Modeling and Analyzing Timing Faults in Transaction Level SystemC Programs

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

[1] Michigan State University
[2] Michigan Technological University

Abstract. Since SoC (System on Chip) systems are getting more complex everyday, they are subject to different types of faults including timing faults. Timing has a significant importance in SoC systems. However, their fault-affected models are not studied extensively. In this paper, we present a method for modeling and analyzing timing faults in SystemC Transaction Level Modeling (TLM) programs. The proposed method includes three steps, namely timed model extraction, fault modeling and timed model checking. We use UPPAAL timed automata to formally model the SystemC TLM programs and monitor how the models behave in the presence of timing faults. We analyze our method using two case studies. The first one uses loosely-timed coding style, which has a loose dependency between timing and data. The second one utilizes approximately-timed coding style, which has a stronger dependency between timing and data. Also, unlike the previous work, our approach preserve the semantics of transactions and timing concerns.

Keywords: Fault Modeling, SystemC, Transaction Level Modeling

1 Introduction

In order to increase design productivity of SoC (System on Chip) systems, there is a need to move from implementation-driven design at Register Transfer Language (RTL) [1] to higher levels of abstraction. This move introduces a shift in the development of electronic systems, which has been put into practice as Electronic System Level (ESL) design. The main advantages of ESL are to enable: system-level design, hardware/software co-design, architecture exploration, virtual prototyping, and co-simulation/co-verification [2]. The importance of ESL has become reality with *Transaction Level Modeling* (TLM) standard TLM-2.0 [3]. The C++-based system modeling language *SystemC* [4] perfectly supports TLM and hence is well-accepted for ESL design in industry. Since the TLM models serve as references for the RTL implementation, it is necessary to study the effect of faults in such models. Also in SoC systems, scheduling and timing play an important role. Hence, analyzing the system in the presence of timing faults is essential. This paper studies the effect of timing faults on SystemC TLM programs.

Formal verification and, in particular, model checking of finite models created from SystemC programs has been studied extensively in the past decade [5–12]. Moreover, some work have focused on transformations from SystemC or SystemC TLM programs into equivalent state machines [13–15]. Regarding fault localization, in [16–20] the authors propose fault/mutation models for SystemC TLM programs. They mainly target fault localization in their methods and do not analyze the SystemC TLM programs in the presence of faults. Also some related work [21–24] has focused on verification of SystemC TLM programs. Specifically, in [21,22], authors focus on timed and untimed fault-free behaviors of SystemC TLM programs respectively. In [23], we have focused on verification of untimed behaviors in the presence of transient faults. In [?], we have focused on verification of timed behaviors in the presence of transient faults, permanent faults and message loss. Nonetheless, none of the aforementioned work addresses the issue of timing faults in the context of SystemC TLM programs. Specifically, [21] focuses on fault-free behaviors and does not support timing faults. Although [21] permits modeling of timing properties (not faults), the generated models are often too large to perform significant analysis. The results in [23] focus on untimed behaviors and does not address timing faults. And, [24] does not address timing faults considered in this paper.

With this observation, the goal of our paper is to develop a methodology to support analysis in the presence timing faults. Since SystemC TLM programming is typically considered in two forms: loosely-timed (LT) and approximately-timed (AT), we consider both coding styles. The LT coding style utilizes b_transport for communication between modules. As the name suggests, in the LT coding style, there is a loose dependency between timing and data. The AT coding style utilizes nb_transport for communication between modules. This captures the global time more precisely. In this coding style, there is a stronger dependency between timing and data.

To model timing faults in SystemC TLM programs, our approach uses three steps: (1) *model extraction*, (2) *timing faults modeling*, and (3) *analyzing the model in the presence of timing faults*.

For the first part, we use the ideas in [21] and [23] to extract an UPPAAL timed automata model from the SystemC TLM program. Towards this end, each component in the SystemC TLM program is mapped to one or more timed automata. These timed automata interaction captures the communication between SystemC TLM components. This translation is based on a set of rules that identify the semantics of b_transport and nb_transport as well as different primitives in the transaction.

For the second part, we consider timing faults and inject them into the extracted timed automata model. Some of the timing faults we consider include transactions (or parts of transactions) that are delayed or finish too quickly. The effect of these faults under LT and AT coding style is different. Specifically, in LT coding style, the effect of a delay would affect all subsequent transactions. However, the order of transactions will not change. By contrast, in AT coding style, such a fault would cause a change transactions to finish in a different order. Also, the choice of timed automata can cause modeling of these faults more complex. For example, in a timed automaton, a clock can either increase continuously (at the fixed rate for all clock variables) or be reset, but it cannot be set to an arbitrary value. This constraint needs to be satisfied in generating the UPPAAL model.

