A FRAMEWORK FOR VERIFICATION OF TRANSACTION LEVEL MODELS IN SYSTEMC

By

Reza Hajisheykhi

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor of Philosophy

2016

ABSTRACT

A FRAMEWORK FOR VERIFICATION OF TRANSACTION LEVEL MODELS IN SYSTEMC

By

Reza Hajisheykhi

Due to their increasing complexity, today's SoC (System on Chip) systems are subject to a variety of faults (e.g., single-event upset, component crash, etc.), thereby their verification a highly important task of such systems. However, verification is a complex task in part due to the large scale of integration of SoC systems and different levels of abstraction provided by modern system design languages such as *SystemC*.

To facilitate the verification of SoC systems, this dissertation proposes an approach for verifying inter-component communication protocols in SystemC Transaction Level Modeling (TLM) programs. SystemC is a widely accepted language and an IEEE standard. It includes a C++ library of abstractions and a run-time kernel that simulates the specified system, thereby enabling the early development of embedded software for the system that is being designed. To enable and facilitate the communication of different components in SystemC, the Open SystemC Initiative (OSCI) has proposed an interoperability layer (on top of SystemC) that enables transaction-based interactions between the components of a system, called *Transaction Level Modeling* (TLM).

In order to verify SystemC TLM programs, we propose a method that includes five main steps, namely **defining formal semantics**, **model extraction**, **fault modeling**, **model slicing**, and **model checking**. In order to extract a formal model from the given SystemC TLM program, first we need to specify the requirements of developing a formal semantics that can capture the SystemC TLM programs while still benefiting from automation techniques for verification and/or synthesis. Based on this intuition, we utilize two model extraction approaches that consider the architecture of the

given program too. In the first approach, we propose a set of transformation rules that helps us to extract a Promela model from the SystemC TLM program. In the second approach, we discuss how to extract a timed automata model from the given program.

When we have the formal model, we model and inject several types of faults into the formal models extracted from the SystemC TLM programs. For injecting faults, we have developed a tool, called *UFIT*, that takes a formal model and a desirable fault type, and injects the faults into the model accordingly.

The models extracted from the SystemC TLM program are usually very complex. Additionally, when we inject faults into these models they become even more complex. Hence, we utilize a model slicing technique to slice the models in the presence or absence of faults. We have developed a tool, called *USlicer* that takes a formal model along with a set of properties that needs to be verified, and generate a sliced model based on the given properties. The results show that verification time and the memory usage of the sliced version of the model is significantly smaller than that of the original model. Subsequently, in some cases where the verification of the original formal models is not even possible, using our model slicing technique makes the verification possible in a reasonable time and space.

We demonstrate the proposed approach using several SystemC transaction level case studies. In each case study, we explain each step of our approach in detail and discuss the results and improvements in each of them.

To my parents, Mastaneh and Dariush.

ACKNOWLEDGMENTS

First and foremost, I feel indebted to my advisor, Professor Sandeep Kulkarni, for his guidance, encouragement, and inspiring supervision throughout the course of this research work. His patience, extensive knowledge, and creative thinking have been the source of inspiration for me. He was available for advice or academic help whenever I needed and gently guided me for deeper understanding, no matter how late or inconvenient the time is. It is hard to express how thankful I am for his unwavering support over the last years.

I would like to take on this opportunity to thank my dissertation committee members Dr. Abdol-Hossein Esfahanian, Dr. Guoliang Xing, and Dr. Subir Biswas who have accommodated my timing constraints despite their full schedules, and provided me with precious feedback on my dissertation presentation. Also, during my Ph.D. studies, I had the pleasure of collaborating with Dr. Ali Ebnenasir. His valuable comments helped me a lot to find the right roadmap in my research. I also would like to thank Dr.

Living in East Lansing without my good friends would not have been easy. I want to thank all my friends in the department and outside the department. I wish I could name you all.

Last but definitely not least, I want to express my deepest gratitude to my beloved parents and my dearest sister. Their love and unwavering support have been crucial to my success, and a constant source of comfort and counsel. Special thanks to my parents for abiding by my absence in last five years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

Introduction

1.1 Motivation

Verification of today's complex SoC (System on Chip) systems is difficult in part due to the huge scale of integration and the fact that capturing crosscutting concerns (e.g., system verification) in the Register Transfer Language (RTL) [76] is non-trivial [20]. Additionally, SoC systems, in practice, are in the presence of faults that makes it even more difficult to verify such systems. More importantly, modern design languages (e.g., SystemC [2]) enable the co-design of hardware and software components, which makes it even more challenging to Verify SoC systems. Thus, enabling the systematic (and possibly automatic) verification of SystemC programs in the presence of faults can have a significant impact.

SystemC is a widely accepted language and an IEEE standard [2]. It includes a C++ library of abstractions and a run-time kernel that simulates the specified system, thereby enabling the early development of embedded software for the system that is being designed. To enable and facilitate the communication of different components in SystemC, the Open SystemC Initiative (OSCI) [2] has proposed an *interoperability* layer (on top of SystemC) that enables transaction-based interactions between the components of a system, called *Transaction Level Modeling* (TLM) [4]. The interoperability layer enables two main abstraction levels (a.k.a. *coding styles*), namely Loosely-Timed (LT) and Approximately-Timed (AT). The LT style of coding is mainly used when designers need fast simulation of a program with little concern about timing issues, whereas the

AT style provides a notion of global time during simulation. Since SoC systems are subject to different types of faults (e.g., single-event upset, hardware aging, etc.), it is desirable to study their behavior in the presence of such faults. However, verification of SystemC TLM programs in the presence of faults is non-trivial as designers have to deal with appropriate manifestations of faults and verification at different levels of abstraction. In this thesis, we will develop a systematic and fully automated method for augmenting existing SystemC TLM programs with verification capability.

There are numerous approaches for fault injection and impact analysis, testing and verification of SystemC programs. These approaches lack a systematic method for verification of SystemC programs in the presence of faults. Testing methods can be classified into two categories: test patterns and verification-based methods. Test patterns [28] enable designers to generate test cases and fault models [44] for SystemC programs at a specific level of abstraction and use the results to test lower levels of abstraction. Verification approaches [14, 47, 55, 62, 74] use techniques for software model checking where finite models of SystemC programs are created (mainly as finite state machines) and then properties of interest (e.g., data race or deadlock-freedom) are checked by an exhaustive search in the finite model. Fault injection methods [20, 27, 64, 70, 75] mainly rely on three techniques of (i) inserting a faulty component between two components; (ii) replacing a healthy component with a faulty version thereof, and (iii) injecting signals with wrong values at the wrong time. Then, they analyze the impact of injected faults in system outputs at different levels of abstraction (e.g., RTL and TLM) [29]. Most of the aforementioned approaches enable the modeling of faults and their impacts with little support for systematic verification that can be captured at different levels of abstraction.

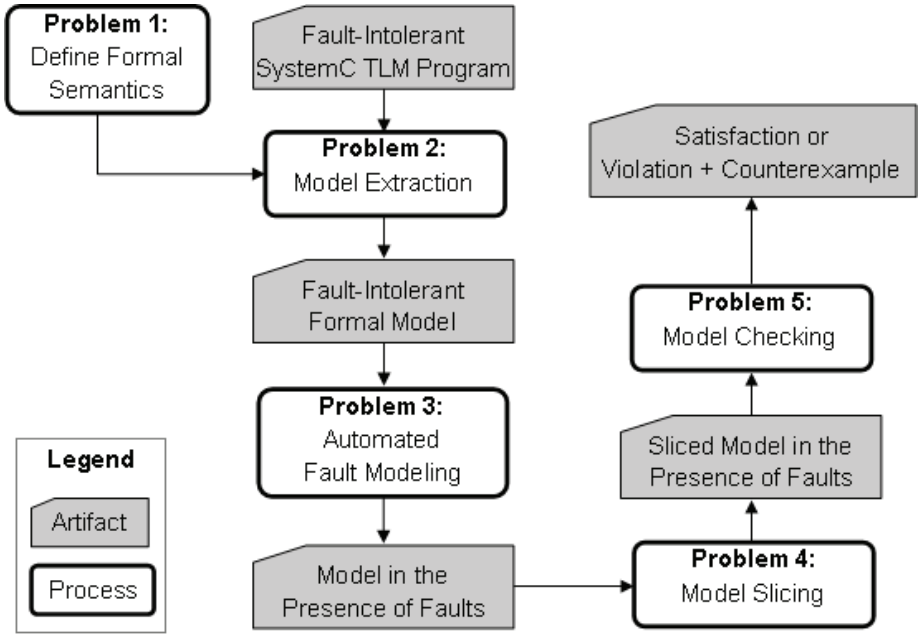

Figure 1.1: Overview of the proposed framework.

1.2 Objectives and Proposed Framework

Our objective is to facilitate the verification of SystemC TLM programs by *building tools to provide automation to the extent feasible*. Our proposed approach exploits model extraction, model checking, and verification techniques to enable a framework for the verification of SystemC TLM programs. Specifically, our approach consists of five steps, namely, *defining formal semantics*, *model extraction*, *fault modeling*, *model slicing*, and *model checking*. These steps are represented by Problem 1, Problem 2, Problem 3, Problem 4, and Problem 5 in Figure 1.1, respectively.

In this framework, we start with a SystemC TLM program that meets its functional requirements, but does not exhibit tolerance in the presence of a specific type of faults (e.g., transient faults, stuck-at faults, component failure, etc.), called the *fault-intolerant* program. Existing testing and verification methods [14, 55, 62, 74] can be used to ensure that a SystemC program meets its functional requirements in the absence of faults. In the first step, we define a formal semantics that can capture the transaction-based semantics of SystemC programs at different levels of ab-

straction while being amenable to automation (Problem 1 in Figure 1.1). Subsequently, we extract a formal model of the SystemC TLM program automatically (Problem 2 in Figure 1.1). Thereafter, our framework facilitates the modeling of different types of faults and their impacts on the formal model. These faults will consider typical faults considered in SystemC programs such as transient faults, stuck-at, component failure, etc (Problem 3 in Figure 1.1). Next, the framework provides model slicing to generate a simplified model based on properties/requirements/specifications of interest (Problem 4 in Figure 1.1). Finally, the sliced model is given to a model checker and the model checker gives us either ”yes”, which means the set of properties is satisfied, or ”no”, which means the properties is violated (Problem 5 in Figure 1.1). In case where the property is violated, the model checker gives us a counterexample too that can be used for revising the model. All of these five steps in our framework are automatic.

In the rest of this chapter, we explain each of the aforementioned problems in some detail and give an outline for the thesis.

Problem 1: Developing a formal semantics that can capture the communication protocols of TLM programs while being amenable to automation. In this step, we develop a formal semantics that preserves the structure/architecture of SystemC TLM programs. This formal semantics should also articulate different communication characteristics along with different coding styles of SystemC TLM programs.

Problem 2: A method for extracting an abstract model from SystemC TLM programs. In order to extract a formal model automatically, in addition to having a C++ compiler, we need to extract architecture of the SystemC TLM program. Having the architecture and behavior information can also assist us to translate back the abstract model to the SystemC program. Different approaches have been proposed to extract a formal model from a SystemC program

automatically [61, 62, 65, 66, 77]. However, to the best of our knowledge, all of them only consider pure SystemC programs while only one of them considers SystemC Transaction level Modeling programs.

Problem 3: An approach for modeling and analyzing the impact of faults on the formal specifications and the behaviors of SystemC TLM programs in the presence of faults. To analyze the impact of faults, we identify how different types of relevant faults (e.g., transient faults, message faults, stuck at faults, etc.) can be represented and injected into the abstract model.

Problem 4: A technique for slicing the formal model. The models extracted are usually very complex. They get even more complex after injecting fault into them in Problem 3. Hence, we utilize program slicing techniques to slice the model and generate a simplified version based on the given set of properties.

Problem 5: Model checking the formal model. In this step, we give the sliced model to a model checker to be model checked. If the specification is violated, the model checker gives us a counterexample that can be utilized to revise the formal model.

1.3 Thesis Overview

In Chapter 2, we give a background on SystemC, Transaction Level Modeling (TLM), UPPAAL timed automata, and Promela. We describe some of the previous work related to this thesis in Chapter 3. In Chapter 4, first we discuss about the requirements of a target formal semantics, and then we introduce a set of transformation rules for transforming a SystemC TLM program into Promela and UPPAAL time automata. We also introduce a tool for extracting timed automata

models from SystemC TLM programs. Afterwards, we use the extracted models for modeling faults in Chapter 5. A tool, called *UFIT*, for modeling faults is explained in Chapter 6. This tool injects different types of faults into UPPAAL timed automata models. In Chapter 7, we discuss how to slice timed automata models and propose our tool, called *USlicer* for slicing UPPAAL timed automata models in Chapter 8. Finally, in Chapter 9, we conclude this dissertation and describe the possible directions for future work.

1.4 Bibliographic Notes

Some of the results in this dissertation have appeared in prior publications. The materials in Chapter 4 are based on the papers published in Conferences ICDCN 2012 (International Conference on Distributed Computing and Networking) [25], SSS 2013 (International Symposium on Stabilization, Safety, and Security) [36], NoCArc 2013 (Network on Chip Architectures) [37], and SEFM 2014 (International Conference on Software Engineering and Formal Methods) [39], and Journal TCS 2013 (Theoretical Computer Science) [26]. The materials in Chapter 5 are based on the papers published in Conferences SSS 2013 (International Symposium on Stabilization, Safety, and Security) [36], NoCArc 2013 (Network on Chip Architectures) [37], ICDCS 2013 (International Conference on Distributed Computing Systems) [38], SEFM 2014 (International Conference on Software Engineering and Formal Methods) [39], and NFM 2015 (NASA Formal Methods) [40]. The material in Chapter 6 is based on the work published in NFM 2015 (NASA Formal Methods) [40]. Finally, the material in Chapters 7 and 8 are based on the work published in DAC 2016 (Design Automation Conference) [41].

Chapter 2

Preliminaries

This chapter provides a brief background on SystemC (Section 2.1), Transaction Level Modeling (Section 2.2), UPPAAL timed automata (Section 2.3), and Promela (Section 2.4). The concepts represented in this chapter are mainly adapted from [2–5, 9].

2.1 SystemC

SystemC [2] was introduced by the Open SystemC Initiative (OSCI) in 1996. The aim of the Open SystemC Initiative was to develop an open industry standard for system-level modeling, design and verification. SystemC can be seen as both a system level design language and a framework for HW/SW co-simulation. It allows the modeling and execution of system level designs on various levels of abstraction, including classical register transfer level hardware modeling and transaction-based design. This allows system-level design from abstract concept down to implementation in a unified framework. SystemC without extensions can only be used for digital HW/SW systems. There also exists an extension for analog and mixed-signal components, namely SystemC-AMS, but this is not in the scope of this thesis.

SystemC is implemented as a C++ class library, which provides the language elements and an event-driven simulation kernel. The language comprises constructs for modularization and structuring, for hardware, software and communication modeling, and for synchronization and coordination of concurrent processes. From a structural point of view, a SystemC design is a

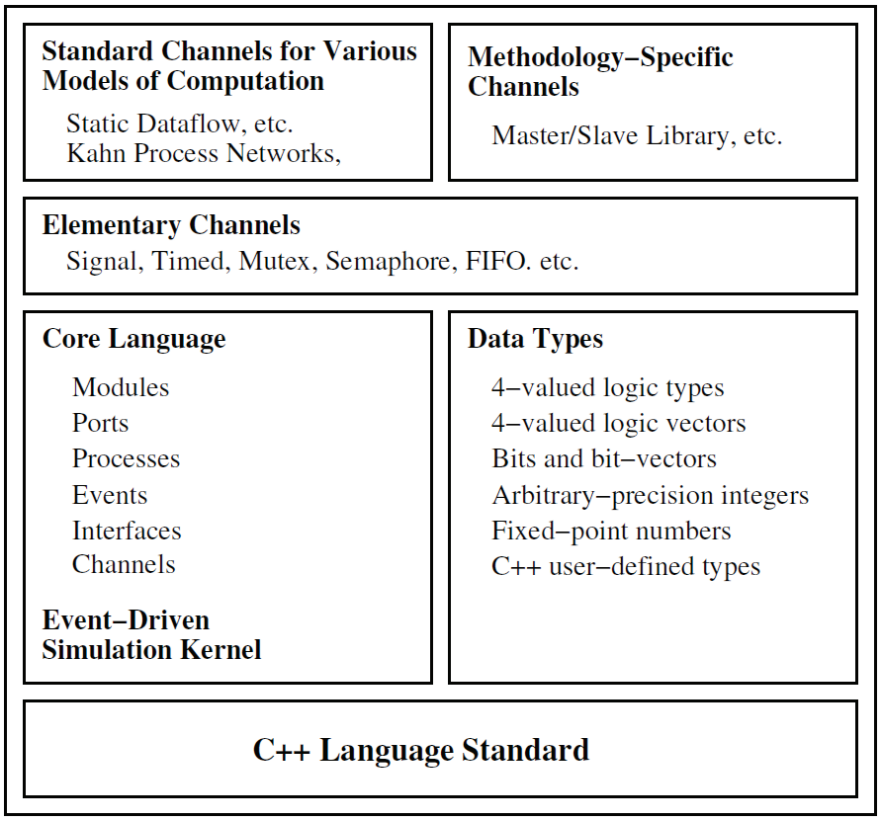

Figure 2.1: SystemC language structure [2].

set of modules, connected by channels. The structure strictly separates between computation and communication units (i. e., modules and channels) and is highly flexible due to a communication concept that allows transaction level modeling and communication refinement. The event-driven simulation kernel regards the SystemC design as a set of concurrent processes that are synchronized and coordinated by events and communicate through channels.

The SystemC language architecture is shown in Figure 2.1 [2]. The SystemC language provides constructs for the modeling of concurrency, time, reactivity, hardware data types, hierarchy and communication. As SystemC is implemented as a C++ class library, the C++ language standard constitutes the base of the language architecture. Above that, the core language of SystemC provides means to describe the structure and the behavior of a system. The structure is described by using modules, channels, ports, and interfaces, the behavior by using processes and events.

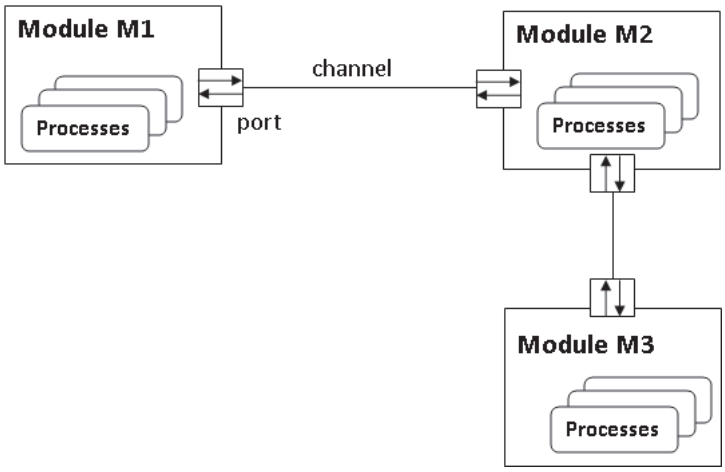

Figure 2.2: Structure of a SystemC design.

Together with the event-driven simulation kernel, the core language defines the semantics of SystemC. Alongside to that, the SystemC language provides a set of hardware data-types. On top of the core language and the dedicated hardware data-types, a set of elementary channels is defined, which can be used for more specific models of computation, e.g., FIFOs for functional or signals for hardware modeling. The topmost layer of the SystemC language architecture consists of design libraries and models needed for more specific design methodologies or models of computation. Note that those are not part of the SystemC standard. The SystemC standard [2] comprises the core language together with the event-driven simulation kernel, the dedicated data-types, and the elementary channels. In the following, we describe both the structure and the behavior of a SystemC design and briefly review the simulation semantics.

2.1.1 Structural Modeling

From a structural point of view, each SystemC program has a *sc_main* function, which is the entry point of the application and is similar to the *main* function of a C++ program. In this function, the designer creates structural elements of the system, called *modules*, and connects them using *channels* (see Figure 2.2). The separation of modules and channels allows the separation of *com-*

putation and *communication*. Together with a flexible communication model based on channels, ports, and interfaces, this allows *Transaction Level Modeling* (TLM) with SystemC. Modules are the basic building blocks that allow a modular and hierarchical design.Each module contains *processes*, *ports*, *internal data*, *channels*, and *interfaces*. A *process* is the main computing element of a module that is executable every time an event is triggered. An *event* is a basic synchronization object that is used to synchronize between processes and modules. The processes in a SystemC program are conceptually concurrent and can be used to model the functionalities of the module. A *port* is an object through which a module communicates with other modules. A *channel* is a communication element of SystemC that can be either a simple wire or a complex communication mechanism like FIFO. A port uses an *interface* to communicate with the channel [2].

2.1.2 Behavioral Modeling

SystemC designs are executed in a discrete-event simulation. The basic execution unit are processes, which are triggered by events. Thus, from a behavioral point of view, a SystemC design can be regarded as a network of concurrent processes, which communicate through channels and synchronize on events. In the following, we describe the main concepts of processes and events and how they are used in the discrete-event simulation.

2.1.2.1 Processes

Processes are contained in modules and use the ports of the containing module to access external channels. SystemC provides two kinds of processes: *method processes* and *thread processes*. A method process, when triggered, always executes its body from the beginning to the end and does not keep an internal execution state. It is not possible to suspend and resume a method process. In contrast to that, a thread process can be suspended at any time by calling a *wait* function. It keeps

its internal execution state and thus can be resumed at the point where it was suspended. Note that a thread process is only started once at the beginning of simulation, whereas a method process may be invoked arbitrary often.

The functionality of processes is described in methods, which contain the executable code of a SystemC design. For execution, the methods are encapsulated into processes, which care for the interactions with the scheduler and the events. As a consequence, methods are either invoked by the encapsulating process, or called by other methods. This includes communication methods, which are called as external methods through the port their channel is bound to.

2.1.2.2 Events

Both thread and method processes are triggered by *events*. An event is an object that determines whether and when a process would be triggered. The triggering of an event is called *event notification*. Whenever an event is notified, this triggers the execution of all processes that are *sensitive* to the event. A process may be sensitive to an event either statically or dynamically. Static sensitivity is allowed for both method and thread processes, dynamic sensitivity is only allowed for thread processes. A static sensitivity list is attached to a process statically within the module constructor, where their static sensitivity lists consist in each case only of the clock event. A static sensitivity list may also contain multiple events. A method process is triggered, whenever an event from its static sensitivity list is notified. While method processes are executed from the beginning to the end whenever an event from their static sensitivity list occurs, thread processes may suspend execution by calling a wait function. This overwrites their static sensitivity list temporarily and is called dynamic sensitivity. For example, if a process calls *wait(e)*, it becomes sensitive to the event e and is resumed at the next occurrence (i.e., notification) of the event e. A process can also be dynamically sensitive to multiple events or for the elapsing of a certain amount of time. Table 2.1

shows the variants of wait calls available in SystemC. As a thread process either runs or is suspended, the only possibility to wait for an event from the static sensitivity list in a thread process is to suspend it with an empty *wait()* statement. If an event object e is notified by its owner, processes that are sensitive to the event start respectively resume execution.

wait(e)	wait for event e to be notified
wait(t)	wait for t time units to elapse
wait(t, e)	wait for event e for maximally t time units
wait()	wait for any event from the static sensitivity list
wait($e1$ & $e2$ & $e3$)	wait for all three events to be notified
wait($e1 \mid e2 \mid e3$)	wait for any of the three events to be notified

Table 2.1: Variants of the *wait* statement.

SystemC supports three types of event notifications. An *immediate notification*, invoked by e.notify(), causes processes to be triggered immediately in the current delta cycle. A *delta-delay notification*, invoked by e.notify(0), causes processes to be triggered at the same time instant, but after updating primitive channels, i.e., in the next delta-cycle. A *timed notification*, invoked by e.notify(t) with t $>$ 0, causes processes to be triggered after the given delay t. If an event is notified that already has a pending notification, only the notification with the earliest expiration time takes effect. That means that immediate notifications override all pending notifications, delta-delay notifications override timed notifications, and timed notifications override pending timed notifications if their delay expires earlier.

2.1.3 Simulation Kernel and Scheduler

SystemC has a simulation kernel that enables the simulation of SystemC programs. The SystemC scheduler is a part of the SystemC kernel that selects one of the processes that has an activated event in its sensitivity list. The *sensitivity list* is a set of events or time-outs that causes a process to

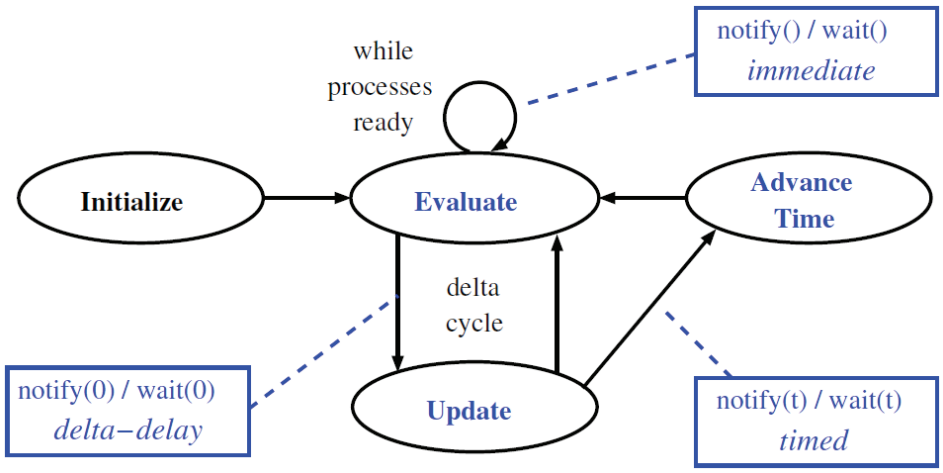

Figure 2.3: SystemC scheduler [2].

be either resumed or triggered. Figure 2.3 illustrates the behavior of the SystemC scheduler. The SystemC scheduler includes the following phases to simulate a system [2]:

1. *Initialization* phase: This phase initiates the primary runnable processes. A process is in a runnable state when one or more events of its sensitivity list have been notified.

2. *Evaluation* phase: In this phase, the scheduler selects one process to either execute or resume its execution from the set of runnable processes. Once a process is scheduled for execution, it will not be preempted until it terminates; i.e., a *run-to-completion* scheduling policy. The scheduler stays in the evaluation phase until no other runnable processes exist.

3. *Update* phase: This phase updates signals and channels.

4. *delta (δ) notification* phase: A delta notification is an event resulting from an invocation of the notify() function with the argument SC_ZERO_TIME. Upon a delta notification, the scheduler determines the processes that are sensitive to events and timeouts, and adds them to the list of runnable processes.

5. *Timed notification* phase: If pending timed notifications or timeouts exist, the scheduler iden-

tifies the corresponding sensitive processes and adds them to the set of runnable processes.

2.2 Transaction Level Modeling

In Transaction Level Modeling (TLM), a *transaction* is an abstraction of the communication (caused by an event) between two SystemC components for either data transfer or synchronization. One of the components initiates the transaction, called the *initiator*, in order to exchange data or synchronize with the other component, called the *target*. The philosophy behind TLM is based on the separation of communication from computation [4]. For example, consider the SystemC TLM program of Figure 2.4. In this example, we have two modules: *initiator* and *target* (Lines 6-15, and 17-32). The *initiator* module includes a process called *initiate*, and the *target* module has the *incModEight* process. The process *incModEight* waits for a notification on the internal event *e* (Line 29) before it updates its local variable d. The *sc_start* statement (Line 39) notifies the simulation kernel to start the simulation. The event e will be notified when the trigger method of the target is called from the initiate process (Line 14).

