A Framework for Verification of SystemC TLM Programs with Model Slicing: A Case Study *

Reza Hajisheykhi
Michigan State University
East Lansing, Michigan, USA
hajishey@cse.msu.edu

Mohammad Roohitavaf
Michigan State University
East Lansing, Michigan, USA
roohitav@cse.msu.edu

Ali Ebnenasir
Michigan Tech. University
Houghton, Michigan, USA
aebnenas@mtu.edu

Sandeep Kulkarni
Michigan State University
East Lansing, Michigan, USA
sandeep@cse.msu.edu

ABSTRACT

In this paper, we evaluate the effectiveness of model slicing to provide assurance about correctness of SystemC TLM programs. The need for such assurance is important since SystemC has become a de-facto standard for building systems with hardware/software co-design. Existing approaches that enable one to transform the given SystemC TLM program into an UPPAAL model that can be verified suffer from models that result in state space explosion. This problem becomes even more complex when verifying fault-tolerance. Model slicing has the potential to provide a solution to this problem. Therefore, we focus on developing a model slicer that extends existing work on model slicing and combines it with tools to generate UPPAAL models from SystemC TLM programs and tools to add the impact of faults to those UPPAAL models. The experimental results show that with the proposed framework, the designer is capable of verifying even very complex SystemC TLM models, which would have been impossible without the proposed approach.

1. INTRODUCTION

With the advances in VLSI technology, several problems in distributed systems and networks that are traditionally thought of as problems solved in software (e.g., routing, producer consumer problem, etc.) are now often designed with the use of hardware/software co-design. Two main areas of research and practice in this context are Systems on Chip (SoC) and Network on Chip (NoC). The initial work in this area focused on the use of Register Transfer Language (RTL) [10]. However, as these systems became more complex, there is a need to move 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 ESL design has become a reality with *Transaction-Level Modeling* (TLM) standard TLM-2.0 [4]. *SystemC* [3] is an IEEE industry standard that 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 verify them in the presence and absence of faults. For this reason, the role of formal methods (especially for verification) is critical for such systems. Hence, the goal of this work is to analyze the effectiveness of different approaches in managing the cost of verification of these systems in the absence and in the presence of faults.

A widely used approach in formal verification is to convert the system level design into a well-defined representation and then make use of an existing formal verification tool to analyze and verify the representation. For instance, [7] proposes a tool, called *STATE*, that takes a SystemC TLM program as an input and generates a timed automata model as an output. However, the timed automata model that STATE generates is often so complex that model checking them is not feasible. Moreover, this issue is made more critical by the fact that these systems are often subject to faults. Hence, we do not only need to verify them in the absence of faults but also in the presence of faults. Additionally, there are some researches on slicing SystemC/SpecC programs by using the ideas of slicing C/C++ programs (e.g. [9]). However, most of the existing work on slicing is for the purpose of simulation/testing and not for purpose of verification.

Our goal in this paper is to identify whether one can reduce the verification time of fault-free and fault-impacted SystemC TLM programs. Specifically, we focus on the use of model slicing considered in [8] to slice timed automata models. In general, slicing is a technique for extracting the parts of a program that effects the values computed at a statement of interest [11]. Model slicing, on the other hand, considers

*This work is supported by by NSF CNS 1329807, NSF CNS 1318678, and XPS 1533802.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org. DAC'16, June 05-09, 2016, Austin, TX, USA. ©2016 ACM. ISBN 978-1-4503-4236-0/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2897937.2897961.

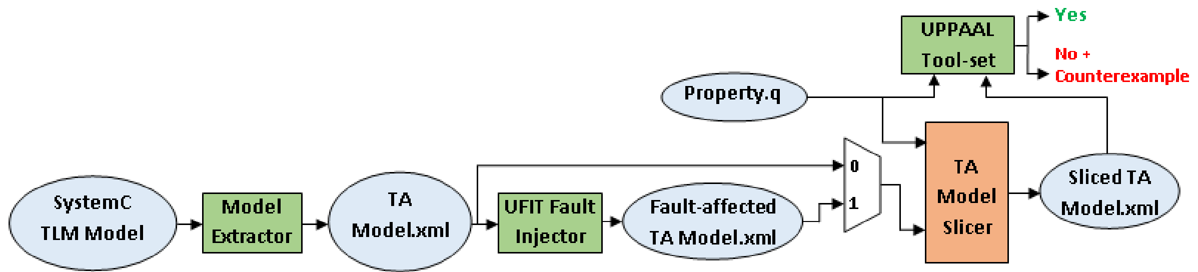

Figure 1: SystemC TLM model slicing framework.

slicing at the model level, e.g., state machine-based models.