In the third part, we use UPPAAL toolset to verify the extracted model and analyze how the model behaves in the presence of timing faults. Specifically, we evaluate whether the UPPAAL model that is subjected to timing faults satisfies (1) deadlock-freedom, (2) timing constraints for the original program, and (3) relaxed timing constraints.

We evaluate our approach with two case studies. Both case studies represent on-chip memory-mapped busses using TLM base protocol. The first case study uses loosely-timed coding style and b_transport interface for communication, whereas the second case study utilizes approximately-timed coding style and nb_transport interface for communication.

Contributions of the paper. We present:

- a set of transformation rules to extract the UUPPAAL timed-automata from SystemC TLM program.
- an approach for modeling timing faults in timed automata for SystemC TLM programs.
- an analysis of the fault-affected model for both loosely-timed and approximately-timed coding styles.
- two case studies that illustrate our approach.

Organization of the paper. The rest of the paper is organized as follows: In Section 2, we give a brief background of UPPAAL and SystemC TLM programs. In Section 3, we describe our approach for extracting UPPAAL model from the given SystemC TLM program. In section 4, we represent the first case study and explain our methodology using this case study. The second case study is explained in Section 5. Finally, the concluding remarks are presented in Section 6.

2 Background: SystemC, Transaction Level Modeling, and UPPAAL Timed Automata

This section provides a brief background on SystemC and its simulation kernel (Section 2.1), Transaction Level Modeling (Section 2.2), and UPPAAL timed automata (Section 2.3). The concepts presented in this section are adapted mainly from [3,4,25].

2.1 SystemC

SystemC is an open-source C++ class library that provides executable models of hardware-software systems at different levels of abstraction. The structural elements of a SystemC program are modules. Each module consists of the following elements: *processes*, *ports*, *internal data*, *channels*, and *interfaces*. Processes are the main computing elements of the module and every time an event is triggered, the process is executable. Events are instantaneous and have neither duration nor value. They are used for synchronization between processes and modules. Ports are objects through which a module communicates. Channels are communication elements that can be either very simple like a wire or complex like a FIFO. Ports use interfaces to communicate with channels.

SystemC provides an event-driven simulation kernel in C++ that enables the simulation of concurrent processes. SystemC scheduler, which is a part of the kernel, selects one of the processes to be executed from the sensitivity list. The sensitivity list consists of a set of events or timeouts that cause a process to be either resumed or triggered. The SystemC scheduler includes the following phases to simulate a system: i) initialization: each process is executed once; ii) evaluation: all processes that are ready-to-run are either executed or resumed in an arbitrary order; iii) update: primitive channels and signals are updated; iv) delta notification: A delta notification is an event resulting from an invocation of the notify() function with the argument

SC_ZERO_TIME. If there are delta-delay notifications, the corresponding processes are triggered; and v) timed notification: if pending timed notifications or timeouts exist, the scheduler identifies the corresponding sensitive processes and adds them to the set of runnable processes.

2.2 Transaction Level Modeling

Transaction Level Modeling (TLM) is an abstraction level above RTL [1] whose primary purpose is to accelerate simulation by utilizing function calls instead of using individual events and pins. In TLM, a *transaction* is an abstraction for an interaction between two or more concurrent processes for either data transfer or synchronization. TLM 2.0, in particular, targets transaction-level memory-mapped bus modeling to support virtual platform modeling of today's complex SoC systems. It is layered on top of the SystemC standard to provide higher simulation speed and interoperability, i.e., the ability to take TLM models from different sources and make them work together. In other words, SystemC constructs wrappers around the individual function calls to enable fast simulation and interoperable modeling. In a TLM model, an *initiator* is a module that initiates new transactions to exchange data or synchronize with the other module, called the *target*. Both the initiator and target try to maintain the interoperability while interacting. In TLM 2.0, the interoperability is introduced as a layer and has a set of main components as follows:

- Core interface which implements blocking and non-blocking interfaces,
- Generic payload which represents transaction objects,
- Sockets that connect initiator and target modules, and
- Base protocol that is a set of rules for using TLM interfaces while sending/receiving generic payload through sockets.

2.3 UPPAAL Timed Automata

Timed Automata (TA) are state machines that enable the modeling of real-time systems [26]. The notion of time is captured by real-valued *clock* variables. The clock values are used to express the timing constraints and can be assigned to locations (vertices) and transitions (edges) of the TA. The semantics of TA is given by an infinite-state transition system where transitions correspond either to a change of location (discrete transition) or to passage of time (time transition). UPPAAL [25] is an integrated tool environment for modeling, simulation, and verification of real-time systems modeled as networks of timed automata, extended with data types. A system in UPPAAL consists of concurrent processes, each of them modeled as a TA. Each process TA has a set of locations and transitions. To control transitions between locations, UPPAAL uses *guards* that limit when process actions can be executed and *synchronization channels* that require multiple processes to coordinate.