While the program in Figure 2.4 illustrates how an initiator and a target module can communicate using SystemC ports and method invocations, the OSCI initiative further facilitates TLM programming by introducing an interoperability layer. The interoperability layer includes a set of core components as follows:

- *Core Interfaces.* The core interfaces comprise a set of methods that mainly support two abstraction levels supported by two coding styles, namely Loosely-Timed (LT) and Approximately-Timed (AT) coding styles. The LT style is mainly used when designers need fast simulation of a program with little care about timing concerns. Such a style of coding heavily relies on a blocking transport interface b_transport() that should be implemented in target modules

```
1  class target_if : virtual public sc_interface {
2  public:
3      virtual void trigger() = 0;
4  };
5
6  class initiator : public sc_module {
7  public:
8      sc_port<target_if> port;
9      SC_HAS_PROCESS(initiator);
10     initiator(sc_module_name name) : sc_module(name) {
11         SC_THREAD(initiate);
12     }
13     void initiate()
14         { port->trigger(); }
15  };
16
17 class target : public target_if, public sc_module {
18 public:
19     short d;
20     sc_event e;
21     SC_HAS_PROCESS(target);
22     target(sc_module_name name) : sc_module(name) {
23         d = 0;
24         SC_THREAD(incModEight);
25         }
26     void trigger()
27         { e.notify(SC_ZERO_TIME); }
28     void incModEight() {
29         wait(e);
30         d = (d+1)%8;
31     }
32 };
33
34 int sc_main (int argc , char *argv[]) {
35     initiator initiator_inst(Initiator);
36     target target_inst(Target);
37
38     initiator_inst.port(target_inst);
39     sc_start();
40     return 0;
41 }
```

Figure 2.4: A simple running example for two communication modules.

and invoked by initiators. The AT style of coding is used when timing issues are important to consider in simulation. In this style of coding, designers benefit from a non-blocking transport interface nb_transport(). The b_transport() and nb_transport() are part of the core interfaces in the interoperability layer. The core interfaces include four other methods, nonetheless, we focus only on the b_transport() interface as the rest of them are beyond the scope of this paper.

- *Generic Payload.* In TLM, transactions are objects captured by a structure, called the *generic payload*, that includes a set of attributes of the transaction object.

- *Sockets.* In the interoperability layer, modules communicate by sending and receiving transactions. Observe that the communication between the initiator and the target in Figure 2.4 is achieved through fine-grained declaration of SystemC ports and method invocations, which requires the initiator to have some knowledge of the internals of the target. The interoperability layer provides *sockets*, which are programming constructs that achieve two goals: connect modules by binding initiator and target sockets together, and facilitate the transmission of transactions between modules by hiding details.

- *Base Protocol.* The base protocol maximizes interoperability by providing a set of rules that can be used by the initiator and target modules when sending/receiving generic payloads through sockets.

To illustrate the SystemC TLM programs using TLM Base protocol and interoperability, consider the following example adapted from [1]. This example models how on-chip memory-mapped busses are captured using the TLM base protocol. In this example (see Figures 2.5 and 2.6), the Initiator module (Lines 1-32 in Figure 2.5) generates a transaction, while the Target module (Lines 33-63 in Figure 2.6) represents a simple memory. The initiator module has a thread process (Lines

```
struct Initiator: sc_module
    {   tlm_utils::simple_initiator_socket<Initiator> socket;
        SC_CTOR(Initiator)   : socket("socket")
    {   SC_THREAD(thread_process);   }

    void thread_process()
    {
        tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
        sc_time delay = sc_time(10, SC_NS);

        tlm::tlm_command cmd = static_cast<tlm::tlm_command>(rand()%2);

        if (cmd == tlm::TLM_WRITE_COMMAND) data = 0xFF000000 | 0;

        trans->set_command(cmd);
        trans->set_address(0);
        trans->set_data_ptr(reinterpret_cast<unsigned char*>(&data));
        trans->set_data_length(4);
        trans->set_streaming_width(4);
        trans->set_byte_enable_ptr(0);
        trans->set_dmi_allowed(false);
        trans->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);

        socket->b_transport(*trans, delay);

        if (trans->is_response_error())
            SC_REPORT_ERROR("TLM-2","Response error");
        wait(delay);
    }

    int data;
};
```

Figure 2.5: The Initiator module.

```
33 struct Memory: sc_module
34     {
35     tlm_utils::simple_target_socket<Memory> socket;
36     enum { SIZE = 256 };
37     SC_CTOR(Memory)  : socket("socket")
38     {
39         socket.register_b_transport(this, &Memory::b_transport);
40         for (int i = 0; i < SIZE; i++)
41             mem[i] = 0xAA000000 | (rand() % 256);
42         }
43
44     virtual void b_transport(tlm::tlm_generic_payload& trans, sc_time& delay)
45     {
46         tlm::tlm_command cmd = trans.get_command();
47         sc_dt::uint64    adr = trans.get_address() / 4;
48         unsigned char*   ptr = trans.get_data_ptr();
49         unsigned int   len = trans.get_data_length();
50         unsigned char*   byt = trans.get_byte_enable_ptr();
51         unsigned int    wid = trans.get_streaming_width();
52
53         if (adr >= sc_dt::uint64(SIZE) || byt != 0 || len > 4 || wid < len)
54          SC_REPORT_ERROR("TLM-2","Target does not support the transaction");
55         if ( cmd == tlm::TLM_READ_COMMAND )
56             memcpy(ptr, &mem[adr], len);
57         else if ( cmd == tlm::TLM_WRITE_COMMAND )
58             memcpy(&mem[adr], ptr, len);
59
60         trans.set_response_status(tlm::TLM_OK_RESPONSE);
61     }
62     int mem[SIZE];
63 };
64
65 SC_MODULE(Top)
66 {   Initiator *initiator;
67     Memory    *memory;
68     SC_CTOR(Top)
69     {
70         initiator = new Initiator("initiator");
71         memory    = new Memory   ("memory");
72         Initiator->socket.bind(memory->socket);
73 } };
74 int sc_main(int argc, char* argv[])
75 {   Top top("top");
76     sc_start();
77     return 0; }
```

Figure 2.6: The Memory, the Top, and the Main module.

6-29 in Figure 2.5) that sends a generic payload to the Target module; i.e., the Memory module.

In Lines 15-22 in Figure 2.5, we initialize the attributes command, address, data, byte_enables, streaming_width, response_status, and DMI hint. To send/receive a transaction to/from the memory module, we need a two-way communication between the modules. Thus, we define an initiator socket in Lines 2-3 in Figure 2.5 and a target socket in Line 35 of Figure 2.6. The initiator sends the transaction out through the initiator socket (Line 24 in Figure 2.5), and the memory communicates with the initiator by first registering a callback method with the socket (Line 39 in Figure 2.6), and then implementing that method (Lines 44-61 in Figure 2.6). The memory module then, in this method, implements the read and write commands by copying data to or from the data area in the initiator (Lines 53-58 in Figure 2.6). The final act of the memory module is to set the response_status attribute of the generic payload to indicate the successful completion of the transaction (Line 60 in Figure 2.6). If not set, the default response_status would indicate to the initiator that the transaction is incomplete (Lines 26-27 in Figure 2.5). In each TLM SystemC program we need a sc_main function (Lines 74-77 in Figure 2.6). Moreover, to connect up the module hierarchy, we use the Top module (Lines 65-73 in Figure 2.6). The top-level module of the hierarchy instantiates one initiator and one memory, and binds the initiator socket in the initiator to the target socket in the target memory (Line 72 in Figure 2.6).

2.3 UPPAAL Timed Automata

Timed Automata (TA) are state machines that enable the modeling of real-time systems [5]. The notion of time is captured by real-valued *clock* variables. The clock values are used to express the timing constraints and can be assigned to locations (vertices) and transitions (edges) of the TA. The semantics of TA is given by an infinite-state transition system where transitions correspond either

to a change of location (discrete transition) or to passage of time (time transition). UPPAAL [9,10] is an integrated tool environment for modeling, simulation, and verification of real-time systems modeled as networks of timed automata, extended with data types. A system in UPPAAL consists of concurrent processes, each of them modeled as a TA. Each process TA has a set of locations and transitions. To control transitions between locations, UPPAAL uses *guards* that limit when process actions can be executed and *synchronization channels* that require multiple processes to coordinate. In the following, we first introduce the semantics of Timed Automata and Networks of Timed Automata. Then, we describe some specialties and extensions of the Uppaal modeling language.

2.3.1 Timed Automata

As typical state automata, timed automata consist of a set of nodes, which are called locations and which are connected by edges. A notion of time is introduced by a set of real-valued clock variables $C : R_{\geq 0}$. They are used in clock constraints to model time-dependent behavior. The clocks are initialized with zero and then run synchronously with the same speed. As an effect of a transition, a clock may be reset, i.e., set to zero. A clock constraint is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$ for $x, y \in C, \sim \in \{\leq, <, =, >, \geq\}, n \in N$. $B(C)$ denotes the set of clock constraints. In *Timed Büchi Automata*, clock constraints are assigned to edges and are interpreted as enabling conditions for the corresponding transitions. They cannot force the transition to be taken. As a consequence, a Timed Büchi Automaton may stay an infinite amount of time in the same location. Alur *et al.* [5] solved this problem by *Büchi acceptance conditions*. A subset of locations is marked as accepting, and only executions passing through an accepting location infinitely often are considered as valid behaviors. A more intuitive solution to the problem of infinite idling is given by Henzinger *et al.* [46] by introducing *Timed Safety*

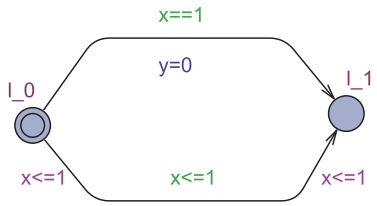

Figure 2.7: A simple timed automaton.

Automata. In Timed Safety Automata, one can distinguish two kinds of clock constraints: *Guards* are assigned to edges and yield conditions, under which the corresponding transition may be taken. In other words, they enable progress. Invariants are assigned to locations and yield conditions, under which one may stay in the corresponding state. The invariants must not be violated, i.e., the location must be left before its invariant is invalidated. In other words, invariants *ensure* progress. In the remainder of this thesis, we refer to *Timed Safety Automata* whenever we use the term *timed automata*.

A simple example for a timed automaton is shown in Figure 2.7. It consists of two locations l_0 and l_1 that are connected by two edges from l_0 to l_1. To l_0 and l_1, the same invariant $x \leq 1$ is assigned. That means that in both locations, the automaton may stay at most for one time unit. The upper edge from l_0 to l_1 has a guard $x == 1$, and the clock y is reset whenever this edge is taken. The lower edge from l_0 to l_1 has a guard $x \leq 1$ and no effect. As a consequence, there are two possibilities to come from location l_0 to location l_1: during time $x \in [0, 1]$, the lower edge may fire without effect, and at $x = 1$, the upper edge may fire and y is reset.

2.3.2 Networks of Timed Automata

Networks of timed automata are used to model systems with concurrent processes. The state of a network of timed automata is defined as a vector of the current locations of all timed automata in the network and all clock valuations. For synchronization, the automata may interchange events. An

event is sent over a channel c, and $c!$ and $c?$ denote sending and receiving an event respectively [11].

2.3.3 Symbolic Semantics of Timed Automata

The semantic state space of timed automata is infinite due to the real-valued clock variables. This makes it impossible to apply automatic verification techniques such as model checking, which explore the whole semantic state space. To solve this problem, the symbolic semantics presented by Bengtsson *et al.* [10] abstracts from certain points of time and uses clock zones instead. As a consequence, a state is then a tuple $(\bar{l},D)$ where D is a difference bound matrix representing a clock zone. The resulting abstract model has a finite state space and can be model checked.

The foundation for a symbolic semantics of timed automata was laid by Alur *et al.* [6]. There, the notion of *region equivalence* was introduced. The idea is that two clock assignments can be considered equivalent, if they have no influence on the possible transitions the timed automaton can take. If only integer variables are used in clock constraints that means that two clock assignments can be considered equivalent, when for each clock

- both are greater than a given maximal constant, also called *clock ceiling*;
- their integer part is equal and both have a fractional part of zero, or
- their integer part is equal and both have a fractional part greater than zero.

In any case, the two clock assignments have to be in the same relation to all other clocks.

2.3.4 UPPAAL

UPPAAL [9, 10] is a tool set for the modeling, simulation, animation and verification of networks of timed automata. The UPPAAL model checker enables the verification of temporal properties,

including safety and liveness properties. The simulator can be used to visualize counter-examples produced by the model checker.

2.3.4.1 Modeling Language

The Uppaal modeling language extends timed automata by introducing parameterized timed automata templates, bounded integer variables, binary and broadcast channels, and urgent and committed location. Timed automata templates provide the possibility to model similar timed automata only once and to instantiate them arbitrary often with different parameters. Timed automata are modeled as a set of locations, connected by edges. The initial location is denoted by  . Invariants can be assigned to locations and enforce that the location is left before they would be violated. Edges may be labeled with selections, guards, updates, and synchronizations. Selections are used to non-deterministically bind a given identifier to a value in a given range. Updates are used to reset clocks and to manipulate the data space, i.e., they provide the actions the automaton may perform. Processes synchronize by sending and receiving events through channels. Sending and receiving via a channel c is denoted by `c!` and `c?`, respectively. Binary channels are used to synchronize one sender with a single receiver. A synchronization pair is chosen non-deterministically if more than one is enabled. If no communication partner is available, both the sender and the receiver are blocked if they synchronize on a binary channel. Broadcast channels are used to synchronize one sender with an arbitrary number of receivers. Any receiver that can synchronize must do so. In contrast to binary communication, a process sending on a broadcast channel is never blocked. Urgent and committed locations are used to model locations where no time may pass. Urgent locations are graphically depicted by the symbol  , committed locations by the symbol  . Leaving a committed location has priority over leaving non-committed locations.

An Uppaal model comprises three parts: global declarations, parameterized timed automata

(TA templates) and a system declaration. In the global declarations section, global variables, constants, channels and clocks are declared. The timed automata templates describe timed automata that can be instantiated with different parameters to model similar process. In the system declaration, the templates are instantiated and the system to be composed is given as a list of timed automata.

2.3.4.2 Query Language

The query language, which is used in UPPAAL to express requirements specifications, is a restricted version of CTL [9]. Like in CTL, the query language consists of path formulas and state formulas. State formulas describe individual states, whereas path formula quantify over paths of the model. Path formula can be classified into reachability, safety, and liveness.

State formulas are expressions that can be evaluated for a given state without looking at the rest of the model. This includes boolean expressions on variables (e.g., $x \leq 4$) and tests whether a particular process is in a given location (e.g., P1.init). A deadlock is expressed using the special state formula deadlock.

Path formulas express either reachability, safety, or liveness properties. The reachability property that some state satisfying a given state formula ϕ is expressed by $E <> \phi$. The safety properties that a state formula ϕ is always true is expressed by $A[]\ \phi$, whereas $A[]\ \phi$ says that there exists a path where ϕ is always true. The classical liveness property that something good will eventually happen is expressed by $A <> \phi$. Additionally, there exists a leads to or response property $\phi - - > \psi$, which expresses that whenever ϕ is satisfied, ψ will eventually be satisfied.

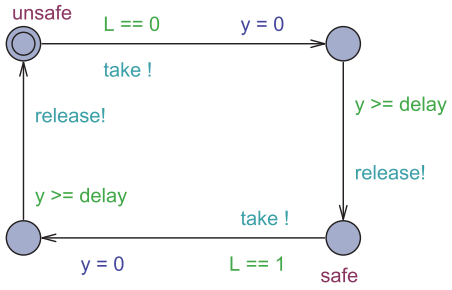

Figure 2.8: The viking automaton [9].

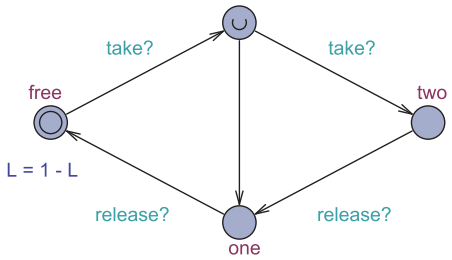

Figure 2.9: The torch automaton [9].

2.3.4.3 An Illustrative UPPAAL Example

An example for an UPPAAL model taken from the demo models included in the free UPPAAL distribution is the riddle of the four vikings. The riddle is as follows: four vikings want to cross a bridge at night, but they have got only one torch and the bridge can only carry two of them. Thus, they can only cross the bridge in pairs and one has to bring the torch back to the other side before the next pair can cross. The vikings have different speeds, the fastest needs 5 minutes, the slowest 25 minutes, and the other two 10 and 20 minutes. The question is whether it is possible that all the vikings cross the bridge within 60 minutes.

To model this problem in UPPAAL, we need two timed automata templates, one for the vikings which is instantiated with the different delays, and one for the torch, see Figures 2.8 and 2.9. The representation of timed automata is a usual automata representation with locations connected by edges. In addition, we have two channels take and release, which model the interaction between the vikings and the torch. Furthermore, we have a data variable L which serves as a semaphore to

ensure that the torch can only be on one side of the bridge at a time, and we have a clock variable y and a clock constraint $y \geq delay$ which models the time it takes the vikings to cross the bridge. A viking is on the other side of the bridge if it is in its safe location.

The question if they all can cross the bridge in 60 minutes can be formalized as an existential quantification over a state where all vikings are in their safe location and time is less or equal than 60 minutes:

```
E<> Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe and time <= 60
```

Note that the example of the four vikings is comparable to the question if a packet can reach its receiver in a given time limit in a communication network or a Network on Chip (NoC) systems.

2.4 SPIN and Promela

SPIN [48] is an efficient verification system for analyzing the logical consistency of distributed systems, specifically of data communication protocols. It has been used to detect design errors in applications ranging from high-level descriptions of distributed algorithms to detailed code for controlling telephone exchanges. The system is described in a modeling language called PROMELA (Process or Protocol Meta Language). The language allows for the dynamic creation of concurrent processes. In this section, we provide a brief description of SPIN and explain the basics of Promela.

2.4.1 Modeling Language

Promela is a verification modeling language. Using Promela we can make abstractions of protocols (or distributed systems in general) that suppress details that are unrelated to process interaction.

The intended use of Spin is to verify fractions of a process behavior that are considered suspect. The relevant behavior is modeled in Promela and verified using Spin. A complete verification is therefore typically performed in a series of steps, with the construction of increasingly detailed Promela models. Each model can be verified with Spin under different types of assumptions about the environment (e.g., message loss, message duplications etc). Once the correctness of a model has been established with Spin, that fact can be used in the construction and verification of all subsequent models.

The syntax of Promela is based on the C programming language. A Promela model comprises (1) a set of (concurrent) processes (2) a set of variables, and (3) a set of message channels. The processes specify the behavior of the model and all processes are global objects. Also, each Promela model has to contain at least one process to be meaningful. The variables are utilized to store the information about the system being modeled and can be declared globally or locally within a process. The global variables define the environment in which the process run. Message channels are used to model the transfer of data from one process to another. Next, we explain processes, variables, and message channels in some detail.

2.4.1.1 Processes

The state of a variable or of a message channel can only be changed or inspected by processes. The behavior of a process is defined in a predefined type, called *proctype*. This type contains the process identifier, formal parameter list, and local variable declaration and statements. The contents falls into the following form of the proftype declaration:

```
proctype process_identifier (formal parameter) {
      local variable declarations
```

```
        and statement
    }
```

The semantics of Promela is based on an operational model that defines how the actions of proctypes are interleaved. An action (also known as a *guarded command*) is of the form $grd \rightarrow stmt$, where the guard grd is an expression in terms of the Promela model's variables and the statement $stmt$ may update some model variables. Actions can be atomic or non-atomic, where an atomic action (denoted by the atomic $\{\}$ blocks in Promela) ensures that the guard evaluation and the execution of the statement are not interrupted. As an illustration for atomic actions, consider the following example.

```
atomic{    /* swap the values of a and b */
        tmp = b;
        b = a;
        a = tmp
        }
```

In the example, the values of two variables a and b are swapped in a sequence of statement executions that is defined to be *uninterruptable*. That is, in the interleaving of process executions, no other process can execute statements from the moment that the first statement of this sequence begins to execute until the last one has completed. It is often useful to use atomic sequences to start a series of processes in such a way that none of them can start executing statements until all of them have been initialized:

```
init {
        atomic {
```

Type	Range
bit	$0..1$
bool	$0..1$
byte	$0..255$
short	$-2^{15}..2^{15}-1$
int	$-2^{31}..2^{31}-1$

Table 2.2: Data types in Promela.

```
        run A(1,2);
        run B(2,3);
        run C(3,1)
        }
}
```

Atomic sequences may be non-deterministic. If any statement inside an atomic sequence is found to be unexecutable, however, the atomic chain is broken, and another process can take over control. When the blocking statement becomes executable later, control can non-deterministically return to the process, and the atomic execution of the sequence resumes as if it had not been interrupted.

2.4.1.2 Variables

Table 2.2 summarizes the five basic data types used in Promela. *Bit* and *bool* are synonyms for a single bit of information. The first three types can store only unsigned quantities. The last two can hold either positive or negative values. The precise value ranges of variables of types short and int is implementation dependent, and corresponds to those of the same types in C programs that are compiled for the same hardware.

2.4.1.3 Message channels

Message channels are declared either locally or globally, for instance as follows:

```
chan qname = [16] of { byte }
```

This declares a channel that can store up to 16 messages of type byte. Channel names can be passed from one process to another via channels or as parameters in process instantiations.
To send the value expression expr to the channel qname, we use the following command that appends the value to the tail of the channel.

```
qname!expr
```

Additionally, to receive the value expression expr from the channel qname, we use the following command that retrieves the expression from the head of the channel.

```
qname?expr
```

Moreover, if we want to send and receive the messages without storing them, we can use *rendezvous* channels. These types of channels are defined as follows.

```
chan qname = [0] of { byte }
```

Consider that rendezvous communication is binary: only two processes, a sender and a receiver, can be synchronized in a rendezvous handshake.

2.4.2 Control Flow

In this section, we identify two important control flow constructs in Promela: *case selection* AND *repetition*.

2.4.2.1 Case selection

The selection structure can contain more that one execution sequence, each preceded by a double colon. Only one sequence from the list will be executed. A sequence can be selected only if its first statement is executable. The first statement is therefore called a *guard*. For instance, in the following selection structure, we have two execution sequences that can be selected non-deterministically.

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

2.4.2.2 Repetition

A logical extension of the selection structure is the repetition structure. Only one option can be selected for execution at a time. After the option completes, the execution of the structure is repeated. The normal way to terminate the repetition structure is with a *break* statement. As an example, the following structure randomly changes the value of the variable $count$ up or downs

```
proctype counter()
{
        do
        :: count = count + 1
        :: count = count - 1
        :: (count == 0) -> break
```

```
        od
    }
```

2.4.3 Verification with SPIN

Given a model system specified in Promela, Spin can either perform random simulations of the system's execution or it can generate a C program that performs a fast exhaustive verification of the system state space. The verifier can check, for instance, if user specified system invariants may be violated during a protocol's execution. For this purpose, a set of properties needs to be given to Spin in *Linear Temporal Logic* (LTL) structure. In LTL, one can encode formula about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, etc. An LTL formula can contain the unary temporal operators $\Box$ (pronounced always), $\Diamond$ (pronounced eventually), and binary temporal operators U (pronounced until).

2.5 Summary

In this section, we presented the relevant background for our framework. To this end, we gave an introduction to SystemC programs and Transaction Level Modeling. We also explained the formal language of UPPAAL timed automata, which comes with a tool suite for modeling, simulation and animation of timed automata and a model checker. Finally, we described Promela language which is the input of SPIN model checker.

Chapter 3

Related Work

This section discusses existing approaches for formal analysis of SystemC TLM programs. Extant work can broadly be classified into four categories: methods for formalizing the semantics of SystemC TLM programs, checkers for SystemC designs, static analysis and stateless model checking of SystemC programs, and model extraction and model checking of SystemC TLM programs. In the next sections, we explain the researches in each category in some detail.

3.1 Formalizing the Semantics

Several researchers focus on assigning formal semantics to SystemC TLM programs. For example, a definition of the simulation semantics based on abstract state machines is given by Müller *et al.* [67] and Ruf *et al.* [72]. The purpose of their work is to provide a precise description of the SystemC scheduler. However, the system design itself, as built from modules, processes and channels, is not covered and therefore cannot be verified with this approach. Niemann and Haubelt [68] provide an approach for specifying the semantics of SystemC TLM programs using deterministic Communicating State Machines. In their approach, only the automaton that explicitly captures the scheduler has non-deterministic behaviors. Patel and Shukla [69] present a formalization of SystemC in abstract state machines and revise Microsoft SpecExplorer for the validation and debugging of SystemC programs. Kroening and Sharygina [53] create abstract models of SystemC programs using Labeled Kripke Structures (LKS), where each SystemC thread is captured by an

LKS. A Labeled Kripke Structures is a directed graph whose nodes represent states annotated by atomic propositions that hold in that state. The arcs of the directed graph denote transitions between states that are labeled by actions. The abstract state of the SystemC program is defined in terms of the local states of its threads, their program counters and the status of each thread in SystemC scheduler. Salem [73] presented a denotational semantics for the SystemC scheduler and for SystemC processes, but only for a synchronous subset. Habibi *et al.* [34, 35] proposed program transformations from SystemC into equivalent state machines. In these approaches, time is ignored, and the transformation is performed manually. Besides, the state machine models do not reflect the structure of the underlying SystemC designs. Traulsen *et al.* [77] proposed a mapping from SystemC to PROMELA, but they only handle SystemC designs at transaction level, do not model the non-deterministic scheduler and cannot cope with primitive channels. Harrath and Monsuez [43] introduced the formalism of SystemC waiting-state automata. Those SystemC waiting-state automata are supposed to allow a formal representation of SystemC designs at the delta-cycle level. However, the approach is limited to the modeling of delta-cycles, the scheduler and complex interactions between processes are not considered and the formal model has to be specified manually. Man [59] presented the formal language SystemC^{FL}, which is based on process algebras and defines the semantics of SystemC processes by means of structural operational semantics style deduction rules. SystemC^{FL} does not take dynamic sensitivity into account, and considers only simple communications. The concept of channels is neglected. A tool to automatically transform SystemC to SystemC^{FL} is presented by Man *et al.* [60]. However, it does not handle any kind of interaction between processes. Karlsson *et al.* [52] verify SystemC designs using a petri-net based representation. This introduces a huge overhead because interactions between subnets can only be modeled by introducing additional subnets.

Herber *et al.* [47] propose an approach to define a formal semantics for SystemC that can handle

relevant SystemC language elements, including process execution, interactions between processes, dynamic sensitivity and timing behavior. The informally defined behavior and the structure of SystemC designs are completely preserved. The mapping from SystemC designs into Uppaal timed automata is fully automated, introduces a negligible overhead, produces compact and comparably small models and enables the use of the Uppaal model checker and tool suite.

3.2 Checkers for SystemC Designs

There has been some work on checkers for SystemC designs. For example, an approach to check temporal assertions for SystemC has been presented by Ruf *et al.* [72]. More related to our work is the work of Drechsler, Große and Kühne [23, 30–33]. In [23], they describe how to convert a gate-level model given in SystemC into BDDs. The BDD is used for forward reachability analysis. In [30], they present a method which allows checking of temporal properties for circuits and systems described in SystemC, not only during simulation. A property is translated into a synthesizable SystemC checker and embedded into the circuit description. This enables the evaluation of the properties during the simulation as well as after the fabrication of the system. In [31, 32], they present an approach to prove that a SystemC model satisfies a given property using bounded model checking and show the applicability of the approach with the co-verification of a RISC CPU implemented in SystemC. In [33], they use a 3-step approach. First, they verify the functional correctness of the underlying hardware using bounded model checking. Then, they verify the HW/SW interface. This means that they verify, that each instruction through which the software can access the hardware has the specified effects on all hardware blocks involved. Finally, assembler programs are verified by constraining the instructions of the program as assumptions in the proof. In other words, the instructions of a given assembler program are translated into assumptions and the

known effects on the hardware are used for the proof.

The main limitation of the work of Drechsler, Große and Kühne is that their approaches are all restricted to synchronous and cycle-accurate models on register-transfer level. As a consequence, they can, in particular, not verify models using SystemC channels, necessary for transaction level modeling (TLM), nor can they handle dynamic or timing sensitivity. With our approach, we can handle SystemC design on low abstraction-levels as well as designs on high abstraction-levels and thus we can support the whole design-process.

