considered on the basis of both states and events.

The basic SCT has been developed in different directions, including decentralized supervisory control (Rudie and Wonham 1992), supervision of infinite behaviour (Thistle and Wonham 1994), supervisory control under partial observation (Lin and Wonham 1995), and supervisor reduction (Su and Wonham 2004). Li (1997) explains the problem of synthesizing deadlock-free modular supervisory. Other endeavors by O'Young (1991) and Brandin and Wonham (1994) resulted in extending SCT to timed automata (will be explained in Section 2.4.1). More recently, a descriptive work by Lafortune (2007) explains how supervisory control can be extended to the problems where local controllers cannot explicitly communicate with each other in real-time.

A more computational work done by Leduc, Lawford and Dai (2006) describes how to cope with the problem due to exponential growth of state-space in large-scale practical systems such as manufacturing systems. It suggests a "hierarchical interface-based" supervisory control for systems with a natural master-slave structure. In another work done by Gaudin and Marchand (2005), the supervisory control and deadlock avoidance problem is discussed for concurrent discrete event systems. The work implies its emphasis on viewing the problem as the entire collection, instead of adding up binary behaviours. Regarding this view, the work done by Gaudin and Marchand (2005) can be compared in part to our view in this research.

2.4 Hybrid Systems

Since a class of hybrid systems is also addressed in this thesis, this section provides a brief explanation of these systems and how they are connected to the previously explained systems.

Hybrid systems are the integration of TDS and DES, and therefore cannot be

modeled by either of TDS or DES modeling alone. In the literature, diverse modeling approaches have been developed to address inconsistency of the two types of modelings.

2.4.1 Timed Automata

By definition, an automaton is a model for DES, and therefore dose not include the concept of time. However, due to the fact that most logically-behaving systems require *time* for measurement and synchronization, the concept of time appears to be necessary in the modeling and control of real-world applications. It was first added to the automata as integer clock events, called ticks, (O'Young 1991) which can be considered as a formal step towards hybrid system modeling. The coarse-timing, however, does not meet the real-world requirements; as a result, a set of real-valued clocks added to the theory of automata by Alur and Dill (1994). The supervisory control for such systems was then explained by Brandin and Wonham (1994).

The theory of timed automata has been implemented mostly in the real-time computing applications, such as on-line transaction processing systems (Kourkouli and Hassapis 2005); however, it has not been widely used in the systems and control community.

2.4.2 Hybrid Automata

While timed automata theory has achieved promising results (Saadatpoor 2004), it still does not include the concept of time-driven *dynamics*. In other words, any physical systems with logical constraints (more generally, with logical behaviours) cannot be modeled by timed automata. This notion shaped the idea of hybrid automata.

The theory of hybrid automata was first developed by Alur, Courcoubetis, Henzinger and Ho (1993) and Henzinger (1996). It has then evolved through a wide range of work done by different researchers including Lygeros, Tomlin and Sastry (1999), Lynch, Segala and Vaandrager (2001), and Cassandras and Lygeros (2006).

The formal definition of hybrid automata will be presented in Chapter 3 (Section 3.6), where more relevant references will also be provided.

Chapter 3

Synchronization of Discrete Event Systems

In this chapter, we consider a system of more than one DES, each accepted to be an independent entity. The normal behaviour of each DES is the language it generates before applying any constraint (see (Wonham 2009) and (Cassandras and Lafortune 1999) for a complete description of language and automata in the DES literature), while the desired behaviour is the language generated after applying a set of constraint, called the specification. For a group of given DES (more than one), two types of specifications are considered in this thesis:

1. **(Event-based specification)** is a behavioural constraint to an individual DES (or a number of DES together, which share common events), such as blocking of specific events in one or more DES,

2. **(State/event-based specification)** is a "state-dependent," and still "event-based" constraint for concurrent DES, such as having a specific transition in a DES as a result of the occurrence of a specific event from the event set of another DES in the group.

More precisely, if we have a group of more than one system, a specification can be applied to the group because of individual desired behaviour and/or concurrent

(intra-systems) desired behaviour. For a given problem one or both of the above specifications can be applied.

As we will see, an event-based specification is normally a separate automaton added to the group, with events from the event set(s) of the system(s) to be controlled, and the states which are not from the state set(s) of the main system(s). This type of specification has been well-developed in the context of supervisory control.

For a state/event-based specification, a set of local functions is proposed in this thesis to locally translate an event, generated by other systems of the group, into an "owned" event while being in a specific state. This type of specification becomes more important in the implementation of real-world applications, where, due to the engineering considerations such as information hiding, the state of a DES is as important as its generated language. Additionally, when a DES is a model for a part of a larger system, each state of the DES can bear a physical concept in behind, regarding the entire system. As a result, a specification may be set up once the DES is in a specific state.

Remark 3.1: (Theory vs. Implementation)

Generally speaking, in systems theory and control, there is almost always a distinction between the theory, which is pure mathematical explanations, and the implementation, which is the way that is preferred because of computational and/or engineering considerations. From the mathematical point of view, different modeling methods may be considered as "equal" provided they generate the same results. However, from the implementation point of view, those methods may be considered as "completely different." because of the reasons ranging from the physical meanings behind the constituents of a model, to the computational complexity (and even possibility). For example, in the context of continuous systems theory, it is said that the state-space model is not unique, which is true. However, engineers prefer to use a model that matches each state variable with a physical concept (say in a high-level modeling

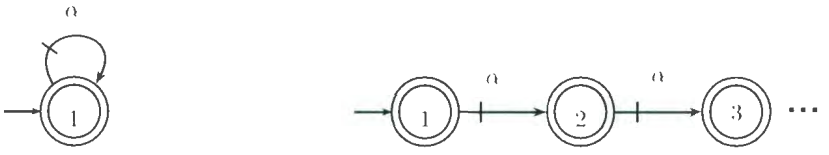

Figure 3-1: Two DES with different topology, which generate and mark the same languages. The one at the right has infinite number of states, thus can not be implemented.

of a car, the state variables can be the car's location, velocity, etc. instead of sheer mathematical variables so that their combinations provides the car's location, etc.). Likewise, in the context of DES, from the mathematical point of view, as long as we are only concerned about the generated and marked languages, the result would not be different from that when we consider implementation issues such as an object-oriented design. Figure 3-1 illustrates two DES, both of which generate and mark the same languages, while only the one on the left can be implemented using a finite amount of memory; behaviourally, these two automata are identical, but only one can be implemented.

Mathematically, it is always possible to "edit" a DES by changing its topology to achieve a desired behaviour. Therefore, both types of specifications can be performed by re-labeling, and/or augmenting, and/or omitting the edges of the main systems, instead of introducing a separate DES and/or local functions. However, in practice, to have an object-oriented design (therefore, benefit from the ideas such as information hiding, hierarchical design, and reuse), a basic premise is *not* to allow for the change of a model (hidden information) after the very first stage of design. This premise puts forward only methodologies which permit this "hiding" and "interface-design" for both types of mentioned specifications.

Figure 3-2 summarizes the theoretical concepts which will be presented in this chapter. We begin with the formal modeling of discrete event systems (which is the concept of "*automaton*"). We then extend the modeling of one single system to a finite set of systems having interaction explicitly by seeing each others' events,

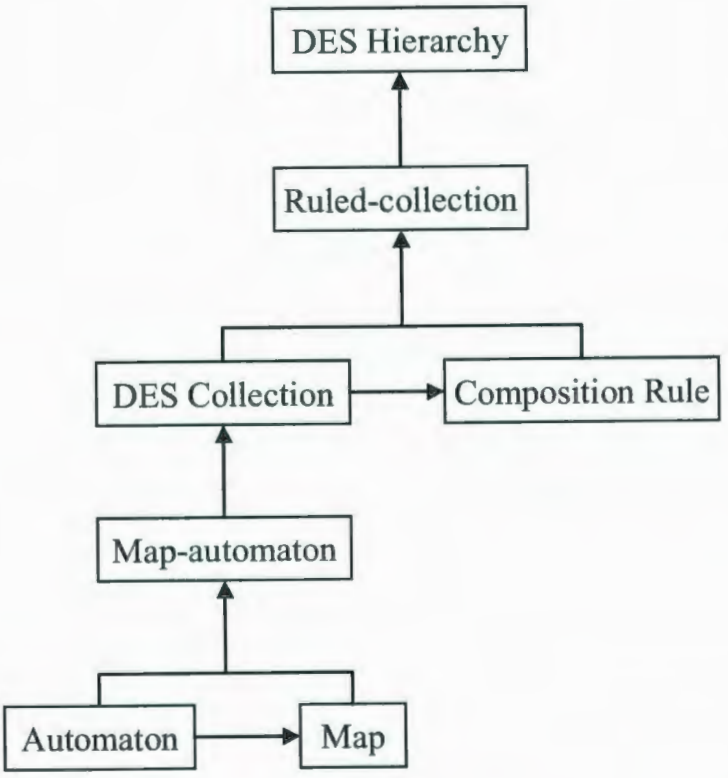

Figure 3-2: The outlook of Chapter 3.

which results in the definition of "*map*". These two together will shape the "*map-automaton*", which in turn is the basis for the definition of a "*DES collection.*" Over a *DES collection* a "*composition rule*" can be defined. A *DES collection* with a defined *composition rule* will shape a "*ruled-collection,*" which is the basis for "*DES hierarchy*" at the final step.

The goal of this chapter is to establish a unified and solid mathematical framework to model multi-DES structures effectively. Both the standard and the introduced definitions and concepts are clarified by simple examples.

3.1 DES Concepts And Methodologies

Definition 3.1.1 (Automata) *An automaton*[1] *(Cassandras and Lafortune 1999) is a six-tuple*

$$G = (Q, \Sigma, \delta, \Gamma, q_0, Q_m) \tag{3.1}$$

where:

- Q *is a set of discrete* states,
- Σ *is the finite set of events associated with transitions in* G,
- $\delta : Q \times \Sigma \rightarrow Q$ *is the* transition function, *which is generally a partial function on its domain,*
- $\Gamma : Q \rightarrow 2^{\Sigma}$ *is the* active event function; $\Gamma(q)$ *is the set of events* σ, *for which* $\delta(q, \sigma)$ *is defined, and is called the* active event set *of* G *at* q,
- q_0 *is the initial state,*
- $Q_m \subseteq Q$ *is the set of* marked states.

For the sake of convenience (Cassandras and Lafortune 1999), δ is always extended from domain $Q \times \Sigma$ to domain $Q \times \Sigma^*$ in the following recursive manner[2]:

$$\delta(q, \epsilon) := q, \tag{3.2}$$

$$\delta(q, s\sigma) := \delta(\delta(q, s), \sigma), \text{ for } s \in \Sigma^* \text{ and } \sigma \in \Sigma, \tag{3.3}$$

[1]The terms *automaton* and *finite automaton* will be used interchangeably through out this text, but both are distinguished from *hybrid automaton*. Additionally, automata theory is a methodology to model DES; however, in this text, automata and DES are both being referred to as in the definition 3.1.1.

[2]Through out this text, the symbol ":=" means "by definition is equal to."

where, ϵ is the empty string. Also, it is important to note that we allow the transition function δ to be partially defined over its domain which is a standard definition in DES area (but a variation over the automata theory in the computer science literature). This definition allows for blocking, which is a basic concept in the DES area (Cassandras and Lafortune 1999).

Definition 3.1.2 (Languages Generated And Marked) *The language generated by* $G = (Q, \Sigma, \delta, \Gamma, q_0, Q_m)$ *is*

$$\mathcal{L}(G) := \{s \in \Sigma^* \mid \delta(q_0, s) \text{ is defined.}\} \tag{3.4}$$

The language marked by G *is*

$$\mathcal{L}_m(G) := \{s \in \mathcal{L}(G) \mid \delta(q_0, s) \in Q_m\}. \tag{3.5}$$

By definition, two automata G_1 and G_2 are said to be *equivalent* and shown as $G_1 = G_2$ if and only if they generate and mark the same languages. Also, the reader may note that, in the DES area, it is usually more desirable to construct the definitions based on the generated and marked languages instead of the automata themselves (see for example the text by Wonham (2009)). However, in this thesis, the theory will preferably be built upon the automata definition, due to the role that states play in the real-world applications (the reader will observe that each state can carry a physical meaning, rather than being a sheer name, especially in the hybrid systems area).

Example 3.1: (Automaton)

Consider the machine explained in example 2.1, the automaton for this machine would be defined as

$$G = (Q, \Sigma. \delta. \Gamma, q_0, Q_m) \quad \text{where,} \tag{3.6}$$

$$Q = \{1, 2, 3. 4\}, \tag{3.7}$$

$$\Sigma = \{a, b, c, d, e, f\}, \tag{3.8}$$

δ is defined as:

$$\delta(1, b) = 4, \tag{3.9}$$

$$\delta(1, c) = 2, \tag{3.10}$$

$$\delta(2, e) = 1, \tag{3.11}$$

$$\delta(2, d) = 4, \tag{3.12}$$

$$\delta(2, f) = 3, \tag{3.13}$$

$$\delta(4, a) = 1. \tag{3.14}$$

and Γ is defined as:

$$\Gamma(1) = \{b, c\}. \tag{3.15}$$

$$\Gamma(2) = \{e, d, f\}, \tag{3.16}$$

$$\Gamma(3) = \emptyset, \tag{3.17}$$

$$\Gamma(4) = \{a\}, \tag{3.18}$$

$$q_0 = 4, \tag{3.19}$$

$$Q_m = \{1, 4\}. \tag{3.20}$$

■

Let us now extend the single-system architecture to a multi-system one, where each DES is supposed to be modeled as a stand-alone automaton (that is, we are *not* going to let the entire system be flattened).

Remember that for a system of more than one DES, the specifications are accepted in two different ways: event-based and state/event-based. The idea behind the implementation of a state/event-based specification is to encode the event-based interaction of multi-DES in local functions called *map*, so that the topology of each DES remains unchanged to preserve information hiding and promote reuse. Consider the following definition:

Definition 3.1.3 (Map) *Given a set of n DES,* $\{G_1, ... G_n\}$, *and* $\Sigma = \bigcup_{p=1}^{n} \Sigma_p$, M_i *is called the* map *for the i-th DES and defined as* [3]*:*

$$M_i : Q_i \times (\Sigma \backslash \Sigma_i) \xrightarrow{par} \Sigma_i, \text{ such that} \tag{3.21}$$

$$M_i(q_i, \sigma) = \sigma' \Rightarrow \sigma' \in \Gamma_i(q_i) \backslash \bigcup_{j \in \{1,....,n\} \backslash \{i\}} \Sigma_j. \tag{3.22}$$

For $q_i \in Q_i$, $\sigma \in \Sigma \backslash \Sigma_i$, and $\sigma' \in \Gamma_i(q_i) \backslash \bigcup_{j \neq i} \Sigma_j$; $M_i(q_i, \sigma) = \sigma'$ provides a translation of event σ to event σ', which can cause a local transition ($\delta_i(q_i, \sigma')$) in system i. Therefore, inside a group of DES, a map can be thought of as a local, one-sided interface of a DES to the rest of the group. Thus, a map hides its DES details. We have not yet talked about the composition rules, which are the formal ways of constructing a new DES from a set of given DES. However, note that a given map (say M_i) only preserves the local *translation* of events, and does not directly affect the transition function of the corresponding system (δ_i). The transition function should then be defined separately by a composition rule, which can take advantage of these locally translated events (for example, if the translated event (σ') is blocked by an event-based specification, which is one of the DES in the group, then *no* transition will occur). We will explain it completely in Section 3.3. Now, consider the following example:

[3] *The symbol* $\xrightarrow{par}$ *indicates the range of a function as partial.*

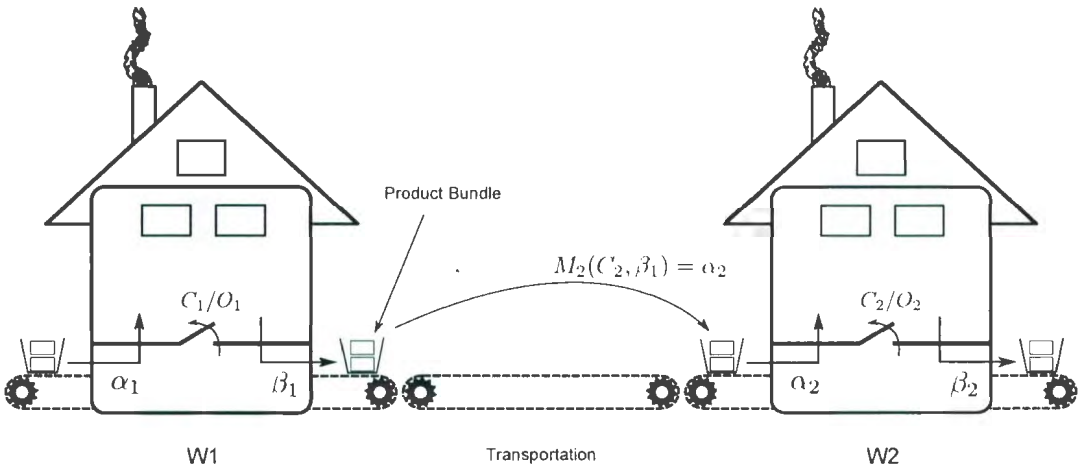

Figure 3-3: Schematic of two warehouses, each has two states of *door-open* (O_i) and *door-close* (C_i); in addition to the two events of *product-arrival* (α_i) and *product-departure* (β_i), and the event ω_i which is not shown. The warehouses are working in series, so that the *product-departure* event of W_1 (β_1) is mapped to the *product-arrival* event of W_2 (α_2) once W_2 is in the state C_2.

Example 3.2: (Two-warehouse System)

As an example, consider two warehouses each of which has two states of *door-opened* (O) and *door-closed* (C); and three events of *product-arrival* (α), *product-departure* (β), and *time-off* (ω) (see Figure 3-3). Suppose these warehouses can be modeled by the automata W_1 and W_2, shown in Figure 3-4. Consider the case that these warehouses are working in series, namely, the product departed from W_1 is sent to W_2 (see Figure 3-3). At this point, we are not going to model the compositional behaviour as a single DES, rather we are interested in modeling the system as a group of two stand-alone DES, each of which has a map.

For $\Sigma_i = \{\alpha_i, \beta_i, \omega_i\}$, $i = 1, 2$, let us define the following *maps* to encode the *in-group* behaviour:

$$M_2(C_2, \beta_1) = \alpha_2, \text{ and } M_1 \text{ is not defined.} \tag{3.23}$$

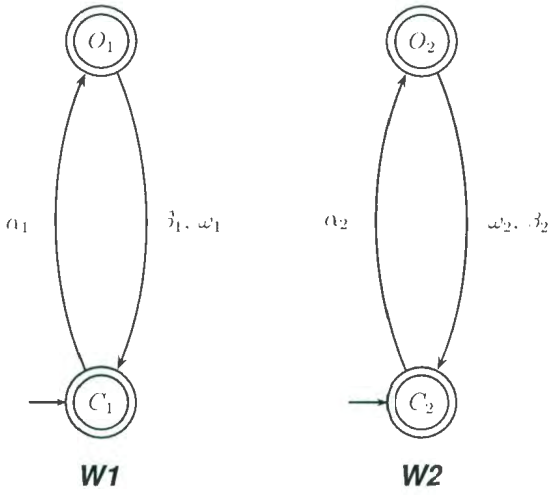

Figure 3-4: Two warehouses of Example 3.2, W_1 and W_2, in which W_2 accommodates the event α_1 of W_1 through its own event β_2.

It means that using M_2, W_2 translates β_1 into α_2, while it is in state C_2. Whereas, W_1 does not perform any translation. With such a definition, the concurrent behaviour is accurately captured without the need to re-model each system from scratch. ■

Having a stand-alone model without having to think of its synchronization with other automata (Figure 3-5) encourages an object-oriented approach to modeling. Indeed, the maps are the local interface between automata, thus the basic object-oriented concepts such as re-use and data-hiding can be taken advantage of, while the maps preserve the in-group information (that is, state/event-based specifications) of the modeled systems. Let us summarize the characteristics of a map:

1. it allows for encoding specific interactions among a set of DES without having to modify each DES model directly,

2. the input event of a map can still be blocked by an event-based specification,

3. it is local, thus it is able to encode non-symmetric translations,

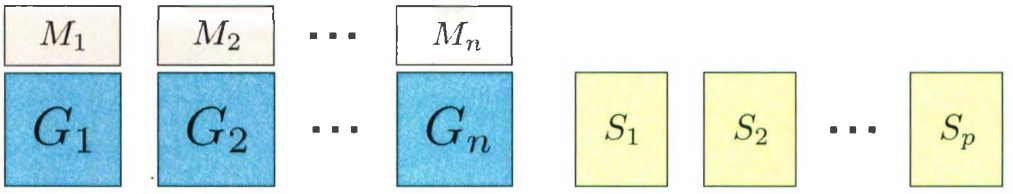

Figure 3-5: A group of DES with both *state/event-based* and *event-based* specifications, which are indicated by the set of maps $\{M_1, ..., M_n\}$ and the set of automata $\{S_1, ..., S_p\}$ respectively.

4. it is also defined over the local (owned) states, therefore it provides the ability to specify the states for which a translation is required,

5. it is not defined over the state of other systems; therefore, the state of each system is still local (hidden from the group), and the result is an event-based concurrency,

6. it is limited in the sense that the result of a map is restricted only to the *exclusive* events in the active event set of the current state.

In the next section, we utilize these concepts to establish a general composition rule which in turn allows for the desired hierarchical structure design.

3.2 Concurrent DES

Once a group of DES is given, along with required information of how each single DES locally translates the events from other DES, each DES can be promoted by its map to include those state/event-based specifications. Consider the following definition:

Definition 3.2.1 (Map-Automata) *Given a set of n DES, $\{G_1, ..., G_n\}$, and a map M_i for each, a* map-automaton $\tilde{G}_i$ *is a tuple*

$$\tilde{G}_i := (G_i, M_i), \quad i = 1, ..., n, \tag{3.24}$$

which is defined for each automaton G_i inside the group.

Therefore, a map-automaton consists of two entities: an automaton, which is normally not allowed to be changed at the run-time; and a map that can be changed (up-dated) if required. This encourages the main thread of hiding (encapsulation) and interface design in object-oriented programming. See the following example:

Example 3.3: (Three-warehouse System)

Let us extend the previous example (Example 3.2) to three warehouses. As illustrated in Figure 3-6, consider the case that the third warehouse (W_3) receives products departed from both W_1 and W_2, while W_2 only receives products from W_1. Additionally, W_1 does not receive products which are departed from the other warehouses (W_2 and W_3). To avoid the conceptual inconsistency in the case that a product departs from W_1 (thus, should be received by both W_2 and W_3), assume each product departed from W_1 is a bundle which has two parts each can be sent to a different warehouse as an *arrival product*. The corresponding automata are illustrated in Figure 3-7. The maps for this case will then be defined as:

$$M_2(C_2, \beta_1) = \alpha_2, \tag{3.25}$$

$$M_3(C_3, \beta_1) = \alpha_3, \tag{3.26}$$

$$M_3(C_3, \beta_2) = \alpha_3, \tag{3.27}$$

M_1, M_2, M_3 are not defined for all other events.