3 Model Extraction

In order to extract the UPPAAL timed automata in loosely-timed coding style programs using b_transport interface, we build on the ideas from [23], where a set of transformation rules is proposed. However, these rules generate untimed Promela models from untimed SystemC TLM programs, while we need to extract UPPAAL timed automata from timed TLM programs. Hence, we utilize the following rules similar to those in [23], where timed and untimed models are the same:

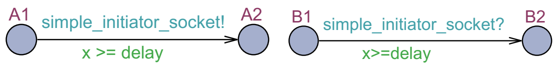
```
Rule 1:   tlm_utils::simple_initiator_socket<Initiator> socket;
          |--
          chan simple_initiator_socket[0];

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

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

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

Since timing issues are abstracted in [23], the rules in [23] ignore the timing parameters in the SystemC TLM program. To model timing faults, in this section, we introduce new rules that consider these timing parameters. We introduce *Rules 4* and *5* for extracting the UPPAAL model in LT coding style (i.e., with b_transport interface).These rules are described later in this section. We utilize rules from [21] for extracting the UPPAAL model in AT coding style (i.e., with nb_transport interface). We describe these rules, next.

Rules for LT coding style. In a transaction using b_transport interface, we need to consider the timing since one of the sending arguments is *delay*. If the transaction is from the Initiator to the Target, this argument describes the point of time in the future where the communication actually starts, and illustrates the ending time of the communication in the response. To transform a b_transport interface into UPPAAL, we define *Rule 4*, where a synchronization channel *simple_initiator_socket* is used (in Figure 1) to synchronize the Initiator and Target of a transaction. Figures 1(a) and 1(b) illustrate the Initiator and Target of the transaction respectively. The communication starts at *delay* timing point. Hence, the Initiator does not synchronize with the Target before that time. We take this constraint into account by defining the guard $x >= delay$ at the Initiator, where x shows the current simulation time *sc_time_stamp()*. In addition, the Target accepts the synchronization after the *delay* timing point. We consider this constraint by defining the guard $x >= delay$ at the Target. It means the communication starts after *delay* timing point.

```
Rule 4:    socket->b_transport( *trans, delay )
            |--
```



(a) (b)

Fig. 1. Transforming a b_transport interface into UPPAAL timed automata.

Moreover, in order to guarantee deterministic execution and increase the timing accuracy, a SystemC TLM program that uses loosely-timed coding style benefits from explicit synchronization points by utilizing calls to *wait()* function. This function is a synchronization-on-demand method that yields the control to the SystemC scheduler. To transform this function into the UPPAAL model, we define a variable *global-clock* that plays the role of the global time in a SystemC TLM program. To transform the wait function into the UPPAAL model, we define *Rule 5* as follows:

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

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

Rules for AT coding style. Having the approximately-timed coding style and nb_transport interface in a SystemC TLM program, we utilize our rules from [23] along with the ideas from [21] to extract the UPPAAL model. The ideas from [21] are used for transforming the SystemC scheduler and the payload event queue (PEQ). We model the PEQ with four different timed automata, namely *timed-ordered list*, *interface*, *event fetch and callback invocation*, and *PEQ event* automata. The first automaton is an ordered-list where tuples of a payload, a delay and a phase can be stored and sorted by their delay expiration time. The interface automaton is called by initiators and targets to insert a new PEQ notification. The third automaton removes elements from the PEQ if their delay expires and invokes the callback method associated with the PEQ (peq_cb). The last automaton models the event object notification.

4 Case Study 1: On-chip Memory-mapped Busses using LT Coding Style

In this section, we present our first case study, adapted from [27], that models an on-chip memory mapped bus using *loosely-timed* (LT) coding style and the *TLM base protocol*. We utilize this case study to formalize the timing faults in SystemC TLM programs using blocking transport interface.

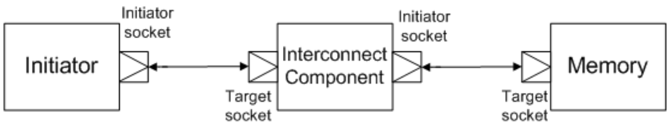


Fig. 2. The architecture of the memory-mapped busses model with an interconnect component.

4.1 Description of the SystemC TLM Program

In this example, we have three modules: Initiator, Router, and Memory. The Initiator module generates a transaction while the Memory (target) module represents a random access memory. The Router is a TLM 2.0 interconnect component that is placed between the Initiator and the Memory. An interconnect component is a

```
struct Initiator: sc_module
    ...
    void thread_process()
    {
        tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
        sc_time delay = sc_time(10, SC_NS);

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

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

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

        socket->b_transport(*trans, delay);

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

    int data;
};
```

Fig. 3. The Initiator module.