3.3 Static Analysis and Stateless Model Checking

Many techniques combine static analysis with controlled scheduling in order to enable stateless model checking, where no explicit-state model is generated and properties are checked as the program executes. For instance, Blanc and Kroening [14] present a compiler that uses model checking to predict race conditions in SystemC programs. They use the results of predictions during simulation in order to reduce the number of interleavings. Kundu *et al.* [55] statically compute the total number of atomic blocks in SystemC code and then analyze the dependency of atomic blocks on each other. An atomic block in SystemC is the code between two consecutive wait() statements. Two blocks are dependent if one of them enables/disables another or one of them writes a shared variable that the other one reads. The main advantage of stateless model checking is that there is no need for model extraction; however, they generally have a bounded nature in that the approach is incomplete (i.e., it may miss some errors). This inherent feature of stateless model checking makes it difficult to model the impact of faults on the entire set of behaviors of a model due to its on-the-fly analysis nature.

3.4 Model Extraction and Model Checking

Model extraction and model checking methods use a set of rules for semantics-preserving transformation of SystemC programs to the modeling language of some model checkers. For example, Moy *et al.* [66] model a SystemC program as the automata-theoretic product of a set of synchronous automata representing SystemC threads along with an automaton representing the simulation scheduler of SystemC. They provide an intermediate formal language, called Heterogeneous Parallel Input/Output Machines, that can capture both synchronous and asynchronous automata. Traulsen *et al.* [77] present a method for transforming a subset of SystemC to the Promela [3] modeling language in order to enable the model checking of asynchronous software threads in the SPIN model checker [48]. Marquet and Moy [62] present a front-end that transforms SystemC programs to an intermediate language in LLVM [57], which is a framework that provides reusable components for compiler construction. Marquet *et al.* [61] present a model extraction scheme from SystemC to Promela where they provide a set of transformation rules for the synchronization primitives of SystemC (i.e., wait and notify statements). Moreover, they avoid explicit modeling of the SystemC scheduler, and present a set of invariant conditions for validating that the transformation rules are semantics-preserving. Cimatti *et al.* [21] present a model checking approach supporting two techniques; one method that generates sequential C programs from SystemC code where SystemC threads and its scheduler are captured as functions and safety properties are checked as assertions, and the other is a hybrid technique where explicit-state modeling is used to capture the behaviors of the SystemC scheduler and SystemC threads are modeled symbolically.

Herber *et al.* [47] propose a toolset based on an automatic transformation of a SytemC TLM program into a semantically equivalent timed automata model. They use this transformation to test a set of safety and liveness properties. However, this work does not consider faults and their

impacts on the model. Their toolset, called *STATE* (SystemC to Timed Automata Transformation Engine), takes a SystemC TLM program as an input and transforms it into formally equivalent UPPAAL timed automata as the output. Hence, it is possible to verify safety, liveness, and timing properties of the given SystemC TLM program using UPPAAL model checker. This toolset handles all SystemC elements such as processes, interaction between them, dynamic sensitivity, and timing behavior. In the transformation, each method is mapped to a single automaton and interactions between processes are modeled by channels. As a front-end, STATE uses *KaSCPar* (Karlsruhe SystemC Parser) [52] that gets a SystemC TLM program and generates an Abstract Syntax Tree (AST) in XML. The back-end of STATE utilizes the AST and generates UPPAAL timed automata in XML. The transformation preserves the behavioral semantics and the structure of a given SystemC design. In particular, it captures the semantics of the TLM core interfaces, including the payload event queue (PEQ).

3.5 Summary

There has been a considerable amount of work in the area of formal verification of SystemC designs. However, all of the presented approaches have their limitations. They are either restricted to subsets of SystemC that preclude them from the application during the whole design process, or they lack formal foundation, or they require a lot of manual effort. To the best of our knowledge, a comprehensive co-verification framework that supports fully automatic verification techniques and yields a high degree of reliability due to the use of formal methods does not exist. With our approach, we provide such a framework.

Chapter 4

Formal Semantics and Model Extraction

In this chapter, first, we introduce the requirements of developing a formal semantics in Section 4.1. These requirements need to capture the essentials of the SystemC TLM models while still benefiting from automation techniques for verification and/or synthesis. Then, in Section 4.2, we discuss the essentials of transformation rules. Considering these requirements, we propose a set of transformation rules in Sections 4.3 to transform a SystemC TLM program into a Promela model. Thereafter, in Section 4.6, we propose a set of rules to transform a Loosely-Timed (LT) SystemC TLM program into a timed automata model. These rules help us extract a timed automata model which is simplified. We also introduce a tool-set, called STATE (SystemC to Timed Automata Transformation Engine), in Section 4.9 that we use to extract timed automata models from Approximately Timed (AT) SystemC TLM programs. To illustrate our model extraction ideas, we apply them on five case studies and explain each of them in detail in Sections 4.4, 4.5, 4.7, 4.8, and 4.10.

4.1 Developing a Formal Semantics

Since verification and synthesis of SystemC programs is in general undecidable, we need their abstract representations that enable automated analysis and revision. To utilize such an abstract representation, we need to develop a formal semantics of SystemC TLM programs. The objectives of this formal semantics are (1) enabling derivation of formal models of SystemC TLM programs

at different levels of abstraction and in the presence of different types of faults, and (2) analysis and revision in the presence of faults. It is desirable that this formal semantics satisfies the following requirements:

- Preserve the structure/architecture of the SystemC TLM program.
- Articulate different communication characteristics in the SystemC TLM program.
- Express different coding styles in the SystemC TLM program.
- Permit efficient analysis with available tool-chains.

Of these, the first requirement is motivated by the fact that transaction based modeling assists in simplifying the design of SystemC programs. We intend to preserve this property while developing the formal semantics. This will also simplify the task of synthesis where we intend to obtain the corresponding fault-tolerant SystemC TLM program.

The second requirement is motivated by the fact that the semantics should be expressive enough to articulate different communication characteristics. Since transactions are central to the model based design methodology, communication characteristics among modules are separated from the details of the implementation of functional units. This separation encapsulates low-level details, e.g., bus models, of the information exchange.

Regarding the third requirement, our focus is on AT (Approximately-Timed) and LT (Loosely-Timed) models. The AT model is suitable for performance modeling and architecture exploration while the LT model is mainly used when designers need fast simulation of a program with little care about timing concerns. The AT model allows us to keep the processes in the given model synchronized to a common clock. In this model, each process runs at a specific time and this time corresponds to the actual time when the corresponding activity is scheduled to occur in the

real system. This creates future events that are created in response to the event that is currently processed. Also, generally, this model utilizes nonblocking transport interface nb_transport() for maximal concurrency. By contrast, the LT coding style allows processes to run as fast as possible with some fairness for all initiators so that they can perform their transactions. This allows for temporal decoupling. Even though Loosely-Timed, this model allows time as an interrupt. Moreover, generally, this coding style utilizes blocking transport interface b_transport() for simplifying system execution.

Another important issue for developing the formal semantics is that it restricts the set of verification back-ends that can be applied. These back-ends can be either in the category of symbolic model checkers like Lustre [42], SMV [18] or Esterel [13] tool-chains, or one of the timed-model checkers like IF [16] or UPPAAL [56], or an explicit-state model checker like SPIN [48].

4.2 Extracting the Formal Semantics from the SystemC TLM Program

In order to extract an abstract formal model from the SystemC TLM program, we need a set of transformation rules. This formal model facilitates verification of the semantic properties developed in Section 4.1. For this purpose, we build on the ideas from [68], where each SystemC TLM program has three basic processes, *Behavior*, *Initiator* and *Target*. The behavior process captures the main functionality of the TLM module. An initiator and a target process participate in transactions. The execution of the TLM program switches between these processes. The extracted model would preserve this structure of the TLM program either with rules that model the scheduler explicitly or implicitly. With explicit scheduler, the scheduler is modeled as a special process whereas with implicit scheduler, the scheduling decisions would be hardcoded in the extracted

model. While the former is more general and would be useful in the context of faults that affect the scheduler, the latter would be useful in improving performance of verification and/or repair of the SystemC TLM program.

Our model extraction focuses on both the LT coding style and the AT coding style. Since the SystemC simulation kernel has a run-to-completion scheduling policy, a thread/process cannot be interrupted until it is either terminated or waits for an event. We use this policy to build up our model extraction while having LT coding style by modeling the scheduler implicitly. An implicit scheduler is sufficient for the case where the given SystemC TLM program contains a small number of processes, does not utilize features such as timed notifications, and the scheduling decisions are simple. For complex algorithms, the scheduler is modeled explicitly. Note that this explicit scheduler captures the scheduling policy and does not directly depend upon the input program. Hence, our models would provide typical schedulers, e.g., first-come-first-serve scheduler, priority based scheduler, used in practice. On the other hand, for AT style it is more desirable to model the scheduler explicitly, since timing issues are important to consider in program execution. Typically, encoding the scheduler explicitly induces additional communications between processes, compared to the original SystemC semantics. This can lead us to additional communications that may prevent verification tools to perform powerful optimizations.

Next, we present two sets of transformation rules. The first set of rules is used for generating Promela models from SystemC TLM programs written based on the TLM base protocol for interoperability. The second set of rules is used to generate UPPAAL timed automata from the SystemC programs.

4.3 Transformation Rules for Generating Promela Models

Our objective in this section is to define a set of rules to transform the processes explained in Section 4.2, i.e., Behavior, Initiator, and Target, to Promela. We specify a transformation rule as X | -- Y, where X is a SystemC construct, and Y is a Promela code snippet. The initiator and the target sockets have to be declared and constructed explicitly. The following rule, **Rule 1**, enables the transformation of sockets:

```
Rule 1:
tlm_utils::simple_initiator_socket<Initiator> socket
     |--
chan simple_initiator_socket = [0] of {mtype, trans}
```

Notice that we model the sockets as synchronous channels in Promela, since the transmission of a transaction from an initiator to a target can be conceived as an access to a memory-mapped bus system. This is done synchronously and need not to be buffered. **Rule 2** transforms the declaration of a SystemC thread to a *proctype* in Promela as follows:

```
Rule 2:
SC_THREAD(thread_process)  |--  proctype thread_process()
```

Moreover, the generic payload has a standard set of bus attributes: *command*, *address*, *data*, *byte enables*, *streaming width*, and *response status*. After setting the attributes, the generic payload is passed through the sockets between the initiator and the target. **Rule 3** transforms the declaration of the generic payload in a SystemC TLM program to Promela. Note that in the SystemC part of this rule, *trans* is a pointer of type $tlm\_generic\_payload$.

```
Rule 3:
tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
```

```
    |--
 typedef trans {  tlm_command cmd;
                  int address;
                  int data_ptr;
                  int data_length;
                  int streaming_width;
                  byte byte_enable_ptr;
                  bool dmi_allowed;
                  tlm_response response_status
    };
```

On the right-hand side, *trans* is defined as a structure in Promela, where the address attribute is the lowest address value from/to which data is to be read or written, the data_ptr attribute points to a data buffer within the initiator, the data_length attribute gives the length of the data array in bytes, the streaming_width attribute specifies the width of a streaming burst where the address repeats itself, and the set_dmi_allowed method is used to indicate to the initiator that it can use the direct memory interface for data transfer. A TLM command is either a TLM read command or TLM write command or a TLM ignore command. Thus, we model it with the **Rule 4**, where each TLM command is defined as a structure in Promela:

```
Rule 4:
tlm::tlm_command cmd = static_cast<tlm::tlm_command>
    |--
typedef tlm_command { bit cmd[2]; };
```

Furthermore, a TLM response could have seven different values. To cover all these seven values in the transformed program, we present **Rule 5** that defines a structure for the TLM response. We

encode these values in the three bits of the response array.

```
Rule 5:

TLM_OK_response = 1,

TLM_incomplete_response = 0,

TLM_generic_erro_response = -1,

TLM_arrdess_error_response = -2,

TLM_command_error_Response = -3,

TLM_burst_error_response = -4,

TLM_byre_enable_error_response = -5

    |--

typedef tlm_response { bit response[3]; };
```

Rule 6 transforms a forward submission of a transaction to a message transmission in a socket channel in Promela.

```
Rule 6:

socket->b_transport( *trans, delay )  |--  simple_initiator_socket!t;
```

t is a transaction of type trans that is sent to the channel simle_initiator_socket.

4.4 Case Study 1: Extracting Promela Model

In this section, we extract the Promela model of the SystemC TLM program explained in Figure 2.4. The extracted Promela model M includes two proctypes named Initiator and Target (see Figure 4.1). Moreover, we consider a separate proctype to model incModEight. To enable communication between the Initiator and the Target modules in the model M, we declare a synchronous channel tgtIfPort (see Figure 4.1). To start a transaction, the Initiator sends the message

startTrans to the Target via tgtIfPort channel and waits until the Target signals the end of the transaction with a message endTrans. The Promela code in Figure 4.1 captures the specification of channels and the Initiator, Target and incModEight proctypes. The incModEight proctype models the *Behavior* process of the Target.

The mtype in Figure 4.1 defines an enumeration on the types of messages that can be exchanged in the synchronous communication channels if2TgtBeh and tgtIfPort. The Initiator and the Target are connected by the channel tgtIfPort and the Target is connected to its Behavior proctype (i.e., incModEight) via the channel if2TgtBeh. Initially, the Initiator sends a startTrans message to the Target. Upon receiving startTrans, the Target sends the message inc to incModEight to increment the value of d modulo 8. The incModEight proctype sends incComplt to Target after incrementing d. Correspondingly, the Target proctype sends a endTrans back to the Initiator indicating the end of the transaction.

4.4.1 Capturing the execution semantics of the simulation kernel

Note that, we have not explicitly modeled the scheduler and the way it would run this program has been implicitly captured by the way we model the *wait()* statement. Since the simulation kernel has a run-to-completion scheduling policy, a thread/process cannot be interrupted until it is either terminated or waits for an event. There are two threads in the program of Figure 2.4: one that is associated with the method initiate() of the initiator class (see Line 11 in Figure 2.4) and the other implements the body of the incModEight() method of the target class (see Line 24 in Figure 2.4). The first statement of the incModEight() method is a wait() statement on a delta notification event because in Line 27 of Figure 2.4 the notify() method is invoked on the SC_ZERO_TIME event. Thus, initially only the initiator thread can execute, which includes an invocation of the trigger() method of the target class via a port in the initiator (see Line 14 in Figure 2.4). Afterwards,

```
1  mtype = {inc, incComplt, startTrans, endTrans} // Message types
2  chan if2TgtBeh = [0] of {mtype} // Declare a synchronous channel
3  chan tgtIfPort = [0] of {mtype}
4  byte d =0;
5  int cnt = 0;  // used to model the occurrence of faults
6
7  active proctype Initiator(){
8     byte recv;
9     waiting:  tgtIfPort!startTrans;
10              tgtIfPort?recv;
11              initRecv = recv;  // initRecv is used to specify
12                                // desired requirements
13    ending:   (recv == endTrans) -> fin: skip;
14  }
15
16 active proctype Target(){
17    byte recv;
18    waiting:  tgtIfPort?recv;
19              tgtRecv = recv;  // tgtRecv is used to specify
20                               // desired requirements
21    starting: (recv == startTrans) -> if2TgtBeh!inc;
22              if2TgtBeh?recv;
23              (recv == incComplt) -> tgtIfPort!endTrans;  }
24
25 active proctype IncModEight(){  // Models the Behavior process
26                               // of the Target
27    byte recv;
28    waiting:  if2TgtBeh?recv;
29              (recv == inc) ->  d = (d + 1) % 8;
30              if2TgtBeh!incComplt;  }
```

Figure 4.1: The extracted functional model.

the initiator thread terminates. The simulation kernel context switches the Target at the end of the current simulation cycle upon the occurrence of delta notification. We have captured this semantics using the synchronous channels in the Promela model. That is why we do not explicitly have a proctype for modeling the behaviors of the simulation kernel. Of course, this does not mean that such an approach would work for all SystemC programs. For example, in models where processes are triggered by time-outs, we need to explicitly model the behaviors of the scheduler in the Timed Notification phase when sensitive processes are added to the set of runnable processes.

4.4.2 Property Specification and Functional Correctness

In order to ensure that the extracted model correctly captures the requirements of the SystemC program, we define a set of macros that we use to specify desired requirements/properties. We only consider the requirements related to the communication between the Initiator and the Target. The SystemC program of Figure 2.4 has two types of requirements. First, once the Initiator starts a transaction, then that transaction should eventually be completed. Second, it is always the case that if the Initiator receives a message from the Target after instantiating a transaction, then that message is an endTrans message. Moreover, if the Target receives a message, then that is a startTrans message. Since the second requirement should always be true in the absence of faults, it defines an *invariant* condition on the transaction between the Initiator and the Target (denoted by the inv macro below). To formally specify and verify these properties in SPIN [48], we first define the following macros in the extracted Promela model.

```
#define  strtTr     initiator@waiting
#define  endTr      initiator@fin
#define  finish     (initRecv == endTrans)
#define  start      (tgtRecv == startTrans)
#define  initEnd    initiator@ending
#define  tgtStart   targetTrigger@starting
#define  inv    ((!initEnd || finish)&&(!tgtStart || start))
```

The macro strtTr is true *if and only if* the control of execution of the Initiator is at the label waiting (see Figure 4.1). Likewise, the macro endTr captures states where the Initiator is at the label fin. Using these two macros, we specify the first requirement as the temporal logic expression $\Box$(strtTr $\Rightarrow$ $\Diamond$ endTr), which means it is always the case (denoted by $\Box$) that if the Initiator is waiting (i.e., has started a transaction), then it will eventually (denoted by $\Diamond$) reach the label fin

(see Line 13 in Figure 4.1); i.e., finish the transaction. We specify the invariant property as the expression □ inv. This property requires that inv is always true (in the absence of faults). Using SPIN, we have verified the above properties for the extracted model of Figure 4.1.

4.5 Case Study 2: Extracting Promela Model using Transformation Rules

In order to extract a Promela model from the SystemC TLM program of Section 2.2 (Figures 2.5 and 2.6), we use the same ideas explained in Section 4.4. Moreover, we use the set of transformation rules of Section 4.3, which helps us to model interoperability. The extracted Promela model M has two proctypes named Initiator (Lines 16-44 in Figure 4.2) and Memory (Lines 46-73 in Figure 4.3). To enable communication between the Initiator and the Memory in the model M, we use **Rule 1** of the transformation rules to declare a synchronous channel simple_initiator_socket (see Figure 4.2). The binding of the initiator and the target sockets in Line 72 of Figure 2.6 is captured as a channel in the Promela model (Line 12 in Figure 4.2). As a result, in the Promela model, we do not explicitly generate anything corresponding to the target socket in the Memory module.

We model the actual memory by an array of bytes in Line 14 of Figure 4.2. Using **Rule 3**, the initiator creates a transaction by setting the attributes of the generic payload (Lines 21-27 in Figure 4.2). Note that, data_ptr is a pointer in the SystemC program, whereas in the Promela model we treat it as the actual data that should be read/written. Since we cannot use pointers in Promela (unless we use c_code blocks which complicates the model), we use data_ptr as the actual data. From the point of view of modeling, if we could model pointers in Promela, all we would do was to access the memory contents. That is why we treat data_ptr as the actual data value. After

setting the attributes of the generic payload, the Initiator sends the message initT to the Memory via simple_initiator_socket channel (Line 28 in Figure 4.2) and waits until the Memory signals the end of the transaction with a message that contains a generic payload with response_status attribute being equal to 1. Consider that, to send the message initT via the simple_initiator_socket channel, we use **Rule 6** to transform the *b_transport* method of the TLM program to model M.

The Initiator module in Figure 4.2 defines a random value, either 0 or 1, for cmd attribute to send the message initT (Lines 22-23 in Figure 4.2). In order to declare the cmd attribute in the Promela model M, we use **Rule 4** of the set of transformation rules. When the cmd is 0, the Initiator is requesting a *read* command. Thus, the Memory module, after checking address range and unsupported features, returns the contents of that address in memory (Line 64 in Figure 4.3). When the cmd attribute of the message memT equals 1, the Initiator is requesting a *write* command. Thus, the Memory writes the value of data_ptr attribute into the memory cell whose address equals the address attribute of the received message memT (Line 65 in Figure 4.3). After reading from/writing to the memory successfully, the Memory module sets the response_statuse attribute of memT message to 1 (Line 68 in Figure 4.3). This means, according to the transformation **Rule 5**, the response_status attribute equals *OK*. Finally, the Initiator, after receiving *OK* response, completes the current transaction (Lines 41-44 in Figure 4.2).

```
typedef tlm_response { bit response[3]; };
typedef tlm_command { bit cmd[2]; };
typedef trans { tlm_command cmd;
                int address;
                int data_ptr;
                int data_length;
                int streaming_width;
                byte byte_enable_ptr;
                bool dmi_allowed;
                tlm_response response_status
                };
chan simple_initiator_socket = [0] of trans;
int cnt =0;  // used to model the occurrence of faults
trans initT; trans memT; byte mem[255];

active proctype Initiator(){
tlm_response res;
trans recv;
int sentData;  int recvData;
waiting:
  if
    :: initT.cmd = 0; initT.data_ptr = 0; // read
    :: initT.cmd = 1; initT.data_ptr= 0; sentData=initT.data_ptr; // write
  fi;
  initT.address = 0;  initT.data_length = 4;
  initT.streaming_width = 4;  initT.byte_enable_ptr = 0;
  initT.dmi_allowed = false;  initT.response_status = 0;
  simple_initiator_socket!initT;  simple_initiator_socket?recv;
ending:
  if
    :: (recv.response_status == -2) ->
        atomic{ printf("response_status is:", recv.response_status);
        goto waiting;}
    :: else -> skip;
  fi;
  if
    :: (initT.cmd == 0) ->  recvData = recv.data_ptr;
    :: else -> skip;
  fi;
transComplete:
  if
    :: (recv.response_status == 1) -> fin: skip;
    :: else -> skip;
  fi; }
```

Figure 4.2: The Initiator module of the extracted functional model.

```
active proctype Memory(){
int data;
bool faultsOccur = false;

waiting:
        simple_initiator_socket?memT;

starting:
        if
           :: ((memT.address >= 256) || (memT.byte_enable_ptr != 0)
           || (memT.data_length > 4)|| (memT.streaming_width < 4))
           ->   atomic { memT.response_status = -2;
                simple_initiator_socket!memT;
                goto waiting; }
           :: else -> skip;
         fi;

        if
           :: (memT.cmd == 0) ->  memT.data_ptr = mem[memT.address] ;
           :: (memT.cmd == 1) -> mem[memT.address] = memT.data_ptr;
        fi;

        memT.response_status = 1;
        data = mem[memT.address];

        simple_initiator_socket!memT;

}
```

Figure 4.3: The Memory module of the extracted functional model.

4.5.1 Property Specification and Functional Correctness.

To ensure that the extracted model M captures the requirements/properties of the TLM program, we specify the requirements that should hold in the absence of faults. For this purpose, we define a macro that captures the requirements related to the read and write actions in the Memory module.

In Figures 2.5 and 2.6, if the Initiator receives a message from the Memory after requesting a write command, the sent data of the Initiator must be the same as the written data in the Memory when the transaction is complete. In addition, after sending a read command, the data_ptr in that message should be equal to the read value in the Memory. We consider this property as an

invariant, since it should be always true in the absence of faults. We define the following macro and verify it in SPIN [48].

```
#define inv1 ((!(Initiator@transComplete && (initT.cmd == 1))
                || (Initiator:sentData == Memory:data))
             &&
              (!(Initiator@transComplete && (initT.cmd == 0))
                || (Initiator:recvData == Memory:data)))
```

This invariant represents that when the execution is at the complete label of the Initiator and we have a write action, the sentData of the Initiator and written data in the Memory are equal. Likewise, when we have a read action, the recvData in the Initiator and the read data from Memory are the same at the complete label in the Initiator. We specify the invariant property as the expression □ inv1. This property requires that inv1 is always true (in the absence of faults). Using SPIN, we have model checked the above properties for the extracted model of Figures 4.2 and 4.3.

4.6 Transformation Rules for Generating UPPAAL Timed Automata

In order to extract the UPPAAL timed automata in Loosely-Timed coding style programs using b_transport interface, we utilize the rules from Section 4.3, where a set of transformation rules is proposed. However, these rules generate untimed Promela models from untimed SystemC TLM programs, while we need to extract UPPAAL timed automata from timed TLM programs. Hence, we utilize the following rules similar to those in Section 4.3, where timed and untimed models are

the same:

```
Rule 1:   tlm_utils::simple_initiator_socket<Initiator> socket;
          |--
          chan simple_initiator_socket[0];


Rule 2:   tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
          |--
          typedef struct { tlm_cmd cmd;
                 int address;  int data_ptr;
                 int data_length;  int streaming_width;
                 int byte_enable_ptr;  bool dmi_allowed;
                 tlm_response response_status;
          }trans;


Rule 3: tlm::tlm_command cmd = static_cast<tlm::tlm_command>
        |--
        typedef { bit cmd[2]; }tlm_command;
```

where in a transformation rule $X|--Y$, X is a SystemC TLM construct, and Y is a UPPAAL code snippet. **Rule 1** enables the transformation of sockets. **Rule 2** transforms the declaration of the generic payload in a SystemC TLM program to UPPAAL, and **Rule 3** transforms the declaration of a TLM command.

In order to study timing behaviors of SystemC TLM programs, we also need to consider timing constraints of the programs. Hence, we introduce new rules that consider these timing constraints. **Rule 4** and **Rule 5** are introduced for extracting the UPPAAL model in LT coding style, i.e., with

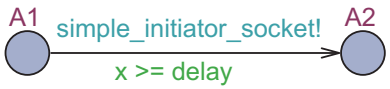

Figure 4.4: Transforming b_transport interface into UPPAAL, the Initiator.

Figure 4.5: Transforming b_transport interface into UPPAAL, the Target.

b_transport interface. In a transaction using b_transport interface, we need to consider the timing since one of the sending arguments is *delay*. If the transaction is from the Initiator to the Target, this argument describes the point of time in the future where the communication actually starts, and illustrates the ending time of the communication in the response. To transform a b_transport interface into UPPAAL, we define **Rule 4**, where a synchronization channel *simple_initiator_socket* is used (in Figures 4.4 and 4.5) to synchronize the Initiator and Target of a transaction. Figures 4.4 and 4.5 illustrate the Initiator and Target of the transaction respectively. The communication starts at *delay* timing point. Hence, the Initiator does not synchronize with the Target before that time. We take this constraint into account by defining the guard $x >= delay$ at the Initiator, where x shows the current simulation time *sc_time_stamp()*. In addition, the Target accepts the synchronization after the *delay* timing point. We consider this constraint by defining the guard $x >= delay$ at the Target. It means the communication starts after *delay* timing point.

```
Rule 4:    socket->b_transport( *trans, delay )
            |--
            Figures 4.4 and 4.5
```

Moreover, in order to guarantee deterministic execution and increase the timing accuracy, a SystemC TLM program that uses Loosely-Timed coding style benefits from explicit synchronization points by utilizing calls to *wait()* function. This function is a synchronization-on-demand

method that yields the control to the SystemC scheduler. To transform this function into the UPPAAL model, we define a variable *global-clock* that plays the role of the global time in a SystemC TLM program. To transform the wait function into the UPPAAL model, we define **Rule 5** as follows:

```
Rule 5:  wait(x)
         |--
         global-clock = global-clock + x;   x = 0;
```

Hence, when the wait function is called, we add the *delay* arguments obtained from Rule 4 to the global-clock variable and reset the local clock x.

4.7 Case Study 3: Extracting UPPAAL Model using Transformation Rules

In this example, we extend the example explained in Section 2.2 to have three modules: Initiator, Router, and Memory. The Initiator module generates a transaction while the Memory (target) module represents a random access memory. The Router is a TLM 2.0 interconnect component that is placed between the Initiator and the Memory. An interconnect component is a component that forwards transactions from an incoming target socket to an outgoing initiator socket (see Figure 4.6 for the architecture). In this example, the initiator socket of the Initiator module is bound to the target socket of the Router and the initiator socket of the Router is bound to the target socket of the Memory module. The Initiator has a thread process (similar to that in Figure 2.5) that starts the communication by sending a generic payload along with a delay parameter to the Router. The Router utilizes its target socket and the b_transport interface to receive the transaction from

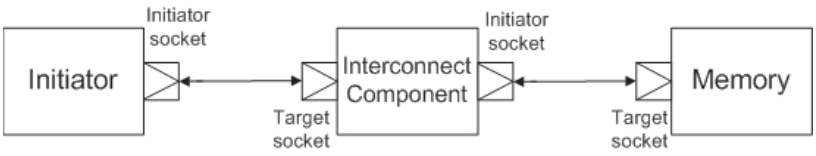

Figure 4.6: The architecture of the memory-mapped busses model.

the Initiator. After decoding the address, if it is needed, the Router forwards the transaction to the Memory. Finally, the Memory uses the b_transport interface to receive the transaction from the Router. Based on the Initiator's request, the Memory either reads from or writes to the data attribute of the generic payload and sends back the corresponding response.