For the group of three warehouses, the three map-automata can then be defined as:

$$\tilde{W}_1 = (W_1, M_1), \quad \tilde{W}_2 = (W_2, M_2), \quad \text{and} \quad \tilde{W}_3 = (W_3, M_3), \tag{3.28}$$

where, M_1 is defined as *null* or *empty*, meaning that there is no translation local to system W_1. ■

In this example, while M_is encode the local translations, the modeling of W_1 and W_2

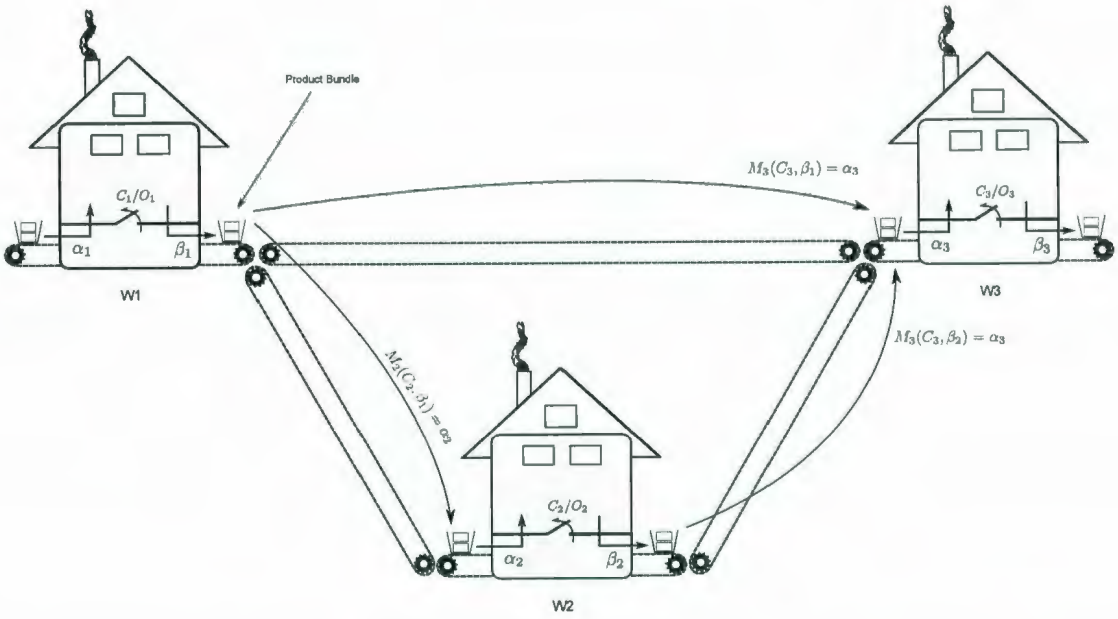

Figure 3-6: Schematic of three warehouses, each has two states of *door-opened* (O_i) and *door-closed* (C_i); in addition to the two events of *product-arrival* (α_i) and *product-departure* (β_i), and the event ω_i which is not shown. The warehouses are working in series, in such a way that the *product-departure* event of W_1 (β_1) is mapped to the *product-arrival* event of W_2 (α_1), and W_3 (α_3), while they are in their *door-closed* states (C_2 and C_3, respectively). Also, *product-departure* of W_2 (β_2) is mapped to *product-arrival* of W_3 (α_3) while it is in its *door-closed* state (C_3).

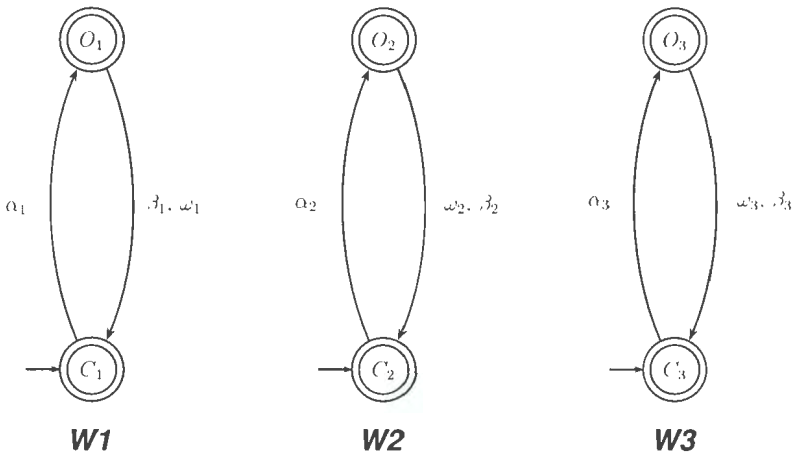

Figure 3-7: Three warehouses of Example 2, W_1, W_2, and W_3, in which both W_2 and W_3 translate the event β_1 of W_1 into their own events α_2 and α_3 respectively. In addition, W_3 also translates the event β_2 of W_2 to its own event α_3.

(the automata of example 1) has left unchanged, that is no re-labeling is required. As mentioned before, this property is an important characteristic since it allows for abstraction and encapsulation in object-oriented programming. We will return to this basic advantage in the next chapter.

Now, inside a larger group of DES, let us collect each set of map-automata as a DES collection. Formally,

Definition 3.2.2 (DES Collection) *Given a set of automata* $\{G_1, ..., G_m\}$, *the set*

$$\mathcal{C} := \{\tilde{G}_1, ..., \tilde{G}_m\} \tag{3.29}$$

is called a DES collection, where $\tilde{G}_i = (G_i, M_i)$ *is the map-automaton corresponding to* G_i *with the map* M_i *defined for the set* $\{G_1, ..., G_m\}$.

Therefore, a given set of DES can be partitioned into a finite number of smaller sets of (possibly one-element) DES, each form a DES collection, provided that inside each set, any DES is defined with a map (possibly empty). See the following example:

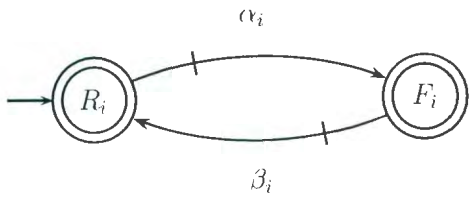

Figure 3-8: Discrete abstraction of two-mode liquid tanks of example 3.4.

Example 3.4: (DES Collection)

Suppose we have n two-mode liquid tanks, which are supposed to be at either the *filling* (F) mode or the *rest* (R) mode. Each tank is equipped with a two-mode tap, which defines whether the tank is in F or in R by two events of *tap open* (α) and *tap closed* (β). The automaton corresponding to the i-th tank is then defined as (see Figure 3-8):

$$G_i = (Q_i, \Sigma_i, \delta_i, \Gamma_i, q_{0i}, Q_{mi}), \tag{3.30}$$

where,

$$Q_i = \{F_i, R_i\}, \tag{3.31}$$

$$\Sigma_i = \{\alpha_i, \beta_i\}, \tag{3.32}$$

$$\delta_i : \quad \delta_i(F_i, \beta_i) = \{R_i\}, \quad \delta_i(R_i, \alpha_i) = \{F_i\}, \tag{3.33}$$

$$\Gamma_i : \quad \Gamma_i(F_i) = \{\beta_i\}, \quad \Gamma_i(R_i) = \{\alpha_i\}. \tag{3.34}$$

Let these tanks be arranged in two groups (with the cardinality of m_1 and m_2, $m_1 + m_2 = n$), where all the tanks of each group have to share a single tap. Suppose each tap is single-user, that is it can only fill one tank at a time, and therefore each tank has to transit to its *rest* mode (say R_i) once any other tank in the group goes to *filling* mode (say F_j). In other words, inside each group, each tank should

have a translation of all other *fillings* to its own *rest*. Formally, if the set of indices $I = \{1, ..., n\}$ is partitioned into two sets, I_1 and I_2, corresponding to the indices of automata in each groups, we have

$$M_i(F_i, \alpha_j) = \beta_i, \quad \text{for } i, j \in I_k,\ i \neq j, \text{ and } k \in \{1, 2\}. \tag{3.35}$$

Aggregating these maps and the original automata forms two collections representing the two groups (see Figure 3-9):

$$\begin{aligned} &\text{for } k = 1, 2, \\ &\mathcal{C}_k = \{\tilde{G}_i \quad | \ \tilde{G}_i = (G_i, M_i), \text{ for } i \in I_k\}. \end{aligned} \tag{3.36}$$

■

3.3 Composition Rules

It would be natural to think of a mathematical n-ary composition rule over a given DES collection. Such a rule, as far as this thesis is concerned, should result in a new DES. Additionally, to the best of our knowledge, the composition rule should satisfy basic mathematical properties to be "consistent" and "useful," while still being considered as a general rule[4]. We then make use of such a rule (as a general term) to define a hierarchical design. Consider the following definition (Cassandras and Lafortune 1999):

Definition 3.3.1 (Accessible Function) *The accessible part of an automaton* $G =$

[4]It is open to discussion what "consistent" and "useful" are, while it can be compromising that such basic properties are more intuitive than based on a firm reasoning. In this text, these properties are mostly influenced by the standard operations in the existing literature.

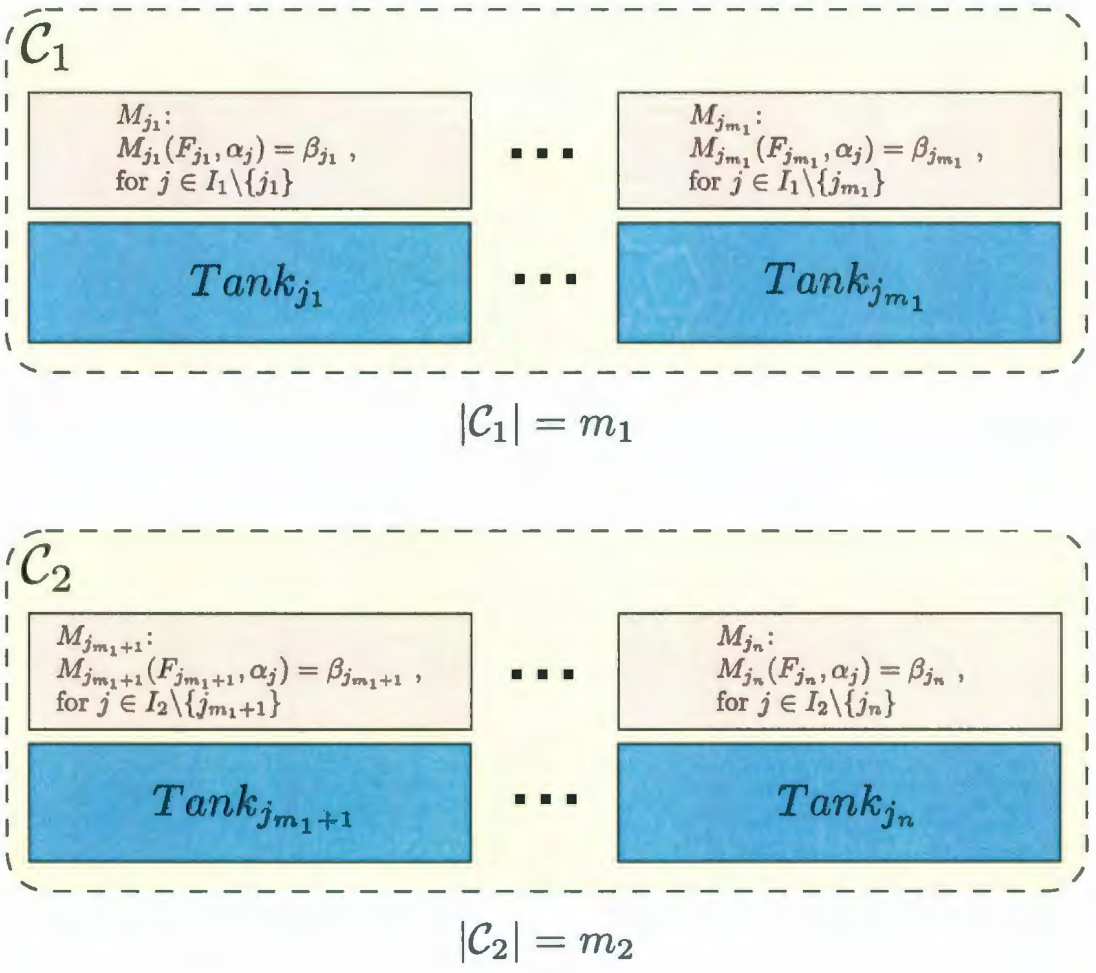

Figure 3-9: Liquid tanks of Example 3.4, grouped into two collections of $\mathcal{C}_1$ (of cardinality $|\mathcal{C}_1| = m_1$) and $\mathcal{C}_2$ (of cardinality $|\mathcal{C}_2| = m_2$).

$(Q, \Sigma, \delta, \Gamma, q_0, Q_m)$ *is denoted by* $Ac(G)$ *and defined as*

$$Ac(G) := (Q_{ac}, \Sigma, \delta_{ac}, \Gamma_{ac}, q_0, Q_{ac,m}), \text{ where} \tag{3.37}$$

$$Q_{ac} = \{q \in Q \mid \exists s \in \Sigma^* \ (\delta(q_0, s) = q)\}, \tag{3.38}$$

$$Q_{ac,m} = Q_m \cap Q_{ac}, \tag{3.39}$$

$$\delta_{ac} = \delta \mid Q_{ac} \times \Sigma \to Q_{ac}, \tag{3.40}$$

$$\Gamma_{ac} = \Gamma \mid Q_{ac} \to 2^{\Sigma}. \tag{3.41}$$

The notation $\delta \mid Q_{ac} \times \Sigma \to Q_{ac}$ means that δ is restricted to the smaller domain of the accessible states Q_{ac}, and so is Γ. Let us now define the n-ary composition rule:

Definition 3.3.2 (N-ary Composition Rule (NCR)) *For a given DES collection* $\mathcal{C} = (\tilde{G}_1, ..., \tilde{G}_m)$, *a function* $*$ *which is defined over every sequence of the mapped-automata of* $\mathcal{C}$ *as:*

$$*\mathcal{C} = \tilde{G}_1 * ... * \tilde{G}_m = \mathcal{G} \tag{3.42}$$

is an n*-ary composition rule (NCR) if the following three axioms hold:*

1. *(DES closure)* $\mathcal{G}$ *is an automaton,*
2. *(Accessibility)* $\mathcal{G} = Ac(\mathcal{G})$,
3. *(In-group commutativity)* $\tilde{G}_1 * ... * \tilde{G}_i * ... * \tilde{G}_j * ... * \tilde{G}_m = \tilde{G}_1 * ... * \tilde{G}_j * ... * \tilde{G}_i * ... * \tilde{G}_m$, *for all* $i, j \in \{1, ..., m\}$.

$\mathcal{G}$ *is called the resulting DES.*

As a result, each composition rule depends also upon how the local maps are defined (because it is defined over the set of map-automata rather than the automata

themselves). In general, the resulting transition function of a rule takes the form of:

$$\delta((q_1, ..., q_n), \sigma) = (\delta'_1(q_1, \sigma), ..., \delta'_n(q_n, \sigma)), \quad \text{for } q_i \in Q_i, \text{ and } \sigma \in \Sigma. \tag{3.43}$$

where, $\delta'_i(q_i, \sigma)$ is a local ruled-transition function and is defined based on the selected rule (which uses the local map M_i). However, the use of maps is not a necessity in the design of a composition rule: an arbitrary composition rule can be defined over a set of automata alone. We will observe how standard product and synchronous product composition operators (which are defined over a set of automata) are composition rules.

Here, a number of composition rules are presented. Special attention will also be placed on the n-ary version of standard operations to be considered as composition rules.

3.3.1 Product Composition

Definition 3.3.3 (Product Composition(Cassandras and Lafortune 1999)) *The* product *of* $G_1, ..., G_n$ *is the automaton*

$$G_1 \times ... \times G_n := Ac(Q_1 \times ... \times Q_n, \bigcap_{i=1}^{n} \Sigma_i, \delta, \Gamma, (q_{01}, ..., q_{0n}), Q_{m1} \times ... \times Q_{mn}), \tag{3.44}$$

where,

$$\delta((q_1, ..., q_n), \sigma) := \begin{cases} (\delta_1(q_1, \sigma), ..., \delta_n(q_n, \sigma)) & \text{if } \sigma \in \bigcap_{i=1}^{n} \Gamma_i(q_i), \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{3.45}$$

Thus, $\Gamma(q_1, ..., q_n) = \bigcap_{i=1}^{n} \Gamma_i(q_i)$.

By definition, product composition holds the first two axioms. And, by re-labeling, it is also verifiable that it is commutative (see (Cassandras and Lafortune 1999)). Therefore,

Lemma 3.3.4 *Product composition ($\times$) is an NCR.*

3.3.2 Synchronous Product Composition

Let us begin with the standard binary operation:

Definition 3.3.5 (Synchronous Product *Operation*) *The* synchronous product[5] operation (binary) *(by Wonham (2009) and Cassandras and Lafortune (1999)) of* G_1, G_2 *is the automaton*

$$G_1||G_2 := Ac(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, \Gamma, (q_{01}, q_{02}), Q_{m1} \times Q_{m2}), \text{ where}$$

$$\delta((q_1, q_2), \sigma) := \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & \text{if } \sigma \in \Gamma_1(q_1) \cap \Gamma_2(q_2), \\ (\delta_1(q_1, \sigma), q_2) & \text{if } \sigma \in \Gamma_1(q_1) \backslash \Sigma_2, \\ (q_1, \delta_2(q_2, \sigma)) & \text{if } \sigma \in \Gamma_2(q_2) \backslash \Sigma_1, \\ \textit{undefined} & \textit{otherwise.} \end{cases} \tag{3.46}$$

Therefore, $\Gamma((q_1, q_2)) = [\Gamma_1(q_1) \cap \Gamma_2(q_2)] \cup [\Gamma_1(q_1) \backslash \Sigma_2] \cup [\Gamma_2(q_2) \backslash \Sigma_1]$. In the literature, the synchronous product is normally defined as a binary operation. Because it is verifiable that the binary synchronous product is both commutative and associative (see (Wonham 2009) and (Cassandras and Lafortune 1999)), it can be generalized to an n-ary version without experiencing any inconsistency, which is desired for our purpose.

The core idea behind the presented n-ary definition is to observe that for a given event $\sigma \in \bigcup_{i=1}^{n} \Sigma_i$, there would be a transition only when all the automata to which σ belongs, enable it too. In that case, all those automata have their local transition $\delta_i(q_i, \sigma)$. In other words, there will be no transition (namely, σ will be blocked) if $\exists i \in \{1, ..., n\} \mid \sigma \in \Sigma_i \wedge \sigma \notin \Gamma_i(q_i)$. Let us define a set of indexes Ω, if all the

[5] Also called *parallel* composition (Cassandras and Lafortune 1999).

automata to which σ belongs, also enable it; such that Ω contains the indexes of all those automata: for a given event $\sigma \in \bigcup_{i=1}^{n} \Sigma_i$,

$$\Omega = \{\forall i \in \{1, ..., n\} \mid (\sigma \in \Sigma_i \Rightarrow \sigma \in \Gamma_i(q_i))\}. \tag{3.47}$$

Note that Ω can be an empty set (if σ belongs to some automata, but not all of those automata enable σ simultaneously). Therefore, if Ω is non-empty then there will be a local transition in all G_i, $i \in \Omega$. Let us now re-write the preceding formula including the logical condition of $(\sigma \in \Sigma_i \Rightarrow \sigma \in \Gamma_i(q_i))$ in an algebraic form of $\Omega \in 2^{\{1,...,n\}} \mid \sigma \in [\bigcap_{k\in\Omega} \Gamma_k(q_k)] \backslash [\bigcup_{k\notin\Omega} \Sigma_k]$. We state this latter clause as the condition for defining the n-ary synchronous product *composition*, and then explain the formal justification of this definition:

Definition 3.3.6 (Synchronous Product Composition) *The* synchronous product *of* $G_1, ..., G_n$ *is the automaton*

$$G_1||...||G_n := Ac(Q, \Sigma, \delta, \Gamma, q_0, Q_m), \text{ where}$$

$$Q = Q_1 \times ... \times Q_n, \tag{3.48}$$

$$\Sigma = \bigcup_{i=1}^{n} \Sigma_i, \tag{3.49}$$

$$q_0 = (q_{01}, ..., q_{0n}), \tag{3.50}$$

$$Q_m = Q_{m1} \times ... \times Q_{mn}, \text{ and} \tag{3.51}$$

$$\delta((q_1, ..., q_2)), \sigma) := (\delta'_1(q_1, \sigma), ..., \delta'_n(q_n, \sigma)), \text{ in which} \tag{3.52}$$

$$\delta'_i(q_i, \sigma) := \tag{3.53}$$

$$\begin{cases} \delta_i(q_i, \sigma) & \text{if } \exists\Omega \in 2^{\{1,...,n\}} \mid i \in \Omega \wedge \sigma \in [\bigcap_{k\in\Omega} \Gamma_k(q_k)] \setminus [\bigcup_{k\notin\Omega} \Sigma_k], \\ q_i & \text{if } \exists\Omega \in 2^{\{1,...,n\}} \mid i \notin \Omega \wedge \sigma \in [\bigcap_{k\in\Omega} \Gamma_k(q_k)] \setminus [\bigcup_{k\notin\Omega} \Sigma_k], \\ \text{undefined} & \text{otherwise.} \end{cases}$$

and δ *is undefined if any one of the* δ' *is undefined.*

Therefore,

$$\Gamma(q_1, ..., q_n) = \bigcup_{\Omega} \left[\bigcap_{k \subset \Omega} \Gamma_k(q_k) \setminus \bigcup_{k \notin \Omega} \Sigma_k \right]. \tag{3.54}$$

Consider the following theorem:

Theorem 3.3.7 *Given a set of automata* $\{G_1, ...G_n\}$.