In this paper, we focus on two case studies to analyze the effectiveness of model slicing in the verification of SystemC TLM programs. We consider a framework that combines [7] to extract an UPPAAL model from the given SystemC TLM program, [6] to add the effect of faults to that model, and the ideas in [8] to slice that model to obtain a smaller model that can improve the time/space efficiency of verification. We pursue this approach since slicing of SystemC programs requires one to consider intricacies of SystemC execution. Also, slicing of UPPAAL models is orthogonal to any optimization achieved by slicing of SystemC TLM program. Moreover, the models generated by [7] can be too complex to verify even for small SystemC TLM programs. Furthermore, to the best of our knowledge, the slicing algorithm in [8] has not been implemented. Hence, its effectiveness on realistic UPPAAL models is unknown. The slicing algorithm in [8] does not allow one to use features such as arrays or functions in UPPAAL and the models generated by [7] often involve arrays and/or functions. Hence, in this work, we implement and extend the algorithm in [8] to deal with functions and arrays as well. Our proposed framework is as follows:

- We utilize the tool *STATE* [7] to extract UPPAAL timed automata [5] from the given SystemC TLM model (*model extractor* module in Figure 1).
- We use the tool, *UFIT* [6], to inject different types of faults into the timed automata model (*UFIT fault injector* in Figure 1). UFIT takes a timed automata model as an input and generates a fault-affected automata model as an output.
- Our *model slicer* takes two inputs: a) the fault-free or fault-affected model, and b) a set of properties which needs to be verified. The slicer, based on the given properties, generates a simpler timed automata model to be given to *UPPAAL* tool-set for further analysis.
- The UPPAAL tool-set model checks the sliced model and gives us *yes* if the property of interests is satisfied. Otherwise it provides a counterexample as to how the model fails to meet the property.

We consider two case studies to evaluate the effectiveness of model slicing. The first case study is based on the producer consumer model and is conducted in Loosely-Timed (LT) coding style, where the designers mainly need fast simulation of a program with little care about timing concerns. The second case study is based on the memory-mapped buses and is conducted in Approximately-Timed (AT) coding style, where timing issues are also important to consider in simulation. In both these examples, we consider fail-stop fault that causes a component to fail and message loss. In all the fault-free and fault-affected models, we find that the model slicing improves the performance substantially. In particular, the reduction in time is 11%–99%. We also observed that in the LT example, model slicing does not help significantly with reduction in memory usage. In the AT example, model slicing is very important since it enables verification of several properties when the corresponding properties could not be verified in the original model due to unavailability of sufficient memory.

Organization of the paper. The rest of the paper is organized as follows: In Section 2, we briefly describe SystemC, TLM, and UPPAAL tool-set. Section 3 introduces our slicing framework. In Section 4, we show the effectiveness of our framework by two examples and analyze them in the presence and absence of faults. Finally, we conclude in Section 5.

2. PRELIMINARIES

This section provides a brief background on SystemC and Transaction-Level Modeling (Section 2.1), and UPPAAL tool-set (Section 2.2). The concepts presented in this section are adapted mainly from [3–5].

2.1 SystemC and Transaction-Level Modeling

SystemC is a C++ library, enabling designers to both implement and simulate a system using the library's structures and any C++ construct. A SystemC model consists of two parts: *elaboration phase* and *simulation phase*. In the elaboration phase, the design of the model is created. The structural elements of a SystemC model are modules. Each module consists of the following elements: *processes*, *ports*, *internal data*, *channels*, and *interfaces*. When the design of the system is ready, in the simulation phase, the resulting system gets executed. 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 a sensitivity list.