We use the rules and approaches explained in Section 4.6 to extract the UPPAAL timed automata model. Next, in Figures 4.7, 4.8, and 4.9 we identify the fault-free version of the program that forms the basis of models generated for timing faults. In extracted UPPAAL models, the green texts show either the *guards* or *synchronization*, the blue texts show the *updates*, and the pink texts represent the *names* and *invariants*. Figure 4.7 represents the Initiator model with the starting state L1. We utilize *Rules 2* and *3* to transfer the generic payload and TLM command (read/write) into the UPPAAL timed automata. Hence, States L2 and L3 represent places where the cmd attribute shows write and read commands respectively. After generating the transaction, we use *Rule 1* to define a synchronization channel sendSocket between the Initiator and Router, and use *Rule 4* to transfer b_transport interface into the UPPAAL model. As a result, in Figure 4.7, the Initiator synchronizes itself with the Router module and changes its state from either L2 or L3 to L4. The Initiator cannot send the transaction later than delay1 timing point, since the communication actually starts at that point. In state L6, the Router receives the transaction and changes its state to L7. Then, the Router synchronizes with the Memory via targetSocket channel and changes its state to L8. Note that the transaction is not actually sent from the channel, since the channels in UPPAAL are only for synchronization purposes. Then, we use *Rules 1* and *4* to transfer the b_transport

interface and timing constraints in the Memory. If the guard $x >= delay$ is true, the Memory receives the transaction and changes its state to L11. The Memory cannot send its response later than $delay1 + delay2$, where $delay2$ shows the delay argument of the response message to the Initiator. After executing the write or read action, if there is no error, the Memory changes its state to either L12 or L13, and synchronizes itself with the Router via targetSocket channel. The Router, receives the response from the Memory, and, using the sendSocket channel, sends the response. Finally, the Initiator receives the response and can initiate another transaction.

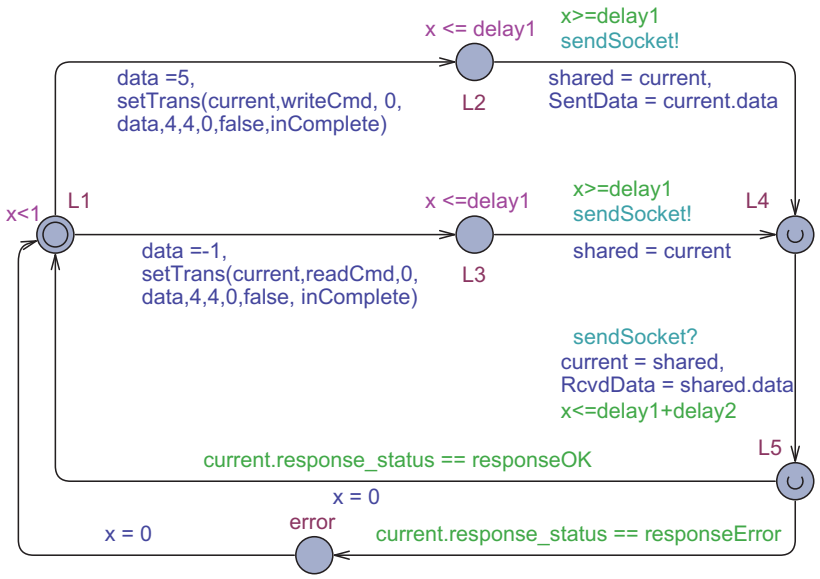

Figure 4.7: Fault-intolerant UPPAAL timed automata model of the Initiator module.

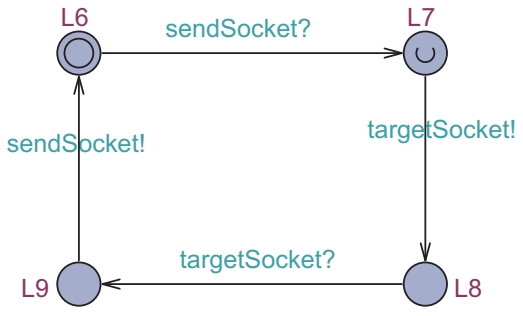

Figure 4.8: Fault-intolerant UPPAAL timed automata model of the Router module.

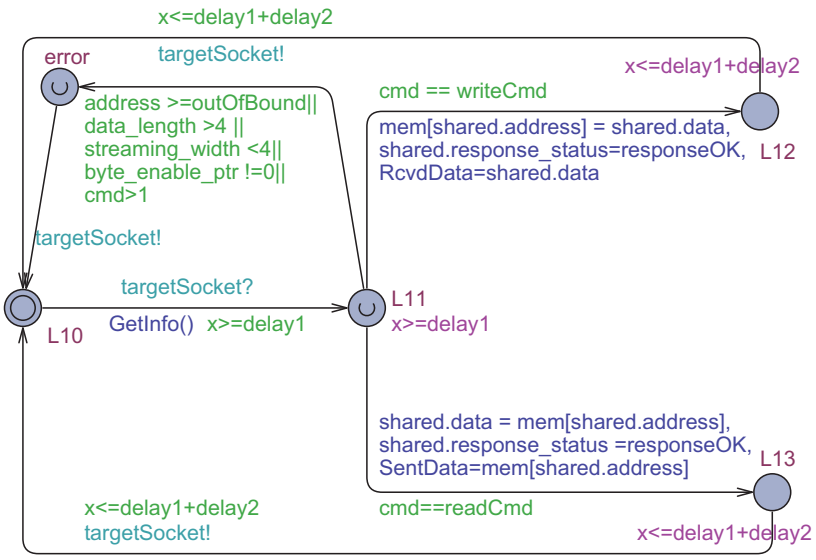

Figure 4.9: Fault-intolerant UPPAAL timed automata model of the Memory module.

4.7.1 Property Specification and Functional Correctness

To ensure that the extracted timed automata model captures the requirements/properties of the SystemC TLM program, we specify the requirements that should hold in the absence of faults. For this purpose, we define the following specifications in a subset of CTL (*Computational Tree Logic*) in Figure 4.10 that captures the requirements related to the timing constraints. These requirements should be always true in the absence of faults.

```
SPEC 1: A[] not deadlock
SPEC 2: Init.L1 --> (Init.L4 and x >= delay1)
SPEC 3: Init.L1 --> (Memory.L11 and x >= delay1)
SPEC 4: (Memory.L12 or Memory.L13) --> (Init.L5 and x <= delay1 + delay2)
SPEC 5: Init.L1 --> (Init.L5 and x <= delay1 + delay2)
```

Figure 4.10: Requirements of Memory Bus System using LT coding style.

The first requirement SPEC 1 represents that the extracted model is free of deadlock. The second requirement SPEC 2 illustrates that the Initiator module will not send the transaction towards the Memory module before the timing point delay1. The SPEC 3 shows that if the Initiator sends a transaction, the Memory will not receive it before the timing point delay1. Also

if the Memory sends the response to the Initiator, it should not be received after the timing point delay1 + delay2. This requirement is represented in SPEC 4. Finally, the SPEC 5 shows that if the Initiator starts a transaction, it should not be finished after the timing point delay1 + delay2. Using UPPAAL, we have model checked the above properties for the extracted model of Figures 4.7, 4.8, and 4.9.

4.8 Case Study 4: Extracting UPPAAL Model of a NoC Switch

In this example, we extend the example explained in Section 2.2 to model a Network on Chip switch. We assume that the switch has eight processing cores that communicate using a router.

To model this switch, we assume we have memory mapped busses with four Initiators and four Targets and a Router as an interconnect component between the Initiators and Targets (See Figure 4.11). Each Initiator module generates a transaction and sends it to one of the Target modules through the Router using b_transport interface. The Router receives a transaction, decodes the address attribute in the transaction, and forwards it to the appropriate Target using the decoded address. The Router also needs to manage the return path from the Targets to the Initiators. In other words, the Router is a component that forwards transactions from an incoming target socket to an outgoing initiator socket. In this example, there are four incoming target sockets connected to four instances of the Initiator, and four outgoing initiator sockets connected to four instances of the Target. Sockets belonging to the Initiators are bound to the target sockets of the Router, and each of the four initiator sockets belonging to the Router is bound to a socket belonging to a different Target. Each initiator-to-target socket connection is point-to-point.

We use the set of rules in 4.6 to extract the UPPAAL timed automata model from the SystemC TLM model. Next, in Figures 4.12 and 4.13, we identify the fault-free version of this model

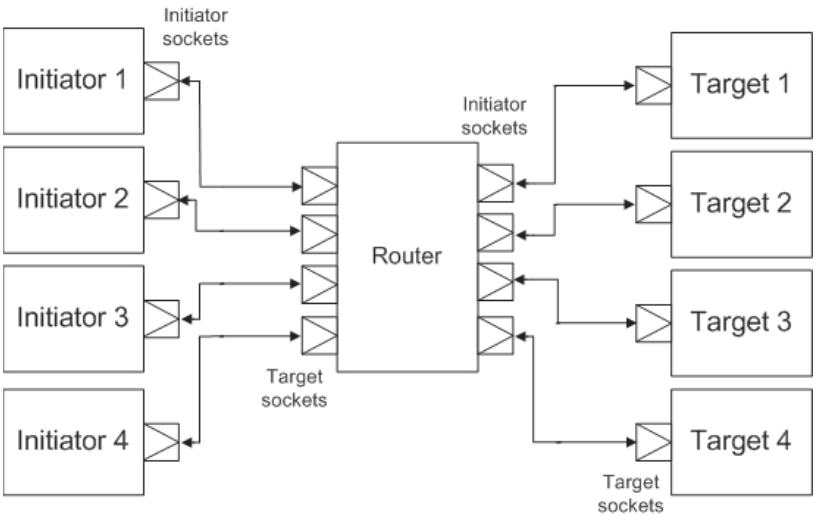

Figure 4.11: Using LT coding style to model NoC switch.

that form the basis of models generated for different types of faults. The extracted models of the Initiators and Targets are the same as those in Section 4.5.

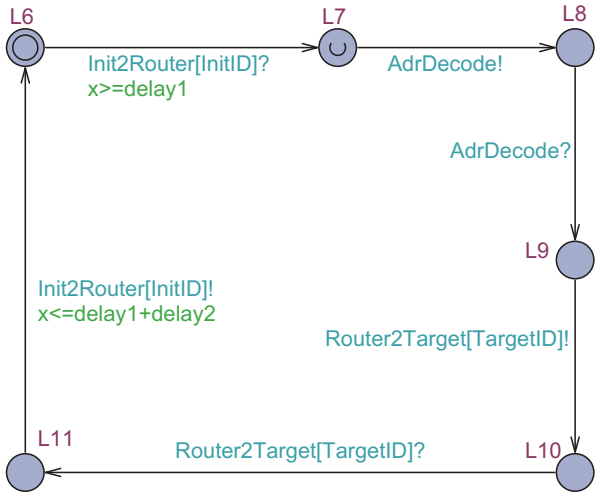

Figure 4.12: The Router module.

Figure 4.12 represents the Router automaton and Figure 4.13 shows the address decoding mechanism used in the Router module. The Router receives a transaction through one of the channels Init2Router and changes its state to L7. This transaction should not be received before delay1 timing point. Note that in the Router automaton, we cannot use the same channel to communicate with Initiators since their socket connections are point-to-point in the SystemC TLM

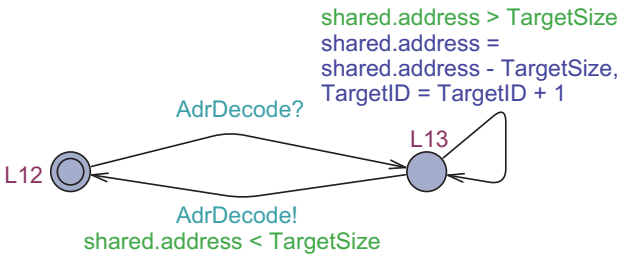

Figure 4.13: The address decoding mechanism.

model. After receiving the transaction, the Router decodes the address (Locations L12 and L13 in Figure 4.13), obtains the TargetID, and forwards the transaction to the appropriate Target. The Router then waits to receive the response of the Target from the same channel (L10) and sends it back to the appropriate Initiator (L11).

4.8.1 Property Specification and Functional Correctness

To ensure that the extracted model captures the requirements/properties of the SystemC TLM program, we specify the properties that should hold in the absence of faults. For this purpose, we define the following CTL (*Computational Tree Logic*) specifications in Figure 4.14.

```
SPEC 1: A[] not deadlock
SPEC 2: Init[id_i].CurrTrans.cmd == readCmd -->
        (Target[id_t].SentData == Router.RcvdData)
        and
        (Router.SentData == Init[id_i].RcvdData)
SPEC 3: Init[id_i].CurrTrans.cmd == writeCmd -->
        (Init[id_i].SentData == Router.RcvdData)
        and
        (Router.SentData == Target[id_t].RcvdData)
SPEC 4: Init[id_i].L1 --> (Init[id_i].L2) or (Init[id_i].L3)
SPEC 5: (Init[id_i].L2) or (Init[id_i].L3) --> Init[id_i].L1
```

Figure 4.14: Properties of the extracted UPPAAL timed automata.

The correctness of requirement SPEC 1 in Figure 4.14 implies that in all paths of the extracted automata model, we do not have any deadlock. The requirements SPEC 2 and SPEC 3 repre-

sent that the communicated data between the Initiator and Router, and the Router and Target are the same in the absence of faults. The SPEC 4 and SPEC 5 show that the Initiator will eventually generate a transaction either with a write request (Location L2) or a read request (Location L3), and will eventually come back to the initial state to generate another transaction. These two requirements together imply that the Initiator module is not blocked. We can extend the set of requirements and define the same requirements as SPEC 4 and SPEC 5 for all modules in the extracted model. Using UPPAAL model checker, we have model checked the requirements of Figure 4.14.

4.9 Using STATE for Extracting Timed Automata Models

In order to consider concurrency in SystemC TLM programs, we need to utilize Approximately-Timed coding style. An Approximately-Timed model breaks down transactions into a number of phases corresponding much more closely to the phasing of particular hardware protocols (e.g., the address and data phases of an AHB (AMBA Advanced High-performance Bus) read or write). On the contrary, a Loosely-Timed model, for which we proposed our transformation rules in Section 4.6, utilizes transactions corresponding to a complete read or write across a bus or network in physical hardware. It provides timing at the level of the individual transaction. Also, in the model extraction proposed in Section 4.6, the SystemC scheduler has been modeled implicitly. In other words, the way the extracted model would run the SystemC program is implicitly captured by the way we model the *wait()* statement. Since the simulation kernel has a run-to-completion scheduling policy, a thread/process cannot be interrupted until it is either terminated or waits for an event. On the other hand, for Approximately-Timed models, we need to model the SystemC scheduler explicitly to be able to manage concurrent transactions. In addition, it is necessary to consider

other SystemC elements as well as TLM components, e.g. events, wait-notify mechanism, and Payload Event Queue (PEQ), in the model extraction. For this purpose, we use *STATE* (SystemC to Timed Automata Transformation Engine) tool-set [47] to transform a SystemC TLM program to its equivalent UPPAAL timed automata model. In the following, we, first, state a few assumptions that defines the subset of SystemC TLM programs supported by STATE. Then, we represent the timed automata templates that STATE extracts for different SystemC TLM elements. The materials in the following sections are mostly adapted from [47].

4.9.1 Assumptions

SystemC supports a very diverse set of models of computation. At the same time, as an extension of C++ , it inherits the full semantic scale of the C++ language. Together, this illustrates that SystemC is an outstandingly expressive languages. To make it nonetheless possible to transform SystemC designs into the more restricted UPPAAL modeling language. Therefore, to utilize STATE, we need to assume that a given SystemC design fulfills the following restrictions.

- UPPAAL supports no dynamic variable or process creation. Thus, dynamic object or process creation are also forbidden in the SystemC design, i.e., a static structure is required. This is a minor restriction because dynamic process creation is not a part of SystemC language definition and can only be used through the corresponding C++ functions. As a consequence, only instantiations and initializations are allowed in constructors and in the sc_main method.

- While SystemC allows hierarchical scopes, the possibility to define scopes is limited to global and local variables in UPPAAL. To avoid name conflicts, we assume that no variable is shadowed (i.e., each variable has a unique identifier in its scope). It is needed to assume that no overloading of methods is used. This assumption as well as the previous

assumption do not restrict the set of possible input designs but require some renaming and code duplication at the most.

- The UPPAAL modeling language only provides the data types int and bool. Most complex data types can be mapped into integers, but the use of pointers is generally impossible in UPPAAL. Thus, we need to assume that the SystemC design does not use any pointers. As a consequence, dynamic memory management is also excluded.

4.9.2 Representation of SystemC TLM Designs in UPPAAL

The general idea of representing a SystemC TLM program in UPPAAL timed automata is that each method is mapped to a single timed automata template. Process automata are used to encapsulate these methods and care for the interactions with events and the scheduler (see Figure 4.15). The scheduler is explicitly modeled and a predefined template is used for events and other SystemC constructs such as primitive channels. The interactions between the processes and the scheduler are modeled by two synchronization channels, *activate* and *deactivate*. The interactions between processes and event objects are modeled by *wait* and *notify*. The interactions between the event objects and the scheduler are used to synchronize their timing. The scheduler informs the event objects when a delta-cycle is completed to release delta-delay notifications, and conversely, the event objects inform the scheduler when time is advanced due to a timed notification.

In the following, we explain the timed automata templates that STATE generate for each of the SystemC structures in Figure 4.15. These automata are needed in an Approximately-Timed model to provide concurrency.

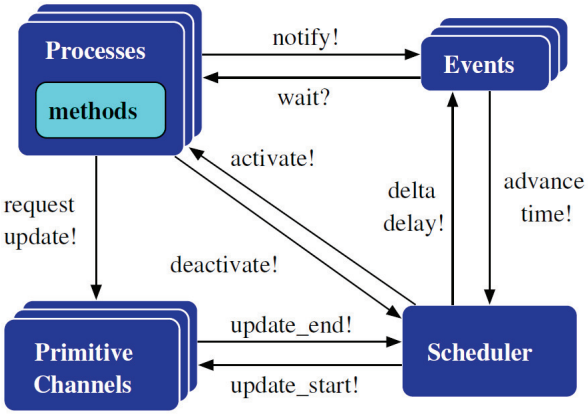

Figure 4.15: Representation of SystemC TLM Designs in UPPAAL.

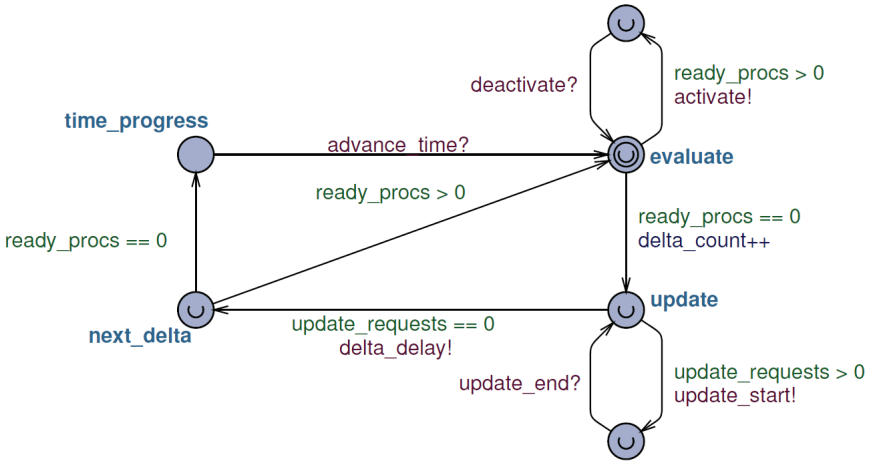

Figure 4.16: Timed automata modeling SystemC scheduler [47].

4.9.2.1 The Scheduler

The scheduler controls the execution of SystemC designs. The basic execution units are processes. The scheduler works in delta-cycles, i.e., in evaluate and update phases. In the evaluate phase, processes that are *ready to run* are executed in non-deterministic order. In the update phase, primitive channels are updated by taking over new values. If there are no more processes *ready to run* when a delta-cycle is finished, time is advanced to the next pending event

The timed automaton that STATE generates for the scheduler is shown in Figure 4.16. Ini-

tialization is implicit in UPPAAL, i.e., processes and methods are executed once before the main simulation loop. As a consequence, the scheduler starts in the evaluation phase depicted by the location evaluate. If there are any processes that are *ready to run*, the scheduler sends an activation event activate!. Processes that are *ready to run* receive this event and resume their execution. STATE uses a binary channel for the activation to ensure that only one process is executed at a time and that processes are executed in a non-deterministic order. To ensure that the scheduler sends the activation event once for each process that is *ready to run*, each process increments a counter ready_procs when triggered, and decrements the counter when suspending itself. When there are no more processes that are *ready to run*, i.e., ready_procs == 0, the scheduler starts the update phase by going to location update.

In the update phase, *update requests* are executed in non-deterministic order using the activation event update_start. Immediate notification is not allowed during the update phase. If there are no more update requests, the scheduler starts the next delta-cycle (see location next_delta in Figure 4.16). When leaving the update phase, the scheduler informs event objects with pending *delta-delay notifications* that a delta-cycle is finished by sending delta_delay!. If there are *delta-delay notifications*, the corresponding processes are immediately triggered and become *ready to run*. They will be executed in the next delta-cycle, which is started by the scheduler without time progress. If there are no processes triggered by *delta-delay notifications*, i.e., ready_procs == 0, simulation time must be advanced to the earliest pending timed notification.

There are two types of *timed notifications* in SystemC: events may be notified with a delay by calling e.notify(t), and processes may be delayed for a given time interval by calling wait(t). In SystemC, the timing behavior is completely managed by the scheduler. In the timed automaton, we have the possibility to wait locally for a given time. Therefore, it is more suitable to model time within processes and event objects. To wait for the earliest pending *timed notification* in the

scheduler, STATE lets the processes and events with timed behavior send a broadcast synchronization advance_time! when their delay expires. The scheduler receives advance_time? and starts a new *delta-cycle*, i.e., executes processes that became ready to run through the *timed notification*.

The timed automaton modeling the scheduler behaves exactly like the SystemC scheduler. The binary channels used to control process execution and channel updates guarantee that UPPAAL model checker considers every possible serialization. The locations used for the execution of delta-cycles are urgent and thus take no simulation time. It is ensured that no scheduling phase is started before the preceding phase is completed using the counters ready_procs and update_procs and committed locations in event notifications. The counters guarantee that pending executions are completed before the next phase is started. The use of committed locations in event notification (which is represented in Figure 4.17) ensures that event triggering is prioritized over state changes in the scheduler.

4.9.2.2 Events

If an event object e is notified by its owner, processes that are sensitive to the event resume execution. SystemC supports three types of event notifications. An *immediate notification*, invoked by e.notify(), causes processes to be triggered immediately in the current delta cycle. A *delta-delay notification*, invoked by e.notify(0), causes processes to be triggered at the same time instant, but after updating primitive channels, i.e., in the next delta-cycle. A *timed notification*, invoked by e.notify(t) with $t > 0$, causes processes to be triggered after a certain delay t. If an event is notified that already has a pending notification, only the notification with the earliest expiration time takes effect. That means that immediate notifications override all pending notifications, delta-delay notifications override timed notifications, and timed notifications override pending timed notifications if their delay expires earlier.

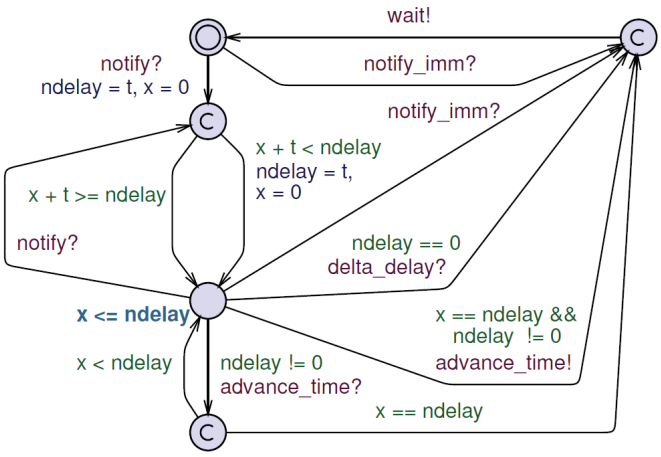

Figure 4.17: Timed automata template for an event object [47].

The modeling of event objects are represented in Figure 4.17 [47]. The timed automata template is instantiated for each event object declared in a given SystemC design. Its template parameters are the synchronization channels notify_imm, notify and wait, and the integer variable t. Initially, the event just waits to be notified. If it is immediately notified, it receives notify_imm?, and immediately sends wait! on a broadcast channel. If the event object is notified by a delta-delay or a timed notification, it receives notify? and copies the parameter t to a local variable ndelay, which yields the notification delay. At the same time, a local clock x is reset. The committed location that is now reached is used to reinitialize ndelay and to reset x if a subsequent delta-delay or timed notification overrides the notification delay. We then have to wait until:

- an immediate notification overrides the current pending notification,
- we receive delta_delay? from the scheduler if ndelay == 0, or
- the current delay expires, i.e., $x == ndelay \;\&\&\; ndelay! = 0$.

Subsequently, we send wait! and go back to the initial location. When a timed notification expires, we have to inform the scheduler to start the next evaluation phase by sending advance_time!.

Due to the use of a broadcast channel advance_time!, only the first advance_time is received by the scheduler if the delays of multiple events expire at the same time.

4.9.2.3 Processes

Processes are the basic execution unit in SystemC. Each process is associated with a method to be executed. There are two types of processes: *method processes* and *thread processes*. A *method process*, when triggered, always executes its method body from the beginning to the end. It is triggered by a set of events given in a static sensitivity list. The timed automata template STATE uses to wrap a method process is in Figure 4.18. It waits for any of the events from the sensitivity list by synchronizing on sensitive?. If one of the events from the sensitivity list occurs, it marks itself as *ready to run* by incrementing ready_procs and by waiting for the activate event. Then, it transfers control to its associated method. When the method returns, it deactivates itself by sending deactivate! to the scheduler and by decrementing ready_procs. Then, it returns to the initial position and waits until it is triggered by one of the events from the sensitivity list again.

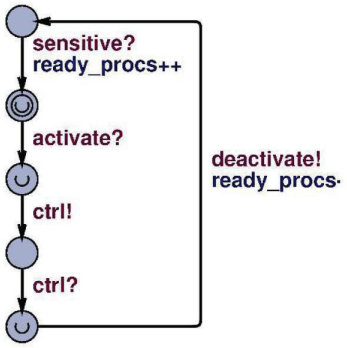

Figure 4.18: Method process template [47].

A *thread process* may suspend its execution and dynamically wait for events or a given time delay. It is triggered only once at the beginning of the simulation and runs autonomously from that time on. The timed automata template STATE uses to start a thread process is given in Figure 4.19.

It just waits to be activated, transfers control flow to its associated method and deactivates itself if the method returns. Note that the control transfer channel is a parameter of the process templates, and thus the same template can be instantiated for arbitrary many process declarations.

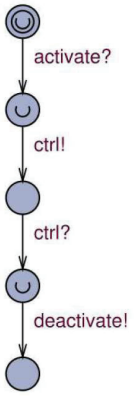

Figure 4.19: Thread process template [47].

4.9.2.4 Payload Event Queue (PEQ)

A PEQ is a time-ordered list of event notifications, where each notification is associated with a transaction object (i.e., a payload and a phase) and a delay. The actual delay of each event notification is calculated from the current simulation time and the annotated delay. The PEQ is connected to a callback method peq_cb, which is executed whenever a notification in the PEQ expires. A PEQ can be used by calling its notify method with a transaction object t and a delay d. This will cause the callback method associated with the PEQ to be executed with t in d time units.