1. $[G_1||G_2]_{n-ary} = [G_1||G_2]_{binary}$
2. $[G_1||G_2||...||G_n]_{n-ary} = [(((G_1||G_2)||G_3)||...)||G_n]_{binary}$

See Appendix A for the proof. In addition that this theorem proves the n-ary definition as the generalization of the binary definition, this is an important theorem since, firstly, it provides a formal justification for using an n-ary synchronous product instead of a flattened binary one. Secondly, it brings the advantages of the binary synchronous product. For example, using the commutativity and associativity properties of binary synchronous product, we can write:

$$G_1||G_2||G_3 = (G_1||G_2)||G_3 = (G_2||G_1)||G_3 = G_2||G_1||G_3. \tag{3.55}$$

Therefore, it is straight forward to show that the third axiom of NRC holds under the synchronous product composition. Also, similar to the product composition, by definition, synchronous product composition holds the first two axioms. Thus,

Lemma 3.3.8 *Synchronous product composition (||) is an NCR.*

The generated and marked languages of synchronous product composition can be found using the binary counterparts (which are described by (Wonham 2009) and

(Cassandras and Lafortune 1999)). Formally,

$$\mathcal{L}(G_1||...||G_n) = \bigcap_{i=1}^{n} P_i^{-1}[\mathcal{L}(G_i)], \tag{3.56}$$

$$\mathcal{L}_m(G_1||...||G_n) = \bigcap_{i=1}^{n} P_i{}^{1}[\mathcal{L}_m(G_i)]. \tag{3.57}$$

where $P_i^{-1}(.)$ is the inverse of *projection* map, P_i, for system i (see (Wonham 2009) and (Cassandras and Lafortune 1999)), defined as:

$$P_i : (\bigcup_{j=1}^{n} \Sigma_j)^* \rightarrow \Sigma_i^*, \text{ for i=1,...,n.} \tag{3.58}$$

Most importantly, Wonham (2009) explains how to apply an event-based specification using synchronous product. See the following example for the implementation of synchronous product:

Example 3.5: (An event-based Specification)

In example 3.3, say for the reason that the second warehouse is full, the specification is "to prevent *product arrival* to the second warehouse (W_2)." For the moment, let us assume that the warehouses are not in series (we will return to the complete version of this example later). The premise here is to define this specification by an automaton and then have a synchronous product of the main system and this automaton. Let the specification be characterized by the automaton S_1 (Figure 3-10). This automaton has only one event α_2 which belongs to system W_2, while the states have no connection to the state of the warehouses (however, they can still be thought of "*good*" state and "*bad*" state for "1" and "2" respectively). The specification S_1 simply shows that the event α_2 goes to an un-marked state, thus it is undesirable. The marked language of the synchronous product will then be our desired behaviour. An alternative automaton is S_2 shown in Figure 3-10 with $\Sigma_{S2} = \Sigma = \Sigma_{W1} \cup \Sigma_{W2} \cup \Sigma_{W3}$. Here, we have all the events in the self-loop, except for α_2. Because α_2 still belongs

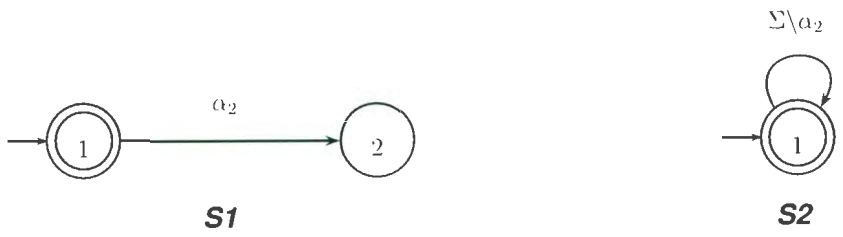

Figure 3-10: Event-based specification of Example 3.5: the event-based specification S_1, which affects the marked language, and the event-based specification S_2, which affects the language of $\mathcal{G} = W_1||W_2||W_3$.

to Σ_{S2}, this supervisor blocks all the α_2 of the main system. Now, the language of the synchronous product is our desired behaviour. ■

Indeed, synchronous product is a powerful mathematical tool in implementing supervisory control. Now, let us extend the concept of synchronous product composition, using the systems' maps.

3.3.3 Accommodating Synchronous Product (ASP) Composition

Definition 3.3.9 (Accommodating Synchronous Product (ASP) Composition) *The* accommodating synchronous product *of* $G_1, ..., G_n$ *which shape a DES collection* $\mathcal{C} = \{(G_1, M_1), ...(G_n, M_n)\} = \{\tilde{G}_1, ..., \tilde{G}_n\}$ *is the automaton*

$$\tilde{G}_1 \perp ... \perp \tilde{G}_n := Ac(Q, \Sigma, \delta, \Gamma, q_0, Q_m),$$

where,

$$Q = Q_1 \times ... \times Q_n, \tag{3.59}$$

$$\Sigma = \bigcup_{i=1}^{n} \Sigma_i, \tag{3.60}$$

$$q_0 = (q_{01}, ..., q_{0n}), \tag{3.61}$$

$$Q_m = Q_{m1} \times ... \times Q_{mn}, \ \textit{and} \tag{3.62}$$

$$\delta((q_1, ..., q_2)), \sigma) := (\delta'_1(q_1, \sigma), ..., \delta'_n(q_n, \sigma)), \ \textit{in which} \tag{3.63}$$

$$\delta'_i(q_i, \sigma) := \tag{3.64}$$

$$\begin{cases} \delta_i(q_i, \sigma) & \textit{if } \exists \Omega \in 2^{\{1,...,n\}} \mid i \in \Omega \wedge \sigma \in \left[\bigcap_{k \subset \Omega} \Gamma_k(q_k)\right] \setminus \left[\bigcup_{k \not\subset \Omega} \Sigma_k\right], \\ q_i & \textit{if } \exists \Omega \in 2^{\{1,...,n\}} \mid i \notin \Omega \wedge \sigma \in \left[\bigcap_{k \subset \Omega} \Gamma_k(q_k)\right] \setminus \left[\bigcup_{k \not\subset \Omega} \Sigma_k\right], \\ & \textit{and } M_i(q_i, \sigma) \textit{ is not defined}, \\ \delta_i(q_i, M_i(q_i, \sigma)) & \textit{if } \exists \Omega \in 2^{\{1,...,n\}} \mid i \notin \Omega \wedge \sigma \in \left[\bigcap_{k \subset \Omega} \Gamma_k(q_k)\right] \setminus \left[\bigcup_{k \not\subset \Omega} \Sigma_k\right], \\ & \textit{and } M_i(q_i, \sigma) \textit{ is defined}, \\ \textit{undefined} & \textit{otherwise.} \end{cases}$$

and δ is undefined if any one of the δ' is undefined.

Therefore,

$$\Gamma((q_1, ..., q_n)) = \bigcup_{\Omega} \left[\bigcap_{k \subset \Omega} \Gamma_k(q_k) \setminus \bigcup_{k \not\subset \Omega} \Sigma_k \right]. \tag{3.65}$$

As a result, for each DES, the ASP composition exactly works as the synchronous product does, unless when the event is an "exclusive" event of another (or another set of) DES, and at the same time, a map is defined for it. In such a case, the DES will perform a transition (namely, $\delta_i(q_i, M_i(q_i, \sigma))$) based on the translated version of σ (which is $M_i(q_i, \sigma)$). Also, note that $\Gamma(q_1, ..., q_n)$ is not changed.

For the case of only two systems, the transition function of the above definition is reduced to the following (see Appendix B for the proof):

$$\delta((q_1,q_2),\sigma) := \tag{3.66}$$

$$\begin{cases} (\delta_1(q_1,\sigma),\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_1(q_1)\cap\Gamma_2(q_2), \\ (\delta_1(q_1,\sigma),\delta_2(q_2,M_2(q_2,\sigma))) & \text{if } \sigma \in \Gamma_1(q_1)\backslash\Sigma_2 \text{ and } M_2(q_2,\sigma) \text{ is defined}, \\ (\delta_1(q_1,\sigma),q_2) & \text{if } \sigma \in \Gamma_1(q_1)\backslash\Sigma_2 \text{ and } M_2(q_2,\sigma) \text{ is not defined}, \\ (\delta_1(q_1,M_1(q_1,\sigma)),\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_2(q_2)\backslash\Sigma_1 \text{ and } M_1(q_1,\sigma) \text{ is defined}, \\ (q_1,\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_2(q_2)\backslash\Sigma_1 \text{ and } M_1(q_1,\sigma) \text{ is not defined}, \\ \textit{undefined} & \textit{otherwise}, \end{cases}$$

and, $\Gamma((q_1,q_2)) = [\Gamma_1(q_1)\cap\Gamma_2(q_2)]\cup[\Gamma_1(q_1)\backslash\Sigma_2]\cup[\Gamma_2(q_2)\backslash\Sigma_1]$.

Lemma 3.3.10 *Accommodating synchronous product ($\perp$) is an NCR.*

Proof. Similar to the previous compositions, by definition, ASP holds the first two axioms of NCR. As for the third axiom, intuitively, since each DES is on its own, the order in the composition is not important. Formally, we can prove the commutativity property by re-labeling:

Let G_1 and G_2 represent any two DES in the set $\{G_1,...,G_n\}$, and suppose $G' = G_1\perp G_2$ and $G'' = G_2\perp G_1$. Since $\forall(q_i,q_j) \in Q_1\times Q_2$, $\exists(q_j,q_i) \in Q_2\times Q_1$, we can relabel both (q_i,q_j) in G' and (q_j,q_i) in G'' with a unique new name λ_k, $k \in \{1,...,|Q_1|\times|Q_2|\}$. Therefore, the state-space of both G' and G'' are equivalent to $\Lambda = \{\lambda_k \mid k \in \{1,...,|Q_1|\times|Q_2|\}\}$. Also, for $q_i' \in Q_1$ and $q_j' \in Q_2$ let us perform the following substitution:

$$\delta_1'(q_i,g) = q_i' \text{ and } \delta_2'(q_j,g) = q_j',$$

then, we can write:

$$\delta_{G'}(\lambda_k, g) = \delta_{G'}((q_i, q_j), g) = (\delta'_1(q_i, g), \delta'_2(q_j, g)) = (q'_i, q'_j) \triangleq \lambda'_k, \tag{3.67}$$

$$\delta_{G''}(\lambda_k, g) = \delta_{G''}((q_j, q_i), g) = (\delta'_2(q_j, g), \delta'_1(q_i, g)) = (q'_j, q'_i) \triangleq \lambda'_k. \tag{3.68}$$

As a result, $\delta_{G'}(\lambda_k, g) \equiv \delta_{G''}(\lambda_k, g)$. Therefore, under this relabeling and because $\Sigma_{G'} = \Sigma_{G''} = \Sigma_1 \cup \Sigma_2$, and also $q_{G',0} = (q_{1,0}, q_{2,0}) = \lambda'_0 \in \Lambda$ and $q_{G'',0} = (q_{2,0}, q_{1,0}) = \lambda'_0 \in \Lambda$, both G' and G'' generate the same languages. On the other hand, in G', $\lambda_{k,m} = (q_{i,m}, q_{j,m})$, and in G'', $\lambda_{k,m} = (q_{j,m}, q_{i,m})$, that is our introduced relabeling can be extended to the marked states. Thus, since the transition function of G' and G'' are equivalent, they generate the same marked language as well, and therefore G' and G'' are equivalent. ■

Remember, a given map (say M_i) only preserves the local *translation* of events (which is state/event-based specifications), but does not directly affect the transition function of the corresponding system (δ_i). Thus, if the translated event (σ') is blocked by an event-based specification (which is characterized by an added DES in the group), then *no* transition will occur. This enables "blocking" to be used. Blocking is one of the basic and important concepts in the DES literature for applying control. The following extensive example demonstrates the case of having both state/event-based and event-based specifications at the same time. All the derivations are presented in detail.

Example 3.6: (ASP Composition)

Let us again return to the warehouses of Examples 3.3, but this time we assume that the warehouses are in series (as explained in the Example 3.3). As explained in Example 3.3, we encode the state/event-based specification by maps M_1, M_2, and M_3 corresponding to each warehouse. Now, we would like to analytically find the resulting DES, namely

$$\mathcal{C}_1 = \{\tilde{W}_1, \tilde{W}_2, \tilde{W}_3\}, \tag{3.69}$$

$$\mathcal{G}_1 = \tilde{W}_1 \perp \tilde{W}_2 \perp \tilde{W}_3 = (Q, \Sigma, \delta, \Gamma, q_0, Q_m). \tag{3.70}$$

First of all, note that $Q = Q_1 \times Q_2 \times Q_3$; therefore, $Q = \{q_1, \ldots, q_8\}$ with

$$\begin{aligned}
q_1 &= (C_1, C_2, C_3),\\
q_2 &= (O_1, C_2, C_3),\\
q_3 &= (C_1, O_2, C_3),\\
q_4 &= (O_1, O_2, C_3),\\
q_5 &= (C_1, C_2, O_3),\\
q_6 &= (O_1, C_2, O_3),\\
q_7 &= (C_1, O_2, O_3),\\
q_8 &= (O_1, O_2, O_3).
\end{aligned}$$

Also (because the event sets are disjoint),

$$\Sigma = \bigcup_{i=1,2,3} \Sigma_i = \{\alpha_1, \alpha_3, \alpha_3, \beta_1, \beta_2, \beta_3, \omega_1, \omega_2, \omega_3\}.$$

For each state $q_i \in Q$, $\Gamma(q_i)$ can be found directly:

$$\begin{aligned}
\Gamma(q_1) &= \Gamma((C_1, C_2, C_3)) \\
&= [\Gamma_1(C_1)\backslash(\Sigma_2 \cup \Sigma_3)] \cup [\Gamma_2(C_2)\backslash(\Sigma_1 \cup \Sigma_3)] \cup [\Gamma_3(C_3)\backslash(\Sigma_1 \cup \Sigma_2)] \\
&= \Gamma_1(C_1) \cup \Gamma_2(C_2) \cup \Gamma_3(C_3) \\
&= \{\alpha_1\} \cup \{\alpha_2\} \cup \{\alpha_3\} \\
&= \{\alpha_1, \alpha_2, \alpha_3\}.
\end{aligned}$$

Likewise, we have

$$\begin{aligned}
\Gamma(q_2) &= \{\beta_1, \omega_1, \alpha_2, \alpha_3\}, \\
\Gamma(q_3) &= \{\alpha_1, \beta_2, \omega_2, \alpha_3\}, \\
\Gamma(q_4) &= \{\beta_1, \omega_1, \beta_2, \omega_2, \alpha_3\}, \\
\Gamma(q_5) &= \{\alpha_1, \alpha_2, \beta_3, \omega_3\}, \\
\Gamma(q_6) &= \{\beta_1, \omega_1, \alpha_2, \beta_3, \omega_3\}, \\
\Gamma(q_7) &= \{\alpha_1, \beta_2, \omega_2, \beta_3, \omega_3\}, \\
\Gamma(q_8) &= \{\beta_1, \omega_1, \beta_2, \omega_2, \beta_3, \omega_3\}.
\end{aligned}$$

The transition function can then be found exhaustively. Consider the following derivations for state q_1, and all the events in $\Gamma(q_1)$:

$$\begin{aligned}
\delta(q_1, \alpha_1) &= \delta((C_1, C_2, C_3), \alpha_1) \\
&= (\delta_1'(C_1, \alpha_1), \delta_2'(C_2, \alpha_1), \delta_3'(C_3, \alpha_1)) \\
&= (\delta_1(C_1, \alpha_1), C_2, C_3) \\
&= (O_1, C_2, C_3) = q_2,
\end{aligned}$$

$$
\begin{aligned}
\delta(q_1.\alpha_2) &= \delta((C_1, C_2, C_3), \alpha_2) \\
&= (\delta_1'(C_1, \alpha_2), \delta_2'(C_2, \alpha_2), \delta_3'(C_3, \alpha_2)) \\
&= (C_1, \delta_2(C_2, \alpha_2), C_3) \\
&= (C_1, O_2, C_3) = q_3, \\
\delta(q_1, \alpha_3) &= \delta((C_1, C_2, C_3), \alpha_3) \\
&= (\delta_1'(C_1, \alpha_3), \delta_2'(C_2, \alpha_3), \delta_3'(C_3, \alpha_3)) \\
&= (C_1, C_2, \delta_3(C_3, \alpha_3)) \\
&= (C_1, C_2, O_3) = q_5.
\end{aligned}
$$

For q_2, and events in $\Gamma(q_2)$ we have:

$$
\begin{aligned}
\delta(q_2, \beta_1) &= \delta((O_1, C_2, C_3), \beta_1) \\
&= (\delta_1'(O_1, \beta_1), \delta_2'(C_2, \beta_1), \delta_3'(C_3, \beta_1)) \\
&= (\delta_1(O_1, \beta_1), \delta_2(C_2, M_{12}(C_2, \beta_1)), \delta_3(C_3, M_{13}(C_3, \beta_1))) \\
&= (C_1, \delta_2(C_2, \alpha_2), \delta_3(C_3, \alpha_3)) \\
&= (C_1, O_2, O_3) = q_7, \\
\delta(q_2, \omega_1) &= \delta((O_1, C_2, C_3), \omega_1) \\
&= (\delta_1'(O_1, \omega_1), \delta_2'(C_2, \omega_1), \delta_3'(C_3, \omega_1)) \\
&= (\delta_1(O_1, \omega_1), C_2, C_3) \\
&= (C_1, C_2, C_3) = q_1, \\
\delta(q_2, \alpha_2) &= \delta((O_1, C_2, C_3), \alpha_2) \\
&= (\delta_1'(O_1, \alpha_2), \delta_2'(C_2, \alpha_2), \delta_3'(C_3, \alpha_2)) \\
&= (C_1, \delta_2(C_2, \alpha_2), C_3) \\
&= (C_1, O_2, C_3) = q_3,
\end{aligned}
$$

$$\begin{aligned}\delta(q_2, \alpha_3) &= \delta((O_1, C_2, C_3).\, \alpha_3) \\ &= (\delta'_1(O_1, \alpha_3), \delta'_2(C_2, \alpha_3).\, \delta'_3(C_3, \alpha_3)) \\ &= (C_1, C_2.\, \delta_3(C_3, \alpha_3)) \\ &= (C_1, C_2, O_3) = q_5.\end{aligned}$$

Similarly, the transition function for all other states and the events in their corresponding active event sets can be derived exhaustively.

Finally, for q_0 and Q_m we can write:

$$\begin{aligned}Q_m &= \{(q_i, q_j, q_k) \mid q_i \in Q_{1,m} \wedge q_j \in Q_{2,m} \wedge q_k \in Q_{3,m}\} \\ &= \{(C_1, C_2, C_3)\} = \{q_1\}. \\ q_0 &= (q_{1,0}, q_{2.0}, q_{3.0}) = (C_1, C_2, C_3) = q_1.\end{aligned}$$

■

Remark 3.2: (Map-dependent Violation of Blocking)

When an event $\sigma \in \Sigma_i$ of G_i is blocked by a supervisor (an event-based specification), by the definition of ASP (Definition 3.3.9), any mapped version of σ will also be blocked (because it falls into the category of "*undefined*"). Similarly, because the range of maps is defined as the exclusively owned events (not in the alphabets of other DES in the collection), any event can be blocked by a supervisor without the fear that it will be violated by a map.

Consider the case that in Definition 3.1.3, the range of a map is defined over its complete active event set (rather than its exclusive subset), namely $M_i : Q_i \times (\Sigma \backslash \Sigma_i) \rightarrow \Gamma_i$. Then, if an event $\sigma \in \Sigma_i$ of G_i is blocked by a supervisor and at the same time σ exists as the output of a map for another event $\sigma' \notin \Sigma_i$ generated by another DES in the collection, then the occurrence of σ' can cause the transition $\delta_i(q_i, M_i(q_i, \sigma')) = \delta_i(q_i, \sigma)$ in system G_i, which must have been blocked. In other

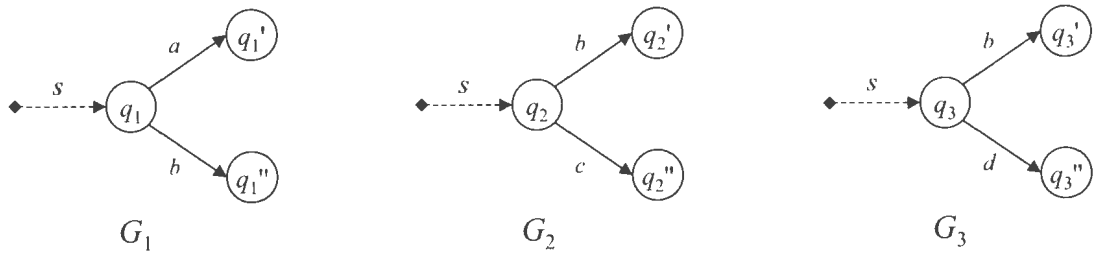

Figure 3-11: Three automata of example 3.7. The only defined map is $M_2(q_2, a) = b$ (for $q_2 \in Q_2$).

words, a "badly-designed" map can cancel the effect of a supervisor once the DES is in some specific states for which the map has been defined. Despite this adverse effect, a map with such a definition is generally more expressive, because it is free to be defined all over the $\Gamma_i(q_i)$ set.

The following example shows the case that a shared event is required to be in the range of a specific map. For simplicity, no supervisor (event-based specification) is presented in this example.

Example 3.7: (ASP Composition with Non-safe Maps)

Consider the ASP composition of three automata, G_1, G_2, G_3, with:

$$\Sigma_1 = \{a, b\}, \tag{3.71}$$

$$\Sigma_2 = \{b, c\}, \tag{3.72}$$

$$\Sigma_3 = \{b, d\}, \text{ and,} \tag{3.73}$$

$$\Sigma_\perp = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3, \tag{3.74}$$

and let only one map be defined for G_2 as $M_2(q_2, a) = b$ (for $q_2 \in Q_2$). Note that, this map is not legitimate by Definition 3.1.3, since b is not an exclusive event of G_2. Also, assume that after a given string $s \in \Sigma_\perp^*$, the current state of each automaton is

q_1, q_2, and q_3 respectively, where:

$$\Gamma_1(q_1) = \{a, b\}, \tag{3.75}$$

$$\Gamma_2(q_2) = \{b, c\}, \tag{3.76}$$

$$\Gamma_3(q_3) = \{b, d\}. \tag{3.77}$$

The case is illustrated in Figure 3-11. Let us investigate the occurrence of events a and b separately.

For a, notice $a \in \Gamma_1(q_1) \backslash [\Sigma_2 \cup \Sigma_3]$ and at the same time $M_2(q_2, a)$ is defined *and* $M_3(q_3, a)$ is not defined. Thus, by definition, we can write

$$\delta_\perp((q_1, q_2, q_3), a) = (\delta_1'(q_1, a), \delta_2'(q_2, a), \delta_3'(q_3, a)) \tag{3.78}$$

$$= (\delta_1(q_1, a), \delta_2(q_2, M_2(q_2, a)), q_3) \tag{3.79}$$

$$= (q_1', q_2', q_3). \tag{3.80}$$

It is important to note that when ASP composition is used, by definition, the map $M_2(q_2, a)$ only changes the consequence of the occurrence of event $\{a\}$ in G_2, and the result of $M_2(q_2, a)$ (which is b) will *not* cause any transition in G_1 and G_3.

For b, observe that $b \in [\Gamma_1(q_1) \cap \Gamma_2(q_2) \cap \Gamma_3(q_3)]$. Therefore,

$$\delta_\perp((q_1, q_2, q_3), b) = (\delta_1'(q_1, b), \delta_2'(q_2, b), \delta_3'(q_3, b)) \tag{3.81}$$

$$= (\delta_1(q_1, b), \delta_2(q_2, b), \delta_3(q_3, b)) \tag{3.82}$$

$$= (q_1'', q_2', q_3''). \tag{3.83}$$

If c or d occurs, since they exclusively belong to one automaton and "no" map is defined for them at the other automata, the transition will only occur at the owned automaton, while the other two automata stay on their current states.

