Theoretical Computer Science 496 (2013) 50–68

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Facilitating the design of fault tolerance in transaction level SystemC programs

Ali Ebnenasir [a,*], Reza Hajisheykhi [b], Sandeep S. Kulkarni [b]

[a] Department of Computer Science, Michigan Technological University, Houghton, MI 49931, USA

[b] Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA

ARTICLE INFO

Keywords:
Fault tolerance
SystemC
Transaction level modeling

ABSTRACT

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 making fault tolerance a highly important property of such systems. However, designing fault tolerance 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. Most existing methods enable fault injection and impact analysis as a means for increasing design dependability. Nonetheless, such methods provide little support for designing fault tolerance. To facilitate the design of fault tolerance in SoC systems, this paper proposes an approach for designing fault-tolerant inter-component communication protocols in SystemC transaction level modeling (TLM) programs. The proposed method includes four main steps, namely model extraction, fault modeling, addition of fault tolerance and refinement of fault tolerance to SystemC code. We demonstrate the proposed approach using a simple SystemC transaction level program that is subject to communication faults. Moreover, we illustrate how fault tolerance can be added to SystemC programs that use the base protocol of the TLM interoperability layer. We also illustrate how fault tolerance functionalities can be partitioned to software and hardware components. Finally, we put forward a roadmap for future research at the intersection of fault tolerance and hardware–software co-design.

1. Introduction

Designing fault tolerance concerns in 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., fault tolerance) in the register transfer language (RTL) [1] is non-trivial [2]. More importantly, modern design languages (e.g., SystemC [3]) enable the co-design of hardware and software components, which makes it even more challenging to capture fault tolerance in SoCs. Thus, enabling the systematic (and possibly automatic) design of fault tolerance in SystemC can have a significant impact as SystemC is a widely accepted language and an IEEE standard [3]. SystemC 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) [3] 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]. Since SoC systems are subject to different types of fault (e.g., single-event upset, hardware aging, etc.), it is desirable to capture fault tolerance in SystemC TLM programs. However,

* Corresponding author. Tel.: +1 906 487 4372.
E-mail address: aebnenas@mtu.edu (A. Ebnenasir).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.11.010

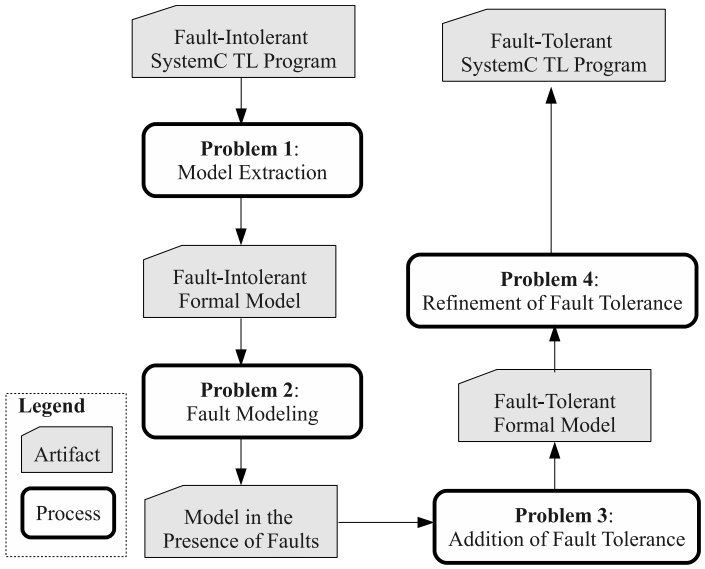

Fig. 1. Overview of the proposed framework.

capturing fault tolerance in SystemC TLM programs is non-trivial, as designers have to deal with appropriate manifestations of faults and fault tolerance at different levels of abstraction. This paper proposes a method for augmenting existing SystemC TLM programs with fault tolerance functionalities.

There are numerous approaches for fault injection and impact analysis, testing and verification of SystemC programs, most of which lack a systematic method for designing fault tolerance concerns in SystemC programs. Testing methods can be classified into two categories: test patterns and verification-based methods. Test patterns [5] enable designers to generate test cases and fault models [6] for SystemC programs at a specific level of abstraction and use the results to test lower levels of abstraction. Verification approaches [7–10] 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 [2,11–14] 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) [15]. Most of the aforementioned approaches enable the modeling of faults and their impacts with little support for systematic design of fault tolerance that can be captured at different levels of abstraction (e.g., assigning fault tolerance functionalities to hardware and software components).

Our objective is to facilitate the design of fault tolerance in SystemC by *separating fault tolerance concerns from functional concerns*. To this end, the proposed approach exploits model extraction, model checking and fault tolerance techniques to enable a framework for the addition of fault tolerance concerns to SystemC TLM programs. The proposed framework (see Fig. 1) includes four steps: (1) *model extraction*, (2) *fault modeling and impact analysis*, (3) *addition of fault tolerance to formal models* and (4) *refinement of fault tolerance from formal models to SystemC code*. The first two steps address Problems 1 and 2 in Fig. 1, and the last two steps respectively focus on Problems 3 and 4.

