TLM Protocol Compliance Checking at the Electronic System Level*

Mohamed Bawadekji Daniel Große Rolf Drechsler
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{bawadekji,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—Design and verification of embedded systems at the Electronic System Level (ESL) is common practice. In particular, Transaction Level Modeling (TLM) is the major reason for the success of ESL design. However, when detailed protocols are modeled at lower levels of TLM, the verification of the communication becomes a critical issue. In this paper, we present an approach for protocol compliance checking of new or detailed protocol implementations. They are checked against user-specified protocol sequences. We also analyze the protocol coverage achieved by the testbench and visualize the results on a protocol sequence graph. Experimental results for a SoC model demonstrate the advantages of our method.**

I. Introduction

SystemC has become an integrating language for the design and verification of embedded systems. In this *Electronic System Level* (ESL) design context the major advantages of SystemC are the development of hardware and software, the support of different levels of abstraction by *Transaction Level Modeling* (TLM) [1], the modeling of heterogeneous systems (containing digital, analog and RF) [2], [3], and the IP integration based on standardized TLM protocols [4].

Looking at a typical (idealized) ESL top-down flow, in the first step the specification is formalized into an algorithmic design. Here, the focus is on pure functionality of the system. Then, the refinement process starts removing degrees of freedom while making architectural decisions. For this task, TLM models in SystemC are created enabling various levels of timing accuracy and simulation speed. For example, loosely-timed TLM models aim early software development since they provide very fast hardware models. Approximately-timed models give more accurate timing and are used for architectural analysis and performance evaluation. Continuing the refinement, cycle-accurate (but not pin-accurate as in RTL) TLM models are built facilitating precise evaluation of the communication architecture at the expense of simulation speed. Finally, these TLM models are refined to RTL.

A key reason for the success of TLM in ESL design is the straightforward integration of IP. The interoperability is provided by the OSCI TLM-2.0 standard [5] enabling communication of models from different sources off-the-shelf, that is without any modification of the models. The main ingredients of the TLM-2.0 standard for modeling communication are the generic payload, the base protocol and the core-interfaces (e.g. *b_transport* and *nb_transport_fw, _bw*). In addition, the usage of standard interfaces and well-defined coding styles offer the capability to abstract or refine a communication protocol. When refining the communication – for example to verify scheduling polices or communication priorities – the lifetime of a transaction is extended. This is done by defining new TLM protocol phases which describe the communication protocol in more detail. However, this procedure increases the complexity of the protocols. Moreover, the more detailed TLM models serve as reference models for RTL and the decisions resulting from the analysis approaches need reliable descriptions.

In this paper we propose the first approach for protocol compliance checking of SystemC TLM models communicating with new or detailed protocol implementations. From compact user specifications of protocol sequences a graph representation is built. During simulation this graph is traversed to check the correct protocol phase transitions as well as the TLM responses. If they differ from the specification, this violation and the executed transaction path is reported to the user. Furthermore, a protocol coverage analysis has been developed to identify unverified sequences automatically. This allows to improve the quality of the testbench and to find more bugs.

Overall, we summarize the contributions of this paper as follows:

- User-friendly protocol sequence specification
  The TLM protocol sequences are specified with a very intuitive C++ interface.
- Irredundant protocol sequence graph generation
  Efficient algorithms have been developed which merge identical protocol sequences resulting in a graph representation which contains no redundant nodes.
- Protocol compliance checking and coverage
  The TLM communication is checked against the user-specified protocol sequences. In addition, uncovered protocol sequences are identified automatically.
- Protocol graph and result visualization
  A graphical representation of the protocol graph is generated showing the TLM protocol phases and responses. In addition, the unexecuted protocol sequences are marked on the graph.

We present experimental results for a TLM SoC model demonstrating the advantages of our approach.

The rest of this paper is structured as follows: Related work is discussed in Section II. Section III describes the preliminaries. The proposed approach is introduced in Section IV. Section V gives the experimental evaluation. Finally, the paper is concluded in the last section.

II. Related Work

For designs modeled at lower levels of abstraction, that is RTL and below, several approaches for protocol compliance verification have been proposed. For example, in [6] properties are represented in a HDL and checked during simulation. The generation has also been automated in [7]. In contrast, formal methods have been used to prove the protocol correctness in a mathematical sense [8], [9], [10]. But all these approaches can not be applied at TLM.

Recently, also approaches for assertion-based verification at TLM have been introduced, see for example [11], [12]. On

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 01M3088.

the one hand these approaches require complex specification languages to describe TLM behavior. On the other hand the support for TLM-2.0 is limited. In contrast, our approach offers a very intuitive and compact way for describing protocol sequences. This results in particular from the developed C++-interface provided for specification.

To the best of our knowledge, only one method has been developed for the task of SystemC TLM protocol verification: the checker from Doulos [13]. However, the method can only be used for checking the TLM-2.0 base protocol. Our approach aims to support the designer when modeling new protocols or refining the current implementation (beyond the TLM base protocol), i.e. additional TLM phases are included.