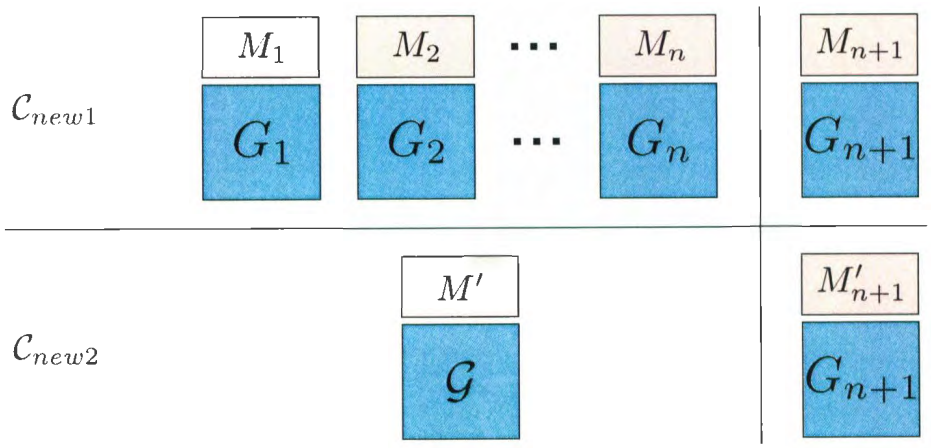

Figure 3-12: Two possibilities of adding a new automaton to a given collection. On the top, an n-ary composition is shown, where the new automaton is added to the group while the existing automata are preserved by updating their maps; and at the bottom, a binary composition is shown, where the new automata is added to the resulting DES of the existing group ($\mathcal{G}$).

In this example, if $b \in \Sigma_3$, but $b \notin \Gamma_3(q_3)$, then it must be blocked. However, one can observe that the occurrence of a can cause $\delta_2(q_2, b)$, which violates the blocking. ■

Remark 3.3: (N-ary vs. Binary)

The set of maps, $\{M_1, ..., M_n\}$, is defined in the group of automata, $\{G_1, ..., G_n\}$, *before* introducing any composition rule. The automata along with their corresponding maps, then shape a collection ($\mathcal{C}$) over which a specific composition rule can be defined. If a new DES (G_{n+1}) requires to be added to the group with "the same" composition rule, as shown in Figure 3-12, there will be two possibilities:

1. $\mathcal{C}_{new1} = \{(G_1, M_1), ..., (G_n, M_n), (G_{n+1}, M_{n+1})\}$,

2. $\mathcal{C}_{new2} = \{(\mathcal{G}, M'), (G_{n+1}, M'_{n+1})\}$.

Where, after addition, in the first line the domain of each M_i ($i \in \{1, ..., n+1\}$) is $Q_i \times \bigcap_{j=1, j\neq i}^{n} \Sigma_j$ (see Figure 3-12). In the second line, $\mathcal{G}$ is the resulting DES of

collection $\mathcal{C}$ and the given composition rule; M' is the map for $\mathcal{G}$ when G_{n+1} is added; and M'_{n+1} is the map for G_{n+1} when it is added to $\mathcal{G}$. To distinguish the difference between the two cases, observe that case (1) is an n-ary composition, while case (2) is a binary one: in $\mathcal{C}_{new1}$, G_{n+1} is added by introducing its own map along with augmenting the required translations to each initial map, while in $\mathcal{C}_{new2}$, G_{n+1} is added to the flattened result $\mathcal{G}$ (with the new state and event sets Q and Σ) by introducing M'_{n+1} and M'. The point is that, in general, $\mathcal{C}_{new1}$ and $\mathcal{C}_{new2}$ are *not* equivalent. M_{n+1} and M'_{n+1} are equivalent because both are defined as $Q_{n+1} \times \Sigma \rightarrow \Sigma_{n+1}$, and they are consistent. Whereas, M' does not have the same effect as the modification of the set $\{M_1, \ldots, M_n\}$ does. The reason is that in $\mathcal{C}_{new1}$, if for a given event of G_{n+1} (say $\sigma \in \Sigma_{n+1}$) we have a defined translation for each of G_1 through G_n (while they are in their current states), then to have the same effect in $\mathcal{C}_{new2}$, we should have a "string" of those translations (a concatenation of all those events) as the translation of σ, which is inconsistent with the definition (by definition, an event can only be translated to a single event, not a string of events). Indeed, $\mathcal{C}_{new1}$ is more general since all the information of previous DES are still in place. The following example demonstrates this issue.

Example 3.8: (Augmenting A New DES to A Given Group: "N-ary" vs. "Binary")

Consider the three automata G_1, G_2, and G_3 shown in Figure 3-13 (left side) with:

$$\Sigma_i = \{\alpha_i, \beta_i\}, \quad \text{for } i \in \{1, 2, 3\}. \tag{3.84}$$

Let G_1 and G_2 be grouped together with empty maps (no translation of events). Therefore, $G' = \tilde{G}_1 \perp \tilde{G}_2$ would be a shuffle of G_1 and G_2 as shown in Figure 3-13

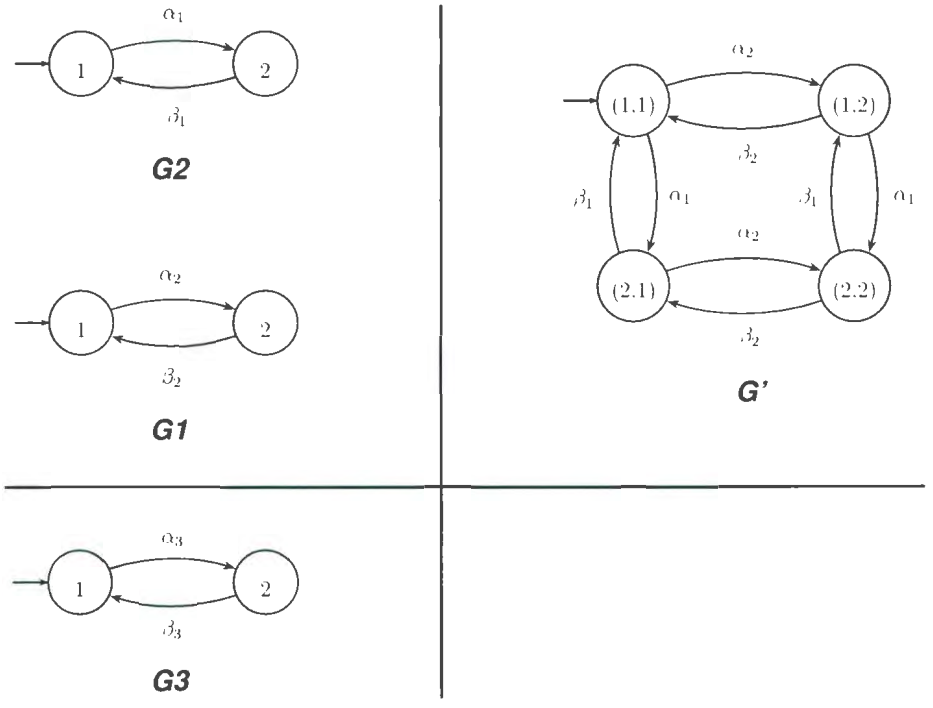

Figure 3-13: Two DES of G_1 and G_2 are grouped together with empty maps. Then, the third DES, G_3 is supposed to be added to the group as an n-ary addition (without flattening), with the maps $M_1(1, \alpha_3) = \alpha_1$, $M_2(1, \alpha_3) = \alpha_2$, and $M_3 = \emptyset$. $G' = G_1 \perp G_2$ shows the resulting automaton of G_1 and G_2 before adding G_3.

(right side). Now, let us explore the addition of G_3 to the n-ary group with the maps:

$$M_1(1, \alpha_3) = \alpha_1, \tag{3.85}$$

$$M_2(1, \alpha_3) = \alpha_2, \tag{3.86}$$

$$M_3 = \emptyset. \tag{3.87}$$

Since there is no shared event, no blocking will occur. For all the events in $\Sigma = \bigcup_{i=1}^{3} \Sigma_i$, except for α_3, the resulting automaton is a shuffle of the three automata. Thus, say G_1, G_2, and G_3 are all in state 1, and α_3 occurs. Because $\alpha_3 \in \Gamma_3(1) \backslash (\Sigma_1 \cup \Sigma_2)$, each DES will perform the following transition:

$$G_1: \quad q_{new1} = \delta_1(1, M_1(1, \alpha_3)) = \delta_1(1, \alpha_1) = 2, \tag{3.88}$$

$$G_2: \quad q_{new2} = \delta_2(1, M_2(1, \alpha_3)) = \delta_2(1, \alpha_2) = 2, \tag{3.89}$$

$$G_3: \quad q_{new3} = \delta_3(1, \alpha_3) = 2. \tag{3.90}$$

Therefore, it is identical to say that G' must go from the state $q' = (1, 1)$ to the state $q'' = (2, 2)$ by the event α_3. Let us probe if there exist a map so that the addition of G_3 to G' can have the same result as the previous case of being at the state $((1, 1), 1)$ and α_3 occurs. It is obvious that there are at least two transitions required in G' to travel from $(1, 1)$ to $(2, 2)$; therefore, it is identical to the result of the n-ary composition. As a result, as mentioned before, the map M' (the map for G') should translate α_3 to the string of either "$\alpha_1\beta_2$" or "$\alpha_2\beta_1$," which is inconsistent with the definition. ■

In this research, due to the goal of having an object-oriented design, it is preferred to preserve (and hide the information of) each DES as a separate entity. Thus, when a new automaton is added to a collection, we prefer to modify the initial maps, rather than flattening the initial collection (we prefer to use $\mathcal{C}_{new1}$).

As a final remark, given a set of automata, if we let maps be empty for all the automata, then the result of ASP will simply reduced to the standard synchronous product composition (it is directly resulted from the definition):

Proposition 3.3.11 *Synchronous product composition is a special case for ASP.*

3.4 Reachability of $\mathcal{G}$ (The Resulting DES)

Let us first epitomize a DES collection by a given composition rule:

Definition 3.4.1 (DES Ruled-collection) *A DES collection* $\mathcal{C} = \{\tilde{G}_1, ..., \tilde{G}_m\}$ *and an* n*-ary composition rule over it, shape a DES ruled-collection* $\tilde{\mathcal{C}}$ *as*

$$\tilde{\mathcal{C}} := (\mathcal{C}, *). \tag{3.91}$$

In DES context, when there is no fear of confusion, it can be simply called *ruled-collection*. Therefore, a ruled-collection always results in an automaton which is the resulting DES of the corresponding composition rule ($\mathcal{G}$ under Definition 3.3.2).

The reachable state set (and the possible transition triplets $Q \times \Sigma \times Q$) of the resulting DES of a ruled-collection (using any arbitrary composition rule) can be found by the following recursive, exhaustive algorithm:

Algorithm 3.1: *Reachability* Algorithm

input : $\mathcal{P} \leftarrow$ (a multiLogic object), $ps \leftarrow$ (the present state), $\mathcal{A} \leftarrow \emptyset$ (the set of visited state)

output: $\mathcal{S}$ (the set of transitions)

```
1  Function Reach(P, ps):
2  if ps ∈ A then
3  |  Return;
4  else
5  |  A ← A ∪ {ps}
6  end
7  foreach σ∈ Γ(ps) do
8  |  ns ← nextState(P,ps,σ);
9  |  S ← S ∪ {(ps, σ, ns)};
10 |  Reach(P, ns);
11 end
12 return S;
```

In which, the function $\texttt{nextState}(\mathcal{P}, ps, \sigma)$ evaluates $\delta_{\mathcal{P}}(ps, \sigma)$ for the selected composition rule. For the sake of visualization, this algorithm can also be used to construct the resulting automaton of a ruled-collection, using a graphical-sketch generator. As an example, using this reachability algorithm, Figure 3-14 illustrates the resulting automaton of the three warehouses of Example 3.6 with the $\wedge SP$ composition rule. To generate this graph the following renaming has been used: the events α_i with $i1$, β_i with $i2$, and ω_i with $i3$. For example, 32 means β_3. Additionally, in the product state, the *door-closed* and *door-opened* states are shown as 1 and 2 respectively. For example, (1 2 2) means W_1 is in *door-closed*, W_2 is in *door-opened*, and W_3 is also in *door-opened*. The details of the software and how to use it, will be provided in the next chapter.

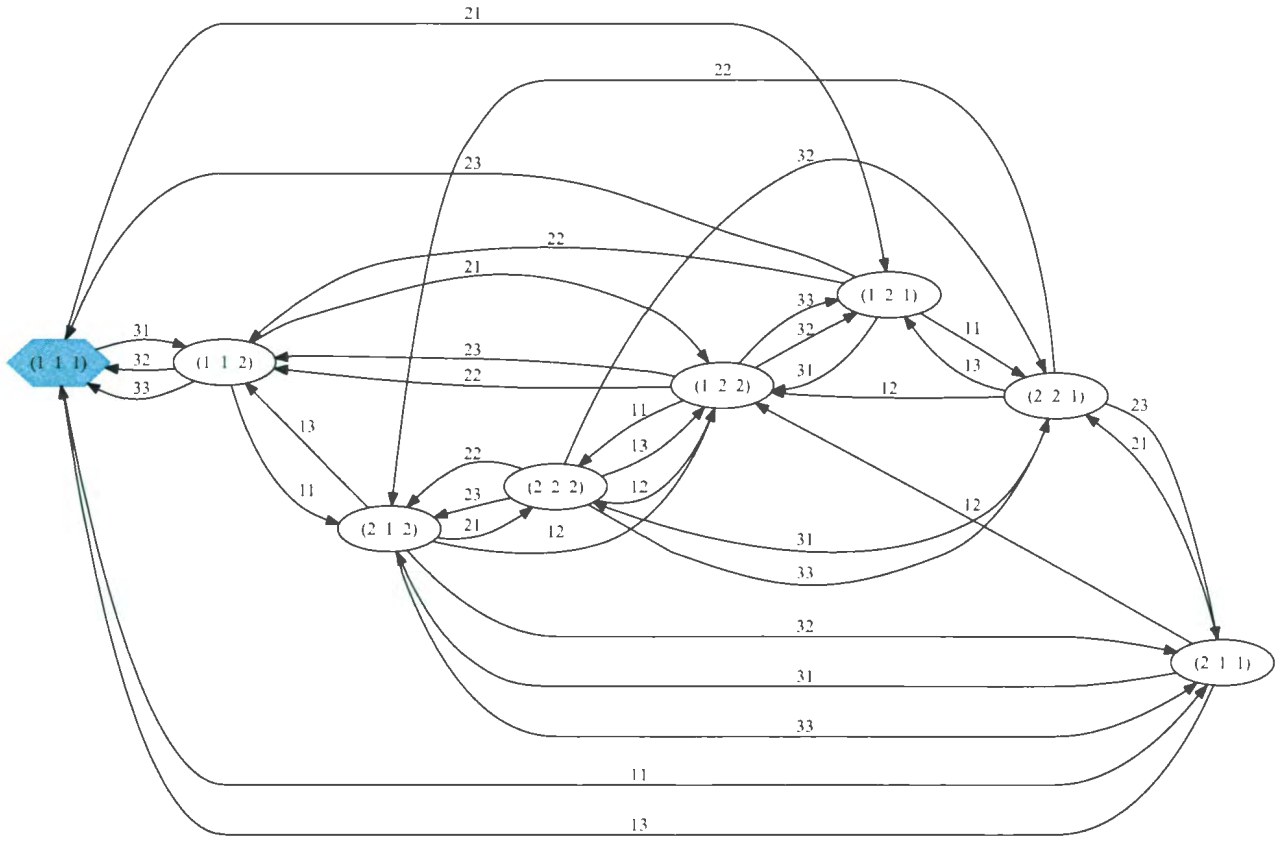

Figure 3-14: The resulting automaton of the three warehouses of Example 3.6, using the reachability algorithm (Algorithm 3.1). The cyan hexagon illustrates the initial state.

3.5 Hierarchical Composition Structures of DES

In this section, the previous concepts are extended to have a set of ruled-collections which are arranged in a hierarchical structure. Although the theory is explained and highlighted by an example, the software implementation of DES hierarchy is beyond the scope of this thesis, and will be left as a future work.

Definition 3.5.1 (DES Hierarchy) *Given a set of DES, $M = \{G_1, ..., G_n\}$, a DES hierarchy is a finite number of levels (with the cardinality of P), where in each level the existing set is partitioned into n_p (p is the level index) number of sets, each shapes a ruled-collection with the cardinality of m_i^p (i is the index of ruled-collection in a level). P is called the order of the DES hierarchy, and by definition, $n_P = 1$.*

Therefore, using NCR, a DES hierarchy defines a hierarchical structure for a finite number of DES, grouped together. Figure 3-15 exemplifies a simple DES hierarchy. The result of the p^{th} level is reducing n_p number of DES to $n_{p+1} < n_p$ number of DES. Let us denote the i^{th} DES ruled-collection in the p^{th} level by $\tilde{C}_i^p$. By definition,

$$m_i^p := |\tilde{C}_i^p|, \tag{3.92}$$

Where, $|.|$ denotes the cardinality operation. It is then straight forward to show

$$\sum_{i=1}^{n_1} m_i^1 = n, \tag{3.93}$$

$$\sum_{i=1}^{n_p} m_i^p = n_{p-1}, \quad p = 2, ..., P. \tag{3.94}$$

Now, consider the following example which illustrates how to construct and modify a DES hierarchy.

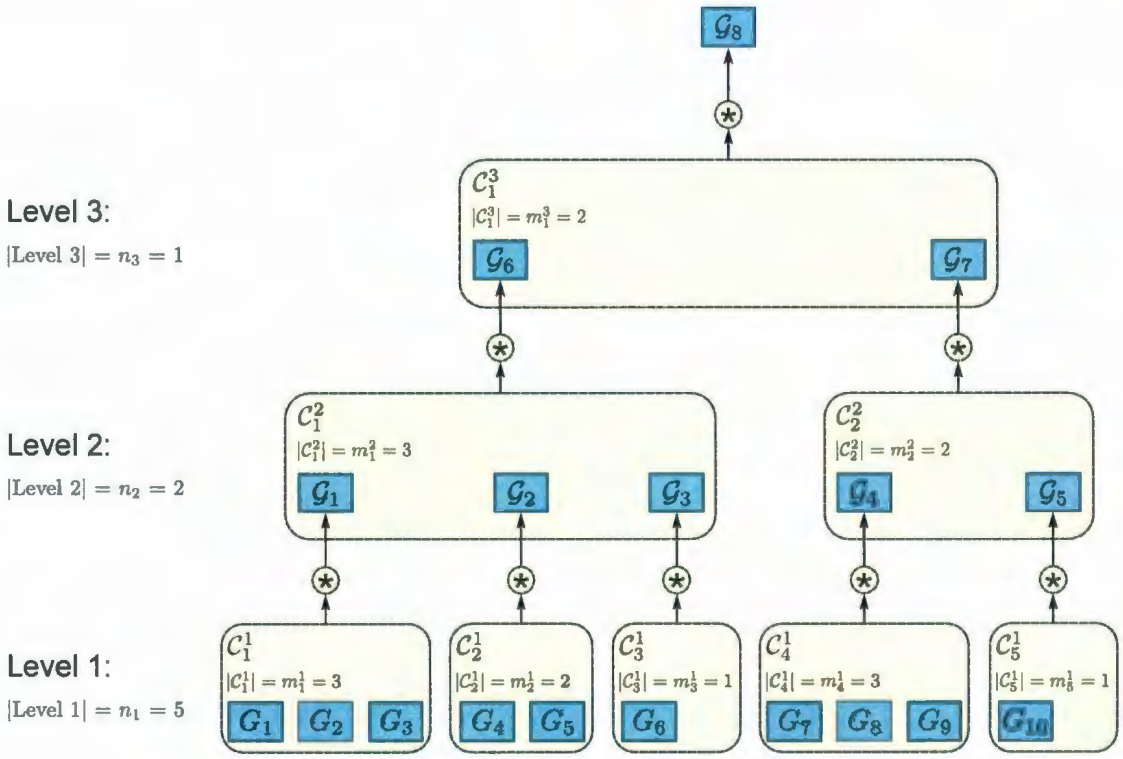

Figure 3-15: A DES hierarchy with three levels. Each blue box illustrates an automaton, and each pink box with dashed-line stroke shows a DES collection. At level one, there are 10 automata, which form five DES collections of $\mathcal{C}_1^1$ to $\mathcal{C}_5^1$, each will then be ruled by a given composition rule $*$, which can be different for each collection. Then, at level two, the resulting DES of these ruled-collections (which are $\mathcal{G}_1$ through $\mathcal{G}_5$) shape the other collections of $\mathcal{C}_1^2$ and $\mathcal{C}_2^2$, and so on. As it can be observed, at the final level, the result of the hierarchy is a single DES ($\mathcal{G}_8$).

Example 3.9: (A Simple Hierarchical Design)

Returning to example 3.4, we would like to shape a hierarchy of order three. At the first level of hierarchy, we have two collections, ruled by ASP composition to satisfy the concurrent behaviour after the mentioned tap-unification. Then, at the second level, we would like to add a supervisor (an event-based specification) to the first collection to block system G_b $(1 \leq b \leq m_1)$ from going to its *filling* state (F_b) (say, because of some technical problems in that tank). Clearly, here the choice is the synchronous product composition (though it can also be performed by ASP composition). Finally, at the third level, we would like to have a shuffle of the resulting automata. Therefore, the choice would be a synchronous product composition (notice since the resulting automata of level two do not share any event, the synchronous product composition results in a shuffle composition).

To capture the hierarchical naming, let us call the collections $\mathcal{C}_1$ and $\mathcal{C}_2$ of Example 3.4 by $\mathcal{C}_1^1$, $\mathcal{C}_2^1$. Their ruled-collections, $\tilde{\mathcal{C}}_1^1$ and $\tilde{\mathcal{C}}_2^1$, are defined as:

$$\tilde{\mathcal{C}}_i^1 = (\mathcal{C}_i^1, \perp) \quad \text{for } i = 1, 2. \tag{3.95}$$

The result of each ruled-collection of level one will then be a new automaton (say $\mathcal{G}_i^1$, $i = 1, 2$). Now, at the second level of hierarchy, we need a product composition of $\mathcal{C}_1^1$ and the following supervisor:

$$S := (Q_S, \Sigma_S, \delta_S, q_{0S}, Q_{mS}), \quad \text{such that,} \tag{3.96}$$

$$Q_S := Q_{mS} := \{q_S\}, \tag{3.97}$$

$$\Sigma_S := \bigcup_{i=1}^{m_1} \Sigma_i, \tag{3.98}$$

$$q_{0S} := q_s, \tag{3.99}$$

$$\forall \sigma \in \Sigma_S \backslash \alpha_b, \quad \delta_S(q_S, \sigma) = q_S. \tag{3.100}$$

That is, S is a single-state supervisor with a self-loop including all the events of the collection except for α_2, and at the same time, $\alpha_2 \in \Sigma_S$. Consequently,

$$\tilde{\mathcal{G}}_1^1 = (\mathcal{G}_1^1, \emptyset), \tag{3.101}$$

$$\tilde{S} = (S, \emptyset), \tag{3.102}$$

$$\mathcal{C}_1^2 = \{\tilde{\mathcal{G}}_1^1, \tilde{S}\}. \tag{3.103}$$

$$\tilde{\mathcal{C}}_1^2 = (\mathcal{C}_1^2, ||), \tag{3.104}$$

$$\tilde{\mathcal{C}}_2^2 = \tilde{\mathcal{C}}_2^1. \tag{3.105}$$

The last line emphasis the fact that the second ruled-collection of level two is that of level one without any change (an exact copy). Computationally, we may need only to make a reference to $\tilde{\mathcal{C}}_2^1$ instead of overloading it. Again, let us call the resulting automata of these ruled-collections as $\mathcal{G}_i^2$, $i = 1, 2$, and finally, at the third level,

$$\tilde{\mathcal{G}}_i^2 = (\mathcal{G}_i^2, \emptyset), \quad i = 1, 2, \tag{3.106}$$

$$\mathcal{C}_1^3 = \{\tilde{\mathcal{G}}_1^2, \tilde{\mathcal{G}}_2^2\}, \tag{3.107}$$

$$\tilde{\mathcal{C}}_1^3 := (\mathcal{C}_1^3, ||). \tag{3.108}$$

■

Let us emphasize the fact that once a DES hierarchy is designed, (1) a DES can be added to and/or deleted from each ruled-collection at any level of hierarchy provided that the maps are updated if required; (2) both the event-based specifications and the maps are allowed to be changed; and (3) once a DES collection is defined, from the rest of hierarchy it is exclusively considered as its resulting DES. Figure 3-16 summarizes the entire design process. The computational implementation and algorithm design for this structure will be explained in Chapter 4.

