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.

Keywords
SystemC, Transaction Level Modeling, Model Slicing

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) [9]. 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.

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 [10]. Model slicing, on the other hand, considers slicing at the model level, e.g., state machine-based models. 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
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 tool-set [7] to transform the program into the corresponding timed-automata model. The front-end of STATE is Karl-sruhe SystemC Parser (KoSCPar) [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 KoSCPar 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 $\phi$ (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.

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 ($q_3' \in R$) then $q_3' = q_3$;
3: else $q_3' = \text{the first reachable location in } R \text{ from } q_3$; end if
4: $T' = \text{out}(R) \text{ s.t. action of each out}(R) \in A$;
5: if (target(out(R)) $\not\in R$) then
6: $\text{target(out(R))} = \text{the first reachable location in } R$; end if
7: $\text{return } M'$;

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 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 L (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:

\[
\begin{align*}
\text{LT1:} & \quad \text{E} \Rightarrow \text{producer.writenBuff} \\
\text{ LT2:} & \quad \text{producer.start} \rightarrow \text{producer.end} \\
\text{LT3:} & \quad \text{A}[\] \text{(producer.writenBuff } \&\& \text{ consumer.readBuffer)} \\
\text{ LT4:} & \quad \text{E} \Rightarrow \text{consumer.readBuffer} \\
\text{ LT5:} & \quad \text{A}[\] \text{(WriteIndex == ReadIndex)} \\
\text{ LT6:} & \quad \text{A}[\text{(WriteIndex==ReadIndex } \&\& \text{ WriteIndex == (ReadIndex+1)}%).]
\end{align*}
\]

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 forth 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 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
and the computer runs out of memory while verifying those properties. We use UPPAAL tool-set to verify the above properties on the same personal computer as that in Section 4.1. However, we cannot verify properties 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 model and check if they are satisfied (s) or violated (v).

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Original Model</th>
<th>Sliced Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Verification Time (ms)</td>
<td>Memory Usage (KB)</td>
</tr>
<tr>
<td>LT1</td>
<td>55</td>
<td>29,288</td>
</tr>
<tr>
<td>LT2</td>
<td>812</td>
<td>32,892</td>
</tr>
<tr>
<td>LT3</td>
<td>412</td>
<td>34,985</td>
</tr>
<tr>
<td>LT4</td>
<td>413</td>
<td>34,966</td>
</tr>
<tr>
<td>LT5</td>
<td>97</td>
<td>40,015</td>
</tr>
<tr>
<td>LT6</td>
<td>411</td>
<td>34,985</td>
</tr>
</tbody>
</table>

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

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 forth 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 AT5 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).

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 81%.

In order to model the message faults, we assume that the messages with Begin Req 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.
slicing is impossible even for simple examples. In case of LT would be essential for AT models where verification without combination in our case studies was substantial. For example, the property (speedup) combination in our case studies was substantial. For example, the property (speedup) was 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 0.) 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.

There are several future directions to this work. One direction is to repair the fault-affected model. Having a fault-affected timed automata model 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.

## 5. DISCUSSION AND CONCLUSION

In this paper, we focused on studying the effectiveness of model slicing in the verification of SystemC TLM programs 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 substantially. We anticipate that slicing would be especially beneficial for larger LT models where verification without slicing is impossible.

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Original Model</th>
<th>Sliced Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification Time (ms)</td>
<td>Memory Usage (KB)</td>
<td>No. of states</td>
</tr>
<tr>
<td>AT1</td>
<td>2,212</td>
<td>991,765</td>
</tr>
<tr>
<td>AT2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>AT3</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>AT4</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>AT5</td>
<td>2,644</td>
<td>991,992</td>
</tr>
<tr>
<td>AT6</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Fault</th>
<th>Location</th>
<th>Property</th>
<th>Original Model</th>
<th>Sliced Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fail-stop</td>
<td>Memory</td>
<td>AT1</td>
<td>v</td>
<td>180</td>
</tr>
<tr>
<td>Fail-stop</td>
<td>Memory</td>
<td>AT2</td>
<td>v</td>
<td>256</td>
</tr>
<tr>
<td>Fail-stop</td>
<td>Memory</td>
<td>AT3</td>
<td>v</td>
<td>282</td>
</tr>
<tr>
<td>Fail-stop</td>
<td>Memory</td>
<td>AT4</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Fail-stop</td>
<td>Memory</td>
<td>AT5</td>
<td>v</td>
<td>187</td>
</tr>
<tr>
<td>Fail-stop</td>
<td>Memory</td>
<td>AT6</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Mag-loss</td>
<td>Initiator</td>
<td>AT1</td>
<td>v</td>
<td>160</td>
</tr>
<tr>
<td>Mag-loss</td>
<td>Initiator</td>
<td>AT2</td>
<td>v</td>
<td>235</td>
</tr>
<tr>
<td>Mag-loss</td>
<td>Initiator</td>
<td>AT3</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Mag-loss</td>
<td>Initiator</td>
<td>AT4</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Mag-loss</td>
<td>Initiator</td>
<td>AT5</td>
<td>v</td>
<td>106</td>
</tr>
<tr>
<td>Mag-loss</td>
<td>Initiator</td>
<td>AT6</td>
<td>v</td>
<td>217</td>
</tr>
</tbody>
</table>

### 6. REFERENCES