Transaction-Level Modeling (TLM) is an abstraction level above SystemC standard 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 components for either data transfer or synchronization. Globally, a TLM component is an encapsulated piece of code that contains active code (processes to be scheduled by the global scheduler) and passive code (functions offered to the external world, that will

be called from a process of another component, by a control flow transfer). Inside such a component, the processes and the functions may share variables and events in order to synchronize with each other. TLM 2.0 supports 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. The AT style of coding is used when timing issues are important to consider in simulation.

2.2 UPPAAL Tool-set

UPPAAL [5] is an integrated tool environment for modeling, simulation, and verification of real-time systems modeled as networks of timed automata, extended with data types. In other words, an UPPAAL model consists of a network of concurrent processes which are created by instantiating the pre-defined timed automaton templates, and these concurrent processes can communicate and synchronize with each other through parameters and channels defined. The system can be seen as a set of automata running concurrently, i.e., when there are multiple transitions enabled in the instance processes, these enabled transitions can take place in non-deterministic order.

3. VERIFICATION VIA MODEL SLICING

This section explains our proposed timed automata model slicing framework. This framework consists of four steps, namely model extraction, fault injection, model slicing, and UPPAAL model checking. We describe these steps in more details in Sections 3.1 and 3.2.

3.1 Model Extraction and Fault Injection

Given a SystemC TLM program, we utilize STATE toolset [7] to transform the program into the corresponding timed-automata model. The front-end of STATE is Karlsruhe SystemC Parser (KaSCPar) [2] that transforms the given SystemC TLM model into an Abstract Syntax Tree (AST). The back-end of STATE contains a set of transformation rules that transform the AST obtained from the KaSCPar into corresponding timed automata model. Afterwards, we use UFIT [6] tool-set to inject faults into the timed automata model. (If the designer needs to study the fault-free model, this step can be skipped.) The input of UFIT is a timed automata model and, based on a set of parameters that the designer specifies, UFIT generates a fault-affected timed automata model. UFIT models five different types of faults namely, message loss, byzantine, stuck-at, fail-stop, and transient faults.

3.2 Timed Automata Model Slicer

Once we have obtained the fault-free timed automata from STATE or the fault-affected model from UFIT, we use it along with a set of given properties as inputs to our model slicer.

Our slicer, similar to [8], uses a two-step approach. In the first step (Algorithm 1), the slicer identifies the locations, say R, and actions, say A (including variables, guards, and statements), that need to be preserved in the sliced automata. This is a recursive procedure where the initial set of states that need to be preserved are determined by the property under consideration, say ϕ (Lines 1-2 in Algorithm 1). For example, if the property under consideration is p *leadsto* q then a location that accesses p and q must be preserved in the sliced automata.

Algorithm 1 Timed-automata model slicing

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

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

Subsequently, the slicer identifies additional locations, variables, guards, and statements that need to be preserved. The reasons for preserving additional details in the sliced model include (1) control dependency, (2) data dependency, and (3) time dependency (Line 5 in Algorithm 1). As an illustration of control dependency, assume that location q_1 is preserved in previous iteration. Now, if the UPPAAL model includes a state such as q_2 such that (1) there is a computation from q_2 that reaches q_1 and (2) there is a computation from q_2 that never reaches q_1. Then, q_2 must also be preserved since we need to know whether the path followed from q_2 will reach q_1 or not. And, deciding whether q_1 is reached or not *can* affect satisfaction (or violation) of the property of interest. As an illustration of time dependency, consider the case where q_1 is preserved in the previous iteration. Suppose there is a path from q_2 to q_1 and the *time spent* in state q_2 can be nonzero then q_2 must also be preserved. (By definition, time spent in states that are marked urgent in UPPAAL is 0.)