Specifically, 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 fault (e.g., transient faults, stuck-at, component failure, etc.), called the *fault-intolerant* program. Existing testing and verification [7–10] methods can be used to ensure that a SystemC program meets its functional requirements in the absence of faults. Then, we present a set of transformation rules for extracting finite models of SystemC TLM programs in the Process Meta Language [16] of the SPIN model checker [17], called Promela. The motivation behind choosing Promela as the target formal language is multi-fold. First, Promela enables the modeling of both synchronous and asynchronous computing systems using communication channels and shared variables. Since synchronous functionalities can easily be captured in hardware design, designers can partition the design of fault tolerance functionalities in Promela to the parts that should be implemented in hardware and the components that can be captured in software. Second, after the implementation of fault tolerance in Promela, developers can use SPIN to simulate the behaviors of the fault-tolerant model and to verify the model with respect to formal specifications of desired fault tolerance requirements. Third, we can leverage the existing work on model extraction in Promela [18–20] to facilitate model generation for fault tolerance. After extracting a model in Promela from the SystemC program, we augment the extracted Promela model with a model of the faults that perturb the SystemC program. The resulting model is a *model in the presence of faults*. Using SPIN, we analyze the impact of faults on functional requirements of the SystemC programs. The results of such an analysis include the scenarios that indicate how functional requirements could be violated due to the occurrence of faults (i.e., counterexamples). We use our previous work on automated addition of fault tolerance [21,22] to design the fault tolerance functionalities that should be added to

the model in the presence of faults. This step can potentially benefit from automation as we have developed software tools for adding fault tolerance [23]. After designing fault tolerance concerns in Promela models, one needs to determine what SystemC constructs should be generated corresponding to the fault tolerance code in Promela. Such refinements should be semantics preserving, in that they should preserve the correctness of fault tolerance aspects once refined to SystemC code. We present a set of reverse transformation rules that enable such refinements.

In order to validate the proposed method, we have conducted several case studies. First, we use a simple SystemC TLM program that captures communication between two simple TLM components without using the interoperability layer of OSCI. For this simple program, we provide a set of rules for model extraction (see Section 3). We also illustrate how fault tolerance functionalities are designed and refined (see Section 4). Second, we present a set of transformation rules (in Section 5) for extracting Promela models from SystemC TLM programs developed based on the interoperability layer of OSCI. We demonstrate our approach in the context of two case studies (Sections 6 and 7) where we design tolerance against transient faults that perturb memory contents and/or control signals between two TLM modules. We discuss related work in Section 8. In Section 9, we make concluding remarks and present a roadmap for future research. In the next section, we present an overview of SystemC and TLM.

2. Background: SystemC and transaction level modeling

This section provides a brief background on SystemC (Section 2.1), its simulation kernel (Section 2.2), and transaction level modeling (Section 2.3). The concepts represented in this section are mainly adapted from [3,4].

2.1. SystemC

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/C++ program. In this function, the designer creates structural elements of the system, called *modules*, and connects them throughout the system hierarchy. A module is the basic building block of SystemC TLM programs that includes *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 modules. 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 [3].

2.2. 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 be either resumed or triggered. The SystemC scheduler includes the following phases to simulate a system [3].

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 time-outs, and adds them to the list of runnable processes.
5. *Timed notification* phase. If pending timed notifications or time-outs exist, the scheduler identifies the corresponding sensitive processes and adds them to the set of runnable processes.

2.3. 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 Fig. 2. 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

```
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 }
```

Fig. 2. A simple running example for two communication modules.

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 example program in Fig. 2 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 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 Fig. 2 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 the target modules when sending/receiving generic payloads through sockets.

Sections 3 and 4 illustrate how we extract Promela models from SystemC TLM programs and how we model faults and fault tolerance in programs that have been developed without using the interoperability layer of OSCI. Sections 5–7 focus on model extraction, fault modeling and designing fault tolerance in SystemC TLM programs developed using the interoperability layer in the loosely timed abstraction level. We also demonstrate how we assign different fault tolerance functionalities to software and hardware components.

3. Model extraction and fault modeling

The proposed approach starts with extracting a model from SystemC TLM programs (see Section 3.1). In Section 3.2, we illustrate how we specify the functional requirements/properties of the communication between components in the TLM program. We then use the SPIN model checker [17] to verify that the extracted model meets its functional requirements. In Section 3.3, we augment the extracted functional model with faults to create a *model in the presence of faults*.

3.1. Model extraction

In order to extract a model from a SystemC TLM program, we build on the ideas from [24], where we consider three basic processes, *Behavior*, *Initiator* and *Target*, for each module in the SystemC TLM program. The *Behavior* process captures the main functionalities of a TLM module. An *Initiator* and a *Target* process are considered for each transaction in which a TLM module is involved. The simulation/execution of TLM programs switches between these three processes by transferring the control of execution. The control transfer is either between (i) the *Behavior* and *Initiator* of the same module or (ii) the *Initiator* of one module and the *Target* of another module [24]. We use Promela [16] as the target formal language in which the extracted model is specified. The syntax of Promela is based on the C programming language. A Promela model comprises (1) a set of variables, (2) a set of (concurrent) processes modeled by a predefined type, called *proctype*, and (3) a set of asynchronous and synchronous channels for inter-process communications. 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.

For the program in Fig. 2, the extracted Promela model *M* includes two proctypes named Initiator and Target (see Fig. 3). 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 Fig. 3). To start a transaction, the Initiator sends the message startTrans to the Target via the tgtIfPort channel and waits until the Target signals the end of the transaction with a message endTrans. The Promela code in Fig. 3 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 Fig. 3 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.

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 Fig. 2: one is associated with the method initiate() of the initiator class (see Line 11 in Fig. 2), and the other implements the body of the incModEight() method of the target class (see Line 24 in Fig. 2). The first statement of the incModEight() method is a wait() statement on a delta notification event because in Line 27 of Fig. 2 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 Fig. 2). Afterwards, 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.

3.2. 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 Fig. 2 has two types of requirement. First,

```
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
26 active proctype IncModEight(){  // Models the Behavior process
27                               // of the Target
28    byte recv;
29    waiting:  if2TgtBeh?recv;
30              (recv == inc) ->  d = (d + 1) % 8;
31              if2TgtBeh!incComplt;
32  }
```

Fig. 3. The extracted functional model.