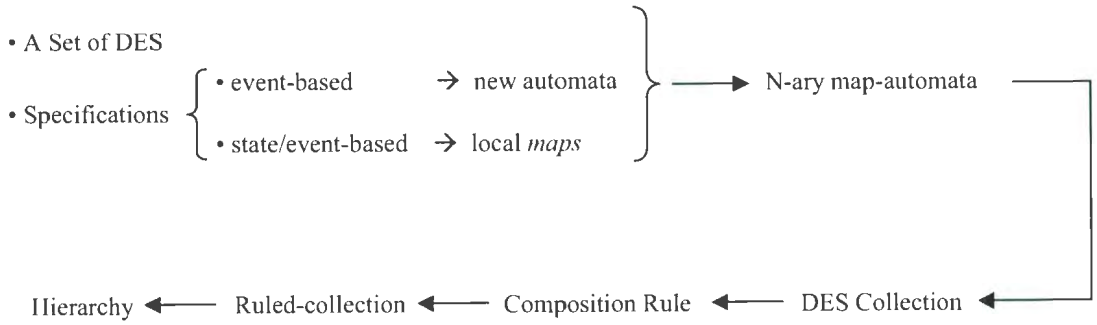

Figure 3-16: A high-level structure of a multi-DES system including two types of specifications.

3.6 A Class of Hybrid Systems

In this section, we briefly extend the results of the previous sections to a class of hybrid systems. A hybrid system is a multi-phased physical plant which is expected or allowed to follow logical behaviours through jumping among its phases. These logical attributes originate from either physical restrictions or our control objectives (or disturbance), and can be best described by the discrete event system (DES) theory. On the other hand, the physical behaviours of a hybrid system are often explained with differential or difference equations, typically derived from physical laws governing the dynamics of the system under consideration. Thus, the interaction between physical and logical behaviours results in the incorporation of discrete and continuous mathematics and modeling issues. To address this challenge, a number of different approaches have been proposed including the theory of hybrid automata by Henzinger (1996); Tomlin, Lygeros and Sastry (2000); and Cassandras and Lygeros (2006), and the switched continuous model by Millan (2006).

3.6.1 Hybrid Automata

In this thesis, a deterministic hybrid automaton (as opposed to stochastic hybrid automaton) is considered as follows:

Definition 3.6.1 (Deterministic Hybrid Automata) *A deterministic hybrid automaton is a septuple*

$$H := (Q, X, f, Dom, r, \Sigma, \delta), \tag{3.109}$$

which characterizes the evolution of continuous state variables $x \in X \subseteq \mathbb{R}^n$, and discrete state variables $q \in Q$ (of cardinality $|Q| = n$) by means of four entities

- *a vector field $f : Q \times X \to X$,*
- *a domain map $Dom : Q \to 2^X$,*
- *discrete event set Σ, which characterizes a collection of functionals with unique alphabetical name (can be integer numbers), and*
- *a reset function $r : Q \times Q \times X \to X$.*

Therefore, a hybrid automaton is a system with discrete modes Q (of cardinality $|Q| = n$) and discrete events Σ. We denote f as the set of continuous dynamics given by the functions f_i, with $i \in \{1, ..., n\}$. Each mode $q_i \in Q$ has dynamics $\dot{x} = f_i(x)$, in which the continuous state is $x \in X$. The transition function δ is deterministic, and indicates the evolution of the discrete state when a transition occurs: $q_{k+1} = \delta(q_k, \sigma)$. The set of events which can enable or force the hybrid system to transition between modes is indicated by Σ. The events in Σ fall into one of three categories (Oishi 2003): they can be controlled, disturbance, or automatic (determined by conditions on the continuous states). In this thesis, however, we only consider automatic events. The map $Dom(q)$ provides the continuous domain of the discrete state q. The initial set is (q_0, x_0), which will then be reset after each transition by the reset function r.

The modeling framework used here is a simplification of that presented in (Tomlin et al. 2000) and (Cassandras and Lygeros 2006). Basic introductions to hybrid systems and hybrid automata can be found in (Branicky 1994) and (Lygeros, Tomlin and Sastry 2008).

Let us separate a hybrid automaton into two entities of physical behaviours (time-driven and continuous dynamics) and logical behaviours (event-driven and discrete dynamics). Namely, a hybrid automaton can be re-written as:

$$H := (P, G), \text{ where} \tag{3.110}$$

$$P := (Q, X, f, Dom, r), \text{ and} \tag{3.111}$$

$$G := (Q, \Sigma, \delta). \tag{3.112}$$

The physical entity, P, is called a *time-driven system model* (TSM), and is a collection of continuous systems, each is labeled by a discrete state, and the logical entity, G, is a DES whose event set Σ is defined from the "automatic" guards (events). The details of partitioning the state-space in order to define the automatic guards, is explained in (Millan 2006). Basically, the state-space of each mode of operation (represented by a discrete state) can be partitioned through a set of functionals. Then, zero-crossing of each functional in a specific direction (either from greater values ($\downarrow$), or from lower values ($\uparrow$), or both) will generate an event, which is called as guards [6]. We do not go much through the details of this well-developed theory; instead, we are interested in deploying the theoretical structure designed in the previous sections for the logical entity of hybrid automata. The following example explains a primitive modeling problem using hybrid automata theory, which helps the understanding of basic concepts before we proceed with the computational design.

[6] Also known as output events.

Example 4.1: (Bouncing Ball)

As an example, consider a bouncing ball, which has two discrete states of *falling* (q_f), and *jumping* (q_j). Let there be no input defined. For the continuous state-space, let us assume that with the ball's vertical position from the floor (h) and the ball's velocity ($\dot{h}$) we can capture the physical dynamics completely. Thus,

$$Q = \{q_f, q_j\}, \tag{3.113}$$

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} h \\ \dot{h} \end{bmatrix}. \tag{3.114}$$

Using the physical laws of motion, the vector fields corresponding to each discrete state will be driven as:

$$f_1: \quad \dot{X} = \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} x_2 \\ -g - \frac{c_d}{m} x_2 \end{bmatrix}, \tag{3.115}$$

$$f_2: \quad \dot{X} = \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} x_2 \\ -g - \frac{c_d}{m} x_2 \end{bmatrix}, \tag{3.116}$$

where, g is the acceleration due to gravity, $c_d > 0$ is the drag coefficient (due to air resistance), and m is the mass of the ball. We will also use a second index to indicate the initial and final values of each state variable (namely, $x_{k,0}$ and $x_{k,f}$ demonstrate the initial and final values of the k-th state variable respectively). Note that the dynamics of both modes are identical in this state-space model. However, to capture the logical (switching) behaviour of the ball, it is more convenient to have two separate discrete states corresponding to *falling* and *jumping* (for the modeling by only one discrete state with a self-loop see (Lygeros et al. 2008)). Then, in each discrete switching, the initial value of velocity ($x_{2,0}$) should be reset to the negative of its final value from the last mode ($-x_{2,f}$). Therefore, the initial value for each discrete state comes from

the reset function:

$$\begin{bmatrix} x_{1,0} \\ x_{2,0} \end{bmatrix} = r(q_m, q_n, \begin{bmatrix} x_{1,f} \\ x_{2,f} \end{bmatrix}) = \begin{bmatrix} x_{1,f} \\ -x_{2,f} \end{bmatrix}, \text{ for } m \neq n \in \{f, j\}, \tag{3.117}$$

where, $x_{1,f}$ and $x_{2,f}$ are the final value of the continuous states in the previous discrete state. Note that, for simplicity, the energy lost due to restitution has not been considered.

Now, we introduce the following guards:

$$\sigma_1 : \quad x_1 = 0, \quad \downarrow, \tag{3.118}$$

$$\sigma_2 : \quad x_2 = 0, \quad \downarrow . \tag{3.119}$$

Where, the downward arrows indicates that the guard considered as an "event" only when the value of x_i reaches zero from positive values. Therefore, $\Sigma = \{\sigma_1, \sigma_2\}$, where, σ_1 indicates the event of reaching the floor, while σ_2 indicates the event of reaching the maximum height. The following automaton indicates the logical behaviour of the ball:

$$G = (Q, \Sigma, \delta, q_0, Q_m), \text{ with} \tag{3.120}$$

$$Q = Q_m = \{q_f, q_j\}, \tag{3.121}$$

$$\Sigma = \{\sigma_1, \sigma_2\}, \tag{3.122}$$

$$q_0 = q_f, \tag{3.123}$$

$$\delta : \quad \delta(q_f, \sigma_1) = q_j, \quad \delta(q_j, \sigma_2) = q_f. \tag{3.124}$$

We assumed that the initial *physical* behaviour is *falling*, and also that both modes of physical behaviours are considered as "desired," thus, both discrete states are marked. This simple automaton effectively captures the logical behaviours of our system. More importantly, it allows for applying a specification. As an example for an event-based

specification, let us assume that at some point, the floor moves out of the way so that the ball no longer has the physical limit. To apply this new constraint without "changing" the main model, it is adequate to add a new DES with the synchronous product to block σ_1 (see Chapter 3, Example 3.5). The specification can be defined by the following DES:

$$S = (Q_S, \Sigma_S, \delta_S, q_{0S}, Q_{mS}), \text{ with} \tag{3.125}$$

$$Q_S = Q_{mS} = \{q_S\}, \tag{3.126}$$

$$\Sigma_S = \Sigma, \tag{3.127}$$

$$q_{0S} = q_S, \tag{3.128}$$

$$\delta_S : \quad \delta_S(q_S, \sigma_2) = q_S. \tag{3.129}$$

■

We mentioned that in this view of the hybrid automata, the physical entity is the source of event generation, while the logical entity modifies the transitions effectively. Now, consider a set of hybrid automata $\{H_1, ..., H_n\}$, which are grouped together by allowing communication only in their logical level of abstraction. As a result, the physical entities are supposed to generate the events independently (without accessing to the state-space and dynamics of other systems), while the event names can be the same (that is, in the logical level of abstraction, different systems can have shared events in their event sets). Thus, we have a set of DES, $\{G_1, ..., G_n\}$, which can shape ruled-collections and hierarchy as explained in the previous sections. Also, note that additional hybrid automata with proper physical entity can be added to the group for event-based specifications, and maps can also be defined to encode the state/event-based specifications. This will be clarified by an example in the next chapter (under Section 4.6).

As a final remark, in order for the software development to be performed efficiently, we need also to consider issues such as parallel computation once solving ordinary differential equations. The computational implementation of this new look to hybrid automata is discussed in the next chapter.

Chapter 4

Computational Design

In this research, a software package has been developed as a testbed for the theories introduced in Chapter 3. The core structure exactly follows the presented definitions. For the sake of software reliability and reusability, an object-oriented design has been considered, in addition to employing standard technologies such as XML as the data transmission format to standard DES software. All the algorithms and platform design are implemented in MATLAB with the consideration of vectorization (as opposed to traditionally scalar software design).

4.1 General Structure

The software integrates two main entities of `multiLogic` and `Generator` as its main "classes," which represent the two main entities of *logical* and *physical* behaviours respectively, as explained in the previous chapter. Technically,

- each *automaton* is represented as an object of the `Logic` class.
- The `Logics` together with their maps then form a *DES collection* defined by an object of the `multiLogic` class. As a *DES collection*, `multiLogic` encapsulates the entire logical part and manages the concurrency of multi-system problems in a memory-efficient manner.

- The *composition rules* (product, synchronous product, and ASP) are applied as the methods of the `multiLogic` class.
- A *ruled-collection* is also characterized by the class `multiLogic`.
- For hybrid systems of the form explained in Section 3.6:
  - All the entities regarding the time-driven behaviours of the hybrid system along with managing simulations and parallel computations of multi-system problems, are encapsulated in the `Generator` class. An object of the `Generator` class is a collection of TSM as explained in Chapter 3 (Section 3.6).
  - `MultiLogic` and `Generator` are then collected as two objects in a general class called `HySys` which in turn applies all the required functionalities by calling the corresponding methods.

This structure allows for both DES and hybrid systems to be added to or deleted from the current n-ary collection, as the theory suggests. In `HySys` (the class representing hybrid systems), the synchronization only occurs in the logical part, which is the `multiLogic` component. Figure 4-1 highlights the object-oriented structure of the software as explained here. The execution process starts by running the `Generator` from an initial value up to reaching a `guard` (an *automatic* event defined as a partitioning functional over the state-space) by any of the physical sub-systems. Then, this `guard` will be passed to the `multiLogic` to be passed properly to each `Logic`. Each `Logic` then specifies a transition (if any) based on its pre-defined transition matrix (called `transArray`), which can come from an XML-file, and the entire *composition rule*, which can be either of `synchronous product`, `ASP`, `product`, or `shuffle`. Finally, the result transition which is selected by the `multiLogic` will be fed back into the `Generator`, and the process will be repeated by resetting the `Generator` to a new mode of operation (with proper initial value inherited from the last mode of operation).

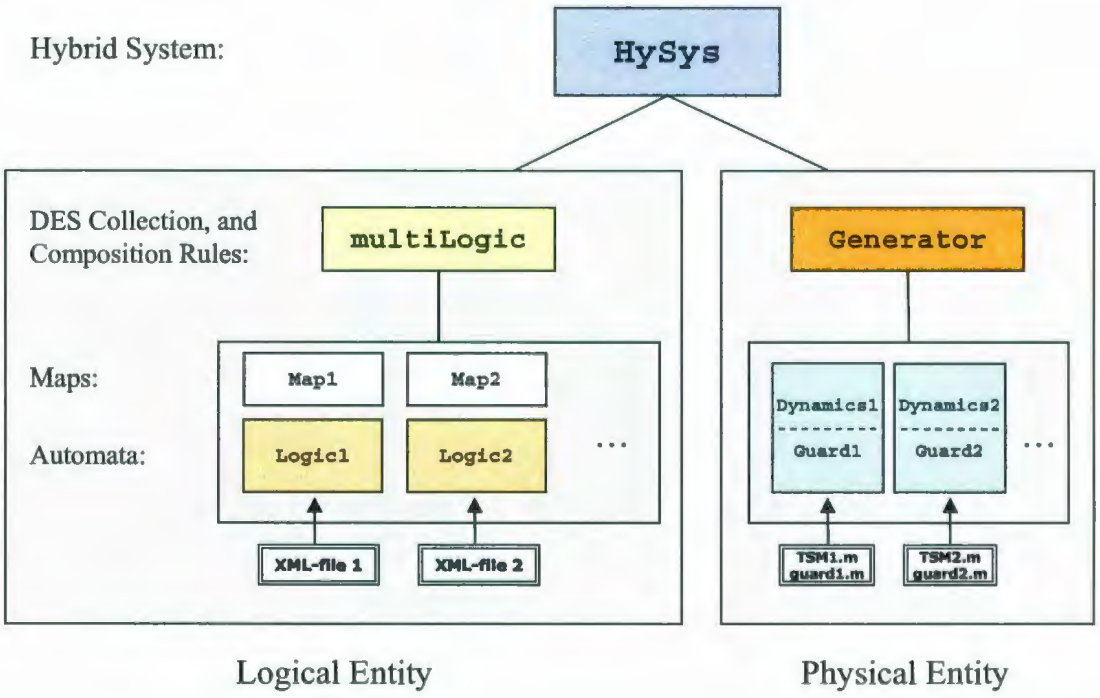

Figure 4-1: Object-oriented structure of the software

For pure DES problems (problems involving only DES without any continuous dynamics behind), the `multiLogic` class can also be used separately. In such a case, again, each DES can be introduced either by an XML file or by a MATLAB script to shape a `Logic`, then collected together as a `multiLogic` object. A set of various *methods* are available for manipulating them (for example, checking for *deadlock* states) in addition to different composition rules as defined in the previous chapters.

4.2 Logical Entity

The software offers a set of different methods for handling a collection of DES as an n-ary. Two main issues have to be considered:

1. the capability of importing data from the standard DES software,

2. memory-usage efficiency in concurrent systems.

4.2.1 Import from standard DES software

Most of the current existing DES software (such as IDES[1] and JFLAP[2]) use the *Extensible Markup Language* (XML) as the format to save information (state names, event names, transitions, properties, and graphical structure). As a result, XML has been selected in our software as the standard for importing of external data (of an automaton model). However, once the XML-files generated by an external software are imported to our software, they will be reformatted to a new data structure, while the graphical/structural information will be ignored.

[1]IDES (Integrated Discrete-Event Systems) is a Java-based software developed by The Open-Symphony Group at Queen's university and under the supervision of K. Rudie (Rudie 2008).

[2]JFLAP is a package of graphical tools which can be used as an aid in learning the basic concepts of Formal Languages and Automata Theory and is under the support of The National Science Foundation (NSF). JFLAP is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License (Rodger and Finley 2006) and (Rodger 2009).

For the tests and examples of this chapter IDES has been selected. Any alternative software can also be used as long as it supports the standard XML 1.0 format. For parsing the XML file, a free-shared package called *xml-tree* is used, which helps importing XML files in MATLAB environment.

4.2.2 Memory-usage Efficiency in Concurrent Systems

Once a new system is added to the current system, the straight-forward method is to perform the composition operation first and then save the resulting flattened automaton. In such a case, the number of states and transitions (and as a result the required memory to hold the information) will be increased dramatically. In this work, in addition to the object-oriented design (which was counted as the main premise in this thesis). for the sake of efficiency in memory-usage, each automaton (can be read from an XML-file) is managed independently in a class called `Logic`. Then, all the objects of the `Logic` class will be managed together in the main `multiLogic` class with appropriate methods. Because, at the end of the day, the entire *DES collection* of a `multiLogic` object with a given *composition rule* should be treated as a single DES (regardless of its non-flattened structure), a convention has been introduced as a standard to numbering the states of the new system (the composition of old systems). This standard is implemented by the private methods `multiLogic.new2old()` and `multiLogic.old2new()`, whose algorithms are provided in Appendix C.

4.2.3 Class Structure of The Logical Entity

Each single logical constituent, which is described by an automaton, is encapsulated by the class `Logic`. The main components of this class are illustrated in Figure 4-2. The properties of a `Logic` object are normally read from an XML-file directly (they can also be modified by a MATLAB script). In such a case, the XML-file should be compatible with the IDES software standard. Then, all the `Logic` objects are encapsulated in the class `multiLogic` ready for an n-ary composition. The components of

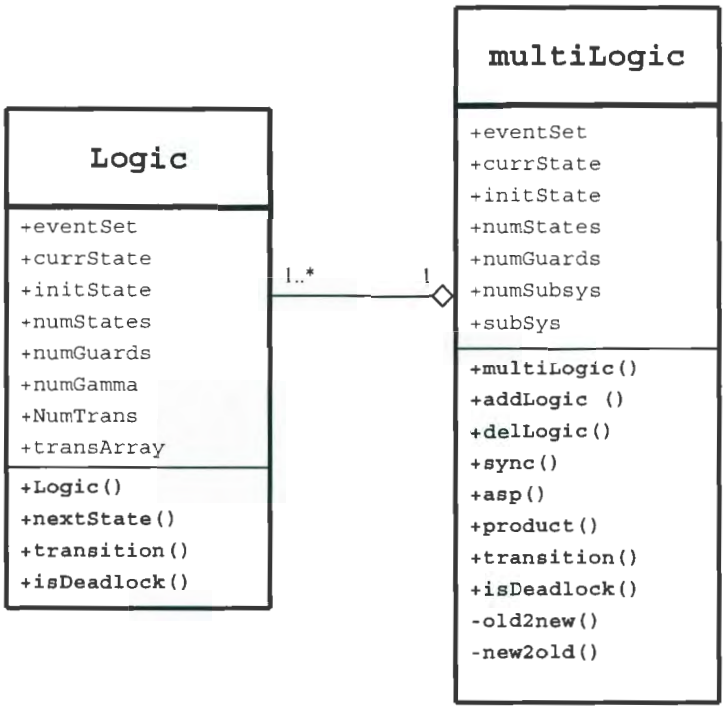

Figure 4-2: Class diagram of `Logic` and `multiLogic` classes (in UML standard).

`Logic` and `multiLogic` are illustrated in Figure 4-2.

A `multiLogic` consists of at least one `Logic` object inside its `subSys` property. In the case of having more than one `Logic` object, it should also be modified with the map vectors for each new `Logic` object. For more details see Section 4.6. The algorithms implementing the main methods are as follows:

4.2.4 Class: `Logic`, method: `nextState`

Algorithm 4.1: *nextState* method of the *Logic* class

input : $\mathcal{G} \leftarrow$ (a DES object), $eventName \in \Sigma$

output: $resultState \in (Q \vee \{False\})$

```
1 Function nextState(G, eventName);
2 if eventName ∈ Γ(G.currentState) then
3 |   resultState ← G.transArray(G.currentState, eventName);
4 else
5 |   resultState ← False;
6 end
7 return resultState;
```

This method evaluates what the next state is for each automaton (which is $\delta_i(q_i, \sigma)$) (abstracted in a `Logic` object); however, it does not perform a transition.

4.2.5 Class: `Logic`, method: `transition`

This method implements the transitions of each single DES (abstracted in a `Logic` object).

Algorithm 4.2: *transition* method of the *Logic* class

input : $\mathcal{G} \leftarrow$ (a DES object), $eventName \in \Sigma$

output: $\mathcal{G}$

```
1 Function transition(G, eventName);
2 temp ← nextState(G, eventName);
3 if temp ≠ False then
4 |   G.currentState ← temp;
5 end
6 return G;
```

Algorithm 4.3: *isDeadlock* method of the *Logic* class

input : $\mathcal{G} \leftarrow$ (a DES object), $stateName \in Q$

output: *result*

```
1 Function isDeadlock(G, stateName);
2 if G.numGamma(stateName) ≠ 0 then
3 |   result ← False;
4 else
5 |   result ← True;
6 end
7 return result;
```

4.2.6 Class: `Logic`, method: `isDeadlock`

This method evaluates if a given state is a deadlock state in each single DES (abstracted in a `Logic` object).

4.2.7 Class: `multiLogic`, method: `transition`

Algorithm 4.4: *transition* method of the *multiLogic* class

input : $\mathcal{G} \leftarrow$ (a `multiLogic` object), $stateName \in Q$ (new)

output: $\mathcal{G} \leftarrow$ (a `multiLogic` object)

1 `Function transition`$(\mathcal{G}, stateName)$;

2 $nextState \leftarrow \mathcal{G}.method(eventName)$;

3 $\mathcal{G}.currState \leftarrow nextState$;

4 $nextStates \leftarrow \mathcal{G}$.`new2old`$(nextState)$;

5 **foreach** $i \in \{1, ..., \mathcal{G}.numSubsys\}$ **do**

6 $\quad \mathcal{G}.subsys.logic\{i\}.currState \leftarrow nextStates(i)$;

7 **end**

8 **return** $\mathcal{G}$;

This method performs a transition in a `multiLogic` object based on a given method which can be one of `sync`, `asp`, and `product`.