```
struct Router: sc_module
{
  tlm_utils::simple_target_socket<Router> target_socket;
  tlm_utils::simple_initiator_socket_tagged<Router>* initiator_socket;

  SC_CTOR(Router)
  : target_socket("target_socket")
  {
    target_socket.register_b_transport(this, &Router::b_transport);
    initiator_socket = new tlm_utils::simple_initiator_socket_tagged<Router>(txt);
    }  }

  virtual void b_transport(tlm::tlm_generic_payload& trans, sc_time& delay)
  {
    sc_dt::uint64 address = trans.get_address();
    sc_dt::uint64 masked_address;
    unsigned int target_nr = decode_address(address, masked_address);

    trans.set_address(masked_address);
    (*initiator_socket[target_nr])->b_transport( trans, delay);
  }
  // address decoding ...
};
```

Fig. 4. The Router modules.

```
53 struct Memory: sc_module
54     { ...
55     virtual void b_transport(tlm::tlm_generic_payload& trans, sc_time& delay)
56     {
57         tlm::tlm_command cmd = trans.get_command();
58         sc_dt::uint64    adr = trans.get_address() / 4;
59         unsigned char*   ptr = trans.get_data_ptr();
60         unsigned int   len = trans.get_data_length();
61         unsigned char*   byt = trans.get_byte_enable_ptr();
62         unsigned int    wid = trans.get_streaming_width();
63 
64         if ( cmd == tlm::TLM_READ_COMMAND )
65             memcpy(ptr, &mem[adr], len);
66         else if ( cmd == tlm::TLM_WRITE_COMMAND )
67             memcpy(&mem[adr], ptr, len);
68 
69         trans.set_response_status(tlm::TLM_OK_RESPONSE);
70     }
71     int mem[SIZE];
72 };
73 
74 
75 SC_MODULE(Top)
76 { Initiator* initiator;
77   Router* router;
78   Memory*    memory;
79 
80   SC_CTOR(Top)
81   { initiator = new Initiator("initiator");
82     router    = new Router("router");
83     memory    = new Memory(txt);
84 
85     initiator->socket.bind( router->target_socket );
86     router->initiator_socket->bind( memory->socket );
87   } };
```

Fig. 5. The Memory and Top modules.

component that forwards transactions from an incoming target socket to an outgoing initiator socket (see Figure 2 for the architecture). The module hierarchy among these three modules is shown by the top-level module (Lines 75-87 of Figure 5). In this module, the initiator socket of the Initiator module is bound to the target socket of the Router (Line 85 in Figure 5), and the initiator socket of the Router is bound to the target socket of the Memory module (Line 86 in figure 5). The Initiator has a thread process that starts the communication by sending a generic payload along with a delay parameter to the Router (Lines 3-26 in Figure 3). The Router utilizes its target socket and the b_transport interface (Lines 42-50 in Figure 4) to receive the transaction from the Initiator. After decoding the address, if it is needed, the Router forwards the transaction to the Memory (Line 49 in Figure 4). Finally, the Memory uses the b_transport interface (Lines 55-70 in Figure 5) to receive the transaction from the Router. Based on the Initiator's request, the Memory either reads from or writes to the data attribute of the generic payload (Lines 65-68 in Figure 5) and sends back the corresponding response.

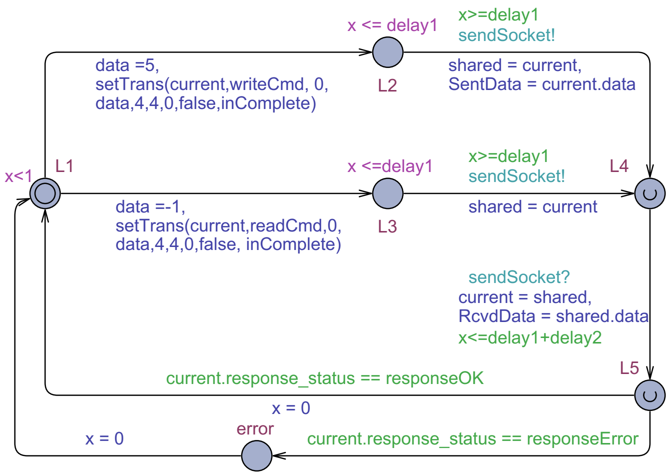


Fig. 6. Fault-intolerant UPPAAL timed automata model of the Initiator module. The green texts show either the *guards* or *synchronization*, the blue texts show the *updates*, and the pink texts represent the *names* and *invariants*.

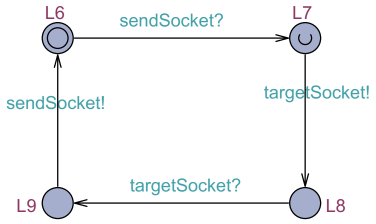


Fig. 7. Fault-intolerant UPPAAL timed automata model of the Router module.

4.2 Extracted Timed Automata Model

We use the rules and approaches explained in Section 3 to extract the UPPAAL timed-automata model from the SystemC TLM program of Figures 3, 4, and 5. Next, in Figures 6, 7, and 8 we identify the fault-free version of the program that forms the basis of models generated for timing faults. Figure 6 represents the Initiator model with the starting state L1. We utilize *Rules 2* and *3* to transfer the generic payload and TLM command (read/write) into the UPPAAL timed automata. Hence, States L2 and L3 represent places where the cmd attribute shows write and read commands respectively. After generating the transaction, we use *Rule 1* to define a synchronization channel sendSocket between the Initiator and Router, and use *Rule 4* to transfer b_transport interface into the UPPAAL model. As a result, in Figure 6, the Initiator synchronizes itself with the Router module and changes its state from either L2 or L3 to L4. The Initiator cannot send the transaction later than delay1 timing point, since the communication actually starts at that point. In state L6, the Router receives the transaction and changes its state to L7. Then, the Router synchronizes with the Memory via targetSocket channel and changes its state to L8. Note that the transaction is not actually sent from the channel, since the channels in UPPAAL are only for synchronization purposes. Then, we use *Rules 1* and *4* to transfer the b_transport interface and timing constraints in the Memory. If the guard $x >= delay$ is true, the Memory receives the transaction and changes its state to L11. The Memory cannot send its response later than $delay1 + delay2$, where $delay2$ shows the delay