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 [17], 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 Fig. 3). 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 $\Box(\text{strtTr} \Rightarrow \Diamond\ \text{endTr})$, 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 Fig. 3); i.e., it will finish the transaction. We specify the invariant property as the expression $\Box$ 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 Fig. 3.

3.3. Model in the presence of faults

The next step for adding fault tolerance is to model the impact of faults on the extracted model and create a model in the presence of faults. First, we identify the type of fault that perturbs the SystemC program under study. Since in this paper we are mainly concerned with the impact of faults on SystemC programs, we do not directly focus on fault diagnosis methods that identify the causes of faults; rather, we concentrate on modeling the impact of faults on programs. 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 Fig. 2 is subject to the type of fault that corrupts 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 paper.) 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 results.

4. Fault-tolerant model and refinement

This section focuses on the next two steps (Problems 3 and 4 in Fig. 1), where we first modify the model in the presence of faults to obtain a fault-tolerant model. Subsequently, in the last step, the fault-tolerant model is refined to obtain a fault-tolerant SystemC program.

4.1. Fault-tolerant Promela model

Upon analysis of the deadlock scenario mentioned in Section 3.3, we find that the deadlock can be handled either by ensuring that the program never reaches a deadlock state (e.g., by preventing certain program actions that reach the deadlock state) or by adding new recovery actions that allow the Initiator and the Target to detect that they have reached a deadlock state and subsequently recover from it to some valid state (such as an initial state). We follow the first approach and modify the Promela model so that, if the Target receives a message other than startTrans, then it ignores it and returns to its initial state, where it waits for another message. Thus, we replace the action in Line 21 of Fig. 3 with the following if fi; statement:

```
if
  :: (recv == startTrans) -> if2TgtBeh!inc;
  :: (recv != startTrans) -> goto waiting;
fi;
```

Likewise, we substitute the action in Line 13 of Fig. 3 with the following statement:

```
if
  :: (recv == endTrans) -> fin: skip;
  :: (recv != endTrans) -> goto waiting;
fi;
```

Fault tolerance property. After modification to the model, we need to verify whether the revised model is fault tolerant. The fault tolerance property states that *it is always the case that, when faults stop occurring, the model recovers to its invariant and any subsequent transaction works correctly*. To express these fault tolerance properties, we first define the macro #define nofaults (cnt $>$ MAX), where nofaults becomes true when the proctype F() terminates; i.e., no more faulty messages are sent to the channel tgtIfPort. Then, we verify whether the revised model satisfies the properties $\Box$ (nofaults $\Rightarrow$ $\Diamond$ inv) and $\Box$ (nofaults $\Rightarrow$ (strtTr $\Rightarrow$ $\Diamond$ endTr)). The first property states that it is always the case that, when no more faults occur, the model will eventually reach an invariant state. The second property stipulates that it is always the case that, when no more faults occur, any initiated transaction will eventually complete. These properties were satisfied by the revised model, thereby resulting in a fault-tolerant model. Next, we should refine the fault-tolerant model to a SystemC program.

4.2. Refinement

The last step in adding fault tolerance to the SystemC program is to refine the fault-tolerant Promela model derived in the earlier step. In this step, we first evaluate the role of the added recovery actions. In general, there are two possibilities that may occur regarding the recovery actions. Some recovery actions may be needed to update original variables in the SystemC program, whereas some recovery actions might require addition of new variables and/or control structures. Depending upon the role of the recovery actions, we augment the SystemC program to declare additional variables and/or control structures.

```
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
10     SC_HAS_PROCESS(initiator);
11     initiator(sc_module_name name) : sc_module(name) {
12
13     SC_THREAD(initiate);
14     }
15   void initiate() { port->trigger(); }
16 };
17
18 class target : public target_if, public sc_module {
19 public:
20     short d;
21     sc_event e;
22     sc_event_finder ef; // Added for detecting the occurrence of faults
23     SC_HAS_PROCESS(target);
24     target(sc_module_name name) : sc_module(name) {
25         d = 0;
26         ef = new sc_event_finder(); // For detection of channel faults
27         SC_THREAD(incModEight);
28    }
29     void trigger() { e.notify(SC_ZERO_TIME); }
30     void incModEight() {
31     waitL: wait(e); // Next line is for implementing recovery
32              if (ef.find_event(target) != SC_ZERO_TIME)  goto waitL;
33              d = (d+1)%8;
34         }
35 };
36
37  int sc_main (int argc , char *argv[]) {
38  initiator initiator_inst(Initiator);
39  target target_inst(Target);
40
41   initiator_inst.port(target_inst);
42   sc_start();
43   return 0;
44  }
```

Fig. 4. Fault-tolerant SystemC program after refinement.

We augment the SystemC code with the constructs that implement the recovery actions generated in the earlier step (see Lines 22, 26 and 32 in Fig. 4). This is achieved using the reverse transformation rules that are the dual of the rules for generating the Promela model in the first step.

Continuing with our example in Fig. 2, we observe that the changes in obtaining the fault-tolerant model included recovery actions to deal with spurious messages sent to the channel. Hence, the tolerant program needs to check whether the events it receives are correct before executing the corresponding action. In the original SystemC program the Target waits for an event of type SC_ZERO_TIME. If this event is perturbed by faults then the Target needs to ensure that it does not execute its (increment) operation. Moreover, when we evaluate the recovery action, upon receiving an unexpected message, the Target goes to a waiting state where it waits for the next message. Fig. 4 illustrates the refined SystemC program.

5. Transformation rules for TLM base protocol

In this section, we present a set of transformation rules for generating Promela models from SystemC TLM programs written based on the TLM base protocol for interoperability. Our objective in this section is to define a set of rules to transform these constructs 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:

```
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 does not need to be buffered. **Rule 2** transforms the declaration of a SystemC thread to a *proctype* in Promela as follows:

```
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*.

```
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:

```
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**, which defines a structure for the TLM response. We encode these values in the three bits of the response array.

```
TLM_OK_response = 1,
TLM_incomplete_response = 0,
TLM_generic_error_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.

