Facilitating the Design of Fault Tolerance in Transaction Level SystemC Programs

Ali Ebnenasir[1], Reza Hajisheykhi[2], and Sandeep S. Kulkarni[2]

[1] Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931, USA
aebnenas@mtu.edu

[2] Computer Science and Engineering Department
Michigan State University
East Lansing, Michigan 48824, USA
{hajishey,sandeep}@cse.msu.edu

Abstract. Due to their increasing complexity, today's SoC (System on Chip) systems are subject to a variety of faults (e.g., soft errors, 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 propose an approach where fault tolerance is designed at the level of inter-component communication protocols in SystemC Transaction Level (TL) models. The proposed method includes four main steps, namely model extraction, fault modeling, addition of fault tolerance and refinement of synthesized fault tolerance to SystemC code. We demonstrate the proposed approach using a simple SystemC transaction level program that is subject to communication faults. We also provide a roadmap for future research at the intersection of fault tolerance and hardware-software co-design.

Keywords: Fault Tolerance, SystemC, Automated 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

L. Bononi et al. (Eds.): ICDCN 2012, LNCS 7129, pp. 91–105, 2012.
© Springer-Verlag Berlin Heidelberg 2012

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* standard (on top of SystemC) that enables transaction-based interactions between the components of a system, called *Transaction Level* (TL) modeling [4]. Since SoC systems are subject to different types of faults (e.g., soft errors, hardware aging, etc.), it is desirable to capture fault tolerance in SystemC TL programs. However, capturing fault-tolerance in SystemC TL 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 TL 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 in 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 TL level) [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.

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 program analysis and fault-tolerance techniques to enable a framework for the addition of fault-tolerance concerns to SystemC TL programs. The proposed framework (see Figure 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 Figure 1, and the last two steps respectively focus on Problems 3 and 4.

Specifically, we start with a SystemC TL program that meets its functional requirements, but does not exhibit tolerance in the presence of a specific type of faults (e.g., soft errors, 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

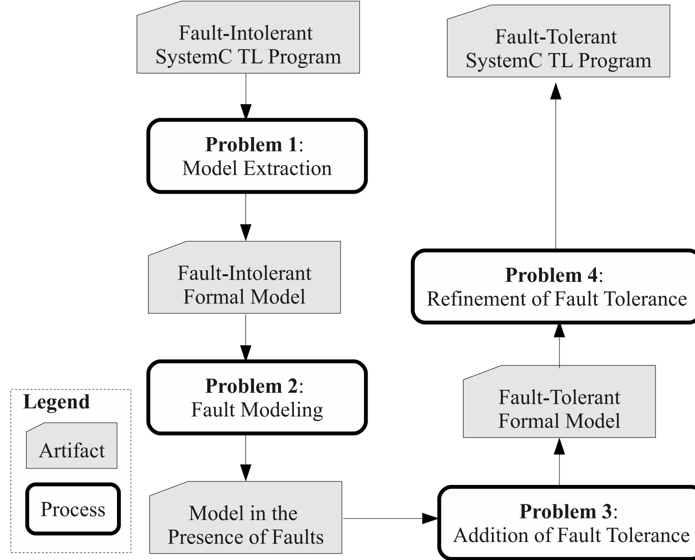

Fig. 1. Overview of the proposed framework

absence of faults. Then, we benefit from program analysis techniques (e.g., program slicing [16, 17] and abstraction [18, 19]) to generate a model of the SystemC TL program in a formal language. We augment the extracted model with a model of the faults that perturb the SystemC program. The resulting model is a *model in the presence of faults*. The *addition of fault-tolerance* requires a library of algorithmic methods for adding fault-tolerance at different levels of abstraction of TL programs in SystemC, namely Loosely-Timed (LT) and Approximately-Timed (AT). The LT abstraction enables fast simulation by having minimal timing requirements, whereas AT abstraction facilitates discrete-event simulation where the actions of all components are synchronized either by events or by a quantum time unit. To capture these levels of abstraction, appropriate formal models should be devised that capture the impact of faults on SystemC programs at different levels of abstraction. After synthesizing fault-tolerance concerns in formal models, one needs to determine what SystemC constructs should be generated corresponding to the components of formal languages. Such refinements should be semantics-preserving in that they should preserve the correctness of fault-tolerance aspects once refined to SystemC code.