STATE models the PEQ with four different timed automata, namely *timed-ordered list* (Figure 4.20), *interface* (Figure 4.21), *event fetch and callback invocation* (Figure 4.22), and *PEQ event* automata (Figure 4.23). The first automaton is an ordered-list where tuples of a payload, a delay and a phase can be stored and sorted by their delay expiration time. The interface automaton is called by initiators and targets to insert a new PEQ notification. The third automaton removes

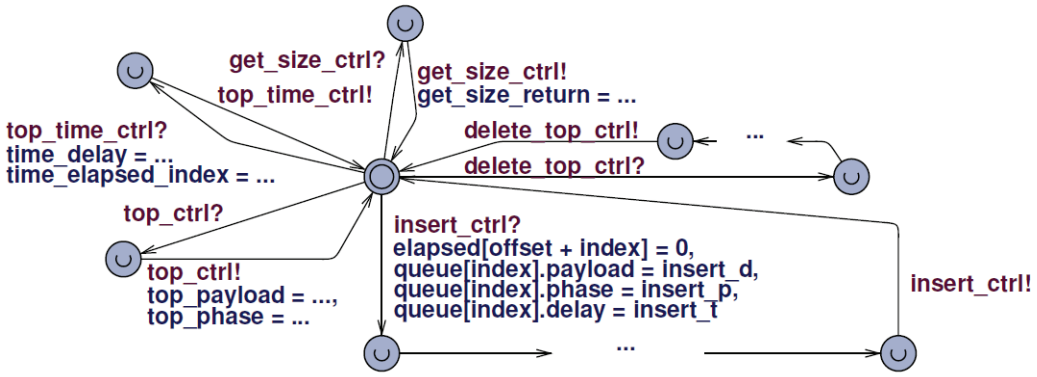

Figure 4.20: Timed automata template of the timed ordered list [47].

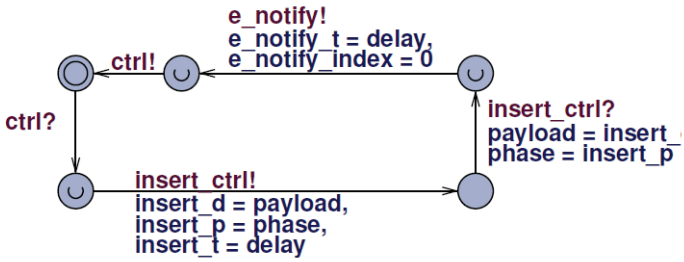

Figure 4.21: Timed automata template of PEQ interface method notify [47].

elements from the PEQ if their delay expires and invokes the callback method associated with the PEQ (peq_cb). The last automaton models the event object notification.

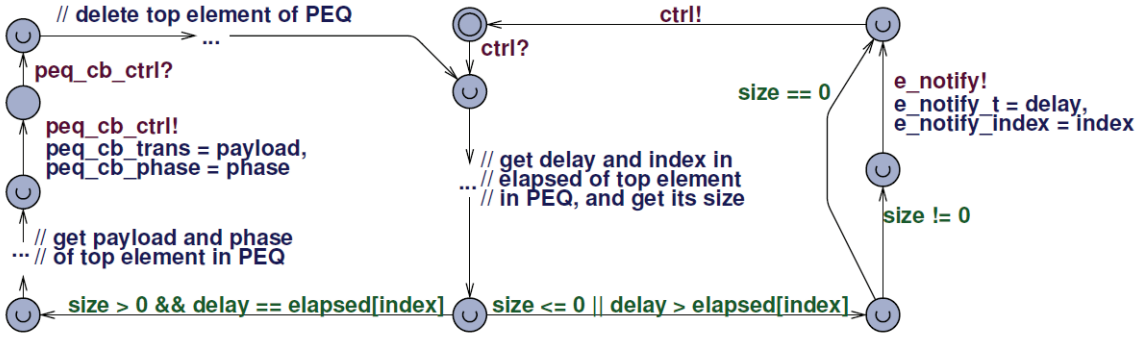

Figure 4.22: Timed automata template of the automaton that processes PEQ elements [47].

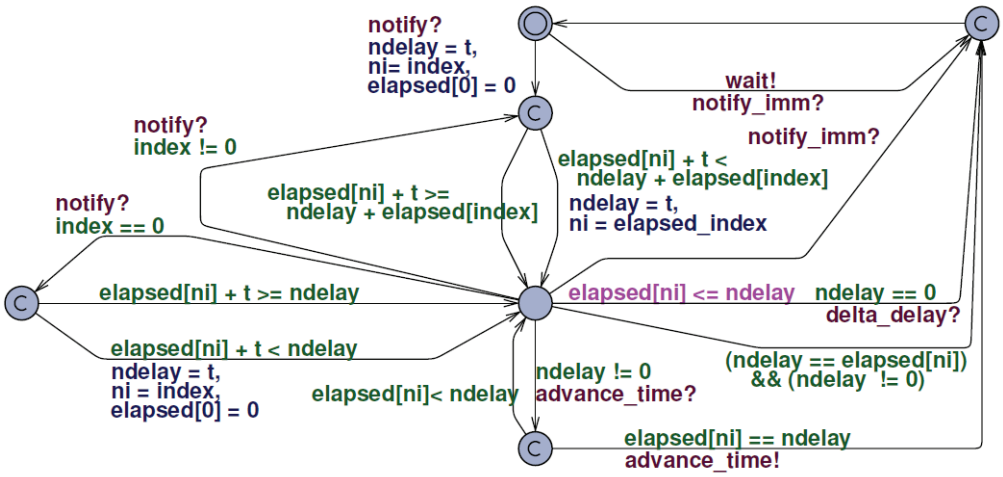

Figure 4.23: Timed automata model of the PEQ events [47].

4.10 Case Study 5: Extracting UPPAAL Model in AT Coding Style

In this section, we present a case study that focuses on an on-chip memory-mapped communication buses between an Initiator and a Memory module. This case study utilizes *Approximately-Timed* (AT) coding style and *TLM base protocol*. We utilize STATE tool-set to extract a timed automata model from the given SystemC TLM program.

In this case study, adapted from [1], the Initiator and the Memory use non-blocking transport (nb_trasport) interface for interaction. The nb_transport interface is intended to support the AT coding style and is particularly suited for modeling pipelined transactions. It breaks down each transaction into four phases, namely BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP, where each phase transition is associated with a timing point (see Figure 4.24). The Initiator generates a transaction and starts the communication by sending a BEGIN_REQ using the forward path nb_transport_fw to the Memory and waits to receive END_REQ or BEGIN_RESP from the backward path nb_transport_bw. After that, the Initiator can finish the

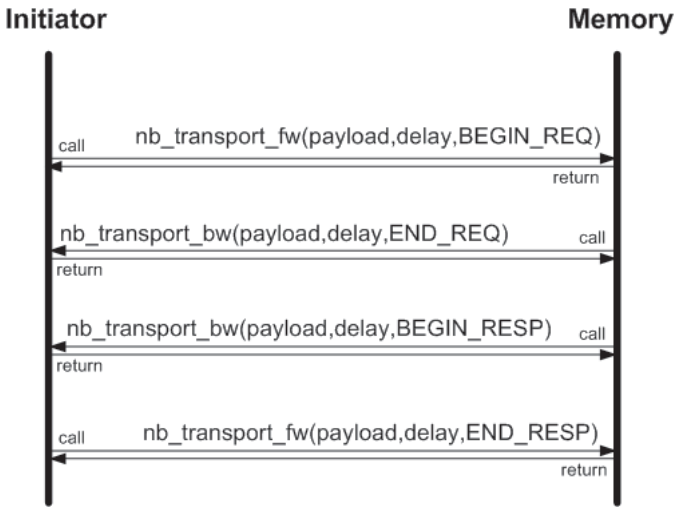

Figure 4.24: Non-blocking transport interface architecture.

transaction by sending END_RESP. The Initiator can also start another transaction by sending a new BEGIN_REQ. Note that during the first two phases we cannot have pipelined transactions. In other words, there is at most one BEGIN_REQ pending in the system, while we can have multiple transactions with BEGIN_RESP phases pending in the system. In this example, during analysis, we restrict the number of pipelined transactions by two.

Each transaction in an nb_trasport interface has three arguments: *generic payload*, *delay*, and *phase*. The generic payload is the transaction object being sent. The delay represents the timing annotation of the communication. The phase, which is a new argument in nb_transport, indicates the state of the transaction, e.g., BEGIN_REQ for starting a transaction, and returns an enumeration value to indicate whether the return from the function also represents a phase transition.

We utilize STATE tool-set, explained in Section 4.9 to extract the UPPAAL timed automata model. Nonetheless, the UPPAAL model generated by considering all possible components is too large to perform exhaustive analysis. Hence, for evaluating the model, we need to utilize model slicing techniques to only consider components that are important for verifying the prop-

erty/requirement of interest. To ensure that the extracted model captures the requirements of the SystemC TLM model, we specify a set of requirements that should hold in the absence of faults. We divide these requirements into two categories: 1) when timing constraints are not important and we want to ensure that message, permanent, and transient faults do not perturb the model, and 2) when training constrains are being verified to ensure that timing faults do not perturb the model. These requirements should be always true in the absence of faults (See Figures 4.25 and 4.26).

```
SPEC 1: A[] not deadlock
SPEC 2: Init.SentBeginReq --> (Memory.RcvdBeginReq)
SPEC 3: (Memory.SentEndReq or Memory.SentBeginResp) --> (Init.EndResp)
SPEC 4: Init.SentBeginReq --> (Init.Initial)
SPEC 5: Init.CurrTrans.cmd == readCmd --> (Target.SentData == Init.RcvdData)
SPEC 6: Init.CurrTrans.cmd == writeCmd --> (Init.SentData == Target.RcvdData)
SPEC 7: (Init.SentBeginReq or Init.EndReq)
    and
    (Memory.RcvBeginReq or Memory.SentBeginResp)
    -->
    Init.CurrTrans.phase == Memroy.CurrTrans.phase
```

Figure 4.25: Requirements of memory bus system using AT coding style.

In Figure 4.25, the first requirement represents that there is no deadlock in the extracted model in the absence of faults. The second Requirement shows that if the Initiator starts a transaction, the Memory module will eventually receive the transaction. Also if the Memory sends a response with either END_REQ or BEGIN_RESP phases, the Initiator will eventually be able to finish the transaction by sending END_RESP. This is shown in the third requirement. The forth requirement checks if a started transaction will eventually finish and the Initiator can start another transaction. This requirement along with the second requirement imply that the Initiator is not blocked. The fifth and sixth requirements represent that the communicated data between the Initiator and Memory are the same in the absence of faults. The last requirement helps to check the execution ordering of transactions while they are executed in pipeline. Using UPPAAL, we have model checked the

above properties for the extracted model.

```
SPEC' 1: A[] not deadlock
SPEC' 2: Init.Initial --> (Init.SentBeginReq and x >= delay1)
SPEC' 3: Init.Initial --> (Memory.RcvdBeginReq and x >= delay1)
SPEC' 4: (Memory.SentEndReq or Memory.SentBeginResp) -->
                  (Init.EndResp and x <= delay1 + delay2)
SPEC' 5: Init.Initial --> (Init.RcvdBeginResp and x <= delay1 + delay2)
SPEC' 6: Payload.ID == peq_fetch$trans.ID -->
                  sc_time_stamp == Payload.delay + Payload.Initialsc_time
```

Figure 4.26: Timing requirements of memory bus system using AT coding style.

In Figure 4.26, the first requirement SPEC' 1 represents that there is no deadlock in the extracted model in the absence of faults. The SPEC' 2 and SPEC' 3 show that the Initiator should not send the transaction with BEGIN_REQ phase before delay1 timing point, and the Memory should not receive the BEGIN_REQ request before delay1 timing point respectively. The SPEC' 4 checks that if the Memory sends either END_REQ or BEGIN_RESP, the Initiator will eventually finish the communication by END_RESP not later than delay1 + delay2 timing point. We ensure if the Initiator receives the response not later than delay1 + delay2 timing point by checking SPEC' 5. The SPEC' 6 represents that each transaction is executed at the right timing point. This requirement helps to check the execution ordering of transactions while they are executed in pipeline. For instance, if transaction *T1* should be executed at x and transaction *T2* should be executed at y, while $x + sc\_time\_stamp() > y + sc\_time\_stamp'()$, *T2* is executed first, where sc_time_stamp() illustrates the simulation time when a transaction is being sent. Using UPPAAL, we have model checked the above properties for the extracted.

4.11 Summary

In this chapter, we explained the requirements that we need to satisfy while extracting a formal model from the give SystemC TLM program. We proposes two sets of transformation rules for extracting formal models from SystemC TLM programs. The first set of rules transforms the SystemC program into a Promela model. The second set of rules generate a loosely-timed timed automata model that also consider the notion of time in the model extraction process. Additionally, we introduce a tool-set, STATE, for transforming concurrent SystemC TLM programs into timed automata models. However, some of the models generated by STATE are complex and need further simplification to be verifiable. Finally, we illustrated our model extraction ideas with five case studies.

Chapter 5

Modeling of Faults

The previous chapter permits model extraction of the given SystemC TLM program. This will allow us to analyze the given SystemC TLM program to verify its desired properties as well as to identify any bugs in it. In this section, we give a brief description of four types of faults considered in this dissertation. These types of faults are based on the discussion in [58] that identifies faults that are typically relevant to a SystemC TLM program. In our work, we distinguish between faults and bugs with the following intuition. A fault is something that we expect to happen in a program and we expect the program to provide desired behavior even if it occurs. Examples of such faults include faults such as message loss (caused due to noise), malicious components, transients, etc. By contrast, a bug is something that we expect to avoid. Examples include uninitialized variables, buffer overflow, incorrect use of blocking or nonblocking interfaces, incorrect use of timed/untimed constructs. With this distinction, intuitively, we want to ensure that the program works correctly even if faults occur. Our work focuses on the former, i.e., it assumes that the designer has decided that it is difficult/impossible to prevent the faults from occurring and, hence, it must be tolerated.

In the following, first, we explain the faults that we consider in this dissertation in Section 5.1. Then, in Section 5.2, we describe modeling of faults in Promela models. For modeling faults in UPPAAL timed automata models, we propose an algorithm in Section 5.3 and use this algorithm to inject different types of faults into the timed automata models extracted. The analysis of the models in the presence of faults are described in Sections 6.3.1, 6.3.2, and 6.3.3.

5.1 Fault Categories

In this dissertation, we consider four different types of faults. These faults are as follows.

1. *Message faults.* Since in SystemC TLM programs transactions are performed via message passing, one of the common faults is a message fault. These faults include message corruption, loss and duplication. In our case studies, we consider message loss. Modeling of message duplication is similar. And, modeling of message corruption is possible using the approach for transient faults.

2. *Permanent faults.* By permanent faults, we mean that the impact of the fault is long-lasting (possibly forever). In this paper, we consider *stuck-at* faults caused in hardware, *component failure*, and *Byzantine* faults. The stuck-at faults cause a signal to gets stuck at a fix value (logical 0, 1, or X) and cannot switch its value. In a component failure fault, the component fails functionally and the other components cannot communicate with it. The Byzantine fault is one where the faulty component continues to run but produces incorrect results. Byzantine faults encompass both *omission failures* such as failing to receive a request and failing to send a response, and *commission failures* such as processing a request incorrectly and sending an incorrect/inconsistent response to a request.

3. *Transient faults.* Transient faults are the most common types of faults that are prevalent in SoC systems [19, 49]. They perturb the state of system components without causing any permanent damage. It is anticipated that most of these faults occur only once (or a small number of times). In this paper we consider *Single Event Upsets* (SEUs). Such events may induce soft errors in storage elements (e.g., SRAM, sequential logic) due to alpha particles generated by the radioactive decay of packaging and interconnect materials.

4. *Timing faults.* A timing fault occurs when an event happens (or does not happen) in a specific time interval. Such timing faults could perturb the state of a system to an illegitimate configuration. For instance, consider a read transaction between an initiator and a memory, where the initiator sends an address and a read signal to fetch a datum from a specific memory cell. If the read signal on the memory side is activated later than required (or deactivated earlier than required), the read operation cannot be performed properly. We will investigate how we will model such timing faults and their impacts on system functionalities.

5.2 Fault Modeling for Promela Models

In this section, we model transient faults and inject them into the Promela models extracted in Sections 4.4 and 4.5. Transient faults can happen at different places of the extracted model. They can change the address, data, phase, etc. To model transient faults, the designer needs to identify variables of concern as far as transient faults are concerned. By default, we consider that all variables could be corrupted.

5.2.1 Case Study 1: Fault Modeling and Impact Analysis for Two Communicating Modules

In this section, we use the extracted model explained in Section 4.4 and analyze this model in the presence of transient faults. To model the transient fault that affects a given variable, we model it as a limited-time corruption of that variable at any reachable state in the program. To this end, we start with a fault-intolerant model in Promela, say M, and a set of actions that describe the *effect* of faults on M, denoted F. Our objective is to create a model M_F that captures the behaviors of M in the presence of faults F. The SystemC program of Figure 2.4 is subject to the type of

faults that corrupt the messages communicated between the Initiator and the Target. To model this fault-type, we include the following proctype in the extracted Promela model:

```
active proctype F() {
   do
     :: (cnt < MAX) -> atomic{ tgtIfPort!startTrans; cnt++;}
     :: (cnt < MAX) -> atomic{ tgtIfPort!endTrans; cnt++;}
     :: (cnt >= MAX) -> break;
   od;
 }
```

The constant MAX denotes the maximum number of times that faults can occur, where each time an erroneous message is inserted into the channel tgtIfPort. The cnt variable is a global integer that we add to the extracted model in order to model the occurrence of faults. For modeling purposes, we need to ensure that faults eventually stop, thereby allowing the program to execute and recover from them. (A similar modeling where one does not assume finite occurrences of faults but rather relies on a fairness assumption that guarantees that the program will eventually execute is also possible. However, it is outside the scope of this report.) Since faults can send messages to the tgtIfPort channel, it is possible to reach a state outside the invariant where the model deadlocks. For instance, consider a scenario where fault F injects endTrans in the channel. Then, the Target receives endTrans instead of startTrans. As such, the Target never completes the transaction and never sends an endTrans message to the Initiator, which is waiting for such a message; hence a deadlock.

5.2.2 Case Study 2: Fault Modeling and Impact Analysis for Memory-Mapped Buses

In this section, we model transient faults similar to that explained in Section 5.2.1, and apply it on the case study explained in Section 4.5. To illustrate the transient faults, we consider two instances (1) perturbing memory contents without causing any permanent damage, and (2) perturbing the read/write command of the generic payload.

5.2.2.1 Perturbing Memory Contents

To model transient faults that perturb memory contents, we define the following proctype that models the impact of this fault-type in the extracted model in Figure 4.3:

```
active proctype memFaults(){
  do
   :: (cnt1 < MAX1) -> atomic{ mem[memT.address] = 0; cnt1++;}
   :: (cnt1 < MAX1) -> atomic{ mem[memT.address] = 1; cnt1++;}
   :: (cnt1 < MAX1) -> atomic{ mem[memT.address] = 2; cnt1++;}
   :: (cnt1 < MAX1) -> atomic{ mem[memT.address] = 3; cnt1++;}
   :: else -> break;
  od;
}
```

Notice that while the mem array is declared inside the Memory module in Line 62 of Figure 2.6, in the model of Figure 4.2, we define it as a global array so we can access its contents from inside the memFaults proctype. To have finite occurrence of faults, we define a constant MAX1 that denotes the maximum number of times faults can occur. Moreover, we use the cnt1 variable to model the occurrence of faults similar to what we did in Section 5.2.1.

5.2.2.2 Control Signal Faults

In order to model the effect of the transient faults that perturb the cmd of the generic payload, we augment the model of Figures 4.2 and 4.3 with the the following proctype:

```
active proctype cmdFaults(){
        do
         :: (cnt2 < MAX2) -> atomic{ memT.cmd = 0; cnt2++;}
         :: (cnt2 < MAX2) -> atomic{ memT.cmd = 1; cnt2++;}
         :: else -> break;
         od;
}
```

The constant MAX2 is the maximum number of times faults can occur, and the cnt2 shows the occurrence of faults. The condition (cnt2 $<$ MAX2) ensures that the faults eventually stop occurring.

5.3 Fault Modeling for UPPAAL Timed Automata Models

In this section, we, first, discuss the generic descriptions of the fault categories introduced in Section 5.1. Then, we identify how these faults can be injected automatically into UPPAAL timed automata models and propose an algorithm for that. This algorithm is used to inject different types of faults into the timed automata models extracted in the last chapter and the fault-affected models will be explained later in this chapter. The reason that we propose our fault modeling approach for timed automata models is that such models consider the notion of times. Since in some of the SystemC TLM programs timing is important for us, we need to focus on a formal model that also supports timed systems. Therefore, in the rest of this dissertation, we study the UPPAAL timed

automata models.

5.3.1 Generic Description of Faults

The generic descriptions of the four aforementioned types of faults are discussed below.

5.3.1.1 Message loss

We present two methods for modeling message loss faults in the UPPAAL timed automata model:

- The first approach injects a new transition T into the UPPAAL timed automata model in parallel with a transition (L_i, L_j), where L_i and L_j are two locations in the extracted model and (L_i, L_j) represents the transition from L_i to L_j. Also the transition (L_i, L_j) corresponds to sending/receiving of a message. The transition T utilizes a channel $loss_m$ for synchronization. However, only the faulty component utilizes this channel and the other components are unaware of it.

- The second approach injects a transition T from location L_i to L_j. This transition does not have any synchronization channel, while the original transition (L_i, L_j) has a channel for synchronization. As a result, the faulty component assumes that the message is sent to other components and waits to receive a response. Nonetheless, the expected receiver does not receive any messages from the faulty component.

5.3.1.2 Permanent faults

As discussed in Section 5.1, we consider three types of permanent faults: fail-stop, Byzantine faults, and stuck-at faults. These faults are modeled as follows:

- To model a *fail-stop*, for each component c, we introduce a variable $down_c$ that denotes whether the component is working $(down_c = 0)$ or failed $(down_c = 1)$. This can be tailored to consider failure of all components or only to a subset of components or to a specific number of components. Furthermore, all component actions of component c are restricted to execute only if $(down_c = 0)$.

- In *Byzantine faults*, one or more components behave maliciously. By default, a malicious component can arbitrarily change the variables it can write. The designer can restrict it to a subset of variables if desired. To model the malicious component, a new transition T' is injected into the component. This transition updates the value of the variable subject to Byzantine faults.

- To model the *stuck-at* faults, we disable all transitions that change the value of the variable (identified by the designer using the same mechanism discussed earlier). This is achieved by revising all actions that change the value of affected variable(s).

5.3.1.3 Transient faults

To model the transient fault that affects a given variable, we model it as a one-time corruption of that variable at any reachable state in the program. (The modeling of transient faults are similar to Byzantine faults except that the transient faults occur only once. By contrast, a Byzantine component can send incorrect data continuously.)

5.3.1.4 Timing Faults

A timing fault occurs when an event happens (or does not happen) in a specific time interval. Such timing faults could perturb the state of a system to an illegitimate configuration. In other

words, the timing faults cause an action/operation to be executed either too early or too late, and, as a result, the operation cannot be performed properly. In a timing fault, we consider the case where an operation takes longer than expected or the case where it takes shorter than expected. In particular, we consider this effect during communication with other components (rather than in internal operations in components). Hence, we first identify the operations that could be subject to timing faults. By default, these are all operations that result in invocation and return of transactions. The UPPAAL model corresponding to these operations have guards that identify conditions under which these operations can be performed. Since UPPAAL model is based on timed automata, the basic constraint used in these guards is of the form "$x\ OP\ c$", where x denotes the local clock, c represents a timing point in the model, and OP is $<$, $\leq$, $>$, etc.

To model the timing faults, we introduce a new variable $delay_t$ and a new transition T_t to the fault-free UPPAAL model that can model both early and late timing problems. The maximum value of $delay_t$ (default value is 1) is identified by the designer. The UPPAAL model is further modified to non-deterministically increase the delay argument in a transaction by $delay_t$ in all processes. To automate the fault injection, we target the transitions with a guard(s). We inject the new transition into the model to be in parallel with the original transition between two locations, say L_i and L_j. Choosing the original transition, the model continues its execution without faults, while choosing the new transition injects the timing faults into the model. Hence, we introduce rules of the following form that utilize the variable $delay_t$ to define the guard(s) of the transition T_t:

1) $if\ \ (x > c)\ \ \rightarrow\ \ x > c + delay_t$

2) $if\ \ (x < c)\ \ \rightarrow\ \ x < c - delay_t$

5.3.2 Automatic Fault Injection

In this section, we describe the automatic fault injection mechanism. Faults are injected based on the following parameters which are specified by designer.

- *The fault type*. Currently, there are three types of faults as explained in Section 5.1.
- *Effect of faults on the program*. The designer needs to specify the variables affected by faults as follows:
  - *Message loss*. For this type, we assume that any of the messages in the model may be lost. The designer can limit it to a subset if desired.
  - *Permanent*. i) *Fail-stop*: For this type of fault, the designer needs to specify the component that is likely to fail. By default, we consider the case where any component can fail; ii) *Stuck-at*: For this type of fault, the designer needs to specify which variable(s) may be corrupted by the stuck-at component and the possible value(s); iii) *Byzantine*: Similar to the stuck-at fault, the designer needs to specify which variable(s) may be corrupted by the Byzantine component and the possible value(s). For instance, in the example of Section 4.7, the variable representing the action (read/write) is affected by faults. This fault can change the requested action and leads to an undesirable state. Hence, the default for this fault is that the variable can be corrupted to any value in its domain.
  - *Transient*. For this type of fault, the designer needs to specify which variables are likely to be affected by a transient fault. The default for this fault is that any variable can be corrupted to any value in its domain.

– *Timing*. For timing faults, the designer needs to identify which clock variables are subject to faults. The designer also needs to define a value for the variable $delay_t$. If a value for this variable is not specified, the default value, which is 1, for the clock variable will be applied.

- *Number of occurrences of faults*. The designer also needs to specify the occurrences of the transient faults. This number denotes the occurrences of transient faults that may take place during the computation. The default setting value is 1.

5.3.2.1 Algorithm Description

The input of Algorithm 1 is a fault-intolerant timed automata model M in XML format and the parameters described above. The output is a fault-affected timed automata model M' in XML format.

Like the TA model, the XML file has a set of locations and transitions, which are respectively defined by the following tags: "$< location >$ *statements* $< /location >$" and "$< transition >$ *statements* $< /transition >$". The *statements* can be a *name*, an *invariant*, or a *type* (e.g., urgent, committed) for locations, and a *source*, a *target*, or *labels* for transitions. The source and target tags represent the position of the transition. The label tag shows whether the transition has a *synchronization* channel, an *assignment* operation, or a *guard* condition.

The Algorithm 1 utilizes three functions $Find$, $Remove$, and $Change$. The function $Find$ takes a model M and a label L and returns a transition T that has label L in model M. The function $Remove$ takes a transition T and a synchronization channel ch and removes the channel ch from T. The function $Change$ takes a transition T and a variable v and returns a transition with a changed value of v.

Algorithm 1 Automatic Fault Injection

Input: A fault-intolerant Timed Automata model M in XML format, variable v subject to faults, type of fault, and counter c.
Output: A fault-affected Timed Automata model M' in XML format.