```
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.

6. Case study: memory faults

This section presents a case study on adding fault tolerance functionalities to a SystemC TLM program that has been developed using the TLM base protocol. In Section 6.1, we introduce the SystemC program under study. Afterwards, we illustrate the applications of the transformation rules presented in Section 5 in extracting a Promela model from the example program. After specifying the properties that should hold in the absence of faults, we present the model in the presence of faults in Section 6.3. Finally, in Sections 6.4 and 6.5, we specify the fault tolerance property of the example program and discuss how fault tolerance functionalities are refined to SystemC code.

```
1  struct Initiator: sc_module
2      {   tlm_utils::simple_initiator_socket<Initiator> socket;
3          SC_CTOR(Initiator)   : socket("socket")
4      {   SC_THREAD(thread_process);   }
5
6      void thread_process()
7      {
8          tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
9          sc_time delay = sc_time(10, SC_NS);
10
11         tlm::tlm_command cmd = static_cast<tlm::tlm_command>(rand()%2);
12
13         if (cmd == tlm::TLM_WRITE_COMMAND) data = 0xFF000000 | i;
14
15         trans->set_command(cmd);
16         trans->set_address(i);
17         trans->set_data_ptr(reinterpret_cast<unsigned char*>(&data));
18         trans->set_data_length(4);
19         trans->set_streaming_width(4);
20         trans->set_byte_enable_ptr(0);
21         trans->set_dmi_allowed(false);
22         trans->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
23
24         socket->b_transport(*trans, delay);
25
26         if (trans->is_response_error())
27             SC_REPORT_ERROR("TLM-2","Response error");
28         wait(delay);
29     }
30
31     int data;
32 };
```

Fig. 5. The initiator module.

6.1. SystemC program

This example, adapted from [25], models how on-chip memory-mapped busses are captured using the TLM base protocol. In this example (see Figs. 5 and 6), the initiator module (Lines 1–32 in Fig. 5) generates a transaction, while the target module (Lines 33–63 in Fig. 6) represents a simple memory. The initiator module has a thread process (Lines 6–29 in Fig. 5) that sends a generic payload to the target module; i.e., the memory module.

In Lines 15–22 in Fig. 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 Fig. 5 and a target socket in Line 35 of Fig. 6. The initiator sends the transaction out through the initiator socket (Line 24 in Fig. 5), and the memory communicates with the initiator by first registering a callback method with the socket (Line 39 in Fig. 6), and then implementing that method (Lines 44–61 in Fig. 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 Fig. 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 Fig. 6). If not set, the default response_status would indicate to the initiator that the transaction is incomplete (Lines 26–27 in Fig. 5). In each TLM SystemC program we need a *sc_main* function (Lines 76–81 in Fig. 6). Moreover, to connect up the module hierarchy, we use the Top module (Lines 65–74 in Fig. 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 Fig. 6).

6.2. Model extraction

In order to extract a Promela model from the SystemC TLM program of Section 6.1, we use the same ideas explained in Section 3.1. Moreover, we use the set of transformation rules of Section 5, which helps us to model interoperability. The extracted Promela model M has two proctypes named Initiator (Lines 16–49 in Fig. 7) and Memory (Lines 50–77 in Fig. 8). 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 Fig. 7). The binding of the initiator and the target sockets in Line 72 of Fig. 6 is captured as a channel in the Promela model (Line 12 in Fig. 7). 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 Fig. 7. Using **Rule 3**, the initiator creates a transaction by setting the attributes of the generic payload (Lines 22–28 in Fig. 7). 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

```
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 };
75
76 int sc_main(int argc, char* argv[])
77 {
78     Top top("top");
79     sc_start();
80     return 0;
81 }
```

Fig. 6. The memory, the top, and the main modules.

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 29 in Fig. 7) 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 Fig. 7 defines a random value, either 0 or 1, for the cmd attribute to send the message initT (Lines 23–24 in Fig. 7). 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 68 in Fig. 8). 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 69 in Fig. 8). After reading from/writing to the memory successfully, the Memory module sets the response_statuse attribute of memT message to 1 (Line 72 in Fig. 8). This means that, according to the transformation **Rule 5**, the response_status attribute equals *OK*. Finally, the Initiator, after receiving the *OK* response, completes the current transaction (Lines 43–47 in Fig. 7).

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.

```
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: %d/n", 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;

}
```

Fig. 7. The initiator module of the extracted functional model.

In Figs. 5 and 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 [17].

```
#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 Figs. 7 and 8.

```
50 active proctype Memory(){
51 int data;
52         bool faultsOccur = false;
53
54 waiting:
55   simple_initiator_socket?memT;
56
57 starting:
58   if
59      :: ((memT.address >= 256) || (memT.byte_enable_ptr != 0)
60            || (memT.data_length > 4)|| (memT.streaming_width < 4))
61            ->   atomic { memT.response_status = -2;
62                 simple_initiator_socket!memT;
63                 goto waiting; }
64      :: else -> skip;
65    fi;
66
67   if
68      :: (memT.cmd == 0) ->  memT.data_ptr = mem[memT.address] ;
69      :: (memT.cmd == 1) -> mem[memT.address] = memT.data_ptr;
70   fi;
71
72   memT.response_status = 1;
73   data = mem[memT.address];
74
75   simple_initiator_socket!memT;
76
77 }
```

Fig. 8. The memory module of the extracted functional model.

6.3. Fault modeling

In this section, we model the impact of faults on the extracted Promela model M in Figs. 7 and 8. Our objective is to create a model M_F that captures the behavior of M in the presence of faults. The TLM program of Figs. 5 and 6 can be perturbed by the transient faults that perturb memory contents without causing any permanent damage. The following proctype models the impact of this fault type in the extracted model:

```
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 Fig. 6, in the model of Fig. 7, 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, similarly to what we did in Section 4.1.