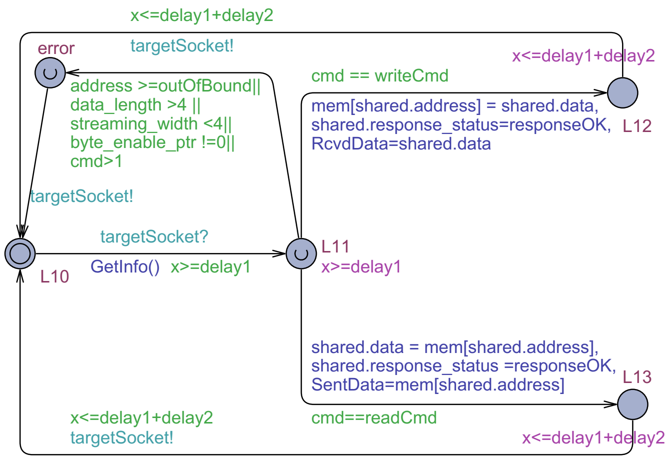


Fig. 8. Fault-intolerant UPPAAL timed automata model of the Memory module.

argument of the response message to the Initiator. After executing the write or read action, if there is no error, the Memory changes its state to either L12 or L13, and synchronizes itself with the Router via targetSocket channel. The Router, receives the response from the Memory, and, using the sendSocket channel, sends the response. Finally, the Initiator receives the response and can initiate another transaction.

Property specification and functional correctness. To ensure that the extracted timed-automata model captures the requirements/properties of the SystemC TLM program, we specify the requirements that should hold in the absence of faults. For this purpose, we define the following specifications in a subset of CTL (*Computational Tree Logic*) in Figure 9 that captures the requirements related to the timing constraints. These requirements should be always true in the absence of faults.

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

Fig. 9. Requirements of Memory Bus System using LT coding style.

The first requirement SPEC 1 represents that the extracted model is free of deadlock. The second requirement SPEC 2 illustrates that the Initiator module will not send the transaction towards the Memory module before the timing point delay1. The SPEC 3 shows that if the Initiator sends a transaction, the Memory will not receive it before the timing point delay1. Also if the Memory sends the response to the Initiator, it should not be received after the timing point delay1 + delay2. This requirement is represented in SPEC 4. Finally, the SPEC 5 shows that if the Initiator starts a transaction, it should not be finished after the timing point delay1 + delay2. Using UPPAAL, we have model checked the above properties for the extracted model of Figures 6, 7, and 8.

4.3 Modeling and Analyzing Timing Faults

In this part, we model timing faults and analyze their impacts on the model extracted above. A timing fault occurs when an event happens (or does not happen) in a specific time interval. Such timing faults could perturb the state of a system to an illegitimate configuration. In other words, the timing faults cause an action/operation to be executed either too early or too late, and, as a result, the operation cannot be performed properly. We introduce a variable $delay_t$ to the fault-free UPPAAL model that can model both early and late timing problems. The maximum value of $delay_t$ (default value is 1) is identified by the designer. The UPPAAL model is further modified to non-deterministically increase the delay argument in b_transport by $delay_t$ in all processes. Regarding executing an action too early, we also introduce a new transition between two locations, let say L_i and L_j, that increases the clock variable by a value less than that of the original transition between L_i and L_j. We utilize the aforementioned variable and transition for modifying the UPPAAL models in Figures 6, 7, and 8. Then, we identify the revised model and evaluate it in the presence of timing faults.

Thus, in our program, we use $delay_t$ to inject a delay to the extracted model. This injection can either change the clock variable x or the transition guards of the model. For instance, in the extracted timed automaton in Figure 6, by injecting the $dealy_t$, we change the guard of the transition (L4,L5) to x + $delay_t \leq$ delay1+delay2. As a result, the guard does not become true and the program gets stuck at location L4. This fault violates SPEC 1, SPEC 4, and SPEC 5 of Figure 9.

Analysis of the fault. If the Initiator executed too late, it is unable to receive the response sent by the Memory on time (at the *delay* timing point set by the Memory in the response). Also if the Memory sets the *delay* timing point but sends it too late to the Initiator, the Initiator does not receive it on time. We model it by injecting the timing faults into one of the locations (L4, L5), (L12, L10), (L13, L10). Injecting the faults into these locations violates requirements 1,4, and 5 of Figure 9. Table 1 represents more experiments based on the causes of some timing faults in the SystemC TLM program and their possible injecting locations in the UPPAAL model. If a requirement is violated, we show it by x_v, and if it is satisfied we show it by x_s, where x is the requirement number defined in the Figure 9. For example, when the Memory executed too late and the affected location is (L4, L5), the requirements SPEC 1, SPEC 4, and SPEC 5 are violated, while SPEC 2 and SPEC 3 are satisfied. The average time for checking all these requirements is 8 ms.

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