4.2.8 Class: `multiLogic`, method: `isDeadlock`

This method evaluates if a given state is a deadlock state in a `multiLogic` object regardless of the existing maps.

Algorithm 4.5: *isDeadlock* method of the *multiLogic* class

input : $\mathcal{G} \leftarrow$ (a `multiLogic` object), $stateName \in Q$ (new)

output: $result \in \{0, 1, 2\}$; 0 means not *deadlock*, 1 means *deadlock*, 2 means some subsystems are at *deadlock*.

```
1  Function isDeadlock(G, stateName);
2  stateSet ← new2old(G, stateName);
3  temp ← zeros(G.numSubsys);
4  foreach i ∈ {1, ..., G.numSubsys} do
5  |  temp(i) ← G.subsys.logic{i}.isDeadlock(stateSet(i));
6  end
7  if sum(temp) = length(temp) then
8  |  result ← 1;
9  else if sum(temp) = 0 then
10 |  result ← 0;
11 else
12 |  result ← 2;
13 end
14 return result;
```

4.3 Physical Entity

In this entity, parallel simulation and resetting of a set of physical systems (with continuous dynamics coming from external m-files) are performed. Each physical system (a time-driven system model (TSM), and its guard generator) is characterized by a pair of m-files representing a TSM and its corresponding Guard. Each guard can have access to only its own TSM (as a local information), that is, by definition, no synchronization/communication is allowed to occur in the physical level of abstraction. Similar to the class `multiLogic`, all the constituents of the physical entity is encapsulated in a class called `Generator`. The main role of `Generator` is to generate events from a given continuous dynamical models. This class reads the continuous dynamics from separate pairs of m-files corresponding to each physical system and its guard. For example, if we have three physical systems we must have three m-files for the TSM's and three others for the corresponding guards. The main components of `Generator` are shown in Figure 4-3. Note that a `Generator` object does not select its current mode of operation: it will perform the appropriate simulations based on a "selected" mode of operation and a proper initial value vector. A simulation can be done by calling the method `simulate`, for which the mode of operation, start time, and initial value should be provided as the input arguments. A `Generator` object can be reset by the method `Generator.reset` (so that the new initial value will be the last value of the state vector, obtained from the last use of `simulate` method).

4.4 Class `HySys`: Modeling of Hybrid Systems

Both the physical (`Generator`) and the logical (`multiLogic`) entities are collected in the class `HySys`. Thus, each object of the class HySys is a DES collection with the corresponding physical systems as the source of event generation. This class has proper methods for adding new hybrid systems, deleting previous hybrid systems, and running the entire system to see the results. It also manages the interface between

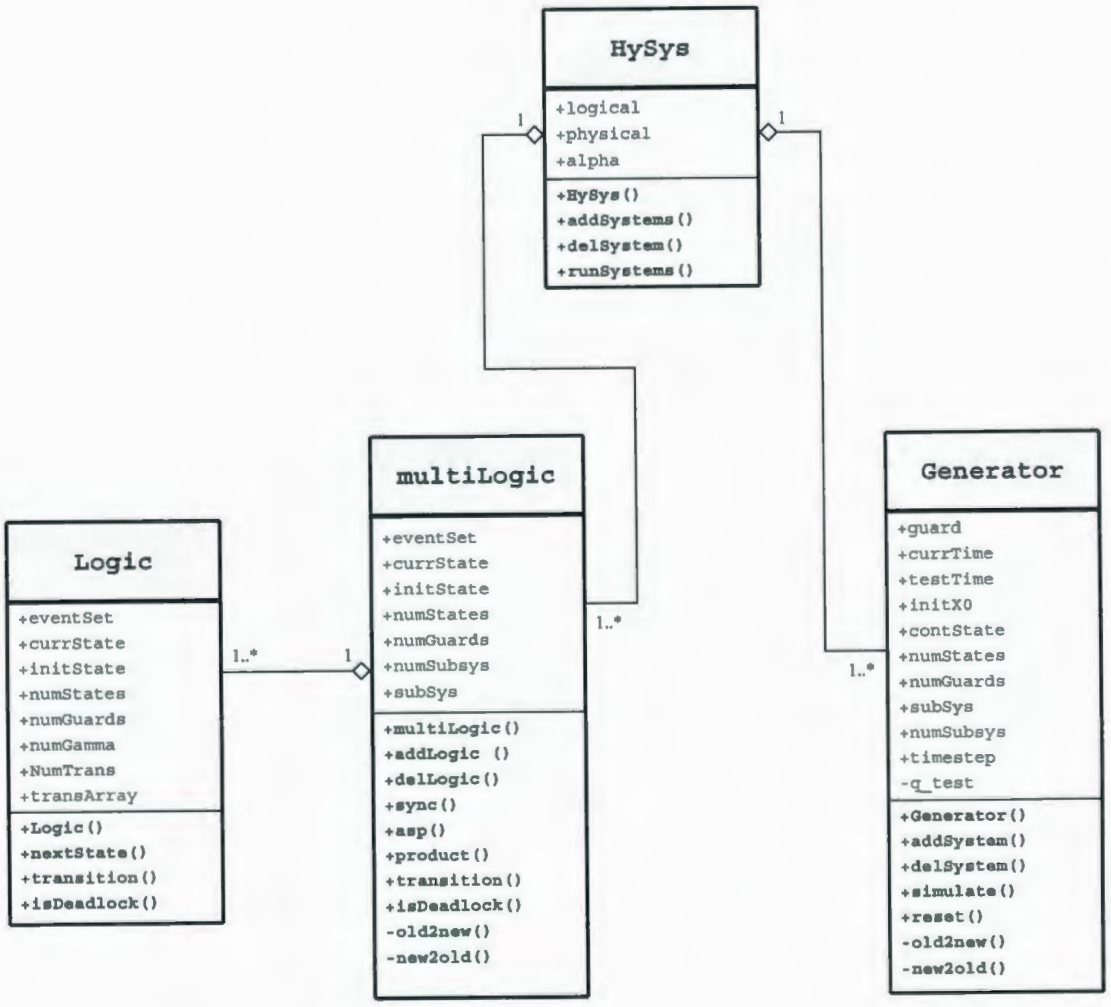

Figure 4-3: Computational class diagram and components (in UML standard).

logical and physical entities, in order to transmit the generated *guards* and the selected *transitions* back and forth, in addition to resetting each layer properly. The main components of `HySys` are shown in Figure 4-3. Note that adding any number of event-based specifications and defining a set of maps for state/event-based specifications are quite feasible in the design of a `multiLogic` object, and `HySys` only manages the entire system after all the specifications are applied in the logical level of abstraction. Thus, the software also enables the design team to see how well the final system will behave (and if any changes is required in the specifications, for example).

4.5 Implementation

For hybrid systems which have only interactions in the logical level of abstraction, the software usage is normally started by defining the automata which represent the logics of each system (can be performed in IDES, for example, or directly by scripting the `logical` property of a `HySys` object. And, the TSM and guard files should be set up for each corresponding physical system. The automata representing event-based specifications (if any) should also be defined as `Logics`. At the final step, from MATLAB command or a script file the components can be called to set up a `HySys` collection and see the results for any desired time duration.

In the next section, the implementation and sample user-codes for an extensive hybrid-systems example is provided. It would also be worth to see the simplicity and flexibility of the final design process on top of all the hidden object-oriented concepts.

4.6 Test and Results

This motivating example highlights the logical concurrency of multi-systems in real-time, using one DES collection and *ASP* as the composition rule. It also demonstrates a realistic logical interaction among physical systems. In addition, this problem tries

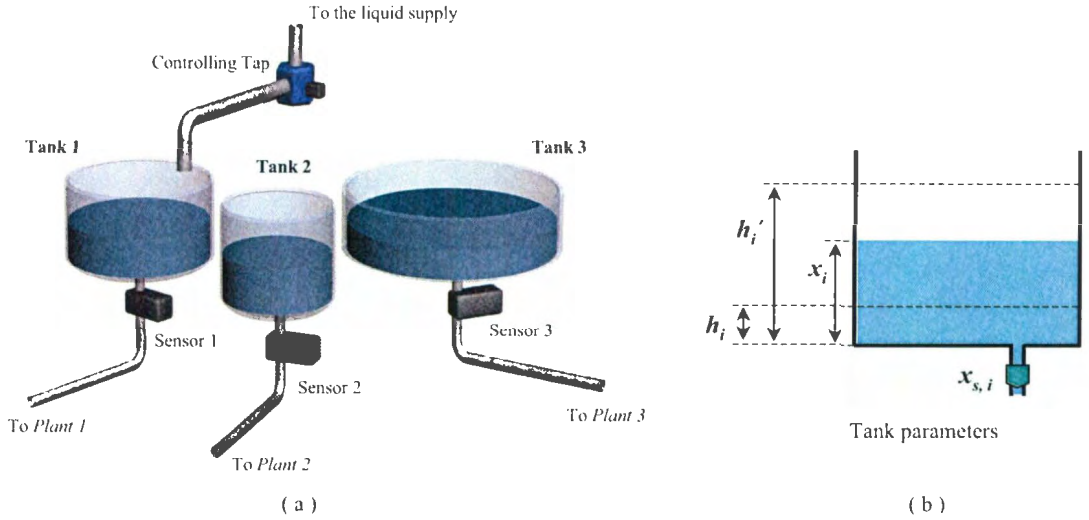

Figure 4-4: Multi-tank example: (a) schematic of the problem, and (b) state variable and other parameters.

to illustrate the simplicity and power of object-oriented algorithm design.

4.6.1 Problem Description

Consider n liquid tanks ($n \geq 2$), each of which is connected to a plant that consumes the liquid from its corresponding tank with a known non-linear flow with the draining factor μ_i for the ith tank, while the consumption (draining) time is unpredictable (Figure 4-4). Each tank is equipped with a sensor that shows whether or not the tank is in the consumption mode. There is only one tap filling all the tanks *one at a time* with a known constant flow of u. It is also assumed that the filling rate is always adequate to fill at least one of the tanks regardless of what the consumption rate is. For simplicity, it is assumed that the tap can be either *on* or *off* and it takes no time to switch to a new tank position (otherwise it has to be defined with new modes of operation for each intermediate case). The only state variable for each tank is its level of liquid (x_i). Different modes of operation are then as follows:

- **mode 1**: (filling), the tap is filling this tank, while the corresponding plant is off (based on the tank sensor): $\dot{x}_i = u$, $x_{0,i}$;
- **mode 2**: (filling, consumption), the tap is filling this tank and the corresponding plant is on (based on the tank sensor); $\dot{x}_i = -\mu_i\sqrt{x_i} + u$, $x_{0,i}$;
- **mode 3**: (consumption), the tap is off, while the corresponding plant is on (based on the tank sensor): $\dot{x}_i = -\mu_i\sqrt{x_i}$, $x_{0,i}$;
- **mode 4**: (rest), both the tap and the plant are *off*: $\dot{x}_i = 0$, $x_{0,i}$.

It is possible that at any time one tank is added to or deleted from the system. The goal is to keep all the tank levels between a minimum and a maximum level (h_i and h'_i respectively) all the times.

4.6.2 Solution and Results

To model this problem, since each tank is a multi-phased system, it has to be modeled as a hybrid system with four discrete states corresponding to each mode of operation. This system then follows a logical behaviour when controlled. However, to capture the behaviour of the sensor, we have to add another state variable $x_{s,i}$, which accepts two values of $\{0, 1\}$ regarding whether or not there is consumption. Note that $x_{s,i}$ does not have any dynamics, rather it receives its value from a sensor with a specific sampling time. Once the value of $x_{s,i}$ changes, the tank must switch to another mode of operation; as a result, $x_{s,i}$ will remain constant during one mode of operation ($\dot{x}_{s,i} = 0$). Based on this state-space, four guards are defined for each tank:

$$\Sigma_i = \{g_{\downarrow,i}, g_{\uparrow,i}, g_{c,i}, g_{n,i}\}. \tag{4.1}$$

Definitions of these guards are provided in Tables 4.1 and 4.2. Consequently, the logical behaviour of a tank based on both our control objectives and physical constraints (the sensor behaviour) can be introduced by an automaton shown in Figure 4-5. This

Table 4.1: Continuous-state Partitioning Functionals.

Label	Guard Name	Functional	Direction	Condition
$i1$	$g_{\downarrow}$	$g_1(\mathbf{x}, t) = x_i - h_i$	$\downarrow$	not enough liquid
$i2$	$g_{\uparrow}$	$g_2(\mathbf{x}, t) = x_i - h_i'$	$\uparrow$	tank is full

Table 4.2: Events Due To The Sensor.

Label	Guard Name	Explanation	Direction	Condition
$i3$	g_c	if $x_{s,i}$ switches to 1	–	consumption
$i4$	g_n	if $x_{s,i}$ switches to 0	–	non-consumption

specification is the synchronous product of the tank and its sensor. The explicit specifications of a tank and a sensor is shown in Figures 4-6 and 4-7 respectively, each of which has two states: tank, *filling* (`F`) and *rest* (`R`); and sensor, *consumption* (`C`) and *non-consumption* (`N`). The total specification represented by the automaton of Figure 4-5 follows from those explicit specifications with the consideration of the fact that in the resulting automaton, the combination of `R` and `N` is `R`; `R` and `C` is `C`; and `F` and `N` is `F`.

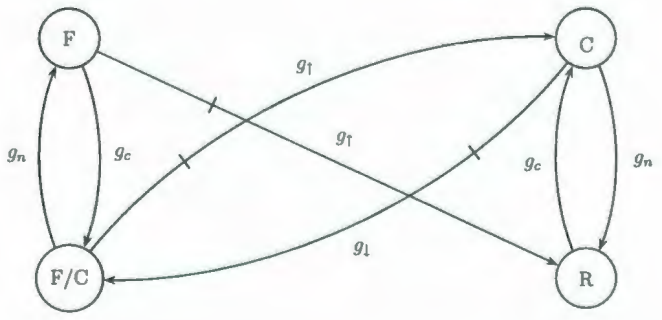

Figure 4-5: An automaton corresponding to the logical behaviour of a single tank with four states of Filling (F), Consumption (C), Filling and Consumption (F/C), and Rest (R).

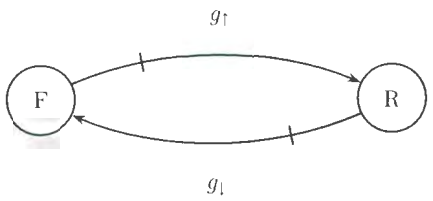

Figure 4-6: The explicit specification for a single tank with two states of *filling* (F) and *rest* (R).

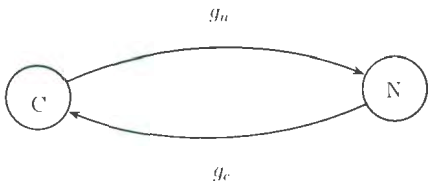

Figure 4-7: The explicit specification for a single sensor with two states of *consumption* (C) and *non-consumption* (N).

Considering a set of n tanks with only one tap, the specification is that if tank i transits to *Filling* mode (that is, if $g_\downarrow$ of a given tank is generated), and at the same time tank j has also been in its *Filling* mode, then tank j must exit its *Filling* mode. This state/event-based specification can be captured by introducing the following maps:

$$\forall i, j \in \{1, ..., n\}, \text{ and } i \neq j$$

$$M_j(F_j, g_{\downarrow,i}) = g_{\uparrow,j}, \tag{4.2}$$

$$M_j((F/C)_j, g_{\downarrow,i}) = g_{\uparrow,j}. \tag{4.3}$$

These maps encode the enforced synchronization among tanks using a single tap, meaning that each tank will see the $g_\downarrow$ of other tanks as its own $g_\uparrow$, which then automatically results in a desired transition. The logical behaviour is then given by

the ASP composition. These explanations can be written formally as:

$$G_i = (Q_i, \Sigma_i, \delta_i, \Gamma_i, q_{0i}, Q_{mi}), \text{for } i = 1, ..., n, \tag{4.4}$$

where, δ_i and Γ_i are as defined in Figure 4-5, $q_{0i} = R_i$, and $Q_{mi} = Q_i$. The collection and ruled-collection are as follows:

$$\mathcal{C} = \{(G_1, M_1), ..., (G_n, M_n)\}, \tag{4.5}$$

$$\tilde{\mathcal{C}} = (\mathcal{C}, \perp). \tag{4.6}$$

Here, a typical case for $n = 3$ is exemplified:

$\mathcal{G}$ (the resulting DES of $\tilde{\mathcal{C}}$) is shown in Figure 4-8. For generating this graph, the reachability algorithm of Chapter 3 (Algorithm 3.1) has been used to generate a DOT-file which then be plotted using *Graphviz*[3]. In this figure, each node on the graph (illustrated by an ellipse) is a product state. The initial state is distinguished by the cyan hexagon on the lower-right part. The parameters for each tank are provided in Table 4.3. The system commences with the first tank for 25 seconds. During this time, a specification, shown in Figure 4-5, will be applied to the systems at the logical layer. Then, another tank is added to the system and both are kept controlled for another 45 seconds, during which the two tanks will accommodate each other for having only one tap (i.e. once one of them requires to be filled, it receives the tap flow, and as a result if the other tank is in the filling mode at the same time, it accommodates the filling of the other tank by going to its appropriate non-filling mode). Based on the problem specifications, the filling priority always goes to the first tank. Afterwards, the third tank would be added and three of them will be controlled for another 30 seconds (again, during this time all three tanks will accommodate each other). Finally, the last tank will be deleted from the system and the remaining two

[3] *Graphviz* is an open source graph visualization software developed by AT&T. It has been distributed under its Common Public License (CPL), which can be reached on-line.

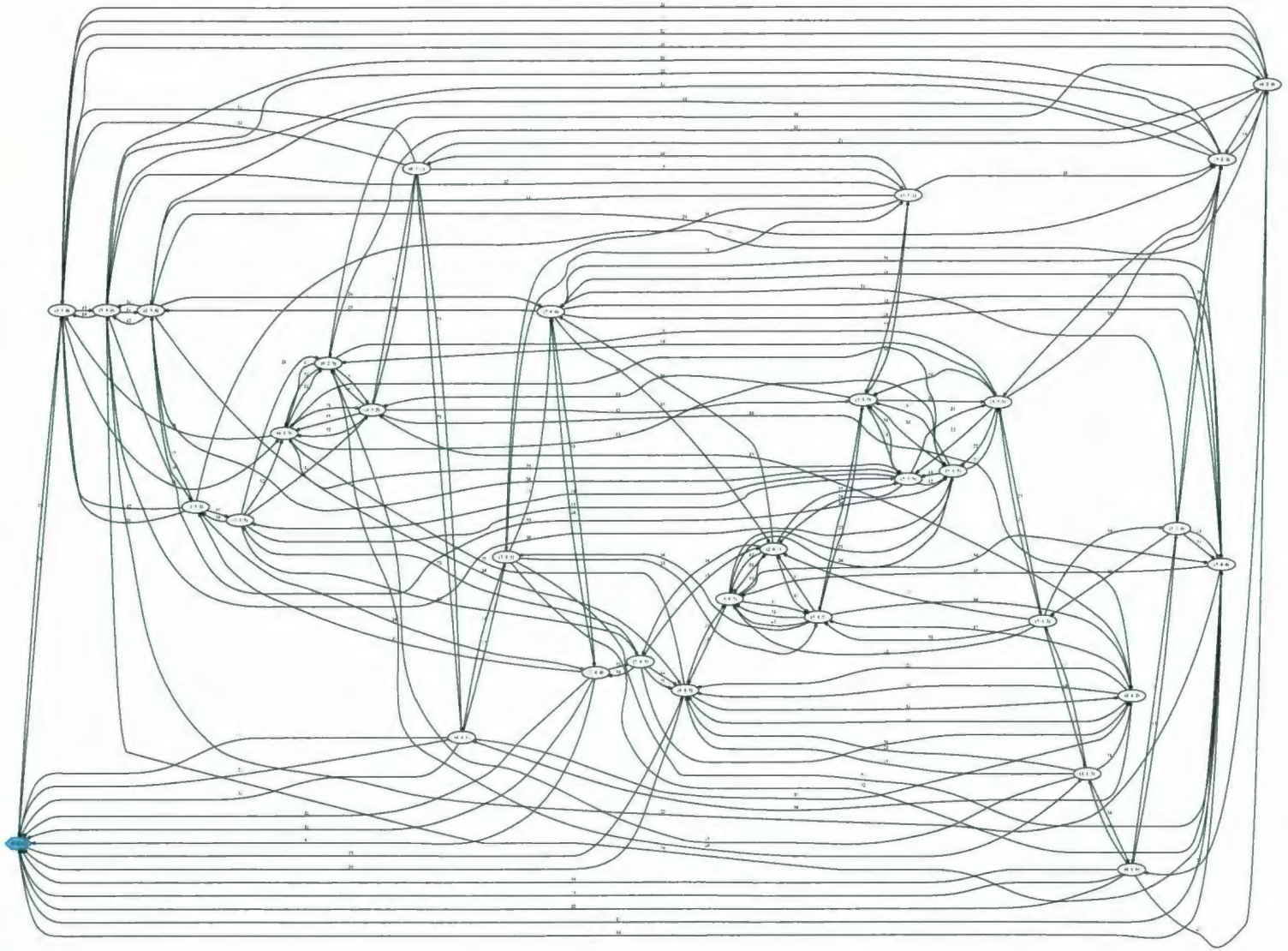

Figure 4-8: $\mathcal{G}$: the resulting DES of $\tilde{\mathcal{C}}$ for three tanks using the ASP composition rule.

Table 4.3: Parameters of the multi-tank problem.

Tank	h_i (m)	h_i' (m)	Initial Level (m)	μ_i $(\sqrt{m}/s)$
1	2	4	3	.03
2	3	3.5	2.5	.03
3	1.5	4.5	4	.06

tanks are kept controlled for another 30 seconds.

For this example, a MATLAB function simulates the sensor which reads the consumption of a hypothetical plant with the sampling rate of one sample per second. In each 10 seconds, the plant consumes liquid for a duration of 4-6 seconds followed by a rest duration of 4-6 seconds unpredictably. To make it more realistic, the plant may also show chattering-like on/off switching rarely in every 10 seconds. The MATLAB code for such a simulation is as follows:

```
function consumption = factorySensor(threshold)
% Plant Sensor Simulation
%
% "threshold" == an integer number in [0,9]
% the factory keeps consuming while the last right digit of the
% external computer clock (in seconds)is between "zero" and
% "threshold" (including threshold), then turns zero upto the end
% of 10 seconds duration and then repeats consumption.
%
% The swiching is on the basis of 1-second integer sampling, while
% over the semi-closed duration of [threshold+1, threshold+2) it may
% show rare and random chaterings.
temp = clock;
temp = fix(temp(6)) - 10*fix(temp(6)/10);
```

```
if temp <= threshold
    consumption = 1;
elseif temp == threshold+1
    % a random selection of {1,0}, most likely to be {0}:
    consumption = fix(rand+.15);
else
    consumption = 0;
end
end % end of function
```

This function will then be called by each TSM appropriately. The m-files representing the TSM and guard for tank1 are shown in Figures 4-9 and 4-10 respectively. A sample user-code for final implementation is also shown in Figure 4-11.