6.4. Adding fault tolerance

Since faults can change the contents of memory cells, the proctype memFaults could perturb the state of the Promela model M to a state outside its invariant. For instance, imagine a scenario where the cmd attribute of the sent message by the Initiator equals 1, demonstrating a write action. Then, assume that after writing the data_ptr value of the memT message in the corresponding memory cell, the faults occur. Thus, the memory contents become different from the variable sentData in the Initiator, thereby violating the inv1.

To ensure recovery to inv1 when faults perturb the state of the model to states outside inv1, we add a proctype called recoverMem to the model M_F that corrects the violation of inv1. The proctype recoverMem continuously executes two actions. The first action is enabled when there has been a write transaction and the data_ptr attribute and the memory contents in the corresponding address are not equal. Thus, we rewrite the data_ptr attribute into the memory cell. Likewise, when the cmd is equal to 0 and after executing a read action the data_ptr attribute and memory cell are not equal, we rewrite the value of the memory cell into the data_ptr attribute. The recoverMem proctype is as follows:

```
active proctype recoverMem(){
 do
  :: ((initT.cmd == 1) && (mem[memT.address] != memT.data_ptr))
          ->  mem[memT.address] = memT.data_ptr;
  :: ((initT.cmd == 0) && (mem[memT.address] != memT.data_ptr))
          ->  memT.data_ptr = mem[memT.address];
 od;
}
```

To verify whether after adding recovery the revised model is fault tolerant, we should guarantee that, when faults stop occurring, the model recovers to its invariant and works correctly. For this purpose, we define the macro #define nofaults1 (cnt1 > MAX1), where nofaults1 becomes true when the proctype memFaults() terminates. Thus, the revised model should always satisfy the *recovery property* $\Box$ (nofaults1 $\Rightarrow$ $\Diamond$ inv1). This property states that *it is always the case that, if faults stop occurring, then the inv1 condition will eventually hold*. We have verified the recovery property with SPIN and the revised model always satisfies it, thereby resulting in a fault-tolerant model.

6.5. Refinement

In order to derive a fault-tolerant TLM SystemC program, we refine the fault-tolerant Promela model obtained in Section 6.4. During refinement, we may augment the TLM SystemC program with additional variables and/or control structures. To perform such refinements systematically, we present a set of reverse transformation rules, some of them being the duals of the rules presented in Section 5. First, we refine a proctype in Promela to a thread in the related module of the SystemC program. For example, the recoverMem() proctype is captured as a thread in the Memory module in the fault-tolerant SystemC program:

```
proctype recoverMem()   |--   SC_THREAD(memRecovery_process)
```

We add this thread to the Memory module (Line 43 and Lines 70–82 in Fig. 9) to check whether, after executing read/write action into/from memory, the values of memory content and data sent by the initiator are equal. Note that such recovery should be achieved in a specific time interval from the time when a read/write operation is performed in the Memory module to the time when the execution of a new transaction is started in Memory. For this reason, we add a new Boolean variable complete to the program (Line 46 in Fig. 9) and set it to false (Line 50 in Fig. 9) in the beginning of the transaction. Then, after performing a read/write, we set this variable to true (Line 67 in Fig. 9). Since memRecovery_process is captured as a separate thread in the Memory module and it needs to have access to the transaction object received from the initiator, we keep a copy of the transaction in the variable transCopy. If the memory content is perturbed by faults and the command demonstrates a read action, then the recovery thread re-reads the content of the memory. Notice that, in the case of a read transaction, if the memory contents are faulty, then the original memory content is lost and cannot be recovered (unless a copy of the memory content is kept somewhere using resource redundancy). Nonetheless, the recovery operation in the added recovery thread satisfies the constraint requiring that the value received by the initiator is equal to the memory content. If there has been a write transaction, then the content of the data pointer is re-written to memory. Fig. 9 illustrates the refined TLM SystemC program.

Software/hardware co-design. The new fault tolerance functionalities added to a SystemC program raises the following questions: *Which part of the new fault tolerance functionalities is captured in software and which part is captured in hardware*? The answer to this question depends on the constraints that affect the design decisions. In this paper, we follow the rule of capturing asynchronous functionalities in software and including the synchronous functionalities in hardware. Thus, the new recovery thread added to the Memory module can be implemented as a software component in the entire system captured by the SystemC program.

7. Case study: control signal faults

As the third case study, we model the effect of a different kind of fault on the explained TLM SystemC program and the extracted model in Section 6. First, in Section 7.1, we specify properties that should hold in the extracted model. Second, we introduce a new type of fault that perturbs the read/write command of the generic payload in Section 7.2. Finally, in Sections 7.3 and 7.4, we revise the extracted model and refine it to the fault-tolerant TLM program.