Table 1. Modeling and analyzing timing faults in the memory bus system while using LT coding style.

5 Case Study 2: On-chip Memory-mapped Busses using AT Coding Style

In this section, we present our second case study, adapted from [27], that focuses on an on-chip memory-mapped communication bus between an Initiator and a Memory module using *approximately-timed* (AT) coding style and *TLM base protocol*.

5.1 Description of the SystemC TLM Program

In this case study, the Initiator and the Memory use non-blocking transport interface (nb_trasport) for interaction. The nb_transport interface is intended to support the AT coding style and is particularly suited for modeling pipelined transactions. Unlike loosely-timed coding style presented in Section 4, the AT coding style cannot generally exploit temporal decoupling because of the need for timing accuracy. Instead, each process typically executes in lock step fashion with the SystemC scheduler. Each transaction in an nb_trasport interface has three arguments: *generic payload*, *delay*, and *phase*. The generic payload is the transaction object being sent. The delay represents the timing annotation of the communication. The phase, which is the only argument that the b_transport interface does not contain, indicates the state of the transaction, e.g., BEGIN_REQ for starting a transaction, and returns an enumeration value to indicate whether the return from the function also represents a phase transition.

This example breaks down each transaction into four phases, namely BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP, where each phase transition is associated with a timing point. The Initiator generates a transaction and starts the communication by sending a BEGIN_REQ using the forward path nb_transport_fw to the Memory. It cannot generate another transaction until receiving the Memory response END_REQ. The Memory uses the backward path nb_transport_bw to send the response END_REQ. After receiving the response, the Initiator can generate the next BEGIN_REQ. The Memory can also execute several requests in pipeline. In this example, we restrict the number of requests that the Memory can execute in pipeline to two. Multiple concurrent requests are implemented using *payload event queue* (PEQ). The PEQ maintains a queue of SystemC event notifications, where each notification contains a transaction object. Each transaction is enqueued into the PEQ with an annotated delay, and each transaction dequeued from the PEQ at a time calculated from the current simulation time plus the annotated delay. Thus, the Memory and Initiator can independently implement the delays of different phases. After implementing the read and write commands by copying data to or from the data attribute of the transaction, the Memory sends back the transaction along with a delay and the phase BEGIN_RESP to the Initiator. Finally, the Initiator ends the transaction by sending END_RESP (See Figure 11 in appendix for the message sequence chart).

5.2 The Extracted Timed Automata Model

We utilize the rules and approaches explained in Section 3 to extract the UPPAAL timed-automata model. In the model extracted, we also assign an identification to each transaction/payload. This identification helps us to track the transactions that executes in pipeline.

To ensure that the extracted timed automata capture the requirements of the SystemC TLM program, we specify a set of requirements that should hold in the absence of faults. These requirements capture the timing constraints and should be always true in the absence of faults.

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

Fig. 10. Requirements of memory bus system using AT coding style.

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

5.3 Modeling and Analyzing Timing Faults

To model the timing faults, we utilize the same approach as that explained in Section 4. Table 2 represents several experiment results based on the causes and locations of the timing faults. For example, if the Initiator executed too early, the affected locations in the UPPAAL model would be in the Initiator automaton and either before sending a transaction with phase BEGIN_REQ or before receiving the transaction with phase END_REQ. Also the average time for verifying the requirements are very similar to that for verifying the requirements in the absence of faults. Since the models generated are too large to add to this paper, we do not give the precise locations or include the models in this paper (The details are available at "`https://www.msu.edu/~hajishey/systemc.html`").

6 Conclusions and Future Work

In this paper, we focused on verification of SystemC TLM models in the presence of timing faults. Timing faults are one of the important faults in SoC systems and occur when an event happens (or does not happen) in a specific time interval.