1: $AddMoreFaults \leftarrow true$, $cnt \leftarrow 0$;
2: **while** ($AddMoreFaults = true$) **do**

3: **Message Loss:**
4: $T \leftarrow$ Find(M, $kind = TransitionKind$);
5: $T' \leftarrow T$;
6: $T' \leftarrow$ Remove(T', channel); {or $T' \leftarrow$ Change(T', channel)}
7: $AddMoreFaults \leftarrow false$;

8: **Fail-stop:**
9: $T \leftarrow$ Find($M, true$);
10: **if** T has an $assignment$ statement **then**
11: add ($down_c \leftarrow 1$) to T's set of assignments;
12: **else**
13: add an $assignment$ statement to T, and add ($down_c \leftarrow 1$) to its set of assignments;
14: **end if**
15: **if** T has a $guard$ statement **then**
16: add ($down_c = 0$) to T's set of guards;
17: **else**
18: add a $guard$ statement to T, and add ($down_c = 0$) to its set of guards;
19: **end if**
20: $AddMoreFaults \leftarrow false$;

21: **Byzantine Fault:**
22: $T \leftarrow$ Find(M, $kind = TransitionKind$);
23: $T' \leftarrow T$;
24: $T' \leftarrow$ Change(T', v); {No need to change $AddMoreFaults$}

25: **Stuck-at Fault:**
26: $T \leftarrow$ Find(M, $kind = TransitionKind$);
27: $T \leftarrow$ Change(T, v);
28: $AddMoreFaults \leftarrow false$;

29: **Transient Fault:**
30: $T \leftarrow$ Find(M, $kind = TransitionKind$);
31: **if** ($cnt \leq c$) **then**
32: $T \leftarrow$ Change(T, v);
33: **else**
34: $AddMoreFaults \leftarrow false$;
35: **end if**

36: **Timing Fault:**
37: $T \leftarrow$ Find(M, $kind = TransitionKind$); $T' \leftarrow T$;
38: $T' \leftarrow$ Change(T', v); $AddMoreFaults \leftarrow false$;

39: **end while**

Based on the type of the fault, the algorithm scans the XML file, finds the corresponding part, and changes it as necessary for the fault. For message loss (Lines 3-7), we identify where the message loss occurs by finding a transition T that has a label $kind = synchronization$. This label represents that T is synchronizing with other modules. Utilizing T, we create T' by removing its synchronization channel, and inject it in parallel with T into the model. In the case studies, we apply this approach to generate several fault-affected models, each model considers the case

where one specific message may be lost. This can be trivially generalized to generate a model that simultaneously loses multiple messages. To model the other approach of modeling message loss described in Section 5.3.1, the synchronization channel of T' should be changed to a faulty channel (by calling function $Change$). After injecting the fault, we use a variable *AddMoreFaults* to terminate the algorithm.

To model a fail-stop fault (Lines 8-20), we use an arbitrary transition T. If T has a label *kind=assigment*, which means T has an assignment statement, we add $down \leftarrow 1$ to its set of assignments. If it does not, we define a new label *kind=assigment* and add $down \leftarrow 1$ to its set of assignment. This step is repeated by every transition in the component subject to fail-stop fault. Also we add the guard *down=0* to the set of T's guards. For modeling the effects of failing a specific component, the locations of transition T (*source* label for the starting location and *target* label for the ending location) needs to be given to the algorithm.

If the fault is a Byzantine fault (Lines 21-24), we inject a new transition T' in parallel to the original transition T, which has an assignment label. The value of the variable v, which is subject to faults, is corrupted in T'. Choosing T', the fault is injected to the model, while by choosing T, the model continues its normal execution. The occurrence of this fault does not terminate the algorithm, while injecting a stuck-at fault (Lines 25-38) terminates the algorithm.

For transient faults (Lines 29-35), we define a counter that controls the number of occurrence of the fault. When the counter is greater than the input c, the algorithm terminates.

To model timing faults (Lines 36-38), we inject a new transition T' in parallel to the original transition T, which has a guard label. Then, we find the guard that has a clock variable in it and change the guard using the variable $delay_t$ defined by the designer.

5.4 Summary

In this paper, we focused on analyzing the effect of different types of faults that are of concern in the SystemC TLM program. We differentiated faults (that need to be tolerated) and bugs (that need to be prevented) and focused on the former.

We began with the extracted model from the given SystemC TLM model. We considered four types of faults, message faults, permanent faults, transient faults, and timing faults. For modeling faults in Promela models, we considered transient faults and injected them into the Promela models extracted from the SystemC TLM programs. For UPPAAL timed automata models, we considered four types of faults introduced in Section 5.1. For each type of faults, we utilized a generic approach to transform the UPPAAL model to obtain a fault-affected model. Subsequently, this model was used in Promela and UPPAAL to obtain a counterexample. We were either able to verify that the original specification is satisfied or find a counterexample demonstrating the violation of the original specification. Moreover, the time for evaluating the effect of faults was comparable (0-57%) to the verification in the absence of faults.

In order to demonstrate our approach for Promela models, we conducted two case studies and studied the Promela models in the presence of different transient faults. The analysis of UPPAAL timed automata models in the presence of faults is discussed in the next chapter.

Chapter 6

The Tool UFIT: The Fault Injector To UPPAAL Timed Automata

In this chapter, we explain our tool, UFIT. This tool is developed based on Algorithm 1 explained in Chapter 5. In the following sections, first, we explain the input of UFIT. Thereafter, using a runtime example, we introduce how UFIT works and inject different types of faults into the example. Finally, we demonstrated our approach with several case studies.

6.1 Input of UFIT

The input of UFIT is a fault-intolerant timed automata model in XML format and a set of parameters. We describe these parameters and our fault modeling approach used in UFIT utilizing a running example from the literature of UPPAAL timed automata, the *Fischer's mutual exclusion protocol* [**?**, 9] (Figure 6.2).

6.1.1 The running example

Fischer's protocol is designed to ensure mutual exclusion among several processes (5 processes here) competing for a critical section using timing constraints and a shared variable *id*. In each process P, the process goes to a request location req if it is the turn for no process to enter the critical section (id==0). After x time units in req ($0 \leq x \leq k$), P goes to the wait location and

sets id to its process ID. Finally, after at least k time units, P enters the critical section cs if it is its turn. The Fischer's protocol satisfies the following set of requirements/properties in the absence of faults:

```
SPEC 1: A[] not deadlock
SPEC 2: P(i).req --> P(i).wait
SPEC 3: A[] P1.cs + P2.cs + P3.cs + P4.cs + P5.cs <=1
```

where SPEC 1 checks wether the system is deadlock-free. The liveness property SPEC 2 checks that whenever a process tries to enter the critical section, it will always eventually enter the waiting location. The safety property SPEC 3 checks for mutual exclusion of the location sc.

6.2 Internal Functionality

To generate the fault-affected model, in addition to the fault-free model, we need to specify the type of the faults and a set of parameters (see Figure 6.1). The fault types that UFIT considers are as follows.

- *Message faults,* where a message may be lost while forwarding from one module to another;
- *Fail-stop* faults, where a module fails functionally and the other modules cannot communicate with it;
- *Byzantine* faults, where the faulty component continues to run but produces incorrect results;
- *Stuck-at* faults, where a signal gets stuck-at a fixed value (logical 0, 1, or X) and cannot switch its value, and

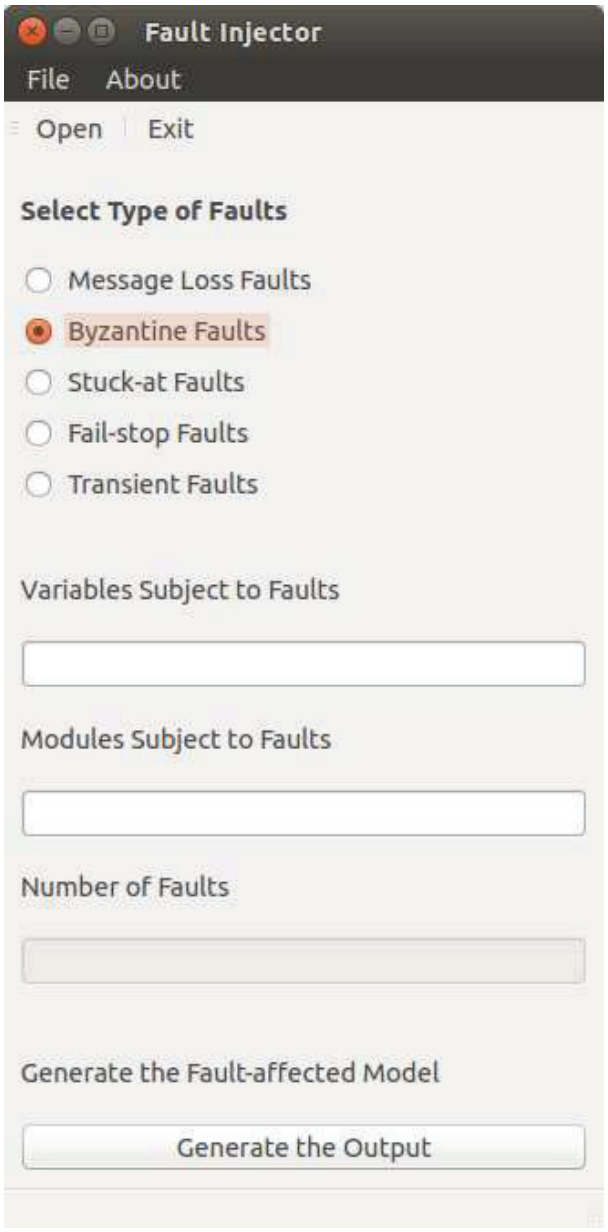

Figure 6.1: The GUI of UFIT.

- *Transient faults,* where the state of system components is perturbed without causing any permanent damage.

In addition to the fault type, the following three discrete variables can be specified:

- *Variable subject to faults.* We are not allowed to increase or decrease the value of the clock variable;
- *Module subject to faults.* We assume any module can be subject to faults, and
- *Number of faults.* The number of occurrences of the transient faults that may take place during the computation needs to be defined. The default setting value is 1.

Remark 6.1 *If any of the above variables is not specified, UFIT will set a value for them arbitrarily. For instance, if the module subject to fail-stop faults is not specified, UFIT will fail one of the modules randomly.*

6.2.1 Brief discussion about modeling of faults in UFIT

Given the parameters and the fault type, intuitively we model the faults as follows. To model a message fault, we inject a new transition into the module subject to faults in parallel to a transition that has a synchronization channel. The set of assignments/guards of the new transition is similar to that of the original transition except that the synchronization channel is changed. To model a fail-stop fault, we define a variable *down* that shows if a module is failed (down=1). For example, Figure 6.3 illustrates that automaton $P1$ is failed since $P1$ cannot go to location wait and has to stay at location req forever. To model stuck-at faults, UFIT finds the location of the variable subject to faults and changes it to a random value. For example, in Figure 6.4, the value of id is stuck at 5, thereby $P1$ cannot enter the critical section. For modeling byzantine faults, UFIT adds a

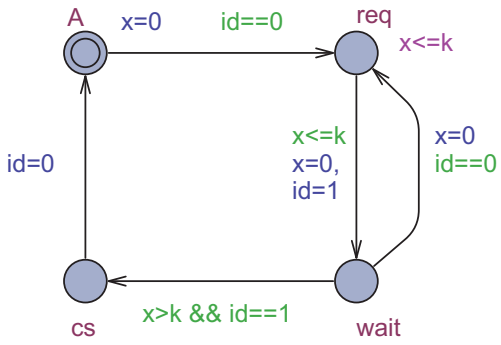

Figure 6.2: Fault-free model of Fischer's mutual exclusion protocol.

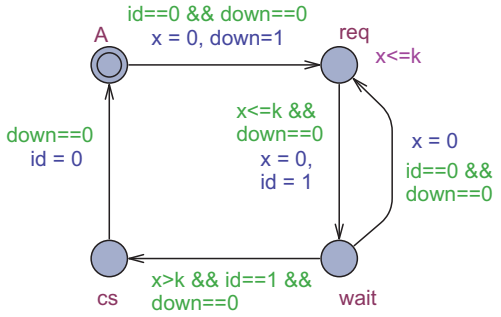

Figure 6.3: Modeling fail-stop fault for Fischer's mutual exclusion protocol.

transition in parallel to that of the original automaton that updates the variable subject to faults and changes its value arbitrarily. Figure 6.5 shows injecting a byzantine faults that changes the value of id, if the faults occur. Modeling of transient faults is similar to that of byzantine faults except that the occurrence of transient faults is limited. UFIT utilizes the number of faults defined in the GUI to limit the number of occurrence of this type of faults.

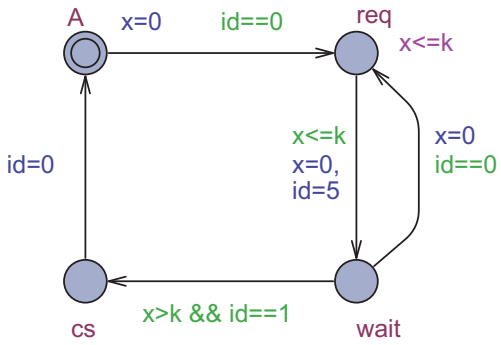

Figure 6.4: Modeling Stuck-at 5 fault for Fischer's mutual exclusion protocol.

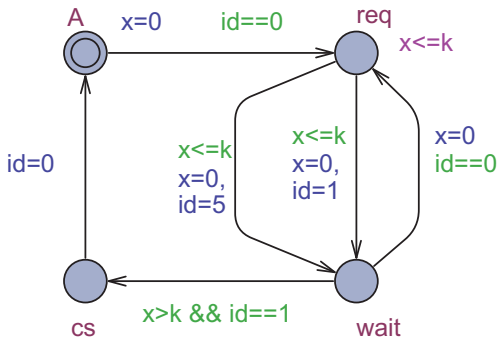

Figure 6.5: Modeling Byzantine fault for Fischer's mutual exclusion protocol.

6.2.2 Analysis of Results

In this section, we analyze the fault-affected models. Also, in addition to Fischer's protocol, we include the results of two other examples adapted from [9]: the train gate and vikings problems. *The train gate problem* is a railway control system which controls access to a bridge for several trains. The bridge is a critical shared resource that may be accessed only by one train at a time. The system is defined as a number of trains and a controller. The model satisfies the following properties in the absence of faults:

```
SPEC 1: A[] not deadlock

SPEC 2: E<> Train(0).Cross and (forall (i : id_t) i != 0 imply Train(i).Stop)

SPEC 3: A[] forall (i : id_t) forall (j : id_t)
                                  Train(i).Cross && Train(j).Cross imply i == j

SPEC 4: Train(0).Appr --> Train(0).Cross
```

where SPEC 2 shows that train 0 can cross bridge while the other trains are waiting to cross. SPEC 3 illustrates that there is never more than one train crossing the bridge (at any time instance), and SPEC 4 provides that train 0 (similarly any other train) approaches the bridge, it will eventually cross.

In the *Vikings problem*, four Vikings want to cross a bridge at night, but they have only one

torch and the bridge can only carry two of them. Thus, they can only cross the bridge in pairs and one has to bring the torch back to the other side before the next pair can cross. Each viking has different speed. The question is whether it is possible that all the vikings cross the bridge within a certain time. This example is comparable to the question if a packet can reach its receiver in a given time limit in a communication network/Network on Chip (NoC) system. The TA model satisfies the following properties in the absence of faults:

```
SPEC 1: A[] not deadlock
SPEC 2: E<> Viking1.safe
SPEC 3: E<> Viking1.safe and Viking2.safe and Viking3.safe and Viking4.safe
```

where SPEC 2 illustrates that the first viking eventually gets to the other side of the river and SPEC 3 shows that all the vikings are in their safe location.

The results of analyzing the examples in the presence of faults are as shown in Table 6.1. In this table, if requirement x is satisfied, we include s in the table, otherwise v.

6.3 Case Studies on Modeling Faults for UPPAAL Timed Automata Models

In this section, we use UFIT to inject different types of faults into three case studies in Sections 6.3.1, 6.3.2, and 6.3.3. The first two case studies are modeled in Loosely-Timed (LT) coding style and the last case study is using Approximately-Timed (AT) coding style.

Protocol	Cause	Affected Locations	SPEC				Total Time (ms)
			1	2	3	4	
Fischer's protocol	Fault-free model	–	s	s	s	–	1250
	Fail-stop	Process P1	v	v	s	–	143
	Transient	Process P1	v	s	s	–	79
	Stuck-at	Process P1	v	s	s	–	81
	Byzantine	Process P1	v	s	s	–	149
Train-Gate protocol	Fault-free model	–	s	s	s	s	218
	Fail-stop	Controller	v	v	s	v	35
		Train 0	v	v	s	v	362
	Message loss	Train to Controller	v	s	s	v	210
		Controller to Train	v	s	s	v	241
Viking protocol	Fault-free model	–	s	s	s	–	25
	Fail-stop	Viking 0	v	v	v	–	23
		Torch	v	v	v	–	15
	Message loss	Viking to Torch	v	v	v	–	17
	Byzantine (L=1)	Torch	v	s	v	–	29
	Stuck-at 0	Torch	v	s	s	–	15
	Stuck-at 1	Torch	v	s	v	–	15
	Transient (L=1)	Torch	v	s	v	–	14

Table 6.1: Modeling and analyzing the impact of faults.

6.3.1 Case Study 1: Fault Modeling and Impact Analysis

In this section, we model timing faults and analyze their impacts on the model extracted in Section 4.7. As described in Section 5.3, we use $delay_t$ to inject a delay to the extracted model. For instance, in the extracted timed automaton in Figure 4.7, by injecting the $dealy_t$, we change the guard of the transition (L4, L5) to x + $delay_t \leq$ delay1+delay2. As a result, the guard does not become true and the program gets stuck at location L4. This fault violates SPEC 1, SPEC 4, and SPEC 5 of Figure 4.10.

6.3.1.1 Analysis of the fault

If the Initiator executed too late, it is unable to receive the response sent by the Memory on time (at the *delay* timing point set by the Memory in the response). Also if the Memory sets the *delay*

timing point but sends it too late to the Initiator, the Initiator does not receive it on time. We model it by injecting the timing faults into one of the locations (L4, L5), (L12, L10), (L13, L10). Injecting the faults into these locations violates requirements 1, 4, and 5 of Figure 4.10. Table 6.2 represents more experiments based on the causes of some timing faults in the SystemC TLM program and their possible injecting locations in the UPPAAL model. If a requirement is violated, we show it by x_v, and if it is satisfied we show it by x_s, where x is the requirement number defined in the Figure 4.10. For example, when the Memory executed too late and the affected location is (L4, L5), the requirements SPEC 1, SPEC 4, and SPEC 5 are violated, while SPEC 2 and SPEC 3 are satisfied. The average time for checking all these requirements is 8 ms.

Cause	Affected Locations	Requirement Status	Total Time (ms)
Initiator executed too late	(L4, L5)	$1_v, 2_s, 3_s, 4_v, 5_v$	8
	(L12, L10)	$1_v, 2_s, 3_s, 4_v, 5_v$	7
	(L13, L10)	$1_v, 2_s, 3_s, 4_v, 5_v$	7
Initiator executed too early	(L2, L4)	$1_v, 2_v, 3_v, 4_s, 5_v$	4
	(L3, L4)	$1_v, 2_v, 3_v, 4_s, 5_v$	4
Memory executed too late	(L4, L5)	$1_v, 2_s, 3_s, 4_v, 5_v$	8
	(L12, L10)	$1_v, 2_s, 3_s, 4_v, 5_v$	7
	(L13, L10)	$1_v, 2_s, 3_s, 4_v, 5_v$	7
Memory executed too early	(L10, L11)	$1_v, 2_s, 3_v, 4_s, 5_v$	5

Table 6.2: Modeling and analyzing timing faults in the memory bus system while using LT coding style.

6.3.2 Case Study 2: Fault Modeling and Impact Analysis

In this section, we present the rules to transform the fault-free model explained in Figures 4.12 and 4.13 into the fault-affected model. For each type of faults, first, we identify a generic approach for modifying the UPPAAL model. Then, we identify the revised model and evaluate its correctness.

6.3.2.1 Message Faults

In our extracted model, we consider an arbitrary number of message loss faults that can lose any of the messages in the system. Hence, we introduce a transition without any synchronization channel to model a message loss. We inject this transition at different components. For example, we inject a new transition into the Router in Figure 4.12 between Locations L9 and L10. As a result, the Router uses this transition and changes its state to L10 and waits to receive the response from one of the Targets. The desired Target, however, does not receive any messages from the Router and waits at its initial state.

Cause	Affected Locations	Requirement Status	Total Time (ms)
Fault-free model	–	$1_s, 2_s, 3_s, 4_s, 5_s$	13
Message loss	Initiator to Router	$1_v, 2_v, 3_v, 4_s, 5_v$	12
	Router to Target	$1_v, 2_v, 3_v, 4_s, 5_v$	12
	Target to Router	$1_v, 2_v, 3_v, 4_s, 5_v$	13
	Router to Initiator	$1_v, 2_v, 3_v, 4_s, 5_v$	13
Component failure	Initiator	$1_v, 2_v, 3_v, 4_z, 5_v$	13
	Router	$1_v, 2_v, 3_v, 4_s, 5_v$	13
	Target	$1_v, 2_v, 3_v, 4_s, 5_v$	14
Byzantine	Initiator	$1_s, 2_z, 3_z, 4_s, 5_s$	14
	Router	$1_s, 2_z, 3_z, 4_s, 5_s$	14
	Target	$1_s, 2_z, 3_z, 4_s, 5_s$	14
Stuck-at	Initiator	$1_s, 2_z, 3_z, 4_s, 5_s$	14
	Router	$1_s, 2_z, 3_s, 4_s, 5_s$	14
	Target	$1_s, 2_z, 3_s, 4_s, 5_s$	14

Table 6.3: Modeling and analyzing faults in the NoC switch while using LT coding style.

We model the message faults while the message is sent between Initiator-Router, Router-Target, Target-Router, and Router-Initiator. The results are as shown in Table 6.3. In this, and subsequent tables, if requirement x is satisfied, we include x_s in the table. If it is violated, we include x_v. If the answer is more complicated, we include x_z and explain the result in the text. Also, SPEC 5 is for all possible Initiators. Hence 5_s means that the requirement for all the Initiators is satisfied,

and 5_v means that the requirement in at least one of the Initiators is violated. The results in Table 6.3 illustrate that the average total time in the fault-affected model (14 ms) is comparable to that in the fault-free model (13 ms). Note that requirement SPEC 5 is defined for the Initiator modules and can be defined for the Router and Targets in the same way.

As discussed in Section 5.1, we consider three types of permanent faults, fail-stop, Byzantine faults and stuck-at faults. Next, we explain modeling of permanent faults on the extracted UPPAAL model.

6.3.2.2 Modeling and analyzing fail-stop faults in the case study

In this example, we consider three types of fail-stop faults: Initiator, Router, and Target failures. The results for failure of different components is as shown in Table 6.3. As expected, a router failure causes all properties to be violated. However, failure of initiator or target does not lead to the whole system failure. Specifically, regarding the Initiator failure, the location of the fault injection affects satisfaction of SPEC 4. If the fault occurs after setting the attributes in the sending transaction, the fault does not violate SPEC 4. If the fault occurs while setting the attributes, the requirement SPEC 4 is violated. Hence, we show it as 4_z.

6.3.2.3 Modeling and analyzing Byzantine faults in the case study

In this case study, we consider the case where the variable of concern is cmd. For this purpose, we inject the faults in the Initiator such that the cmd attribute is non-deterministically changed. In other words, fault causes the initiator to behave maliciously by corrupting the cmd variable from 0 to 1 and vice versa. The effects of these faults on the program are as shown in Table 6.3. As shown in Table 6.3, the resulting program satisfies SPEC 1, SPEC 4 and SPEC 5 and the satisfaction of SPEC 2 and SPEC 3 depends upon the effect of Byzantine fault. Specifically when cmd is

changed from 0 to 1 (respectively 1 to 0), SPEC 2 (respectively SPEC 3) is violated and SPEC 3 (respectively, SPEC 2) is satisfied. We have also considered Byzantine failure at the Router and Target. The results are as shown in Table 6.3.

6.3.2.4 Modeling and analyzing stuck-at faults in the case study

Modeling of the stuck-at faults is similar to that in Byzantine faults except that once the fault occurs, the affected variables cannot change. We consider the stuck-at fault for the variable cmd to 1 in Table 6.3, which means the Initiator is always requesting a write action. As a result, when a write action is requested, the effects of stuck-at faults cannot be found and SPEC 3 is satisfied.

6.3.3 Case Study 3: Fault Modeling and Impact Analysis

In this section, we model and analyze the impact of all four types of faults explained in Section 5.1 on the model extracted in Section 4.10.

6.3.3.1 Message Faults

The modeling of message loss in this case study is similar to that in Section 6.3.2 with the exception that the program is using nb_transport_fw and nb_transport_bw for forwarding and receiving transactions. There are four types of messages in this system, BEGIN_REQ,END_REQ, BEGIN_RESP and END_RESP. Hence, we consider the case where any one of these messages is lost. These faults are modeled by identifying the locations where the message is sent and adding new transitions as described in Section 6.3.2.1. The experimental results are represented in Table 6.4.

As an illustration, consider the case when the END_RESP is lost. In this case, the Initiator would not get blocked and is able to initiate a new transaction, since we have pipelined transactions

in the system. This requirement corresponds to satisfaction of SPEC 1 and SPEC 4 in Figure 4.25. These properties are validated in Table 6.4 in the presence of a message loss of END_RESP.

Cause	Affected Locations	Requirement Status	Total Time (s)
Fault-free model	–	$1_s, 2_s, 3_s, 4_s, 5_s, 6_s, 7_s$	5
Message loss	Initiator-sending BEGIN_REQ	$1_v, 2_v, 3_s, 4_v, 5_v, 6_v, 7_s$	4
	Initiator-sending END_RESP	$1_s, 2_s, 3_s, 4_s, 5_s, 6_s, 7_s$	4.5
	Memory-sending END_REQ	$1_v, 2_s, 3_v, 4_v, 5_v, 6_v, 7_s$	4.2
	Memory-sending BEGIN_RESP	$1_s, 2_s, 3_v, 4_s, 5_v, 6_v, 7_s$	4.2
Component failure	Initiator	$1_v, 2_v, 3_v, 4_v, 5_v, 6_v, 7_s$	4
	Memory	$1_v, 2_v, 3_v, 4_v, 5_v, 6_v, 7_s$	4
Byzantine	Initiator	$1_s, 2_s, 3_s, 4_s, 5_z, 6_z, 7_s$	5.5
	Memory	$1_s, 2_s, 3_s, 4_s, 5_z, 6_z, 7_s$	5.5
Stuck-at	Initiator-stuck-at 1	$1_s, 2_s, 3_s, 4_s, 5_z, 6_s, 7_s$	5.5
	Memory-stuck-at 1	$1_s, 2_s, 3_s, 4_s, 5_z, 6_s, 7_s$	5.5
	Initiator-stuck-at 0	$1_s, 2_s, 3_s, 4_s, 5_s, 6_z, 7_s$	5.5
	Memory-stuck-at 0	$1_s, 2_s, 3_s, 4_s, 5_s, 6_z, 7_s$	5.5
Transient	Initiator-cmd attribute	$1_s, 2_s, 3_s, 4_s, 5_z, 6_s, 7_s$	5.5
	Memory-cmd attribute	$1_s, 2_s, 3_s, 4_s, 5_z, 6_s, 7_s$	5.5
	Initiator-phase attribute	$1_z, 2_z, 3_z, 4_z, 5_v, 6_v, 7_v$	5.7
	Memory-phase attribute	$1_s, 2_s, 3_s, 4_s, 5_z, 6_z, 7_v$	5.7

Table 6.4: Modeling and analyzing faults in the memory bus system while using AT coding style.

6.3.3.2 Permanent Faults

We model failure of components, Byzantine faults, and stuck-at faults in this case study. The structural changes performed for these faults to obtain the fault-affected UPPAAL model is similar to that in Section 6.3.2. In particular, the Byzantine and stuck-at faults are modeled like that in Section 6.3.2.

In modeling component failure, either the Initiator or the Memory can fail. The location of injecting the variable $down_c$ can be different and does not change the results of Table 6.4. Failure of one of the components blocks the whole system and only the last requirement explained in Figure 4.25 is satisfied, since the message ordering is not changed.

In Byzantine faults, we consider the case where the variable of interest is the cmd variable. And, the Byzantine component can change it from 0 to 1 and vice versa. The effects of these faults on the model are as shown in Table 6.4.

In this example, we consider the stuck-at fault for the variable cmd to either 0 or 1. When cmd is stuck-at 1 (respectively 0) SPEC 6 (respectively, SPEC 5) in Figure 4.25 is trivially satisfied.

6.3.3.3 Transient Faults

To model transient faults, the designer needs to identify variables of concern as far as transient faults are concerned. By default, we consider that all variables could be corrupted. In this case study, we consider the case where transient faults can change the command or phase.

To model the transient fault that affects a given variable, we model it as a one-time corruption of that variable at any reachable state in the program. (The modeling of transient faults are similar to Byzantine faults except that the transient faults occur only once. By contrast, a Byzantine component can send incorrect data continuously.) To illustrate the transient faults, we consider two instances: (1) where cmd attribute is corrupted, and (2) where phase argument is corrupted.