Organization of the Paper. The rest of the paper is organized as follows: In Section 2, we present an overview of SystemC and Transaction Level Modeling. In Section 3, we present the approach for model extraction from SystemC, verification of functional requirements and fault modeling. Section 4 focuses on the addition of fault-tolerance and the refinement of fault-tolerance concerns from formal models to SystemC programs. Finally, Section 5 discusses issues raised by our work and Section 6 makes concluding remarks.

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++ program. In this function, the designer creates structural elements of the system and connects them throughout the system hierarchy. Each SystemC program has one or more *modules*. A module is the basic building block of SystemC TL programs that includes *processes*, *ports*, *internal data*, *channels*, and *interfaces*. A *process* is the main computation 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 functionality 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 [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 from the sensitivity list to be executed. 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 Figure 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 kernel to start the simulation. The event e will be notified when the trigger method of the target is called from the initiate process.

3 Model Extraction and Fault Modeling

The proposed approach starts with extracting a model from SystemC TL 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 [20] 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 TL program, we build on the ideas from [21], where we consider three basic processes *Behavior*, *Initiator* and *Target* for each module in the SystemC TL program. The *Behavior* process captures the main functionalities of a TL module. An *Initiator* and a *Target* process is considered for each transaction in which a TL module is involved. The simulation/execution of TL programs switches between these three processes by transferring the control of execution. The control transfer is either between (i) *Behavior* and *Initiator* of the same module or (ii) *Initiator* of one module and the *Target* of another module [21]. We use Promela (Process Meta Language) [22] 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

```
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

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 is not-interrupted.

The motivation behind using Promela is multifold. First, learning Promela is easy due to its C-like syntax. Second, developers can use the SPIN model checker [20] to simulate/model check extracted intolerant models and fault-tolerant models to gain a better understanding of faults and their impact on regular functionalities. Third, since there are already approaches that generate Promela models from SystemC programs for verification [23–25], the task of model extraction

could benefit from existing approaches. For instance, since a Promela program consists of proctypes, channels, and variables, the model extraction task transforms the given TL model into corresponding proctypes, channels and variables. Specifically, the proctypes are global in the Promela model, but channels and variables can be either global or local within a proctype. The proctypes capture the behavior of a model while variables and channels model the interface of proctypes.

For the program in Figure 2, the extracted Promela model M includes two proctypes named Initiator and Target (see Figure 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 Figure 3). 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 3 captures the specification of channels and the Initiator, Target and incModEight proctypes. The incModEight proctype models the *Behabior* process of the Target.

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

```
mtype = {inc, incComplt, startTrans, endTrans} // Message types
chan if2TgtBeh = [0] of {mtype} // Declare a synchronous channel
chan tgtIfPort = [0] of {mtype}
byte d =0;
int cnt = 0;  // used to model the occurrence of faults

active proctype Initiator(){
   byte recv;
       waiting:  tgtIfPort!startTrans;
                 tgtIfPort?recv;
                 initRecv = recv;  // initRecv is used to specify
                                   // desired requirements
       ending:   (recv == endTrans) -> fin: skip;       // (A)
 }

active proctype Target(){
   byte recv;
       waiting:  tgtIfPort?recv;
                 tgtRecv = recv;  // tgtRecv is used to specify
                                   // desired requirements
       starting: (recv == startTrans) -> if2TgtBeh!inc;     // (B)
                 if2TgtBeh?recv;
                 (recv == incComplt) -> tgtIfPort!endTrans;
  }

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

Fig. 3. The extracted functional model

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 Figure 2 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 [20], 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 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 $\Box(\mathsf{strtTr} \Rightarrow \Diamond\ \mathsf{endTr})$, 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 label A in Figure 3); i.e., 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 Figure 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 faults 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 Figure 2, can be perturbed by the type of faults that corrupt the messages communicated between the Initiator and the Target. To capture 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 a endTrans message to the Initiator, which is waiting for such a message; hence a deadlock.