7.1. Property specification and functional correctness

In this section, we introduce another requirement that must be met by the SystemC program presented in Figs. 5 and 6. Specifically, once the memory receives a transaction and before it executes the transaction, the following conditions must hold: (1) if the datum on the bus is invalid (e.g., the data bus is in a tri-state), then the requested operation by the initiator should be a read command, and (2) if there is valid datum being sent to the memory, then the requested operation by the

```
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         SC_THREAD(memRecovery_process); // Added for fault tolerance
44     }
45     tlm::tlm_generic_payload transCopy; // Added new vars
46     bool complete = false;  // Added new vars
47
48     virtual void b_transport(tlm::tlm_generic_payload& trans, sc_time& delay)
49     {
50         complete = false;    // Added for fault tolerance
51         transCopy = *trans;  // Added for fault tolerance
52
53         tlm::tlm_command cmd = trans.get_command();
54         sc_dt::uint64    adr = trans.get_address() / 4;
55         unsigned char*   ptr = trans.get_data_ptr();
56         unsigned int   len = trans.get_data_length();
57         unsigned char*   byt = trans.get_byte_enable_ptr();
58         unsigned int    wid = trans.get_streaming_width();
59
60         if (adr >= sc_dt::uint64(SIZE) || byt != 0 || len > 4 || wid < len)
61             SC_REPORT_ERROR("TLM-2","Target does not support the transaction");
62         if ( cmd == tlm::TLM_READ_COMMAND )
63             memcpy(ptr, &mem[adr], len);
64         else if ( cmd == tlm::TLM_WRITE_COMMAND )
65             memcpy(&mem[adr], ptr, len);
66
67         complete = true;       // Added for fault tolerance
68         trans.set_response_status(tlm::TLM_OK_RESPONSE);
69     }
70     void memRecovery_process()  // Recovery Thread
71     { while (true) {
72             if (complete) {
73                 tlm::tlm_command cmd = transCopy.get_command();
74                 sc_dt::uint64 adr = transCopy.get_address() / 4;
75                 unsigned char* ptr = transCopy.get_data_ptr();
76                 if ((cmd == tlm::TLM_WRITE_COMMAND) && (mem[adr] != *ptr))
77                     mem[adr] = *ptr;
78                 if ((cmd == tlm::TLM_READ_COMMAND) && (mem[adr] != *ptr))
79                     *ptr = mem[adr];
80           }
81       }
82    }
83    int mem[SIZE];
84 };
```

Fig. 9. The memory module of the fault-tolerant SystemC program after refinement.

initiator should be a write command. We represent an invalid datum by -1. We specify these constraints as the following macros in the model captured by Figs. 7 and 8.

```
#define inv2
   (!(memory@starting) || !(memT.data_ptr == -1) || (memT.cmd == 0))
   &&
   (!(memory@starting) || !(memT.data_ptr != -1) || (memT.cmd == 1))
```

The invariant condition inv2 must always be true in the absence of faults. Nonetheless, if transient faults occur, and the command signal is flipped, then inv2 could be violated. If the Memory is at the starting label and the data_ptr attribute of the message memT equals -1, then the cmd attribute of that message is a read command. If the Memory is at the starting label and the data_ptr is not -1, then the cmd attribute is a write command. Note that, if the generic payload contains a read command, then, when the Memory module is at the starting label, the value of the data_ptr attribute is irrelevant. However, in the case of a write command, the value of data_ptr is written in memory. We specify this property as the expression $\Box$ inv2, which requires that inv2 is always true (in the absence of faults). Using SPIN, we have verified the above properties for the extracted model of Figs. 7 and 8.

7.2. Fault modeling

In order to model the effect of the transient faults that perturb the cmd of the generic payload, we augment the model of Figs. 7 and 8 with 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 cnt2 shows the occurrence of faults. The condition (cnt2 $<$ MAX2) ensures that the faults eventually stop occurring.

7.3. Adding fault tolerance

Adding the cmdFaults proctype to the extracted model, it is possible to reach a state outside inv2. For instance, consider a scenario where the cmd attribute in the receiving message memT by the Memory is 0, which means a read command (see Figs. 7 and 8). Moreover, data_ptr is -1, since it is a read command. When the Memory is at the starting point, the fault occurs and the value of cmd is changed to 1. Thus, since data_ptr is -1 and cmd is 1, inv2 is violated. As the second scenario imagine that the Memory is at the starting label in Fig. 8 and that cmd equals 1 and data_ptr is not equal to -1. Then the occurrence of faults changes cmd to 0, thereby violating inv2. Since exactly at the time of reading or writing the right operation must be performed by the Memory module, we ensure that, even if the command signal is perturbed by transient faults, the appropriate memory operation will take place. To this end, we replace the action in Line 68 of Fig. 8 with the following if fi statement:

```
if
 :: ((memT.cmd == 0) || ((memT.cmd == 1) &&
  (memT.data_ptr == -1))) -> memT.data_ptr = mem[memT.address];
 :: else -> skip;
fi;
```

Instead of the action in Line 69 of Fig. 8, we have the following if fi statement:

```
if
 :: ((memT.cmd == 1) || ((memT.cmd == 0) &&
  (memT.data_ptr != -1))) -> mem[memT.address] = memT.data_ptr;
 :: else -> skip;
fi;
```

The new constraints added to the actions in the if fi; statements detect whether there is a mismatch between the command signal and the data bus. Thus, the required action takes place if faults perturb the command signal. To model the finiteness of the occurrence of faults, we define the macro #define nofaults2 (cnt2 $>$ MAX2), where nofaults2 becomes true when the proctype cmdFaults() terminates. The revised model should always satisfy the property $\Box$ (nofaults2 $\Rightarrow$ $\Diamond$ inv2). We have verified that the revised model always satisfies $\Box$ (nofaults2 $\Rightarrow$ $\Diamond$ inv2). Furthermore, the property $\Box$ (nofaults2 $\Rightarrow$ (strtTr $\Rightarrow$ $\Diamond$ cmplTr)) is always satisfied by the revised model; i.e., when no more faults occur, any initiated transaction will eventually complete.

We would like to note that, even if both types of fault (that affect the command signal and the memory contents) occur, the model resulting from the model of Figs. 7 and 8, the proctype recoverMem() and the preceding revised actions provides recovery to inv1 $\wedge$ inv2. We call such a model a *multitolerant* model [26].