Regarding a transient fault that affects cmd, satisfaction of SPEC 5 and SPEC 6 in Figure 4.25 depends upon whether cmd is corrupted from 0 to 1 or from 1 to 0. Hence, Table 6.4 represents it as 5_z and 6_z.

Regarding a transient fault that affects phase attribute, satisfaction of SPEC 1, SPEC 2, SPEC 3 and SPEC 4 in Figure 4.25 depends upon the actual effect of the transient fault. For example, if BEGIN_REQ is perturbed to END_RESP then SPEC 3 is violated. However, if END_RESP is perturbed to BEGIN_REQ then SPEC 3 is satisfied. Hence, Table 6.4 represents this as $1_z, 2_z, 3_z$ and 4_z.

6.3.3.4 Timing Faults

To model the timing faults, we utilize the same approach as that explained in Section 6.3.1. Table 6.5 represents several experiment results based on the causes and locations of the timing faults. For example, if the Initiator executed too early, the affected locations in the UPPAAL model would be in the Initiator automaton and either before sending a transaction with phase BEGIN_REQ or before receiving the transaction with phase END_REQ. Also the average time for verifying the requirements are very similar to that for verifying the requirements of Figure 4.26 in the absence of faults.

Cause	Affected Locations	Requirement Status	Total Time (s)
Initiator executed too early	Before sending begin_req	$1_v, 2_v, 3_v, 4_s, 5_v, 6_s$	11.8
	Before receiving end_req	$1_v, 2_s, 3_s, 4_v, 5_v, 6_s$	12
Initiator executed too late	After dispatching from peq	$1_v, 2_s, 3_s, 4_v, 5_v, 6_v$	16.9
	While inserting into peq	$1_v, 2_s, 3_s, 4_v, 5_s, 6_v$	16.9
	Before sending end_resp	$1_v, 2_s, 3_s, 4_v, 5_s, 6_s$	17
Memory executed too early	While inserting into peq	$1_v, 2_s, 3_s, 4_s, 5_v, 6_v$	14.2
	During the 2_{nd} insertion into peq	$1_v, 2_s, 3_s, 4_v, 5_v, 6_v$	16.1
Memory executed too late	After the 2_{nd} dispatch from peq	$1_v, 2_s, 3_s, 4_v, 5_v, 6_v$	16.9
	While inserting into peq	$1_v, 2_s, 3_s, 4_v, 5_s, 6_v$	16.9

Table 6.5: Modeling and analyzing timing faults in the memory bus system while using AT coding style.

6.4 Summary

In this chapter, we presented the tool UFIT and explained how it models different types of faults in timed automata models. For each type of faults, we utilized a generic approach to transform the UPPAAL model to obtain a fault-affected model. Subsequently, this model was used in UPPAAL to conclude tolerance to faults or to obtain a counterexample. We were either able to verify that

the original specification is satisfied or find a counterexample demonstrating the violation of the original specification. Moreover, the time for evaluating the effect of faults was comparable ($<$ 165%) to the verification in the absence of faults.

UFIT models one type of faults at a time. We can also inject multiple fault types into the model by giving the fault-affected model obtained from UFIT to UFIT and inject a new type of faults.

Chapter 7

Model Slicing Timed Automata Models

In this chapter, we present a model slicing technique for reducing the verification time and space of fault-free and fault-impacted timed automata models extracted from SystemC TLM programs. Specifically, we focus on the use of model slicing considered in [51] to slice timed automata models. For this purpose, first, we explain a brief history of model slicing. Then, we introduce a running example that we utilize to explain our model slicing technique. Finally, we discuss our slicing algorithm and illustrate it with the running example.

7.1 Model Slicing

Program slicing is a source code analysis and manipulation technique, in which a subprogram is identified based on a user-specified slicing criterion. The criterion captures the point of interest within the program, while the process of slicing consists of following dependencies to locate those parts of the program that may affect the slicing criterion [78]. Program slicing has been successfully applied in the context of model checking of untimed systems. Millett and Teitelbaum [63] study slicing of Promela models and propose the so called *imprecise slice*. However, they do not formalize their slicing methods. Hatcliff *et al.* [45] present a formal study of slicing sequential programs preserving LTL and extend their techniques to concurrent Java programs [71]. Slicing is also present in the *IF* project [16] concerning timed systems. Nonetheless, it is defined for its untimed subset only [15].

The closest related work on using static analysis in timed system verification concerns the concept of *influence information* [17]. The technique can be understood as slicing I/O Timed Components, timed safely automata extended with interfaces. The approach does not use the notion of slicing criterion, and, instead, it preserves the branching structure of a transition system up to the propositional assignment given over the external observer. Another related methods are the *active-clock* reduction technique [22] and more general *relevant guard abstraction* [8] for timed safety automata. Since they focus on clocks reduction they are orthogonal to ours and can be combined with it. Finally, Janowska and Janowski in [50] target the slicing in the context of timed systems considering reduction of intermediate language of Verics [50]. The formalism is a specification language with no explicit clock variables, but restricting the time of transitions executions by means of delays.

7.2 The Running Example: The Alternating Bit Protocol

In this section, we present a version of the well-known *alternating bit protocol* [7] as a running example to explain our slicing technique better. This protocol provides reliable communication over a network service that sometimes looses messages. It uses a one-bit sequence number (which alternates between 0 and 1) in each message and an acknowledgment to determine whether the message must be retransmitted. The system consists of three automata running in parallel, *Sender*, *Receiver* and *Faulty-Buffer*, as shown in Figures 7.1, 7.2, and 7.3.

The Sender transmits portions of data, which represents some computations performed in real systems. In our example they are succeeding numbers modulo N. Sender sends each portion accompanied with the bit to Faulty-Buffer. Then it waits for an acknowledgment. If the value of the acknowledgment is the same as the value of the bit, then the message is treated as delivered and the

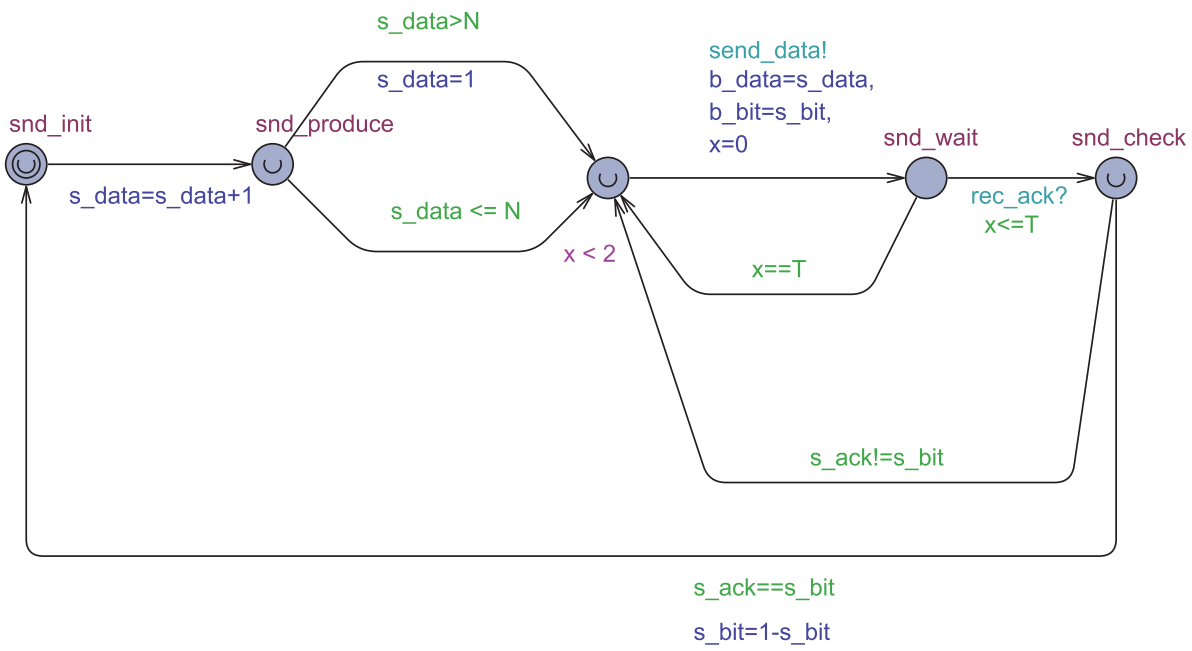

Figure 7.1: The Sender automaton for the alternating bit protocol.

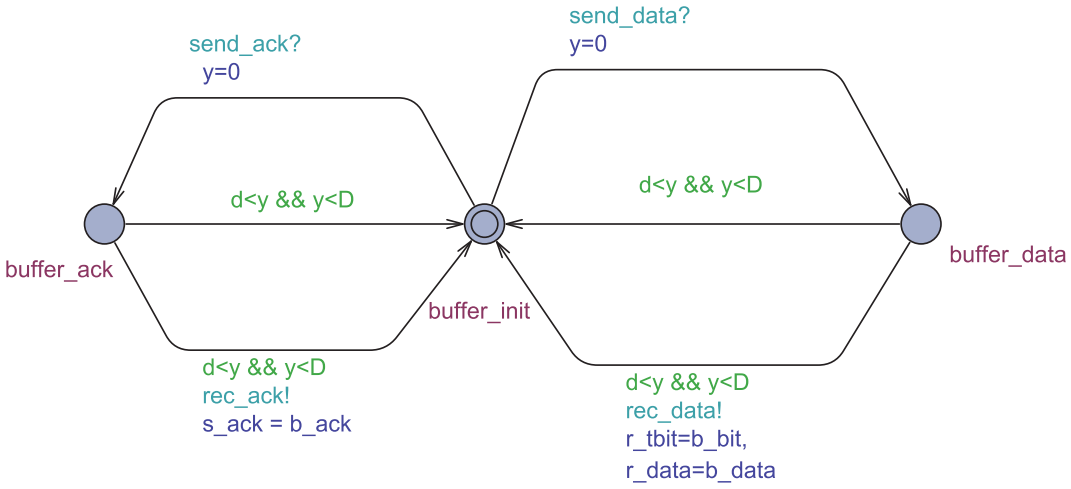

Figure 7.2: The Faulty Buffer automaton for the alternating bit protocol.

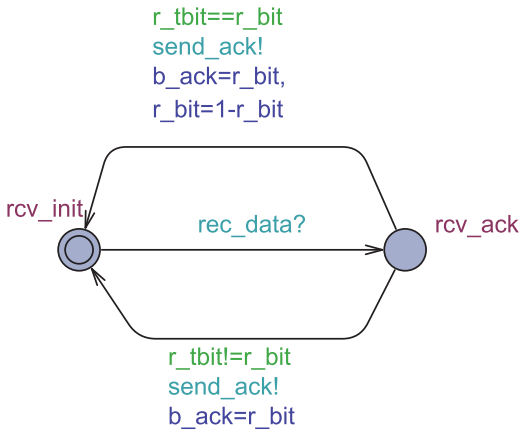

Figure 7.3: The Receiver automaton for the alternating bit protocol.

value of the bit flips. Otherwise the message is retransmitted. The message is also retransmitted, if no acknowledgment comes within T time units (the timeout is modeled by the clock x). Receiver waits until it gets the message from Faulty-Buffer, then it acknowledges receipt of the message and compares its sequence number with the bit value. If they are equal, it changes the value of the bit and accepts the message. Otherwise it waits for another message. Faulty-Buffer accepts a message from Sender or an acknowledgment from Receiver and forwards it respectively or looses it. The clock y is used to model transmission delays, which are between d and D time units.

For the alternating bit protocol, we define a property ϕ as follows:

$$\phi = A[]\ (Sender.snd\_init \rightarrow s\_bit == r\_bit)$$

This property verifies that if Sender is in the location s_init, the value of Sender's bit is equal to the value of Receiver's bit. The important point to note here is that this property does not depend on the value of variables s_data, b_data, and r_data.

7.3 UPPAAL Timed Automata Model Slicing

In this chapter, we explain our model slicing algorithm. Our slicer gets an UPPAAL timed automata model along with a set of properties as inputs and generate a sliced version of the timed automata model as as output. Our slicer, similar to [51], uses a two step approach . In the first step (Algorithm 2), the slicer identifies the locations, say L, and actions, say A (including *guards*, *updates*, and *assignments*), that need to be preserved in the sliced automata. This is a recursive procedure where the initial set of states that need to be preserved are determined by the property under consideration, say ϕ (Lines 1-2 in Algorithm 2). For example, if the property under consideration is p *leadsto* q then a location that accesses p and q must be preserved in the sliced automata. Next, we explain these two steps in detail.

Algorithm 2 Timed Automata Model Slicing

Input: UPPAAL model $M = (Q, q^0, X, T)$, property ϕ;
Output: Sliced UPPAAL model M';

1: L_{init} = locations in ϕ and their immediate predecessors;
2: A_{init} = enabling actions defining variables in ϕ;
3: $L := L_{init}$; $A := A_{init}$;
4: **while** (L or A gets updated) **do**
5: Utilize the dependencies to update L and A; **end while**
6: **return** $M' =$ slicer-builder(L, A, M);

7.3.1 Identifying the set of relevant locations and actions (L and A)

In order to identify L and A, first, our algorithm needs to check if the model contains any *arrays* and *functions*. Intuitively, the UPPAAL model may have two types are arrays: (a) an array of automata and (b) an array of variables. When we have an array of automata with n entries then essentially, we replace it by n different automata. In each automaton, we need to replicate local variables but the global variables remain the same in all n automata. Similarly, for handling an

array of variables with n entries, we replace it by n different variables. Subsequently, we need to replace every entry in every automaton that uses the array so that the array reference is replaced by the appropriate variable. It also requires replicating the local variables in each automaton. This is acceptable since UPPAAL already does this in the verification process.

Regarding functions, we consider the syntactic code involved in each function to identify variables that are accessed during that function. Since our goal is to slice the model, we do not need to evaluate the function (this would be done by UPPAAL as part of verification). Instead, we need to identify if the function is accessing/changing any variables of interest. This can potentially introduce some false dependencies, i.e., dependencies that do not exist in reality but are suspected by the slicer. However, this is acceptable as well since any errors caused in this fashion would result in a larger (but still correct) model.

After considering arrays and functions, our algorithm utilizes the property under consideration and generates the set of *initial relevant locations* and *initial relevant actions*, say L_{init} and A_{init}, respectively. The L_{init} consists of the locations in Φ and their immediate predecessors. The A_{init} consists of all the actions that update any of the variables included in Φ. As an instance, for the alternating bit protocol in Figures 7.1, 7.2, and 7.3, $A_{init} = \{s\_bit = 1 - s\_bit, r\_bit = 1 - r\_bit\}$ and $L_{init} = \{s\_init, s\_check\}$.

Subsequently, the slicer identifies additional locations, variables, guards, and statements that need to be preserved. The reasons for preserving additional details in the sliced model include (1) control dependency, (2) data dependency, and (3) time dependency (Line 5 in Algorithm 2). As an illustration of control dependency, assume that location q_1 is preserved in previous iteration. Now, if the UPPAAL model includes a state such as q_2 such that (1) there is a computation from q_2 that reaches q_1 and (2) there is a computation from q_2 that never reaches q_1. Then, q_2 must also be preserved since we need to know whether the path followed from q_2 will reach q_1 or not.

And, deciding whether q_1 is reached or not *can* affect satisfaction (or violation) of the property of interest. As an illustration of time dependency, consider the case where q_1 is preserved in the previous iteration. Suppose there is a path from q_2 to q_1 and the *time spent* in state q_2 can be nonzero then q_2 must also be preserved. (By definition, time spent in states that are marked urgent in UPPAAL is 0.)

Algorithm 3 Slice Builder

Input: The sets of locations L and actions A, and Model M;
Output: Sliced timed automata model $M' = (Q', q^{0'}, X', T')$;
1: $Q' = R$;
2: **if** $(q^0 \in L)$ **then** $q^{0'} = q^0$;
3: **else** $q^{0'} =$ the first reachable location in L from q^0; **end if**
4: $T' = \bigcup out(L)$ s.t. action of each $out(L) \in A$;
5: **if** $(target(out(L)) \notin L)$ **then**
6: $target(out(L)) =$ the first reachable location in L; **end if**
7: **return** M';

7.3.2 Building the sliced model

When the set of relevant locations and actions (L, A) is ready, in the second step, the slicer builds a revised model that only includes the relevant locations and actions (Algorithm 3). While building the sliced model, if the initial location of an automaton is not included in L, the first reachable location in L becomes the new initial state (Lines 2-3 in Algorithm 3). Also, if the target of an outgoing transition of a location in L is not included, the first reachable location in L becomes the target of that outgoing transition (Lines 5-4 in Algorithm 3). As an illustration, consider Figure 7.4 where q_1 and q_4 are relevant locations that need to be preserved and q_2 and q_3 are locations that are not relevant. In that case, the outgoing transition of q_1 goes into q_4. Moreover, the actions of each transition are those which are included in A (Line 4 in Algorithm 3). The proof of correctness of the model slicing approach is discussed in [51].

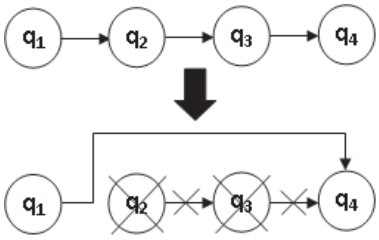

Figure 7.4: Building the sliced model.

The set of locations of each automaton of the sliced system consists of relevant locations of the original automaton. If an automaton has no relevant locations, it means that the whole automaton is not relevant in context of considered properties. The set of variables of the slice consists of variables of the original system which appear in relevant operations. In fact, the only clocks that are reduced are used exclusively to ensure that time cannot progress in some locations. For each automaton the set of transitions is composed of transitions of the original automaton going out of relevant locations. If an original transition goes to a non relevant location, then the target of its counterpart in the slice is the relevant location to which an invisible path exits. It can be shown that for a relevant location and each of its outgoing transitions there exists exactly one such location.

7.4 Applying the Model Slicing on the Alternating Bit Protocol

Let us present how our algorithm works for our example. The construction of the sets L and A starts with the initial sets L_{init} and R_{init}. Hence, at the beginning $A = \{s\_bit = 1 - s\_bit, r\_bit = 1 - r\_bit\}$ and $L = \{s\_init, s\_check\}$. According to Lines 4 and 5 of the algorithm, the following operations are added to set A: $s\_ack == s\_bit$, $(s\_ack! = s\_bit)$, $r\_tbit == r\_bit$, and $r\_tbit! = r\_bit$. The operations $s\_ack == s\_bit$ and $s\_ack! = s\_bit$ are added since the location s_init is in L and control depends on the location s_check. The operations $r\_tbit == r\_bit$ and $r\_tbit! = r\_bit$ are added since the operation $(r\_bit = 1 - r\_bit)$ is in A.

Then, set A is successively augmented by the operations that depend on the operations currently included in A. These newly augmented operations are $s\_ack = b\_ack$, $r\_tbit = b\_bit$, $b\_bit = s\_bit$, and $b\_ack = r\_bit$. The operation $s\_ack = b\_ack$ is added since $s\_ack == s\_bit$ is included in A. Likewise, the operation $r\_tbit = b\_bit$ is added as $r\_tbit == r\_bit$ is included in A. Also, the operation $b\_bit = s\_bit$ is added as the operation $r\_tbit == b\_bit$ is included in A. Finally, the operation $b\_ack = r\_bit$ is added since the operation $s\_ack == b\_ack$ is included in A.

Next, the following locations are added to set L: snd_send, rcv_ack, $buffer\_data$ and $buffer\_ack$. These locations are added to L since their outgoing transitions contain operations that are included in A. Additionally, the algorithm adds the locations snd_wait, rcv_init, and $buffer\_init$ to set L since there are locations currently include in L that are time dependent on them. Finally, the second iteration does not change any of the sets A and L and, as a result, the loop ends.

When the sets of relevant locations and actions, L and A, are ready, we utilize Algorithm 3 to build the sliced timed automata model. The automata built for Sender and Faulty Buffer automata are shown in Figures 7.5 and 7.6 respectively. In Sender automaton (Figure 7.5), the location *snd_produce* disappears in the sliced version since it appears to be *non-relevant* as no location depends on it. Also, there are no variables *s_data*, *r_data* and *b_data* in either Sender automaton or Faulty Buffer automaton (Figure 7.6) as they do not occur in any of relevant actions. Additionally, the Receiver automaton remains the same since none of the aforementioned non-relevant variables is used in this automaton.

7.5 Summary

In this chapter, we specified our modeling slicing technique and explained it with a running example. Additionally, We identified how to conduct functions and arrays in a timed automata model.

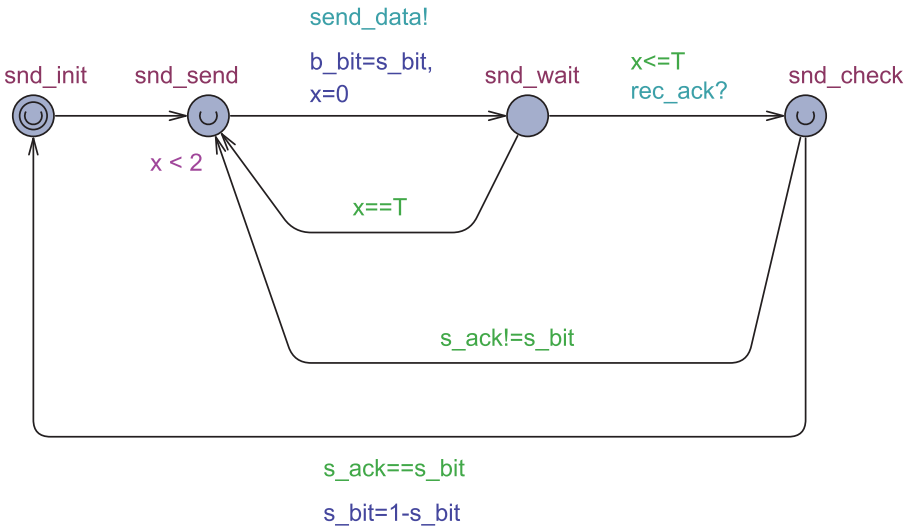

Figure 7.5: The *sliced* Sender automaton for the alternating bit protocol.

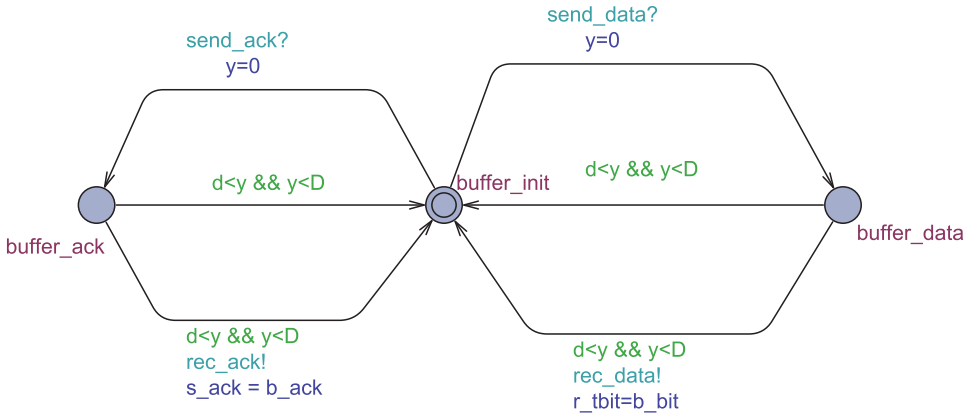

Figure 7.6: The *sliced* Faulty Buffer automaton for the alternating bit protocol.

We have also developed a tool for our model slicing technique which will be introduced in the next chapter.

Chapter 8

USlicer: A Tool for Model Slicing UPPAAL Timed Automata Models

In this chapter, we present the tool USlicer (*Uppaal Slicer for timed automata*) and explain its effectiveness on verifying timed automata models. Given the fault-free or fault-affected timed automata model along with a set of properties, USlicer generates a sliced version of the model based on the property under consideration. Our results show that, in some cases that the verification of the model is not possible due to complexity, utilizing USlicer helps us to make the verification possible in a reasonable time and space.

8.1 Internals of USlicer

USlicer targets UPPAAL timed automata models and slices them based on a set of properties of interest. It is written in *Python* and its source code is publicly available. The input of USlicer is a timed automata model in *XML* format. For parsing the XML file, we utilize *XML ElementTree* library of python. The *Element* type is a flexible container object, designed to store hierarchical data structures, such as simplified XML infosets, in memory. The ElementTree wrapper type adds code to load XML files as trees of Element objects, and save them back again. Next, we explain the XML file that USlicer and UPPAAL tool-set accept as an input in some detail.

8.1.1 XML format

XML is a markup language that is used to describe data. The basic building block of an XML file is an element, defined by tags. An XML file that represents an UPPAAL timed automata model contains the following main tags:

- "$< location >$ *statements* $< /location >$" and
- "$< transition >$ *statements* $< /transition >$".

The former shows the locations and the latter represents the transitions of the timed automata model. Also, the *statements* can be a *name*, an *invariant*, or a *type* (e.g., urgent, committed) for locations, and a *source*, a *target*, or *labels* for transitions. The source and target tags represent the position of the transition. The label tag shows whether the transition has a *synchronization* channel, an *assignment* operation, and/or a *guard* condition. USlicer utilizes *XML ElementTree* library to parse the XML file of the given program.

As explained in Algorithms 2 and 3 in Chapter 7, USlicer utilizes a 2-step approach for slicing the timed automata model. To evaluate the effectiveness of USlicer, we consider two case studies in the following sections. The first case study is based on the producer-consumer program and is conducted in Loosely-Timed (LT) coding style. Such a style of coding heavily relies on a blocking transport interface *b_transport()*. The second case study is based on the memory-mapped buses and is conducted in Approximately-Timed (AT) coding style. In this style of coding, designers benefit from a non-blocking transport interface *nb_transport()*. In general, the blocking transport interface is only able to model the start and end of a transaction, whereas the non-blocking interface allows a transaction to be broken down into multiple timing points.

8.2 Case Study 1: Producer-Consumer Program

In this example, a producer and a consumer communicate through a blocking transport. The producer generates a piece of data, puts it into a shared fixed-size (3 here) buffer and waits for the consumer to consume the data. When the data is consumed, the producer generates the next piece of data. Given the SystemC TLM program of this example, first, we extract the timed-automata model (as explained in Chapter 4). To ensure that the timed-automata model captures the requirements of the TLM program, we specify the following properties/requirements that should hold in the absence of faults:

```
LT1: E<> producer.writenBuff
LT2: producer.start --> producer.end
LT3: A[] (producer.writenBuff && consumer.readBuffer)
        imply  WriteIndex == ReadIndex
LT4: A<> (WriteIndex == ReadIndex))
LT5: E<> consumer.readBuffer
LT6: A[] (WriteIndex==ReadIndex || WriteIndex == (ReadIndex+1)%n)
```

The first property shows that the producer eventually generates some data. The second property represents that when the producer starts generating some data, the data will be eventually consumed by the consumer and the producer can start generating the next piece of data. The third property ensures that consumer consumes the data which is currently generated by the producer and the consumer won't try to remove data from an empty buffer. The fourth property shows that always consumer consumes the data generated by the producer. The fifth property represents that the consumer eventually consumes the data. Finally, the last property illustrates that the consumer's and producer's indices are never more than one apart. We have model checked these properties

using UPPAAL and the results are available in Table 8.1. For the model checking, we use a personal computer with quad core CPU (2.8 GHZ each) and 6 GB memory. Next, we compare the verification time and memory usage for verifying the above properties of the timed automata model and its sliced model in the absence and presence of faults.

Property	Original Model				Sliced Model			
	Verification Time (ms)	Memory Usage (KB)	No. of states	No. of variables	Verification Time (ms)	Memory Usage (KB)	No. of states	No. of variables
LT1	55	29,288	117	90	40	20,532	85	29
LT2	812	32,892	117	90	187	22,212	106	31
LT3	312	33,985	117	90	5	21,888	12	7
LT4	313	33,966	117	90	4	21,876	10	7
LT5	57	30,015	117	90	41	20,532	85	29
LT6	311	33,985	117	90	5	21,521	12	7

Table 8.1: Comparison of the original and sliced models in the absence of faults while using LT coding style.