The results for this implementation is plotted in Figure 4-12. The yellow globes on the first trajectory indicate `ticks` of five seconds (which is an event generated by the first tank in every five seconds to demonstrate the time evolution), the red squares demonstrate when the level of liquid exceeds its prescribed levels (h_i, and h_i'), and the green diamonds show when the sensor status of each tank is changed. Note that, when x_i exceeds h_i' (its maximum level), the controller stops filling, by specification; however, since there is no control on the *draining*, it is possible that at the same time the corresponding plant is *off*, and therefore x_i stays at the exceeded level (in a margin which comes from computational time steps) for a while, then start decreasing. Other than this, at all the times, the liquid level in all the working tanks have been properly kept between their allowed limits.

```
function output = TSM1(tspan, q_current, x0)
switch q_current
    case 1
        f = '@f1';
    case 2
        f = '@f2';
    case 3
        f = '@f3';
    case 4
        f = '@f4';
    otherwise
        error('Discrete state value is illegal');
end
% tspan = t0:.0005:t0 + 1.005;
% options = odeset('RelTol',1e-8,'AbsTol',1e-8);
[t, x] = ode15s(eval(f), tspan, x0([1;3]));
output = [x(end,1);factorySensor(6);x(end,2)];
end
function xdot = f1(t,x) %% q1
global tap;
xdot = [tap
        1];
end
function xdot = f2(t,x) %% q2
mu1 = .03;
global tap;
xdot = [-mu1*sqrt(x(1)) + tap
        1];
end
function xdot = f3(t,x) %% q3
mu1 = .03;
xdot = [-mu1*sqrt(x(1))
        1];
end
function xdot = f4(t,x) %% q4
xdot = [0
        1];
end
```

Figure 4-9: A MATLAB code characterizing the TSM for tank 1.

```
function val = guard1(t_end,x,preFactorySensor)
%
% position hypersurfaces/events
% val is the event detection value; i.e. when val = [],
% no event is detected, otherwise the event id will
% return.

tol = 1e-6;    % computational tolerance
val = [];
tick = 5;      % ticks of 5 seconds
h1 = 2;
h1_prime = 4;

if (x(1) - h1) < tol              % liquid height is less
than h1
    val(end+1,1) = 11;
elseif (x(1) - h1_prime) > tol  % liquid height is
greater than h1, less than h1'
    val(end+1,1) = 12;           % liquid height is
greater than h1'
end

if abs(x(2) - preFactorySensor) >= tol % if sensor status
has changed
    if x(2)==1  % '0' or '1' (the new status)
        val(end+1,1) = 13;
    else % x(2)==0
        val(end+1,1) = 14;
    end
end

if abs(rem(t_end,tick)) < tol
    val(end+1,1) = 1;
elseif abs(rem(t_end,tick)-tick) < tol
    val(end+1,1) = 1;
end

end
```

Figure 4-10: A MATLAB code characterizing the Guard for tank 1.

```
%% controller:
global tap;
tap = .08;

%% System 1
p1 = 'tank_1_spec.xmd';
x01 = [3;0;0];
alpha = .1;  % latency number (sec)
hs = HySys(p1,x01,alpha,'asp')
plot(1,1,'w')
hold on; xlim([0 200]); ylim([0 5]); grid;
hs = hs.runSystem(25) % simulation

%% adding System 2
disp('A new System is now added...'); pause;
p2 = 'tank_2_spec.xmd';
x02 = [2.5;0;hs.physical.currTime];
m1 = {[1 12;2 12] 0 0 0};
m2 = {0 [1 22;2 22] 0 0 0};
hs = hs.addSystem(p2,m2,m1,x02)
hs = hs.runSystem(45) % simulation

%% adding System 3
disp('A new System is now added...'); pause;
p3 = 'tank_3_spec.xmd';
x03 = [4;0;hs.physical.currTime];
m3 = {0 [1,32;2,32] 0 0 0 [1 32;2 32] 0 0 0};
m1 = {[1 12 1 22;2 12 2 22] 0 0 0};
hs = hs.addSystem(p3,m3,m1,x03)
hs = hs.runSystem(30) % simulation

%% adding an event-based specification
disp('Adding an event-base specification...'); pause;
S = 'specification_tank_2.xmd';
x04 = [0;0;0];
m_S = {0 0 0 0 0 -1 0 0 0 0 0 0 0};
m1 = {-1 [0 0 1 22 0 0;0 0 2 22 0 0]};
hs = hs.addSystem(S,m_S,m1,x04)
hs = hs.runSystem(35) % simulation

%% deleting the Supervisor
disp('The Supervisor is now deleted...'); pause;
hs = hs.delSystem(4);  % Delete the third sub-system
hs = hs.runSystem(30); % simulation

%% deleting System 3
disp('The last System is now deleted...'); pause;
hs = hs.delSystem(3);  % Delete the third sub-system
hs = hs.runSystem(30); % simulation
```

Figure 4-11: Sample MATLAB user-code for example one.

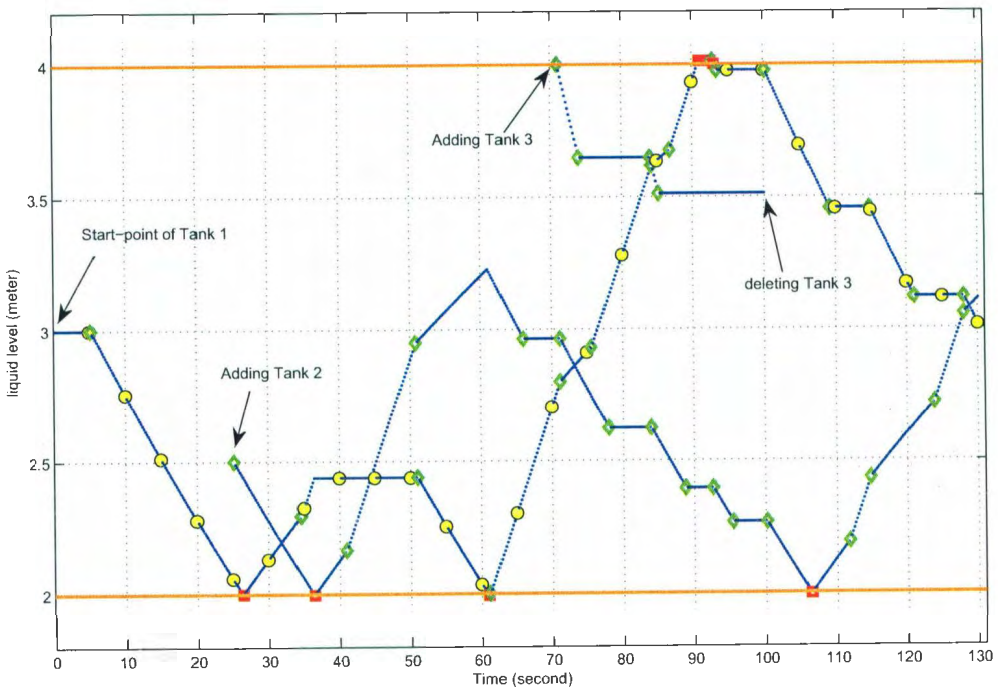

Figure 4-12: Simulation results for three concurrent tanks controlled by one tap. The yellow circles on the first trajectory indicate `ticks` of five seconds, the red squares demonstrate when the level of liquid exceeds its prescribed levels (h_i, and h_i'), and the green diamonds show when the sensor status of each tank is changed.

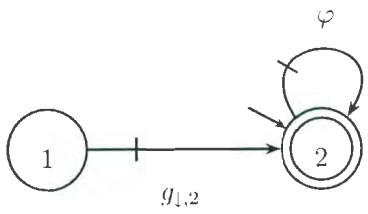

Figure 4-13: An automaton representing the specification to block $g_{\downarrow,2}$.

Let us now extend this example by assuming that at the time $t = 101$ sec., due to a technical problem, the second tank should be prevented from filling. Therefore, two steps should be done: (1) if the second tank is in its state `F` or `F/C` it should be forced to transition to `R` or `C` respectively, and (2) the event $g_{\downarrow,2}$ of the second tank should be blocked (disabled) by a supervisor (an event-based specification) in order for the second tank to stay in the $\{\texttt{R}, \texttt{C}\}$ subset of its discrete state-space. Figure 4-13 demonstrates the automaton which characterizes this specification. The event φ will be mapped to the event $g_{\uparrow,2}$, while the second tank is in `R` or `C`. Thus, after the ASP composition, once φ occurs, it will cause the second tank to have the desired transition. Additionally, note that $g_{\downarrow,2}$ belongs to the event set of this specification, but not to the active event set of its current state ($g_{\downarrow,2} \notin \Gamma_S(2)$); therefore, this specification always blocks $g_{\downarrow,2}$. We also need a physical system to generate φ in the desired time. This physical system can be formed by a simple dynamics and a guard to fire φ at the requested time. This physical system and the automaton of Figure 4-13 will then shape a hybrid system to be added to our existing `HySys` object.

Figure 4-14 shows the entire simulation from the beginning. This time, the new mentioned specification will be added in the time $t = 101$ sec (in real-time), before the third system is deleted. The entire system continues running for another 35 seconds. It can be observed that once the second tank reaches the height of 2 meters, it no longer switches to its *filling* mode. After this 35 seconds, the specification is deleted and it can be seen that the second tank immediately switches to the *filling* mode.

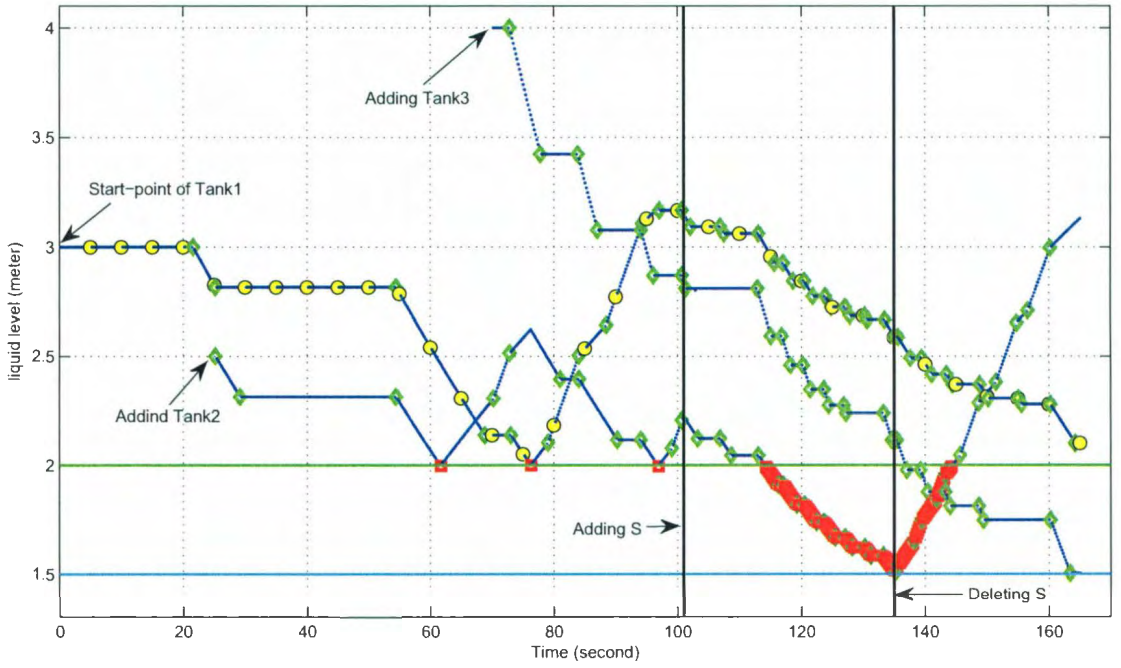

Figure 4-14: Simulation results for three concurrent tanks controlled by one tap. The yellow globes on the first trajectory indicate `ticks` of five seconds, the red squares demonstrate when the level of liquid exceeds its prescribed levels (h_i, and h_i'), and the green diamonds show when the sensor status of each tank is changed. The horizontal green and cyan thick lines represent the minimum allowable levels. Additionally, the two vertical black thick lines show the time of adding and deleting the supervisor S.

The system keeps running for another 30 seconds.

Chapter 5

Conclusions

5.1 Summary

After a high-level description in Chapter 2, which is mostly meant for a reader with limited background in DES and hybrid systems areas, Chapter 3 has provided the required theory in detail. In this chapter, the concepts of map and ASP have been introduced with the goal of implementing the state/event-based specifications properly. The reachability of the resulting DES of a ruled-collection (using any arbitrary composition rule) was also derived through an exhaustive, recursive algorithm. All the details were also clarified by simple and related examples. Moreover, DES hierarchy has been explained in this chapter. Finally, the DES concepts have been extended to a class of hybrid systems which are allowed to have synchronization only at the logical level of abstraction.

Because of the importance of implementation, the idea of using object-oriented concepts (such as information hiding) has been maintained through out the thesis. As explained in Chapter 4, in this thesis, a MATLAB-based software package has been developed, which implements the theoretical concepts. The chapter has discussed the object-oriented structure of the software. Efficient methods and algorithms to overcome memory overflow have also been discussed in Chapter 4. Finally, an ex-

tensive example, which demonstrates the basic concepts targeted in this research, was introduced. Additionally, this example highlighted how both mentioned types of specifications (namely, event-based and state/event-based) can be applied together in concurrent systems.

5.2 Contributions

As explained in Chapter 1, the work presented in this thesis contributes in the following two items:

1. The existing theory of DES has been developed to meet the computational requirements for an n-ary concurrent system design (Chapter 3):
    - The concept of map has been introduced to capture the specifications which are defined based on both events and states (called state/event-based specifications).
    - An n-ary composition rule, called accommodating synchronous product (ASP), has also been introduced to formally address the state/event-based specifications. This rule is an extension of the n-ary version of the standard synchronous product, and is reduced to synchronous product when the maps are set to be empty.
2. Software has also been designed as a testbed for the theory (Chapter 4):
    - Main properties:
        - Encompassing the concepts of the theory
        - Object-oriented structure
        - Importing data from standard DES software
        - The ability of real-time addition/deletion of new system(s) to/from a given collection

- the implementation of the software has been described in solving an extensive hybrid-system problem.

5.3 Future Work

Based on the desired theoretical, computational, and practical aspects, this work can be extended in different directions. As explained before, the theory provided in Chapter 3 is, for the most part, meant for computational design. A research extension is still reasonable to cover other theoretical aspects such as a collection of "distributed" DES and partial observation. In the computational part, at the current stage, an end-user software design is beyond the scope of this work. The software also is not meant to be a general-purpose software, although it includes a considerable number of different methods to tackle a given problem. As a result, another future step can be to develop the software in a more general format. From a theoretical and computational point of view, some possible future work can be itemized as:

1. extending the theory to cover other theoretical issues in the DES area, such as partial event-observation,
2. enhancing the software so as to encompass the DES hierarchy completely,
3. improving the software to meet end-user software requirements, such as having GUI and being stand-alone, and
4. developing the theory to a more general class of hybrid systems to include concurrency also in the physical level of abstraction.

From a practical point of view, on the other hand, it should be noted that DES is a logical level of abstraction, that enables logical policies to be modeled in an appropriate way. Additionally, the modeling of most of today's complex systems involve continuous dynamics with phase jumping, which can be best described by

hybrid systems. Therefore, a framework that formally formulates the concurrency of such systems would be of great importance and can be developed in most applications which involve logical interaction among more than one discrete-event or hybrid system.

References

Alur, R., Courcoubetis, C., Henzinger, T. A. and Ho, P.: 1993, *Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems*, Vol. 736 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, chapter Hybrid Systems, pp. 209–229.

Alur, R. and Dill, D. L.: 1994, A theory of timed automata, *Theoretical Computer Science* **126**, 183–235.

Brandin, B. and Wonham, W. M.: 1994, Supervisory control of timed discrete event systems, *IEEE Transactions on Automatic Control* **39**(2), 329–342.

Branicky, M.: 1994, *Control of Hybrid Systems*, PhD thesis, Department of Electrical Engineering and Computer Sciences, Massachusetts Institute of Technology.

Cassandras, C. G. and Lafortune, S.: 1999, *Introduction to Discrete Event Systems*, Kluwer Academic Publishers.

Cassandras, C. and Lygeros, J.: 2006, *Stochastic Hybrid Systems*, Vol. 24, CRC Press and IEEE Press.

Cellier, F. E.: 1991, *Continuous System Modeling*, Springer-Verlag.

Fonseca, I. and Leoni, G.: 2007, *Modern Methods in the Calculus of Variations: Lp Spaces*, Springer.

Freeman, R. A. and Kokotovic, P. V.: 2008, *Robust Nonlinear Control Design: State-Space and Lyapunov Techniques*, Birkhuser.

Gaudin, B. and Marchand, H.: 2005, Supervisory control and deadlock avoidance control problem for concurrent discrete event systems, pp. 2763 2768.

Henzinger, T.: 1996, The theory of hybrid automata, *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society Press, pp. 278–292.

Kalman, R. E.: 1960, On the general theory of control systems, *First IFAC Congress on Automatic Control, Moscow*, Butterworths, pp. 481 492.

Khalil, H. K.: 2002, *Nonlinear Systems*, third edn, Prentice Hall.

Kourkouli, M. and Hassapis, G.: 2005, Application of the timed automata abstraction to the performance evaluation of the architecture of a bank online transaction processing system, *the 2nd South-East European Workshop on Formal Methods (SEEFM05)*, pp. 142–153.

Lafortune, S.: 2007, *Advances in Control Theory and Applications*, Springer Berlin / Heidelberg, chapter On Decentralized and Distributed Control of Partially-Observed Discrete Event Systems, pp. 171 184.

Leduc, R., Lawford, M. and Dai, P.: 2006, Hierarchical interface-based supervisory control of a flexible manufacturing system, *Control Systems Technology, IEEE Transactions on* **14**(4), 654 668.

Li, Y.: 1997, On deadlock-free modular supervisory control of discrete-event systems, *Automatic Control, IEEE Transactions on* **42**(12), 1705 1708.

Lin, F. and Wonham, W. M.: 1995, Supervisory control of timed discrete-event systems under partial observation, *Automatic Control, IEEE Transactions on* **40**(3), 558 562.

Luenberger, D. G.: 1979, *Introduction to Dynamic Systems: Theory, Models, and Applications*, John Wiley Sons.

Lygeros, J., Tomlin, C. and Sastry, S.: 1999, Controllers for reachability specifications for hybrid systems, *Automatica* pp. 349–370.

Lygeros, J., Tomlin, C. and Sastry, S.: 2008, *Hybrid Systems: Modeling, Analysis and Control*.

Lynch, N., Segala, R. and Vaandrager, F.: 2001, Hybrid i/o automata revisited, *Proceedings Fourth International Workshop on Hybrid Systems: Computation and Control (HSCC'01*, Springer-Verlag, pp. 403 417.

Millan, J. P.: 2006, *Online Discrete Event Control of Hybrid Systems*, PhD thesis, Memorial University of Newfoundland.

Ogata, K.: 2001, *Modern Control Engineering*, Prentice Hall.

Oishi, M.: 2003, *User-Interfaces for Hybrid Systems: Analysis And Design Through Hybrid Reachibility*, PhD thesis, Stanford University.

O'Young, S.: 1991, On the synthesis of the supervisors for timed discrete event processes, *Technical Report 9107*, Department of Electrical and Computer Engineering, University of Toronto.

Pichler, F. and Moreno-Diaz, R. (eds): 1990, *Computer Aided Systems Theory-EUROCAST'89*, Lecture Notes in Computer Science, Springer-Verlag.

Princeton University: 2009, Wordnet®, `http://wordnet.princeton.edu/`.

Radatz, J.: 1997, *The IEEE Standard Dictionary of Electrical and Electronics Terms*, IEEE Standards Office, New York, NY, USA.

Ramadge, P. J. and Wonham, W. M.: 1987, Supervisory control of a class of discrete event processes, *SIAM Journal on Control and Optimization* **25**, 206 230.

Rodger, S. H.: 2009, JFLAP official wesite, `http://www.jflap.org/`.

Rodger, S. H. and Finley, T. W.: 2006, *JFLAP: An Interactive Formal Languages and Automata Package*, Jones Bartlett Publishers.

Rudie, K.: 2008, IDES official wesite, `https://qshare.queensu.ca/Users01/rudie/www/software.html`.

Rudie, K. and Wonham, W. M.: 1992, Think globally, act locally: decentralized supervisory control, *Automatic Control, IEEE Transactions on* **37**, 1692 1708.

Saadatpoor, A.: 2004, *State based control of timed discrete event systems using binary decision diagrams*, Master's thesis, Department of Electrical and Computer Engineering, University of Toronto.

Su, R. and Wonham, W. M.: 2004, Supervisor reduction for discrete-event systems, *Discrete Event Dynamic Systems* **14**(1), 31 53.

Thistle, J. G. and Wonham, W. M.: 1994, Supervision of infinite behavior of discrete-event systems, *SIAM J. Control Optim.* **32**(4), 1098 1113.

Tomlin, C., Lygeros, J. and Sastry, S.: 2000, A game theoretic approach to controller design for hybrid systems, *Proceedings of the IEEE* **88**(7).

Wikipedia.org: 2009, Wikipedia.org, `http://en.wikipedia.org/`.

Wonham, W.: 2009, *Supervisory Control of Discrete-Event Systems*, University of Toronto, Toronto, Canada. Notes for the course ECE1636F/1637S, Control of Discrete-Event Systems.

Appendices

Appendix A

Proof of Consistency of Binary and N-ary Synchronous Product

A.1 Part I:

In the definition 3.3.6, for $n = 2$, Q, Σ, q_0, and Q_m will be directly reduced to those of the binary operator (definition 3.3.5). For δ and Γ, by definition,

$$\delta((q_1, ..., q_2)), \sigma) := (\delta'_1(q_1, \sigma), ..., \delta'_n(q_n, \sigma)), \text{ in which} \tag{A.1}$$

$$\delta'_i(q_i, \sigma) := \tag{A.2}$$

$$\begin{cases} \delta_i(q_i, \sigma) & \text{if } \exists \Omega \in 2^{\{1,...,n\}} \mid i \in \Omega \wedge \sigma \in [\bigcap_{k\in\Omega} \Gamma_k(q_k)] \backslash [\bigcup_{k\notin\Omega} \Sigma_k], \\ q_i & \text{if } \exists \Omega \in 2^{\{1,...,n\}} \mid i \notin \Omega \wedge \sigma \in [\bigcap_{k\in\Omega} \Gamma_k(q_k)] \backslash [\bigcup_{k\notin\Omega} \Sigma_k], \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

and δ is undefined if any one of the δ' is undefined.