7.4. Refinement

In order to have a fault-tolerant SystemC program, we refine the fault-tolerant Promela model presented in Section 7.3. Fig. 10 illustrates the refined program. Notice that the new constraints introduced in the Promela model have been implemented in Lines 55 and 58 of Fig. 10 to ensure that the correct read/write operation is performed even if the command signal is perturbed by faults.

Hardware/software co-design. This case study is an example of fault tolerance functionalities that can be captured in hardware since they must be executed in a synchronous fashion. More specifically, at the same time as a read/write is about to take place in memory, the corruption of the command signal must be detected. Thus, the timing of executing the fault tolerance functionalities is important; it is pointless to detect/correct the command signal after the read/write operation is finished.

```
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            ((cmd == tlm::TLM_WRITE_COMMAND) && (*ptr == INVALID)))
57             memcpy(ptr, &mem[adr], len);
58         else if ((cmd == tlm::TLM_WRITE_COMMAND) ||
59            ((cmd == tlm::TLM_READ_COMMAND) && (*ptr != INVALID)))
60             memcpy(&mem[adr], ptr, len);
61
62          trans.set_response_status( tlm::TLM_OK_RESPONSE );
63     }
64     int mem[SIZE];
65 };
```

Fig. 10. Fault-tolerant SystemC program after refinement — the memory module.

8. Related work

This section discusses existing approaches for formal analysis of SystemC/TLM programs. Extant work can broadly be classified into three categories: methods for formalizing the semantics of SystemC/TLM, static analysis and stateless model checking, and model extraction and model checking of SystemC/TLM programs. Several researchers focus on assigning formal semantics to SystemC/TLM programs. For example, Niemann and Haubelt [24] 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 [27] present a formalization of SystemC in abstract state machines and revise Microsoft SpecExplorer for the validation and debugging of SystemC programs. Kroening and Sharygina [28] create abstract models of SystemC programs using labeled Kripke structures (LKSs), where each SystemC thread is captured by an LKS. A labeled Kripke structure 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.

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 [9] 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. [7] 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.

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. [29] 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. [18] present a method for transforming a subset of SystemC to the Promela [16] modeling language in order to enable the model checking of asynchronous software threads in the SPIN model checker [17]. Marquet and Moy [10] present a front-end that transforms

SystemC programs to an intermediate language in LLVM [30], which is a framework that provides reusable components for compiler construction. Marquet et al. [19] 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. [31] present a model checking approach supporting two techniques: one method 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.

While the preceding methods inspire our work to some extent, our objective is to extract models that capture the impact of a specific type of fault on a SystemC program. As such, we plan to leverage the existing model extraction methods to generate models that are dependent upon the type of fault. For example, in the SystemC example in Fig. 2, the internal activities of the Target module for increasing the value of d are irrelevant to the recovery required for tolerating faults. Thus, such functionalities should be sliced out in the abstraction and model extraction. We are currently investigating the development of compilers that automate model extraction from SystemC for the addition of fault tolerance. Towards this end, we leverage our previous work on model extraction from parallel C programs [32].

9. Conclusions and future work

In this paper, we have presented a methodology for facilitating the design of fault tolerance in SystemC transaction level modeling (TLM) programs. Our methodology involved four main steps. The first step obtains an abstract model of the SystemC program. We chose Promela as the target modeling language since it allowed us to evaluate the effect of faults with the model checker SPIN [16]. Although we did not address the automation of this step, this step can in fact be automated by leveraging existing works such as [18,32]. Subsequently, in the second 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 [16]. Subsequently, we analyze the impact of faults using the SPIN model checker [17] to identify scenarios where faults perturb the model to error states and potentially cause failures. We then analyze the failures to revise the Promela model towards adding fault tolerance. This step may require several iterations, and it terminates when one identifies a Promela model that is fault tolerant. There is a potential to automate this step as well. In particular, techniques such as those in [22,33] have shown feasibility of adding fault tolerance to transition systems. However, such methods should be adapted in the context of the proposed approach to ensure that the revised program can be transformed to SystemC. Finally, we transformed the fault-tolerant Promela models to SystemC programs.

We illustrated our methodology with a transaction level SystemC program that was subject to 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 methodology 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 applied our methodology for designing fault tolerance in SystemC TLM programs that benefit from the Open SystemC Initiative (OSCI) interoperability layer. In particular, we presented a set of transformation rules that enable model extraction from programs developed using OSCI's base protocol. We demonstrated the proposed approach in the context of two case studies for tolerating transient faults that cause spontaneous bit-flips without any permanent damage.

A roadmap for future research. We propose several directions for future research on the software/hardware co-design of fault tolerance in SystemC TLM programs.

- There is a need for automating different steps of the proposed approach in this paper. Towards this end, the transformation rules for model extraction should be customized towards generating fault-dependent models. Moreover, the model extraction step can facilitate the refinement step by including additional information in the generated model. For instance, a fault-tolerant Promela model may detect the truth value of a condition that depends upon the internal state of multiple modules. The refinement of such detection functionalities back to the level of SystemC code can only be realized if there is sufficient information about the interconnection of modules with each other and whether it is feasible to add new modules to the design. We are currently investigating the development of compilers that automate model extraction from SystemC for the addition of fault tolerance.
- Since TLM enables modeling at different levels of abstraction (e.g., loosely timed (LT) and approximately timed (AT) modeling), we should devise methods that facilitate the modeling of faults and fault tolerance at different levels of abstraction. For instance, in the AT level, components may communicate under timing constraints. Thus, faults that cause delays have to be considered. However, such faults lose their significance in an LT context since timing concerns are not considered in order to enable faster simulation of design.
- We plan to extend our previous work on automated addition of fault tolerance [21,22] in order to automate the design of fault tolerance in the extracted Promela 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 the 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. In this paper, we used 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 the co-design of fault tolerance.