Algorithm 2 Slice-Builder

Input: The sets of locations R and actions A, and Model M;
Output: Sliced timed automata model $M' = (Q', q^{0'}, X', T')$;

1: $Q' = R$;
2: **if** $(q^0 \in R)$ **then** $q^{0'} = q^0$;
3: **else** $q^{0'}$ = the first reachable location in R from q^0; **end if**
4: $T' = \bigcup out(R)$ s.t. action of each $out(R) \in A$;
5: **if** $(target(out(R)) \notin R)$ **then**
6: $target(out(R))$ = the first reachable location in R; **end if**
7: **return** M';

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

Intuitively, the UPPAAL model may have two types are arrays: (a) an array of automata and (b) an array of variables. When we have an array of automata with n entries then essentially, we replace it by n different automata. In each automaton, we need to replicate local variables but the global variables remain the same in all n automata. Similarly, for handling an array of variables with n entries, we replace it by n different variables. Subsequently, we need to replace every entry in every automaton that uses the array so that the array reference is replaced by the appropriate

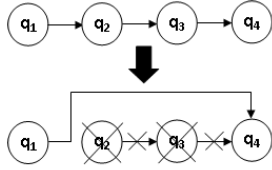

Figure 2: Building the sliced model.

variable. It also requires replicating the local variables in each automaton. This is acceptable since UPPAAL already does this in the verification process.

When the set of relevant locations and actions (R, A) is ready, in the second step, the slicer builds a revised model that only includes the relevant locations and actions (Algorithm 2). While building the sliced model, if the initial location of an automaton is not included in R, the first reachable location in R becomes the new initial state (Lines 2-3 in Algorithm 2). Also, if the target of an outgoing transition of a location in R is not included, the first reachable location in R becomes the target of that outgoing transition (Lines 5-4 in Algorithm 2). As an illustration, consider Figure 2 where q_1 and q_4 are relevant locations that need to be preserved and q_2 and q_3 are locations that are not relevant. In that case, the outgoing transition of q_1 goes into q_4. Moreover, the actions of each transition are those which are included in A (Line 4 in Algorithm 2). The proof of correctness of the model slicing approach is similar to [8], but it is omitted due to lack of space.

4. EXPERIMENTS

In this section, we apply our model slicing technique on two examples. The first one, Section 4.1, utilizes a LT coding style for modeling the SystemC TLM program. Such a style of coding heavily relies on a blocking transport interface *b_transport()*. The second example, Section 4.2, uses AT coding style for modeling the SystemC TLM program. In this style of coding, designers benefit from a non-blocking transport interface *nb_transport()*. In general, the blocking transport interface is only able to model the start and end of a transaction, whereas the non-blocking interface allows a transaction to be broken down into multiple timing points.

4.1 Case Study 1: Producer-consumer

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

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

The first property shows that the producer eventually generates some data. The second property represents that when the producer starts generating some data, the data will be eventually consumed by the consumer and the producer can start generating the next piece of data. The third property ensures that consumer consumes the data which is currently generated by the producer and the consumer won't try to remove data from an empty buffer. The fourth property shows that always consumer consumes the data generated by the producer. The fifth property represents that the consumer eventually consumes the data. Finally, the last property illustrates that the consumer's and producer's indices are never more than one apart. We have model checked these properties using UPPAAL and the results are available in Table 1. For the model checking, we use a personal computer with quad core CPU (2.8 GHZ each) and 6 GB memory. Next, we compare the verification time and memory usage for verifying the above properties of the timed automata model and its sliced model in the absence and presence of faults.

4.1.1 Slicing in the absence of faults

Once we have the fault-free timed automata model, we use the model and properties provided above to slice the model. Consider that we do not use UFIT since we want to study the model in the absence of faults. For each property, we generate a sliced model and compare the verification time, memory usage, number of states, and number of variables of the original/fault-free model and the sliced model generated by our model slicer. We observe that our slicing technique helps to simplify the model and reduce the time and memory needed for verifying the properties (see Table 1). For example, for verifying property $LT3$, the verification time, memory usage, number of states, and number of variables are reduced by 98%, 35%, 89%, and 92% respectively.

4.1.2 Slicing in the presence of faults

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

4.2 Case Study 2: Memory-mapped Buses

In this section, we present an example that utilizes AT coding style for modeling an on-chip memory-mapped communication buses between an initiator module and a target/memory module. In this example, adapted from [1], the initiator and the memory modules communicate through a non-blocking transport. The non-blocking transport is implemented according to the TLM base protocol, i.e., it breaks down each transition into four phases, namely *Begin_Req*, *End_Req*, *Begin_Resp*, and *End_Resp*, where each phase in a

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

Table 1: Comparison of the original and sliced models in the absence of faults while using LT coding style.