If $n = 2$, then $\Omega \in 2^{\{1,2\}} = \{\{1\}, \{2\}, \{1, 2\}\}$. Each possibility of Ω makes a unique condition for σ:

Case 1: $\Omega = \{1\}$

$$i \in \Omega \quad (\Rightarrow i = 1): \quad \delta_1'(q_1, \sigma) = \delta_1(q_1, \sigma) \qquad \text{if} \quad \sigma \in \Gamma_1(q_1) \backslash \Sigma_2, \tag{A.3}$$

$$i \notin \Omega \quad (\Rightarrow i = 2): \quad \delta_2'(q_2, \sigma) = q_2 \qquad \text{if} \quad \sigma \in \Gamma_1(q_1) \backslash \Sigma_2. \tag{A.4}$$

Case 2: $\Omega = \{2\}$

$$i \in \Omega \quad (\Rightarrow i = 2): \quad \delta_2'(q_2, \sigma) = \delta_2(q_2, \sigma) \qquad \text{if} \quad \sigma \in \Gamma_2(q_2) \backslash \Sigma_1, \tag{A.5}$$

$$i \notin \Omega \quad (\Rightarrow i = 1): \quad \delta_1'(q_1, \sigma) = q_1 \qquad \text{if} \quad \sigma \in \Gamma_2(q_2) \backslash \Sigma_1. \tag{A.6}$$

Case 3: $\Omega = \{1, 2\}$

$i \in \Omega$:

$$(i = 1): \quad \delta_1'(q_1, \sigma) = \delta_1(q_1, \sigma) \qquad \text{if} \quad \sigma \in [\Gamma_1(q_1) \cap \Gamma_2(q_2)], \tag{A.7}$$

$$(i = 2): \quad \delta_2'(q_2, \sigma) = \delta_2(q_2, \sigma) \qquad \text{if} \quad \sigma \in [\Gamma_1(q_1) \cap \Gamma_2(q_2)], \tag{A.8}$$

$i \notin \Omega$ not possible.

Otherwise (none of the above cases which defined for all the possible choices of Ω), both δ_1' and δ_2' are undefined, by the definition.

Combining equations A.3 and A.4; equations A.5 and A.6; and equations A.7 and A.8, results in:

$$(\delta_1'(q_1, \sigma), \delta_2'(q_1, \sigma)) = (\delta_1(q_1, \sigma), q_2) \qquad \text{if} \quad \sigma \in \Gamma_1(q_1) \backslash \Sigma_2, \tag{A.9}$$

$$(\delta_1'(q_1, \sigma), \delta_2'(q_1, \sigma)) = (q_1, \delta_2(q_2, \sigma)) \qquad \text{if} \quad \sigma \in \Gamma_2(q_2) \backslash \Sigma_1, \tag{A.10}$$

$$(\delta_1'(q_1, \sigma), \delta_2'(q_1, \sigma)) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) \qquad \text{if} \quad \sigma \in \Gamma_1(q_1) \cap \Gamma_2(q_2). \tag{A.11}$$

For the Γ, by definition:

$$\Gamma(q_1, ..., q_n) = \bigcup_{\Omega} \left[\bigcap_{k \in \Omega} \Gamma_k(q_k) \backslash \bigcup_{k \notin \Omega} \Sigma_k \right]. \tag{A.12}$$

Again, if $n = 2$, then $\Omega \in 2^{\{1,2\}} = \{\{1\}, \{2\}, \{1,2\}\}$. Thus,

$$\begin{aligned} \Gamma(q_1, ..., q_n) = & \left[\bigcap_{k \in \{1\}} \Gamma_k(q_k) \backslash \bigcup_{k \notin \{1\}} \Sigma_k \right] \\ & \cup \left[\bigcap_{k \in \{2\}} \Gamma_k(q_k) \backslash \bigcup_{k \notin \{2\}} \Sigma_k \right] \\ & \cup \left[\bigcap_{k \in \{1,2\}} \Gamma_k(q_k) \backslash \bigcup_{k \notin \{1,2\}} \Sigma_k \right] \end{aligned} \tag{A.13}$$

$$= [\Gamma_1(q_1) \backslash \Sigma_2] \cup [\Gamma_2(q_2) \backslash \Sigma_1] \cup [\Gamma_1(q_1) \cap \Gamma_2(q_2)]. \tag{A.14}$$

which completes the proof.

■

A.2 Part II:

For the sake of simplicity and clarity let us introduce the following notations:

- q_i:= the current state of G_i, $q_i \in \Sigma_i$,
- $G^{(j)} := G_1 || G_2 || ... || G_j$,
- $\Gamma_i := \Gamma_i(q_i)$:=active event set of the current state of G_i,
- $\Gamma^{(j)} := \Gamma_{G_1||...||G_j}(q_1, ..., q_j)$,
- $\Sigma^{(j)} := \Sigma_{G_1||...||G_j} = \bigcup_{k=1}^{j} \Sigma_k$,
- $\delta^{(j)} := \delta_{G_1||...||G_j} = (\delta'_1, \delta'_2, ..., \delta'_j)$.

We prove the second part of the theorem by induction: given the correct case of $n = 2$, $G^{(n)}$ and $\Gamma^{(n)}$, we show $G^{(n+1)}$ is equivalent to $G_{new} := G^{(n)}||G_{n+1}$, and $\Gamma^{(n+1)}$ is equivalent to $\Gamma_{G^{(n)}||G_{n+1}}$. The base of induction ($n = 2$) was proved in the first part of the theorem. Also,

$$\Sigma_{new} = \Sigma_{G^{(n)}} \cup \Sigma_{n+1} \tag{A.15}$$

$$= \Sigma^{(n)} \cup \Sigma_{n+1} \tag{A.16}$$

$$= \bigcup_{i=1}^{n} \Sigma_i \cup \Sigma_{n+1} \tag{A.17}$$

$$= \bigcup_{i=1}^{n+1} \Sigma_i = \Sigma^{(n+1)}. \tag{A.18}$$

Suppose $\exists \Omega \in 2^{\{1,\ldots,n\}} \mid \sigma \in [\bigcap_{k\subset\Omega} \Gamma_k(q_k)] \backslash [\bigcup_{k\notin\Omega} \Sigma_k]$. Using the binary product definition and the introduced notation we have:

$$\delta_{G^{(n)}||G_{n+1}} := \begin{cases} (\delta^{(n)}, \delta_{n+1}) & \text{if } \sigma \in \Gamma^{(n)} \cap \Gamma_{n+1} \\ (\delta^{(n)}, q_{n+1}) & \text{if } \sigma \in \Gamma^{(n)} \backslash \Sigma_{n+1} \\ (q^{(n)}, \delta_{n+1}) & \text{if } \sigma \in \Gamma_{n+1} \backslash \Sigma^{(n)} \\ undefined & \text{otherwise.} \end{cases} \tag{A.19}$$

On the other hand, using the n-ary definition we have:

$$\delta_{G^{(n+1)}} := \delta^{(n+1)} := (\delta'_1, \ldots, \delta'_n, \delta'_{n+1}) = (\delta^{(n)}, \delta'_{n+1}), \text{ in which} \tag{A.20}$$

$$\delta'_{n+1} := \tag{A.21}$$

$$\begin{cases} \delta_{n+1}(q_{n+1}, \sigma) & \text{if } \exists \Omega' \in 2^{\{1,\ldots,n+1\}} \mid n+1 \in \Omega' \wedge \sigma \in \bigcap_{k\in\Omega'} \Gamma_k \backslash \bigcup_{k\notin\Omega'} \Sigma_k, \\ q_{n+1} & \text{if } \exists \Omega' \in 2^{\{1,\ldots,n+1\}} \mid n+1 \notin \Omega' \wedge \sigma \in \bigcap_{k\in\Omega'} \Gamma_k \backslash \bigcup_{k\notin\Omega'} \Sigma_k, \\ undefined & \text{otherwise.} \end{cases}$$

We used the symbol Ω' to distinguish this case from the case of n, where we used Ω. Thus, $\delta^{(n+1)} = (\delta^{(n)}, \delta'_{n+1})$ should be evaluated for different possibilities of Ω': having

$\Omega \in 2^{\{1,\ldots,n\}}$, we should show that first, for each case, there exists a specific Ω' for which equation A.20 results in equation A.19, and second, the union of all the Ω' is equal to $2^{\{1,\ldots,n+1\}}$. To this end, let us partition $2^{\{1,\ldots,n+1\}}$ into four sets:

$$2^{\{1,\ldots,n+1\}} = \{\Omega \cup \{n+1\}, \quad \forall \Omega \in 2^{\{1,\ldots,n\}}, \Omega \neq \emptyset\} \cup 2^{\{1,\ldots,n\}} \cup \{n+1\} \cup \{\emptyset\}. \tag{A.22}$$

Now, let us investigate each case separately:

Case 1: $\Omega' = \Omega \cup \{n+1\}$

$$\Rightarrow \left[\bigcap_{k\subset\Omega'} \Gamma_k\right] \setminus \left[\bigcup_{k\notin\Omega'} \Sigma_k\right] = \left[\left(\bigcap_{k\subset\Omega} \Gamma_k\right) \cap \Gamma_{n+1}\right] \setminus \left[\bigcup_{k\notin\Omega, k\neq n+1} \Sigma_k\right] \tag{A.23}$$

$$= \left[\left(\bigcap_{k\subset\Omega} \Gamma_k\right) \setminus \left(\bigcup_{k\notin\Omega, k\neq n+1} \Sigma_k\right)\right] \cap \Gamma_{n+1} \tag{A.24}$$

$$= \Gamma^{(n)} \cap \Gamma_{n+1}, \tag{A.25}$$

which is the condition in the first line of equation A.19. In the second equality, we have used the fact that $(A \cap B) \setminus C = (A \setminus C) \cap B$ [1]. For this case, since $\Omega' = \{n+1\} \cup \Omega$, from equation A.21 we have $\delta'_{n+1} = \delta_{n+1}$; therefore from equation A.20 we have:

$$\delta^{(n+1)} = (\delta^{(n)}, \delta'_{n+1}) = (\delta^{(n)}, \delta_{n+1}), \tag{A.26}$$

which is identical to the first line of the transition function of $G^{(n)} || G_{n+1}$ (equation A.19).

[1] It can be simply proved as: $(A \cap B) \setminus C = (A \cap B) \cap C^c = (A \cap C^c) \cap B = (A \setminus C) \cap B$.

Case 2: $\Omega' = \Omega$

$$\Rightarrow \left[\bigcap_{k\subset\Omega'} \Gamma_k\right] \setminus \left[\bigcup_{k\not\subset\Omega'} \Sigma_k\right] = \left[\bigcap_{k\subset\Omega} \Gamma_k\right] \setminus \left[\bigcup_{k\not\subset\Omega} \Sigma_k\right] \tag{A.27}$$

$$= \left[\bigcap_{k\subset\Omega} \Gamma_k\right] \setminus \left[\bigcup_{k\not\subset\Omega, k\neq n+1} \Sigma_k \cup \Sigma_{n+1}\right] \tag{A.28}$$

$$= \left(\left[\bigcap_{k\subset\Omega} \Gamma_k\right] \setminus \left[\bigcup_{k\not\subset\Omega, k\neq n+1} \Sigma_k\right]\right) \setminus \Sigma_{n+1} \tag{A.29}$$

$$= \Gamma^{(n)} \setminus \Sigma_{n+1}, \tag{A.30}$$

which is the condition in the second line of equation A.19. In the third equality, we have used the fact that $A\setminus(B \cup C) = (A\setminus B)\setminus C$ [2]. For this case, from equation A.21 we have: $\delta'_{n+1} = q_{n+1}$. Therefore,

$$\delta^{n+1} = (\delta^{(n)}, \delta'_{n+1}) = (\delta^{(n)}, q_{n+1}). \tag{A.31}$$

which is identical to the second line of the transition function of $G^{(n)}||G_{n+1}$ (equation A.19).

Case 3: $\Omega' = \{n+1\}$

$$\Rightarrow \left[\bigcap_{k\in\Omega'} \Gamma_k\right] \setminus \left[\bigcup_{k\not\subset\Omega'} \Sigma_k\right] = \Gamma_{n+1} \setminus \bigcup_{k\neq n+1} \Sigma_k \tag{A.32}$$

$$= \Gamma_{n+1} \setminus \bigcup_{k\in\Omega} \Sigma_k = \Gamma_{n+1} \setminus \Sigma^{(n)}, \tag{A.33}$$

[2] It can be proved using De Morgan's law:

$$A\setminus(B\cup C) = A \cap (B\cup C)^c = A \cap (B^c \cap C^c) = (A\cap B^c)\cap C^c = (A\setminus B)\cap C^c = (A\setminus B)\setminus C.$$

which is the condition in the third line of equation A.19. For this case, since $\Omega' = n+1 \Rightarrow \sigma \notin \bigcup_{i=1}^{n} \Sigma_i$, in other words, $\sigma \notin \Sigma_i$, for $i = 1, ..., n$, by definition we have:

$$\delta'_{n+1} = \delta_{n+1}, \tag{A.34}$$

$$\delta^{(n)} = (\delta'_1, ..., \delta'_n) = (q_1, ..., q_n) = q^{(n)}, \tag{A.35}$$

therefore,

$$\delta^{(n+1)} = (\delta^{(n)}, \delta'_{n+1}) = (q^{(n)}, \delta_{n+1}), \tag{A.36}$$

which is identical to the third line of the transition function of $G^{(n)}||G_{n+1}$ (equation A.19).

Case 4: $\Omega' = \emptyset$

$$\Rightarrow \left[\bigcap_{k \in \Omega'} \Gamma_k\right] \setminus \left[\bigcup_{k \notin \Omega'} \Sigma_k\right] = \emptyset \tag{A.37}$$

$$\Rightarrow \text{ no transition is allowed,} \tag{A.38}$$

which is corresponding to the forth line of the transition function of $G^{(n)}||G_{n+1}$ (equation A.19).

The validity of $\Gamma^{n+1} = \Gamma_{G^{(n)}||G_{n+1}}$ also follows the same arguments of the four described cases above.

■

Appendix B

Proof of Consistency of Binary and N-ary Accommodating Synchronous Product

Similar to Appendix A, for $n = 2$ in the definition 3.3.9, δ and Γ could be derived. By definition,

$$\delta((q_1, ..., q_2)), \sigma) := (\delta'_1(q_1, \sigma), ..., \delta'_n(q_n, \sigma)), \text{ in which} \tag{B.1}$$

$$\delta'_i(q_i, \sigma) := \tag{B.2}$$

$$\begin{cases} \delta_i(q_i, \sigma) & \text{if } \exists \Omega \in 2^{\{1,...,n\}} \mid i \in \Omega \wedge \sigma \in [\bigcap_{k \in \Omega} \Gamma_k(q_k)] \backslash [\bigcup_{k \notin \Omega} \Sigma_k], \\ q_i & \text{if } \exists \Omega \in 2^{\{1,...,n\}} \mid i \notin \Omega \wedge \sigma \in [\bigcap_{k \in \Omega} \Gamma_k(q_k)] \backslash [\bigcup_{k \notin \Omega} \Sigma_k], \\ & \text{and } M_i(q_i, \sigma) \text{ is not defined,} \\ \delta_i(q_i, M_i(q_i, \sigma)) & \text{if } \exists \Omega \in 2^{\{1,...,n\}} \mid i \notin \Omega \wedge \sigma \in [\bigcap_{k \in \Omega} \Gamma_k(q_k)] \backslash [\bigcup_{k \notin \Omega} \Sigma_k], \\ & \text{and } M_i(q_i, \sigma) \text{ is defined,} \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

and δ is undefined if any one of the δ' is undefined.

If $n = 2$, then $\Omega \in 2^{\{1,2\}} = \{\{1\}, \{2\}, \{1, 2\}\}$. Each possibility of Ω makes a unique condition for σ:

Case 1: $\Omega = \{1\}$

$$i \in \Omega \quad (\Rightarrow i = 1): \quad \delta_1'(q_1, \sigma) = \delta_1(q_1, \sigma) \quad \text{if} \quad \sigma \in \Gamma_1(q_1) \backslash \Sigma_2, \tag{B.3}$$

$$i \notin \Omega \quad (\Rightarrow i = 2) \text{ and } M_2(q_2, \sigma) \text{ is undefined :} \tag{B.4}$$

$$\delta_2'(q_2, \sigma) = q_2 \quad \text{if} \quad \sigma \in \Gamma_1(q_1) \backslash \Sigma_2,$$

$$i \notin \Omega \quad (\Rightarrow i = 2) \text{ and } M_2(q_2, \sigma) \text{ is defined :} \tag{B.5}$$

$$\delta_2'(q_2, \sigma) = \delta_2(q_2, M_2(q_2, \sigma)) \quad \text{if} \quad \sigma \in \Gamma_1(q_1) \backslash \Sigma_2.$$

Case 2: $\Omega = \{2\}$

$$i \in \Omega \quad (\Rightarrow i = 2): \quad \delta_2'(q_2, \sigma) = \delta_2(q_2, \sigma) \quad \text{if} \quad \sigma \in \Gamma_2(q_2) \backslash \Sigma_1, \tag{B.6}$$

$$i \notin \Omega \quad (\Rightarrow i = 1) \text{ and } M_1(q_1, \sigma) \text{ is undefined :} \tag{B.7}$$

$$\delta_1'(q_1, \sigma) = q_1 \quad \text{if} \quad \sigma \in \Gamma_2(q_2) \backslash \Sigma_1,$$

$$i \notin \Omega \quad (\Rightarrow i = 1) \text{ and } M_1(q_1, \sigma) \text{ is defined :} \tag{B.8}$$

$$\delta_1'(q_1, \sigma) = \delta_1(q_1, M_1(q_1, \sigma)) \quad \text{if} \quad \sigma \in \Gamma_2(q_2) \backslash \Sigma_1.$$

Case 3: $\Omega = \{1, 2\}$

$i \in \Omega$:

$$(i = 1): \quad \delta_1'(q_1, \sigma) = \delta_1(q_1, \sigma) \qquad \text{if} \quad \sigma \in [\Gamma_1(q_1) \cap \Gamma_2(q_2)], \tag{B.9}$$

$$(i = 2): \quad \delta_2'(q_2, \sigma) = \delta_2(q_2, \sigma) \qquad \text{if} \quad \sigma \in [\Gamma_1(q_1) \cap \Gamma_2(q_2)], \tag{B.10}$$

$i \notin \Omega$ not possible.

Otherwise (none of the above cases which defined for all the possible choices of Ω), both δ_1' and δ_2' are undefined, by the definition.

Combining equations B.3 and B.4; equations B.3 and B.5; equations B.6 and B.7;

equations B.6 and B.8; and equations B.9 and B.10 respectively, results in:

$$\delta((q_1,q_2),\sigma) := \tag{B.11}$$

$$\begin{cases} (\delta_1(q_1,\sigma),q_2) & \text{if } \sigma \in \Gamma_1(q_1)\backslash\Sigma_2 \text{ and } M_2(q_2,\sigma) \text{ is not defined,} \\ (\delta_1(q_1,\sigma),\delta_2(q_2,M_2(q_2,\sigma))) & \text{if } \sigma \in \Gamma_1(q_1)\backslash\Sigma_2 \text{ and } M_2(q_2,\sigma) \text{ is defined,} \\ (q_1,\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_2(q_2)\backslash\Sigma_1 \text{ and } M_1(q_1,\sigma) \text{ is not defined,} \\ (\delta_1(q_1,M_1(q_1,\sigma)),\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_2(q_2)\backslash\Sigma_1 \text{ and } M_1(q_1,\sigma) \text{ is defined,} \\ (\delta_1(q_1,\sigma),\delta_2(q_2,\sigma)) & \text{if } \sigma \in \Gamma_1(q_1)\cap\Gamma_2(q_2). \end{cases}$$

The proof of Γ is identical to that of synchronous product. By definition:

$$\Gamma(q_1,...,q_n) = \bigcup_{\Omega}\left[\bigcap_{k\in\Omega}\Gamma_k(q_k)\backslash\bigcup_{k\notin\Omega}\Sigma_k\right]. \tag{B.12}$$

Again, if $n=2$, then $\Omega \in 2^{\{1,2\}} = \{\{1\},\{2\},\{1,2\}\}$. Thus,

$$\begin{aligned}\Gamma(q_1,...,q_n) = &\left[\bigcap_{k\in\{1\}}\Gamma_k(q_k)\backslash\bigcup_{k\notin\{1\}}\Sigma_k\right] \\ &\cup\left[\bigcap_{k\in\{2\}}\Gamma_k(q_k)\backslash\bigcup_{k\notin\{2\}}\Sigma_k\right] \\ &\cup\left[\bigcap_{k\in\{1,2\}}\Gamma_k(q_k)\backslash\bigcup_{k\notin\{1,2\}}\Sigma_k\right]\end{aligned} \tag{B.13}$$

$$= [\Gamma_1(q_1)\backslash\Sigma_2]\cup[\Gamma_2(q_2)\backslash\Sigma_1]\cup[\Gamma_1(q_1)\cap\Gamma_2(q_2)]. \tag{B.14}$$

which completes the proof.

■

Appendix C

Numbering of Product States in An N-ary System

In a product state, the counting of the entire system's state is first performed by the older DES, then by the added DES. For example, if the first `Logic` has four states, then the `multiLogic` states are counted from 1 to 4 corresponding to the `Logic`'s states. Once a new `Logic` added (say with four states), the `multiLogic` will have 16 states, which by the convention are corresponding to '$1 \equiv (1,1)$', '$2 \equiv (2,1)$', ..., '$5 \equiv (1,2)$', '$6 \equiv (2,2)$', ..., '$16 \equiv (4,4)$'. The following algorithms (Algorithms C.1 and C.2) implement this convention (for example, if the two mentioned `Logics` have been added in a `multiLogic` object called `testMultiLogic`, then `testMultiLogic.old2new`([3, 1]) gives 3, and `testMultiLogic.new2old`(15) gives the vector [3, 4]):

Algorithm C.1: *old2new* method of the *multiLogic* class

input : $\mathcal{G} \leftarrow$ (a `multiLogic` object), $oldState \in Q_1 \times ... \times Q_n$

output: $newState \in Q$

```
1 Function oldnew(G, oldState);
2 newState ← oldState(1);
3 newNumState ← 1;
4 foreach i ∈ {1, ..., G.numSubsys − 1} do
5 |   newNumState ← newNumState * G.subsys.logic{i}.numStates;
6 |   newState ← (oldState(i + 1) − 1) * newNumState + newState;
7 end
8 return newState;
```

Algorithm C.2: *new2old* method of the *multiLogic* class

input : $\mathcal{G} \leftarrow$ (a `multiLogic` object), $newState \in Q$

output: $oldState \in Q_1 \times ... \times Q_n$

```
1  Function newold(G, newState);
2  num ← 1;
3  foreach i ∈ {1, ..., G.numSubsys − 1} do
4  |  num ← num * G.subsys.logic{i}.numStates;
5  end
6  oldState ← zeros(1, G.numSubsys);
7  foreach i ∈ {1, ..., G.numSubsys} do
8  |  qn ← floor(newState/num);
9  |  md ← mod(newState, num);
10 |  if md ≠ 0 then
11 |  |  oldState(end − i + 1) ← qn + 1;
12 |  |  newState ← md;
13 |  else
14 |  |  oldState(end − i + 1) ← qn;
15 |  |  newState ← num;
16 |  end
17 |  if i ≠ G.numSubsys then
18 |  |  num ← num/G.subsys.logic{end − i}.numStates;
19 |  end
20 end
21 return oldState;
```