We considered the effect of such timing faults in models that utilize both the LT (loosely timed) and AT (approximately timed). Regarding LT model, we presented an approach for model extraction that extended the approach in [23] by modeling of timing faults. This approach also utilized UPPAAL (as opposed to Promela considered in [23]) so that accurate timing properties can be verified. Regarding AT model, we utilized the approach in [21]. We further simplified the resulting UPPAAL

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

Table 2. Modeling and analyzing timing faults in the memory bus system while using AT coding style.

model by removing components unaffected by timing faults. This abstraction enabled verification of timing faults in the AT model.

We considered the faults when components execute too early or too late. To evaluate our approach, we considered on-chip memory mapped busses. In this example, both in LT and AT coding style, the time for verification in the presence of timing faults was small. However, there was a substantial gap between the verification time of LT and AT examples. For example, the average verification time for the faults where the Initiator executed too late in the LT model was 7.5 ms, whereas in the AT model was 17 s.

One future work in this area is to automate the analysis of timing faults completely. Currently, our analysis of timing faults is rule-based and identifies the changes that need to be done to the original UPPAAL model. We are currently working on automating these rules to develop a fully automated approach that takes the description of the timing faults from users and automates the analysis.

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. B. B. Grant Martin, A. Piziali, ESL Design and Verification: A Prescription for Electronic System Level Methodology, Morgan Kaufmann/Elsevier, 2007.
3. Transaction-Level Modeling (TLM) 2.0 Reference Manual, `http://www.systemc.org/downloads/standards/`.
4. Open SystemC Initiative (OSCI): Defining and advancing SystemC standard IEEE 1666-2005, `http://www.systemc.org/`.
5. 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.
6. A. Sen, Mutation operators for concurrent SystemC designs, in: International Workshop on Microprocessor Test and Verification, 2000.
7. 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.
8. 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.

9. 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.
10. 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.
11. K. Marquet, B. Jeannet, M. Moy, Efficient Encoding of SystemC/TLM in Promela, Tech. Rep. TR-2010-7, Verimag, France (2010).
12. 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.
13. A. Habibi, H. Moinudeen, S. Tahar, Generating finite state machines from systemc, in: G. G. E. Gielen (Ed.), DATE Designers' Forum, European Design and Automation Association, Leuven, Belgium, 2006, pp. 76–81.
14. J. Ruf, D. W. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiel, W. Müller, The simulation semantics of systemc, in: DATE, 2001, pp. 64–70.
15. A. Habibi, S. Tahar, An approach for the verification of systemc designs using asml, in: D. Peled, Y.-K. Tsay (Eds.), ATVA, Vol. 3707 of Lecture Notes in Computer Science, Springer, 2005, pp. 69–83.
16. H. M. Le, D. Große, R. Drechsler, Automatic tlm fault localization for systemc, IEEE Trans. on CAD of Integrated Circuits and Systems 31 (8) (2012) 1249–1262.
17. H. M. Le, D. Große, R. Drechsler, Scalable fault localization for systemc tlm designs, in: DATE, 2013.
18. N. Bombieri, F. Fummi, G. Pravadelli, A mutation model for the systemc tlm 2.0 communication interfaces, in: DATE, IEEE, 2008, pp. 396–401.
19. N. Bombieri, F. Fummi, G. Pravadelli, On the mutation analysis of systemc tlm-2.0 standard, in: MTV, IEEE Computer Society, 2009, pp. 32–37.
20. N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton, F. Letombe, Functional qualification of tlm verification, in: DATE, IEEE, 2009, pp. 190–195.
21. P. Herber, M. Pockrandt, S. Glesner, Transforming systemc transaction level models into uppaal timed automata, in: S. Singh, B. Jobstmann, M. Kishinevsky, J. Brandt (Eds.), MEMOCODE, IEEE, 2011, pp. 161–170.
22. 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.
23. A. Ebnenasir, R. Hajisheykhi, S. S. Kulkarni, Facilitating the design of fault tolerance in transaction level systemc programs, Theoritical Computer Science(ELSEVIER 2012)doi:http://dx.doi.org/10.1016/j.tcs.2012.11.010.
24. J. Chen, S. Kulkarni, Smt-based model checking for stabilizing programs, Tech. Rep. MSU-CSE-12-13, Department of Computer Science, Michigan State University, East Lansing, Michigan (October 2012).
25. G. Behrmann, A. David, K. G. Larsen, A tutorial on uppaal, in: M. Bernardo, F. Corradini (Eds.), SFM, Vol. 3185 of Lecture Notes in Computer Science, Springer, 2004, pp. 200–236.
26. R. Alur, D. L. Dill, A theory of timed automata, Theor. Comput. Sci. 126 (2) (1994) 183–235.
27. Getting started with tlm-2.0, http://www.doulos.com/knowhow/systemc/tlm2/.

Appendix

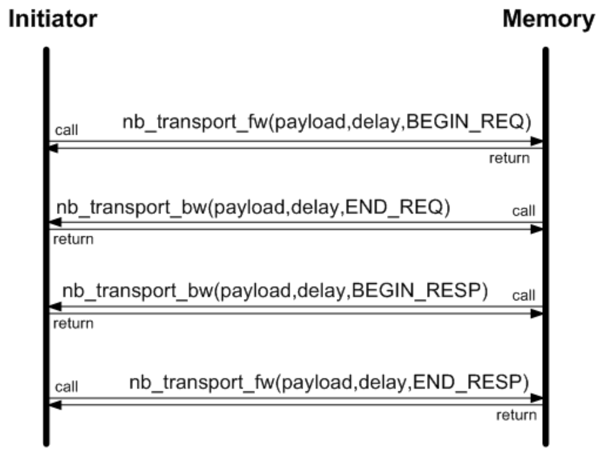


Fig. 11. Message sequence chart for non-blocking transport interface.