Acknowledgement

This work was partially sponsored by the National Science Foundation grants CCF-1116546 and CNS-0914913, and the grant FA9550-10-1-0178 from the Air Force Office of Scientific Research.

References

[1] D.E. Thomas, E.D. Lagnese, J.A. Nestor, J.V. Rajan, R.L. Blackburn, R.A. Walker, Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench, Kluwer Academic Publishers, Norwell, MA, USA, 1989.
[2] Y.-Y. Chen, C.-H. Hsu, K.-L. Leu, SoC-level risk assessment using FMEA approach in system design with SystemC, in: International Symposium on Industrial Embedded Systems, 2009, pp. 82–89.
[3] Open SystemC Initiative (OSCI): Defining and advancing SystemC standard IEEE 1666–2005. http://www.systemc.org/.
[4] Transaction-Level Modeling (TLM) 2.0 Reference Manual. http://www.systemc.org/downloads/standards/.
[5] A. Fin, F. Fummi, M. Martignano, M. Signoretto, SystemC: A homogenous environment to test embedded systems, in: Proceedings of the Ninth International Symposium on Hardware/Software Codesign, CODES'01, 2001, pp. 17–22.
[6] I.G. Harris, Fault models and test generation for hardware–software covalidation, IEEE Design and Test of Computers 20 (4) (2003) 40–47.
[7] S. Kundu, M. Ganai, R. Gupta, Partial order reduction for scalable testing of SystemC TLM designs, in: Proceedings of the 45th Annual Design Automation Conference, Design Automation Conference, 2008, pp. 936–941.
[8] A. Sen, Mutation operators for concurrent SystemC designs, in: International Workshop on Microprocessor Test and Verification, 2000.
[9] N. Blanc, D. Kroening, Race analysis for SystemC using model checking, ACM Transactions on Design Automation of Electronic Systems 15 (3) (2010) 21:1–21:32.
[10] K. Marquet, M. Moy, PinaVM: A SystemC front-end based on an executable intermediate representation, in: International Conference on Embedded software (EMSOFT), 2010, pp. 79–88.
[11] S. Misera, H.T. Vierhaus, A. 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, 2007, pp. 587–595.
[12] R.A. Shafik, P. Rosinger, B.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, 2008, pp. 99–104.
[13] A. da Silva Farina, S.S. 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, 2009, pp. 237–244.
[14] J. Perez, M. Azkarate-askasua, A. Perez, Codesign and simulated fault injection of safety-critical embedded systems using SystemC, in: Proceedings of the 2010 European Dependable Computing Conference, 2010, pp. 221–229.
[15] B. Giovanni, C. Bolchini, A. Miele, Multi-level fault modeling for transaction-level specifications, in: Proceedings of the 19th ACM Great Lakes Symposium on VLSI, 2009, pp. 87–92.
[16] Spin language reference, http://spinroot.com/spin/Man/promela.html/.
[17] G.J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5) (1997) 279–295.
[18] C. Traulsen, J. Cornet, M. Moy, F. Maraninchi, A SystemC/TLM semantics in Promela and its possible applications, in: SPIN Workshop, 2007, pp. 204–222.
[19] K. Marquet, B. Jeannet, M. Moy, Efficient Encoding of SystemC/TLM in Promela, Tech. Rep. TR-2010-7, Verimag, France (2010).
[20] D. Campana, A. Cimatti, I. Narasamdya, M. Roveri, An analytic evaluation of systemc encodings in Promela, in: International SPIN Workshop on Model Checking Software (SPIN), 2011, pp. 90–107.
[21] S.S. Kulkarni, A. Arora, Automating the addition of fault-tolerance, in: Formal Techniques in Real-Time and Fault-Tolerant Systems, Springer-Verlag, London, UK, 2000, pp. 82–93.
[22] A. Ebnenasir, Automatic synthesis of fault tolerance, Ph.D. Thesis, Michigan State University, 2005.
[23] A. Ebnenasir, S.S. Kulkarni, A. Arora, FTSyn: a framework for automatic synthesis of fault-tolerance, International Journal on Software Tools for Technology Transfer 10 (5) (2008) 455–471.
[24] B. Niemann, C. Haubelt, Formalizing TLM with communicating stat machines, in: Proceedings of Forum on Specification and Design Languages 2006, FDL 2006, 2006, pp. 285–292.
[25] Getting started with tlm-2.0, http://www.doulos.com/knowhow/systemc/tlm2/.
[26] A. Ebnenasir, S.S. Kulkarni, Feasibility of stepwise design of multitolerant programs, ACM Transactions on Software Engineering and Methodology (TOSEM) 21 (1) (2011) 1:1–1:49.
[27] H.D. Patel, S.K. Shukla, Model-driven validation of SystemC designs, in: C-Based Design of Heterogeneous Embedded Systems 2008, EURASIP Journal on Embedded Systems (2008) 4:1–4:14.
[28] D. Kroening, N. Sharygina, Formal verification of systemc by automatic hardware/software partitioning, in: ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), 2005, pp. 101–110.
[29] M. Moy, F. Maraninchi, L. 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), 2005, pp. 26–35.
[30] C. Lattner, V.S. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2004, pp. 75–88.
[31] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, M. Roveri, KRATOS: A software model checker for SystemC, in: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV), 2011, pp. 310–316.
[32] A. Ebnenasir, UPC-SPIN: A Framework for the Model Checking of UPC Programs, in: Fifth Partitioned Global Address Space Conference (PGAS), 2011.
[33] B. Bonakdarpour, S.S. Kulkarni, Exploiting symbolic techniques in automated synthesis of distributed programs, in: EEE International Conference on Distributed Computing Systems(ICDCS), 2007, pp. 3–10.