III. Preliminaries

In this section we provide the preliminary terms needed to describe our approach[1]. Typically, the communication between SystemC TLM modules is performed through channel's interfaces and via ports/exports of the modules. To increase the interoperability between IP modules, the communication between SystemC TLM modules has been standardized by the *Open SystemC Initiative* (OSCI) by providing generic API interfaces. The current version of the TLM-2.0 standard [5] defines two types of transport interfaces, namely the *blocking* and *non-blocking* transport interface.

The blocking transport interface is defined such that the lifetime of each transaction is restricted to only two timing points, corresponding to the call to and return from the *b_transport* method. The blocking transport is generally used for modeling a hardware design at high level of abstraction without considering implementation details. By this a so-called *loosely-timed* model is provided which is used for early software development.

The non-blocking transport interface allows the lifetime of a transaction to be extended to multiple protocol-specific phases. Therefore, multiple function calls are required to execute a full transaction. Two non-blocking transport methods are defined: *nb_transport_fw* for *forward path* and *nb_transport_bw* for *backward path* (see also Fig. 1). Both method calls are performed via corresponding sockets. An initiator (respectively target) socket contains a port for interface method calls on the forward (respectively target) path and an export for interface method calls on the backward (respectively forward) path. The non-blocking transport is used when exploring new TLM communication protocols with different degrees of accuracy as well as when promising design alternatives are further refined to the so-called *approximately-timed* model.

In addition, TLM-2.0 introduced a standard transaction *payload* considering a set of private attributes such as command, address, data and response status. It supports also the extension of the generic payload by providing the extension mechanism.

A non-blocking method returns a value from the set $\{TLM\_ACCEPTED, TLM\_UPDATED, TLM\_COMPLETED\}$ to indicate whether the transaction object has been updated by making a phase transition. If the callee accepted the call, both, the transaction and the phase should remain unchanged. Finally, the completion of the transaction can be explicitly indicated by returning *TLM_COMPLETED*.

IV. TLM Protocol Compliance Checking

In this section the proposed approach is presented. At first, the general idea and the overall flow are given. Then, the specification mechanism for protocol sequences is introduced. Next, the algorithms for protocol graph generation, protocol compliance checking and the protocol sequence coverage are described.

[1]Due to space limitation we refer the reader for a general introduction into SystemC and TLM to [14], [15].

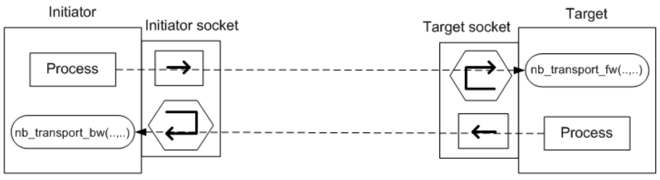

Fig. 1. OSCI TLM-2.0 connectivity between initiator and target

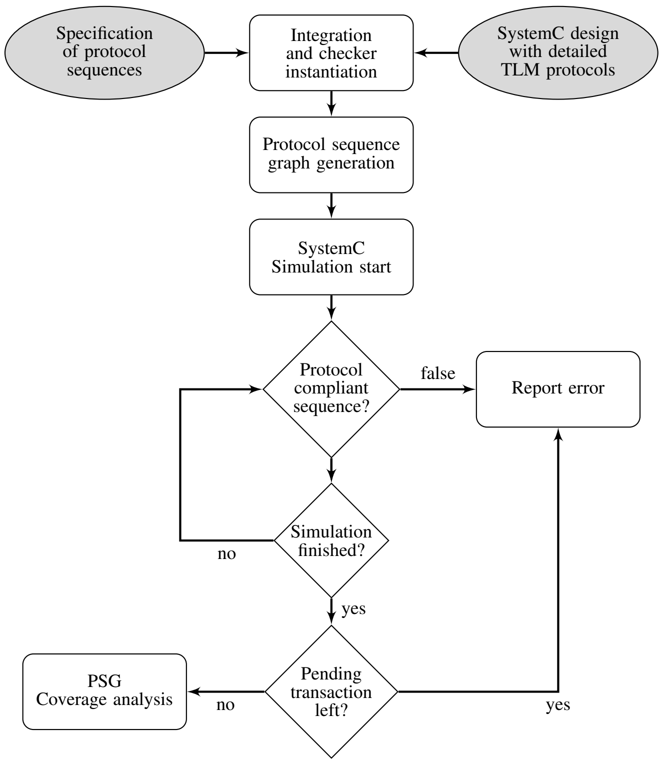
Specification of protocol sequences
Integration and checker instantiation
SystemC design with detailed TLM protocols
Protocol sequence graph generation
SystemC Simulation start
Protocol compliant sequence?
false
Report error
Simulation finished?
no
yes
Pending transaction left?
PSG Coverage analysis
no
yes

Fig. 2. Overall flow for protocol compliance checking

A. General Idea and Overall Flow

When the design team begins to create TLM models for architectural analysis and performance evaluation, the communication of the actual models needs to be refined. More precisely, instead of only relying on the TLM base protocol for communication the non-blocking interfaces are used and new timing points are added by introducing additional TLM phases. Obviously, the complexity of the corresponding protocol implementation increases. Hence, ensuring the functional correctness becomes more difficult. Since TLM-2.0 provides only the modeling features we propose an approach for automatically checking the protocol compliance of the implementation against user-specified protocol sequences.