4 Fault-Tolerant Model and Refinement

This section focuses on the next two steps (Problems 3 and 4 in Figure 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 add the following recovery action to the Target after the label B in Figure 3.

```
(recv != startTrans) -> goto waiting;
```

Likewise, we add the following action to the Initiator to ensure recovery. This statement is inserted after the label A in Figure 3.

```
(recv != endTrans) -> goto waiting;
```

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. 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. We augment the SystemC code with the constructs that implement the recovery actions generated in the earlier step. This is achieved using the reverse transformation rules that are the dual of the rules for generating Promela model in the first step.

Continuing with our example in Figure 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 waiting state where it waits for the next message. Figure 4 illustrates the refined SystemC program.

5 Discussion

This section discusses some concerns regarding fault tolerance in TLM, the level of automation, and fault modeling. Transaction level modeling is based on the principle of separating inter-component communications from computations. Due to the very large scale of integration in today's SoCs, hardware systems are subject to transient faults that cause temporary bit-flips in the circuitry of SoCs. Such faults have internal and external causes such as cosmic rays, hardware aging, etc. Since the occurrence of faults could perturb inter-component communications, it is increasingly important to design systems that tolerate such faults; hence the significance of designing fault tolerance in TLM.

```
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 implementing fault tolerance
23              SC_HAS_PROCESS(target);
24              target(sc_module_name name) : sc_module(name) {
25                      d = 0;
26                      ef = new sc_event_finder();
27                                      // Added for detection of channel faults
28                      SC_THREAD(incModEight);
29              }
30              void trigger() { e.notify(SC_ZERO_TIME); }
31              void incModEight() {
32              waitL:  wait(e);
33                      if (ef.find_event(target) != SC_ZERO_TIME)  goto waitL;
34                                      // Added for implementing recovery
35                      d = (d+1)%8;
36              }
37  };
38
39   int sc_main (int argc , char *argv[]) {
40          initiator initiator_inst("Initiator");
41          target target_inst("Target");
42
43          initiator_inst.port(target_inst);
44          sc_start();
45          return 0;
46   }
```

Fig. 4. Fault-tolerant SystemC program after refinement

While this paper proposes a methodology for facilitating the design of fault tolerance in TLM, there is a need for automating different steps of the proposed approach. Towards this end, we need abstraction rules that determine how SystemC constructs should be modeled in Promela or any other target modeling

language. Such rules specify semantics-preserving transformations that enable the extraction of models from SystemC programs. Moreover, such rules should be customized towards capturing fault tolerance in TLM. For example, in the SystemC example in this paper, 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 [26].

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 a LT context since timing concerns are not considered in order to enable faster simulation of design.

6 Conclusions and Future Work

In this paper, we presented a methodology for facilitating the design of fault-tolerance in Transaction Level SystemC programs. Using SystemC programming helps designers to model complex systems, which are a hybrid between hardware and software. The SystemC simulation kernel and C++ library of abstractions also help the designer to simulate concurrent processes conveniently.

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 [22]. Although we did not address the automation of this step, this step can in fact be automated using existing works such as [23, 26]. Subsequently, in the second step, we augmented the extracted model with faults. This step requires us to model the impact of faults on SystemC TL programs and capture them in the context of Promela [22]. Subsequently, we analyze the impact of faults using the SPIN model checker [20] 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 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 [27, 28] have shown feasibility of adding fault-tolerance to transition systems. However, this work needs to be adapted in this context to ensure that the revised program can be transformed to SystemC. Finally, we transformed the fault-tolerant Promela model to SystemC. To extend this step, we will devise a set of reverse transformation rules that enable the refinement of Promela models to SystemC TL 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.

There are several possible extensions of this work. The most direct extension is to automate the four steps involved in our method. As mentioned above, we expect that there is a significant potential to automate the first three steps although the last step may be difficult to automate fully. Hence, one future work in this context is to develop sufficient guidelines that will simplify the last step. There are new issues that have to be addressed in automating the first three steps. Specifically, regarding the first step, we need to annotate the abstract model in such a way that it would facilitate the generation of fault-tolerant model in the last step. Regarding the second step, the designer may want to enforce *fault-containment*. In particular, the designer may wish to 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. Regarding the fourth step, new rules need to be developed to ensure that any code added to the model 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.