```
struct Initiator: sc_module, tlm::tlm_bw_transport_if<payload>
{ tlm::tlm_initiator_socket<32, payload, 1> socket;
  ...
  void thread_process()
  {... if (request_in_progress) wait(end_request_event);
      request_in_progress = tran;
      phase = tlm::BEGIN_REQ; delay = sc_time(100, SC_PS);
      tlm::tlm_sync_enum status;
      status = socket->nb_transport_fw( *tran, phase, delay );
      if (status == tlm::TLM_UPDATED)
      { m_peq.notify( *tran, phase, delay ); }
      ...
  tlm::tlm_sync_enum nb_transport_bw( payload& tran, int& phase, sc_time& delay )
  { m_peq.notify( tran, phase, delay );
    return tlm::TLM_ACCEPTED;  }

  void peq_cb(payload& tran, const int& phase)
  { if (phase == tlm::END_REQ || (&tran == request_in_progress && phase == tlm::
       BEGIN_RESP))
    { request_in_progress = 0;
      end_request_event.notify();  }

    if (phase == tlm::BEGIN_RESP)
    { check_transaction( tran );
      int fw_phase = tlm::END_RESP;
      sc_time delay = sc_time(100, SC_PS);
      socket->nb_transport_fw( tran, fw_phase, delay );
    } }
 };
```

Fig. 12. The initiator of the fault-intolerant SystemC TLM program using AT coding style.

```
struct Target: sc_module, tlm::tlm_fw_transport_if<payload>
{ tlm::tlm_target_socket<32, payload, 1> socket;

  payload*  next_response_pending;
  tlm_utils::peq_with_cb_and_phase<Target, payload> m_peq;
  tlm::tlm_sync_enum nb_transport_fw( payload& tran, int& phase, sc_time& t )
  { m_peq.notify( tran, phase, t);
    return tlm::TLM_ACCEPTED; }

  void peq_cb(payload& tran, const int& phase)
  { tlm::tlm_sync_enum status;
    sc_time delay;
    int bw_phase;

    switch (phase) {
    case tlm::BEGIN_REQ:
    ...
    status = socket->nb_transport_bw( tran, bw_phase, delay );
    if (status == tlm::TLM_COMPLETED)
    {  tran.release();  }
    delay = delay + sc_time(100, SC_PS);
    m_peq.notify( tran, internal_ph, delay );
      break;
    case tlm::END_RESP:
      tran.release();
      break;
    case tlm::END_REQ:
    case tlm::BEGIN_RESP:
      break;
    default:
      if (phase == internal_ph)
      { int cmd = tran.command;
        ...
     bw_phase = tlm::BEGIN_RESP;
     delay = SC_ZERO_TIME;
  status = socket->nb_transport_bw( tran, bw_phase, delay );
     if (status == tlm::TLM_UPDATED)
     {  m_peq.notify( tran, bw_phase, delay); }
    else if (status == tlm::TLM_COMPLETED)
     { tran.release();  }
        break;
      } } } ...
};
```

Fig. 13. The memory of the fault-intolerant SystemC TLM program using AT coding style.