Fault	Location	Property	Original Model				Sliced Model			
			status v/s	Verification Time (ms)	Memory Usage (KB)	No. of variables	status s/v	Verification Time (ms)	Memory Usage (KB)	No. of variables
Fail-stop	Consumer	LT1	s	55	30,112	91	s	39	20,535	30
Fail-stop	Consumer	LT2	v	45	33,023	91	v	40	21,221	32
Fail-stop	Consumer	LT3	s	335	35,654	91	s	5	21,810	8
Fail-stop	Consumer	LT4	v	26	35,361	91	v	1	21,093	8
Fail-stop	Consumer	LT5	v	35	30,112	91	v	32	20,435	30
Fail-stop	Consumer	LT6	v	54	34,120	91	v	1	21,354	8
Msg-loss	Producer	LT1	v	48	30,855	92	v	40	20,615	31
Msg-loss	Producer	LT2	v	51	32,102	92	v	45	22,333	33
Msg-loss	Producer	LT3	s	344	36,342	92	s	5	24,109	9
Msg-loss	Producer	LT4	v	15	36,345	92	v	1	23,021	9
Msg-loss	Producer	LT5	v	48	30,855	92	v	40	20,615	31
Msg-loss	Producer	LT6	s	381	34,350	92	s	5	24,109	9

Table 8.2: Comparison of the original and sliced models in the presence of faults while using LT coding style.

8.2.1 Slicing in the absence of faults

Once we have the fault-free timed automata model, we use the model and properties provided above to slice the model. Consider that we do not use UFIT since we want to study the model in

the absence of faults. For each property, we generate a sliced model and compare the verification time, memory usage, number of states, and number of variables of the original/fault-free model and the sliced model generated by our model slicer. We observe that our slicing technique helps to simplify the model and reduce the time and memory needed for verifying the properties (see Table 8.1). For example, for verifying property $LT3$, the verification time, memory usage, number of states, and number of variables are reduced by 98%, 35%, 89%, and 92% respectively.

8.2.2 Slicing in the presence of faults

To study the model in the presence of faults, we consider two types of faults in this example: (1) fail-stop faults, where a module fails functionally and the other modules cannot communicate with it, and (2) message faults, where a message may be lost while forwarding from one module to another. We utilize UFIT to inject these faults into the fault-free model generated by STATE. For fail-stop, we consider the scenarios where the consumer fails and is not able to consume any data from the buffer. For the message faults, we assume that the messages may get lost while the producer is writing them into the buffer. Table 8.2 represents the results for verifying the original model and its sliced model in the presence of faults. We do not include the number of states in this table since UFIT does not introduce new states into the model. We notice that the verification time for finding the violation, memory usage, and the number of variables in the sliced models are reduced by 11%–99%, 29%–32%, and 66%–92% respectively compare to those in the original model. Consider that, when the property under verification is violated in the presence of faults, the verification time may be smaller than that in the original model since the verification is terminated upon finding the violation.

8.3 Case Study 2: Memory-Mapped Buses

In this section, we present an example that utilizes AT coding style for modeling an on-chip memory-mapped communication buses between an initiator module and a target/memory module. In this example, adapted from [1], the initiator and the memory modules communicate through a non-blocking transport. The non-blocking transport is implemented according to the TLM base protocol, i.e., it breaks down each transition into four phases, namely *Begin_Req*, *End_Req*, *Begin_Resp*, and *End_Resp*, where each phase in a transition is associated with a timing point. Moreover, in an AT coding style, each module has a queue called Payload Event Queue (PEQ). The PEQ is a time-ordered list of event notification in the TLM model. Utilizing STATE, we generate the timed automata model from the given SystemC TLM program (as explained in Chapter 4). We also define a set of properties to ensure that the generated model is correct in the absence of faults. These properties are as follows:

```
AT1: E<> Init.SentBeginReq and Memory.RcvdBeginReq
AT2: Initiator.SentBeginReq --> Memory.RcvdBeginReq
AT3: A[] (Initiator.sentBeginReq && request_in_progress ==0)
                imply (Memory.SentEndReq or Memory.SentBeginResp)
AT4: (Memory.SentEndReq or Memory.SentBeginResp) --> (Init.EndResp)
AT5: E<> Init.EndResp
AT6: scheduler.inititate --> scheduler.execute
```

The first property represents that the initiator eventually initiates a transaction and the memory eventually receive it. The second property shows that whenever the initiator starts a transaction, the memory module will eventually receive it. The third property ensures that if the initiator has sent a transaction and the PEQ is empty, the memory is in a state where either the *End_Req* message

or the *Begin_Resp* message has been sent. In addition, if the memory sends a response with either *End_Req* or *Begin_Resp* phases, the initiator will eventually be able to finish the transaction by sending *End_Resp*. This is shown in the fourth property. The fifth property shows that at least one of the transactions will be executed completely and the initiator will eventually send a message with an *End_Resp* phase. Finally, the last property represents that the scheduler eventually executes some process. Next, we compare the time and memory needed for verifying these properties in the absence and presence of faults.

8.3.1 Slicing in the absence of faults

We use UPPAAL tool-set to verify the above properties on the same personal computer as that in Section 8.2. However, we are not able to verify properties $AT2$, $AT3$, $AT4$, and $AT6$ since the model generated by STATE is too complex and the computer runs out of memory while verifying those properties (see Table 8.3). Also, the memory needed to verify $AT1$ and $AT5$, which are only reachability properties, is 0.99 GB. Therefore, we utilize our slicing technique to simplify the model based on the properties given. Using UPPAAL, we are able to verify all the properties in the sliced models and check if they are satisfied (s) or violated (v). For example, the verification of property $AT3$ in the corresponding sliced model takes 1 s and 476 ms, and the memory usage is 51.5 MB. Also the number of variables needed for verifying this property in the sliced model in 50, which is reduced by 81%.

8.3.2 Slicing in the presence of faults

We utilize UFIT to inject message and fail-stop faults into the timed automata model generated by STATE. Regarding the fail-stop faults, we consider the scenarios where the memory module is

failed and the initiator module is not able to communicate with it. Since injecting the faults into the model makes the model more complex, verification of some properties (i.e., $AT1$ and $AT4$) is not feasible. Therefore, we give the fault affected model and the desirable property to the slicer, and the slicer generates a simplified model based on the property. Surprisingly, we are able to verify all the properties mentioned above in the sliced models (see Table 8.4). As an illustration, verification of property $AT4$, which was not feasible in the original model, takes 1 s and 250 ms and needs 49.9 MB memory. Also, the number of the variables in the sliced model is reduced by 79%.

In order to model the message faults, we assume that the messages with *Begin_reg* phase may get lost when the initiator is forwarding them to the memory module. Having this fault injected to the model, we are able to verify all the above properties in the sliced models (see Table 8.4). For instance, verifying property $AT2$ takes 201 ms and need 43.9 MB memory in the sliced model. This verification has reduced the time and memory usage by 14% and 96% respectively.

Property	Original Model				Sliced Model			
	Verification Time (ms)	Memory Usage (KB)	No. of states	No. of variables	Verification Time (ms)	Memory Usage (KB)	No. of states	No. of variables
AT1	2,212	991,765	188	276	350	38,980	122	43
AT2	N/A	N/A	188	276	821	43,950	130	51
AT3	N/A	N/A	188	276	1,476	51,509	137	50
AT4	N/A	N/A	188	276	1,250	49,898	136	57
AT5	2,643	994,592	188	276	354	38,875	121	43
AT6	N/A	N/A	188	276	815	43,657	129	47

Table 8.3: Comparison of the original and sliced models in the absence of faults while using AT coding style.

8.4 Summary

In this chapter, we introduced our model slicer, called USlicer, which is developed using the algorithms explained in the last chapter. We studied the effectiveness of USlicer on two case studies.

Fault	Location	Property	Original Model				Sliced Model			
			status v/s	Verification Time (ms)	Memory Usage (KB)	No. of variables	status s/v	Verification Time (ms)	Memory Usage (KB)	No. of variables
Fail-stop	Memory	AT1	v	180	655,950	277	v	150	38,910	44
Fail-stop	Memory	AT2	v	255	898,750	277	v	200	43,990	52
Fail-stop	Memory	AT3	v	282	1,350,746	277	v	256	51,656	51
Fail-stop	Memory	AT4	N/A	N/A	N/A	277	s	1,266	49,910	58
Fail-stop	Memory	AT5	v	187	656,870	277	v	152	38,990	44
Fail-stop	Memory	AT6	N/A	N/A	N/A	277	s	817	43,670	48
Msg-loss	Initiator	AT1	v	160	655,950	278	v	155	38,910	45
Msg-loss	Initiator	AT2	v	235	1,165,655	278	v	201	43,990	53
Msg-loss	Initiator	AT3	N/A	N/A	N/A	278	s	1,480	51,721	52
Msg-loss	Initiator	AT4	N/A	N/A	N/A	278	s	1,252	49,923	59
Msg-loss	Initiator	AT5	v	165	657,750	278	v	155	38,992	45
Msg-loss	Initiator	AT6	v	217	899,677	278	v	195	43,673	49

Table 8.4: Comparison of the original and sliced models in the presence of faults while using AT coding style.

In each case study, we studied three types of properties: reachability ($LT1$, $LT5$, $AT1$, and $AT5$), liveness ($LT2$, $LT4$, $AT2$, $AT4$, and $AT6$), and safety ($LT3$, $LT6$, and $AT3$) properties. In the LT coding style, in general, verification times are small since the LT models are efficient in nature. In spite of this, the verification time was reduced by 11%–99%. Nevertheless, slicing the AT case study was essential since we were not able to verify any of the liveness or safety properties in the original model. The only type of property we could verify was reachability property since the verification terminates upon finding the first solution in verifying such properties. By contrast, with the help of slicing, it was possible to verify all properties of interest in a reasonable time. The speedup associated with verification of safety and liveness properties was substantial. For example, the property (speedup) combination in our case studies was $LT1$ (1.375), $LT2$ (4.34), $LT3$ (62.4), $LT4$ (78.25), $LT5$ (1.39), and $LT6$ (62.2). The slicing was especially effective with AT models since verification of certain properties ($AT2$, $AT3$, $AT4$, and $AT6$) was impossible without slicing. Hence, we anticipate that slicing would be essential for AT models where verification without slicing is impossible even for simple examples. In case of LT models, verification without slicing was possible. However, the reason for considering this example was to quantify the benefit of slicing. (AT models do not provide an opportunity to quantify this benefit since verifi-

cation time without slicing is essentially ∞.) In LT models, slicing improved the verification time substantially. We anticipate that slicing would be especially beneficial for larger LT models where verification without slicing is impossible.

Consider that, the programs considered in our case studies are the most *optimal* in terms of the (SystemC) source code and, hence, slicing algorithms will not change them. What we discussed in this chapter is that it is possible to reduce the cost of verification further in these contexts via slicing the timed automata models extracted from the SystemC programs. It follows that one can utilize existing methods to slice the SystemC program to obtain the *smallest* SystemC code and then utilize our approach to reduce the verification time and space of that *smallest program*.

Chapter 9

Conclusion and Future Work

The rise in complexity, size and heterogeneity of modern embedded system designs has pushed modeling to new abstraction levels above RTL. Transaction Level Modeling using SystemC has emerged as a new paradigm for system modeling. On the other hand, SoC design is being adapted to combine the best features of top down and bottom up system design. Although the models in SystemC TLM are designed carefully, their verification is an important task. Moreover, many industrial and academic institutions support and use SystemC and Transaction Level Modeling for software/hardware co-design. Thus, a systematic (and possibly automatic) approach for verification of SystemC TLM programs has a significant impact. To have a systematic method for verification of such programs, in this dissertation, we presented a framework that crosscuts several fields such as compilers, verification, fault tolerance, and model checking. In particular our framework involved five main steps, namely *defining formal semantics*, *model extraction*, *fault modeling*, *model slicing*, and *model checking*. The first two steps obtain an abstract model of the SystemC program. We chose Promela and UPPAAL timed automata as the target modeling languages since they allowed us to evaluate the effect of faults with the model checkers SPIN [48] and UPPAAL [9]. We considered two types are coding style, Loosely-Timed (LT) and Approximately-Timed (AT), in our framework. Targeting Promela models, we proposed a set of transformation rules that help us extract a Promela model from a Loosely-Timed SystemC TLM programs. However, the Promela models extracted are untimed. Regarding UPPAAL timed automata models, we consider the notion of time and target both Loosely-Timed and Approximately-Timed coding

styles. In particular, we propose a set of transformation rules for extracting timed automata models from Loosely-Timed programs. We also utilized a tool, called STATE, for extracting timed automata models from Approximately-Timed SystemC TLM program.

Subsequently, in the third step, we augmented the extracted model with faults. This step requires us to model the impact of faults on SystemC TLM programs and capture them in the context of Promela [3] or timed automata [5]. We studied four different types of faults in this dissertation namely, message faults, permanent faults, transient faults, and timing faults. We proposed a tool, called UFIT, that models the aforementioned faults in UPPAAL timed automata models and generates a fault-affected model.

The models extracted from the SystemC TLM programs are mostly complex and get even more complex after injecting faults into them. Hence, in the fourth step, we proposed a model slicing technique for slicing the timed automata models. We developed a tool, called USlicer, that gets a timed automata model along with a property of interest and generates a simplified version of the model. This step improves the verification time and space. Finally, in the last step, we model check the model and study the behavior of the models in the presence of faults.

We illustrated our framework with several case studies. These case studies covered programs that utilized LT and AT coding styles. In each case study, we extracted either the Promela model or the timed automata model from the given SystemC TLM program. Thereafter, we modeled different types of faults and injected the faults into the models. In particular, We analyzed the untimed extracted Promela models in the presence of communication faults. Since transaction level modeling is based on the principle of separating inter-component communications from computations using the notion of transactions, designing fault-tolerant communication protocols is fundamental to transaction level modeling. This example illustrates the role of our framework in dealing with faults that occur in such inter-component communications. A similar approach can also be easily

applied to other communication errors in such applications. We also analyzed the timed UPPAAL models in presence of all four types of faults. We were either able to verify that the original specification is satisfied or find a counterexample demonstrating the violation of the original specification. Moreover, the time for evaluating the effect of faults was comparable (0-57%) to the verification in the absence of faults. We also used USlicer to slice the timed automata models and improve the verification time and space. In the LT coding style, in general, verification times are small since the LT models are efficient in nature. Hence, although utilizing model slicing improves the time and space efficiency, it is not essential. Nevertheless, slicing the AT case study was essential since we were not able to verify any of the liveness or safety properties in the original model. The only type of property we could verify was reachability property since the verification terminates upon finding the first solution in verifying such properties. Also, the sliced model could be verified in a reasonable time both in the fault-free and fault-affected models. This case study illustrates one of the main advantages of using our model slicing technique where the slicing enabled verification whereas the original model was too large to verify.

9.1 A roadmap for future research.

We propose several directions for future research on this dissertation.

- We plan to extend our previous work on automated addition of fault tolerance [24, 54] in order to automate the design of fault tolerance in the extracted Promela and UPPAAL models. In addition to facilitating the design of fault tolerance, we would like to enable *fault-containment* mechanisms, where designers can guarantee that faults do not get propagated to several components at once. This information can be used to add restrictions on the communication amongst components to ensure compliance with this requirement.

- We will extend our work with a set of reverse transformation rules to ensure that the code added to models in order to capture fault tolerance can indeed be realized in SystemC program. For example, we need rules that specify how atomic recovery actions will be captured in SystemC while preserving atomicity and recovery. One way to achieve this is to refine the atomic actions to a code block between two wait statements in SystemC. This rule relies on the fact that the scheduler of SystemC simulator has a run-to-completion policy for context switching.

- We will investigate different scenarios under which fault tolerance functionalities added to models can be partitioned and assigned to software and hardware. One possibility is to use the rule that any fault tolerance functionality that can be executed asynchronously with the rest of the model can be captured as a software component, whereas synchronous functionalities can be included in hardware. Nonetheless, the decision of including a piece of new functionalities in hardware/software may depend upon other factors such as timing issues, energy consumption, overall system modularity, etc. We plan to investigate the impact of such factors on co-design of fault tolerance.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Getting started with tlm-2.0. http://www.doulos.com/knowhow/systemc/tlm2/.

[2] Open SystemC Initiative (OSCI): Defining and advancing SystemC standard IEEE 1666-2005. http://www.systemc.org/.

[3] Spin language reference. http://spinroot.com/spin/Man/promela.html/.

[4] Transaction-Level Modeling (TLM) 2.0 Reference Manual. http://www.systemc.org/downloads/standards/.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[6] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the Real-Time Systems Symposium. Raleigh-Durham, NC, December 1993*, pages 2–11, 1993.

[7] Keith A. Bartlett, Roger A. Scantlebury, and Peter T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–261, 1969.

[8] Gerd Behrmann, Patricia Bouyer, Emmanuel Fleury, and Kim Guldstrand Larsen. Static guard analysis in timed automata verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 254–277, 2003.

[9] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Bernardo and Corradini [12], pages 200–236.

[10] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*, pages 232–243, 1995.

[11] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in*

September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned], pages 87–124, 2003.

[12] Marco Bernardo and Flavio Corradini, editors. *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*. Springer, 2004.

[13] Gérard Berry. The foundations of esterel. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 425–454. The MIT Press, 2000.

[14] Nicolas Blanc and Daniel Kroening. Race analysis for SystemC using model checking. *ACM Transactions on Design Automation of Electronic Systems*, 15(3):21:1–21:32, 2010.

[15] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. Using static analysis to improve automatic test generation. In *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, pages 235–250, 2000.

[16] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The if toolset. In Bernardo and Corradini [12], pages 237–267.

[17] Víctor A. Braberman, Diego Garbervetsky, and Alfredo Olivero. Improving the verification of timed systems using influence information. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pages 21–36, 2002.

[18] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[19] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Springer, New York, USA, 2011.

[20] Yung-Yuan Chen, Chung-Hsien Hsu, and Kuen-Long Leu. SoC-level risk assessment using FMEA approach in system design with SystemC. In *International Symposium on Industrial Embedded Systems*, pages 82–89, 2009.

[21] Alessandro Cimatti, Alberto Griggio, Andrea Micheli, Iman Narasamdya, and Marco Roveri. KRATOS: A software model checker for SystemC. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, pages 310–316, 2011.

[22] Conrado Daws and Sergio Yovine. Reducing the number of clock variables of timed automata. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), December 4-6, 1996, Washington, DC, USA*, pages 73–81, 1996.

[23] Rolf Drechsler and Daniel Große. Reachability analysis for formal verification of systemc. In *2002 Euromicro Symposium on Digital Systems Design (DSD 2002), Systems-on-Chip, 4-6 September 2002, Dortmund, Germany*, pages 337–340, 2002.

[24] Ali Ebnenasir. *Automatic Synthesis of Fault Tolerance*. PhD thesis, Michigan State University, 2005.

[25] Ali Ebnenasir, Reza Hajisheykhi, and Sandeep S. Kulkarni. Facilitating the design of fault tolerance in transaction level systemc programs. In *Distributed Computing and Networking - 13th International Conference, ICDCN 2012, Hong Kong, China, January 3-6, 2012. Proceedings*, pages 91–105, 2012.

[26] Ali Ebnenasir, Reza Hajisheykhi, and Sandeep S. Kulkarni. Facilitating the design of fault tolerance in transaction level systemc programs. *Theor. Comput. Sci.*, 496:50–68, 2013.

[27] Antonio da Silva Farina and Sebastián Sánchez Prieto. On the use of dynamic binary instrumentation to perform faults injection in transaction level models. In *Proceedings of the 2009 Fourth International Conference on Dependability of Computer Systems*, pages 237–244, 2009.

[28] Alessandro Fin, Franco Fummi, Maurizio Martignano, and Mirko Signoretto. SystemC: A homogenous environment to test embedded systems. In *Proceedings of the ninth international symposium on Hardware/software codesign*, CODES '01, pages 17–22, 2001.

[29] Beltra Giovanni, Cristiana Bolchini, and Antonio Miele. Multi-level fault modeling for transaction-level specifications. In *Proceedings of the 19th ACM Great Lakes symposium on VLSI*, pages 87–92, 2009.

[30] Daniel Große and Rolf Drechsler. Checkers for systemc designs. In *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*, pages 171–178, 2004.

[31] Daniel Große and Rolf Drechsler. Checksyc: an efficient property checker for RTL systemc designs. In *International Symposium on Circuits and Systems (ISCAS 2005), 23-26 May 2005, Kobe, Japan*, pages 4167–4170, 2005.

[32] Daniel Große, Ulrich Kühne, and Rolf Drechsler. HW/SW co-verification of a RISC CPU using bounded model checking. In *Sixth International Workshop on Microprocessor Test and*

Verification (MTV 2005), Common Challenges and Solutions, 3-4 November 2005, Austin, Texas, USA, pages 133–137, 2005.

[33] Daniel Große, Ulrich Kühne, and Rolf Drechsler. HW/SW co-verification of embedded systems using bounded model checking. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI 2006, Philadelphia, PA, USA, April 30 - May 1, 2006*, pages 43–48, 2006.

[34] Ali Habibi, Haja Moinudeen, and Sofiène Tahar. Generating finite state machines from systemc. In Georges G. E. Gielen, editor, *DATE Designers' Forum*, pages 76–81. European Design and Automation Association, Leuven, Belgium, 2006.

[35] Ali Habibi and Sofiène Tahar. An approach for the verification of systemc designs using asml. In Doron Peled and Yih-Kuen Tsay, editors, *ATVA*, volume 3707 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2005.

[36] Reza Hajisheykhi, Ali Ebnenasir, and Sandeep S. Kulkarni. Modeling and analyzing timing faults in transaction level systemc programs. In *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, pages 344–347, 2013.

[37] Reza Hajisheykhi, Ali Ebnenasir, and Sandeep S. Kulkarni. Modeling and analyzing timing faults in transaction level systemc programs. In *Network on Chip Architectures, NoCArc '13, in conjunction with the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7, 2013*, pages 65–68, 2013.

[38] Reza Hajisheykhi, Ali Ebnenasir, and Sandeep S. Kulkarni. Analysis of permanent faults in transaction level systemc models. In *34th International Conference on Distributed Computing Systems Workshops (ICDCS 2014 Workshops), Madrid, Spain, June 30 - July 3, 2014*, pages 154–160, 2014.

[39] Reza Hajisheykhi, Ali Ebnenasir, and Sandeep S. Kulkarni. Evaluating the effect of faults in systemc TLM models using UPPAAL. In *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, pages 175–189, 2014.

[40] Reza Hajisheykhi, Ali Ebnenasir, and Sandeep S. Kulkarni. UFIT: A tool for modeling faults in UPPAAL timed automata. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 429–435, 2015.

[41] Reza Hajisheykhi, Mohammad Roohitavaf, Ali Ebnenasir, and Sandeep S. Kulkarni. A framework for verification of SystemC TLM programs with model slicing: A case study. In

To be appeared Design Automation Conference - 53rd ACM/EDAC/IEEE, DAC 2016, Austin, TX, USA, June 5-9, 2016, Proceedings, 2016.

[42] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Software Eng.*, 18(9):785–793, 1992.

[43] Nesrine Harrath and Bruno Monsuez. Systemc waiting state automata. *IJCCBS*, 3(1/2):60–95, 2012.

[44] Ian G. Harris. Fault models and test generation for hardware-software covalidation. *IEEE Design and Test of Computers*, 20(4):40–47, 2003.

[45] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[46] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

[47] Paula Herber, Marcel Pockrandt, and Sabine Glesner. Transforming systemc transaction level models into uppaal timed automata. In Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt, editors, *MEMOCODE*, pages 161–170. IEEE, 2011.

[48] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[49] Ravishankar K. Iyer, David J. Rossetti, and Mei-Chen Hsueh. Measurement and modeling of computer reliability as affected by system activity. *ACM Trans. Comput. Syst.*, 4(3):214–237, 1986.

[50] Agata Janowska and Pawel Janowski. Slicing timed systems. *Fundam. Inform.*, 60(1-4):187–210, 2004.

[51] Agata Janowska and Pawel Janowski. Slicing of timed automata with discrete data. *Fundam. Inform.*, 72(1-3):181–195, 2006.

[52] Daniel Karlsson, Petru Eles, and Zebo Peng. Formal verification of systemc designs using a petri-net based representation. In *DATE*, pages 1228–1233, 2006.

[53] Daniel Kroening and Natasha Sharygina. Formal verification of systemc by automatic hardware/software partitioning. In *ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 101–110, 2005.

[54] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, London, UK, 2000. Springer-Verlag.

[55] Sudipta Kundu, Malay Ganai, and Rajesh Gupta. Partial order reduction for scalable testing of SystemC TLM designs. In *Proceedings of the 45th annual Design Automation Conference*, Design Automation Conference, pages 936–941, 2008.

[56] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, 1997.

[57] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 75–88, 2004.

[58] Hoang M. Le, Daniel Große, and Rolf Drechsler. Automatic tlm fault localization for systemc. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(8):1249–1262, 2012.

[59] Ka L. Man. Formal communication semantics of systemc$^{\text{fl}}$. In *Eighth Euromicro Symposium on Digital Systems Design (DSD 2005), 30 August - 3 September 2005, Porto, Portugal*, pages 338–345, 2005.

[60] Ka Lok Man, Andrea Fedeli, Michele Mercaldi, Menouer Boubekeur, and Michel P. Schellekens. SC2SCFL: automated systemc to systemc$^{\text{fl}}$ translation. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 7th International Workshop, SAMOS 2007, Samos, Greece, July 16-19, 2007, Proceedings*, pages 34–45, 2007.

[61] K. Marquet, B. Jeannet, and M. Moy. Efficient Encoding of SystemC/TLM in Promela. Technical Report TR-2010-7, Verimag, France, 2010.

[62] Kevin Marquet and Matthieu Moy. PinaVM: A SystemC front-end based on an executable intermediate representation. In *International conference on Embedded software (EMSOFT)*, pages 79–88, 2010.

[63] Lynette I. Millett and Tim Teitelbaum. Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. *STTT*, 2(4):343–349, 2000.

[64] Silvio Misera, Heinrich Theodor Vierhaus, and Andre Sieber. Fault injection techniques and their accelerated simulation in SystemC. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 587–595, 2007.

[65] M. Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, 2005.

[66] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 26–35, 2005.

[67] Wolfgang Müller, Jürgen Ruf, and Wolfgang Rosenstiel. *SystemC: Methodologies and Applications, chapter An ASM based SystemC Simulation Semantics*. Kluwer Academic Publishers, 2003.

[68] B. Niemann and Ch. Haubelt. Formalizing TLM with Communicating Stat Machines. In *Proceedings of Forum on Specification and Design Languages 2006 (FDL 2006)*, pages 285–292, 2006.

[69] Hiren D. Patel and Sandeep K. Shukla. Model-driven validation of SystemC designs. *EURASIP Journal on Embedded Systems - C-Based Design of Heterogeneous Embedded Systems*, 2008:4:1–4:14, January 2008.

[70] Jon Perez, Mikel Azkarate-askasua, and Antonio Perez. Codesign and simulated fault injection of safety-critical embedded systems using SystemC. In *Proceedings of the 2010 European Dependable Computing Conference*, pages 221–229, 2010.

[71] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using indus and kaveri. *STTT*, 9(5-6):489–504, 2007.

[72] Jürgen Ruf, Dirk W. Hoffmann, Joachim Gerlach, Thomas Kropf, Wolfgang Rosenstiel, and Wolfgang Müller. The simulation semantics of systemc. In *DATE*, pages 64–70, 2001.

[73] Ashraf Salem. Formal semantics of synchronous systemc. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10376–10381, 2003.

[74] Alper Sen. Mutation operators for concurrent SystemC designs. In *International Workshop on Microprocessor Test and Verification*, 2000.

[75] Rishad Ahmed Shafik, Paul Rosinger, and Bashir M. Al-Hashimi. SystemC-based minimum intrusive fault injection technique with improved fault representation. In *Proceedings of the 2008 14th IEEE International On-Line Testing Symposium*, pages 99–104, 2008.

[76] Donald E. Thomas, Elizabeth D. Lagnese, John A. Nestor, Jayanth V. Rajan, Robert L. Blackburn, and Robert A. Walker. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*. Kluwer Academic Publishers, Norwell, MA, USA, 1989.

[77] Claus Traulsen, Jerome Cornet, Matthieu Moy, and Florence Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *SPIN Workshop*, pages 204–222, 2007.

[78] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.