References

1. Thomas, D.E., Lagnese, E.D., Nestor, J.A., Rajan, J.V., Blackburn, R.L., Walker, R.A.: Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench. Kluwer Academic Publishers, Norwell (1989)
2. Chen, Y.-Y., Hsu, C.-H., Leu, K.-L.: SoC-level risk assessment using FMEA approach in system design with SystemC. In: International Symposium on Industrial Embedded Systems, pp. 82–89 (2009)
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. Fin, A., Fummi, F., Martignano, M., Signoretto, M.: SystemC: A homogenous environment to test embedded systems. In: Proceedings of the Ninth International Symposium on Hardware/Software Codesign, CODES 2001, pp. 17–22 (2001)
6. Harris, I.G.: Fault models and test generation for hardware-software covalidation. IEEE Design and Test of Computers 20(4), 40–47 (2003)
7. Kundu, S., Ganai, M., Gupta, R.: Partial order reduction for scalable testing of SystemC TLM designs. In: Proceedings of the 45th Annual Design Automation Conference, pp. 936–941 (2008)
8. Sen, A.: Mutation operators for concurrent SystemC designs. In: International Workshop on Microprocessor Test and Verification (2000)

9. Blanc, N., Kroening, D.: Race analysis for SystemC using model checking. ACM Transactions on Design Automation of Electronic Systems 15(3), 21:1–21:32 (2010)
10. Marquet, K., Moy, M.: PinaVM: A SystemC front-end based on an executable intermediate representation. In: International Conference on Embedded Software (EMSOFT), pp. 79–88 (2010)
11. Misera, S., Vierhaus, H.T., Sieber, A.: Fault injection techniques and their accelerated simulation in SystemC. In: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, pp. 587–595 (2007)
12. Shafik, R.A., Rosinger, P., Al-Hashimi, B.M.: SystemC-based minimum intrusive fault injection technique with improved fault representation. In: Proceedings of the 2008 14th IEEE International On-Line Testing Symposium, pp. 99–104 (2008)
13. da Silva Farina, A., Prieto, S.S.: 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, pp. 237–244 (2009)
14. Perez, J., Azkarate-askasua, M., Perez, A.: Codesign and simulated fault injection of safety-critical embedded systems using SystemC. In: Proceedings of the 2010 European Dependable Computing Conference, pp. 221–229 (2010)
15. Giovanni, B., Bolchini, C., Miele, A.: Multi-level fault modeling for transaction-level specifications. In: Proceedings of the 19th ACM Great Lakes Symposium on VLSI, pp. 87–92 (2009)
16. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Higher-Order and Symbolic Computation 13(4), 315–353 (2000)
17. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: 7th International Workshop on SPIN Model Checking and Software Verification, pp. 113–130 (2000)
18. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. ACM SIGPLAN Notices 36(5), 203–213 (2001)
19. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: International Conference on Software Engineering (ICSE), pp. 439–448 (2000)
20. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
21. Niemann, B., Haubelt, C.: Formalizing TLM with Communicating Stat Machines. In: Proceedings of Forum on Specification and Design Languages 2006 (FDL 2006), pp. 285–292 (2006)
22. Spin language reference, `http://spinroot.com/spin/Man/promela.html/`
23. Traulsen, C., Cornet, J., Moy, M., Maraninchi, F.: A SystemC/TLM semantics in Promela and its possible applications. In: SPIN Workshop, pp. 204–222 (2007)
24. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: KRATOS – A Software Model Checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 310–316. Springer, Heidelberg (2011)
25. Campana, D., Cimatti, A., Narasamdya, I., Roveri, M.: An analytic evaluation of systemc encodings in promela. In: International SPIN Workshop on Model Checking Software (SPIN), pp. 90–107 (2011)
26. Ebnenasir, A.: UPC-SPIN: A Framework for the Model Checking of UPC Programs. In: Fifth Partitioned Global Address Space Conference, PGAS (to appear, 2011)
27. Ebnenasir, A.: Automatic Synthesis of Fault Tolerance. PhD thesis, Michigan State University (2005)
28. Bonakdarpour, B., Kulkarni, S.S.: Exploiting symbolic techniques in automated synthesis of distributed programs. In: EEE International Conference on Distributed Computing Systems(ICDCS), pp. 3–10 (2007)