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

Table 2: Comparison of the original and sliced models in the presence of faults while using LT coding style.

transition is associated with a timing point. Moreover, in an AT coding style, each module has a queue called Payload Event Queue (PEQ). The PEQ is a time-ordered list of event notification in the TLM model. Utilizing STATE, we generate the timed automata model from the given SystemC TLM program. We also define a set of properties to ensure that the generated model is correct in the absence of faults. These properties are as follows:

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

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

4.2.1 Slicing in the absence of faults

We use UPPAAL tool-set to verify the above properties on the same personal computer as that in Section 4.1. However, we are not able to verify properties $AT2$, $AT3$, $AT4$, and $AT6$ since the model generated by STATE is too complex and the computer runs out of memory while verifying those properties (see Table 3). Also, the memory needed to verify $AT1$ and $AT5$, which are only reachability properties, is 0.99 GB. Therefore, we utilize our slicing technique to simplify the model based on the properties given. Using UPPAAL, we are able to verify all the properties in the sliced models and check if they are satisfied (s) or violated (v). For example, the verification of property $AT3$ in the corresponding sliced model takes 1 s and 476 ms, and the memory usage is 51.5 MB. Also the number of variables needed for verifying this property in the sliced model in 50, which is reduced by 81%.

4.2.2 Slicing in the presence of faults

We utilize UFIT to inject message and fail-stop faults into the timed automata model generated by STATE. Regarding the fail-stop faults, we consider the scenarios where the memory module is failed and the initiator module is not able to communicate with it. Since injecting the faults into the model makes the model more complex, verification of some properties (i.e., $AT1$ and $AT4$) is not feasible. Therefore, we give the fault affected model and the desirable property to the slicer, and the slicer generates a simplified model based on the property. Surprisingly, we are able to verify all the properties mentioned above in the sliced models (see Table 4). As an illustration, verification of property $AT4$, which was not feasible in the original model, takes 1 s and 250 ms and needs 49.9 MB memory. Also, the number of the variables in the sliced model is reduced by 79%.

In order to model the message faults, we assume that the messages with *Begin_reg* phase may get lost when the initiator is forwarding them to the memory module. Having this fault injected to the model, we are able to verify all the above properties in the sliced models (see Table 4). For instance, verifying property $AT2$ takes 201 ms and need 43.9 MB memory in the sliced model. This verification has reduced the time and memory usage by 14% and 96% respectively.

5. DISCUSSION AND CONCLUSION

In this paper, we focused on studying the effectiveness of model slicing in the verification of SystemC TLM programs

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

Table 3: Comparison of the original and sliced models in the absence of faults while using AT coding style.

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

Table 4: Comparison of the original and sliced models in the presence of faults while using AT coding style.

in the absence and presence of faults. The need for verification of such programs is due to the fact that advances in VLSI technology is enabling implementation of several problems that were thought of as software instances can now be designed with hardware/software co-design and SystemC is a de-facto standard for the same. Since the correctness of these SystemC programs is crucial for applications built on top of them, it is important to provide assurance about their correctness.

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

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

There are several future directions to this work. One direction is to repair the fault-affected model. Having a fault-affected timed automata mode and a set of properties which are violated in the sliced model, we are working on repairing the model automatically to generate a model that eventually satisfies the set of violated properties while preserving the set of satisfied properties.

6. REFERENCES

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

[2] Karlsruhe SystemC Parser Suite (KaSCPar). http://forge.greensocs.com/en/Projects/KaSCPar/.

[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] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In *SFM*, pages 200–236, 2004.

[6] R. Hajisheykhi, A. Ebnenasir, and S. Kulkarni. UFIT: A tool for modeling faults in UPPAAL timed automata. In *NFM*, pages 429–435, 2015.

[7] P. Herber, M. Pockrandt, and S.e Glesner. Transforming SystemC Transaction Level Models into UPPAAL timed automata. In *MEMOCODE*, pages 161–170, 2011.

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

[9] K. Tanabe, S. Sasaki, and M. Fujita. Program slicing for system level designs in SpecC. In *IASTED*, pages 252–258, 2004.

[10] D. Thomas., E. Lagnese, J. Nestor, J. Rajan, R. Blackburn, and R. Walker. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic Publishers, Norwell, MA, USA, 1989.

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