The overall flow of our approach is depicted in Fig. 2. At first, the user specifies the legal TLM protocol sequences via a very intuitive C++-interface and by this also integrates the specification with the SystemC TLM design. This results in carrying out the generation of the protocol sequence graph representing all specified protocol sequences. Moreover, the protocol checker as well as the system modules have to be instantiated. Now, after starting the simulation of the system, each TLM transaction is checked to behave according to the specification. If a concrete check during the lifetime of the transaction fails, e.g. an incorrect phase transition is performed or the TLM return value is invalid, this error is directly reported to the user. Otherwise, all sequences have finished as intended or it is possible that a transaction has not reached the final state (indicated as *pending* results by our checker algorithm). In the latter case an error is reported. In the former case a coverage analysis is performed by our approach to determine unverified protocol sequences.

In the following, all these steps and the respective algorithms are described in detail.

B. Protocol Sequence Specification and Checker Instantiation

For specifying a TLM protocol sequence we use the following grammar:

$$\begin{aligned} S \;&::= S \cdot S \mid Line \\ Line &::= \{\; phase\;,\; \{retV\}\; \} \end{aligned}$$

where $phase$ is an element of the set of all TLM phases defined for the user-specified protocol at SystemC TLM level and $retV$ is a list of possible TLM return-values for the current phase, i.e. $retV$ can contain at most $\mathit{TLM\_ACCEPTED}$, $\mathit{TLM\_UPDATED}$, and $\mathit{TLM\_COMPLETED}$. This allows to describe valid TLM phase transitions and possible TLM return values which are described line by line. Note that the protocol sequences can be passed to the checker interface either individually by calling the checker interface several times (where a protocol sequence describes a single protocol path only) or in a combined manner (where a protocol sequence defines several protocol paths).

Consequently, we decided that the user specifies the protocol sequences via a C++-interface. Therefore, we used the new C++0X standard draft [16] which is supported already by the GNU compiler since version 4.4.0 for implementing the interface. This allows us to directly map protocol sequences to a variadic function, i.e. one which accepts a variable number of arguments. We give a concrete example where our checker has been instantiated at top-level before:

Example 1: An example protocol sequence for an initiator-target communication using the developed C++-interface is shown in Fig. 3 and 4, respectively. In both figures at first, in Line 2 a unique number (ID) for the current protocol sequence has to be defined. The concrete sequence is defined from Line 4 to Line 10. The protocol sequence in Fig. 3 defines a single protocol path only since the return value list in each line contains at most one element. In contrast, the protocol sequence of Fig. 4 shows how to specify several protocol paths, namely eight paths, very compact. Basically, each possible return value is a predecessor node for the following phase. However, according to the TLM-2.0 standard if the callee returns *TLM_UPDATED* as result from a forward call, the callee directly sets a new phase in the transaction. Hence, no backward call from the callee is necessary which has to be accepted by the initiating caller. Therefore, in this case a written *TLM_ACCEPTED* in the next line of the protocol sequence is ignored (see e.g. Line 5 in Fig. 4). In typical bus communication this corresponds to cases where the callee is a bus which can either directly grant the bus (corresponding to returning *TLM_UPDATED*) or recognizing the request and handling it later actively (corresponding to returning *TLM_ACCEPTED* and making an own call later which has to be accepted).

```
1    Top.checker->set_protocol_sequence(
2        0,
3        {
4            { { BUS_REQ     } , { TLM_UPDATED  } },
5            { { GRANT_BUS   }                   },
6            { { BEGIN_REQ   } , { TLM_ACCEPTED } },
7            { { END_REQ     } , { TLM_ACCEPTED } },
8            { { BEGIN_DATA  } , { TLM_UPDATED  } },
9            { { END_DATA    }                   },
10           { { UNGRANT_BUS} , { TLM_COMPLETED} }
11       }
12     );
```

Fig. 3. Example for describing a single protocol sequence

```
1   Top.checker->set_protocol_sequence(
2       0,
3       {
4           { { BUS_REQ     } , { TLM_ACCEPTED, TLM_UPDATED } },
5           { { GRANT_BUS   } , { TLM_ACCEPTED              } },
6           { { BEGIN_REQ   } , { TLM_ACCEPTED, TLM_UPDATED } },
7           { { END_REQ }   } , { TLM_ACCEPTED              } },
8           { { BEGIN_DATA  } , { TLM_ACCEPTED, TLM_UPDATED } },
9           { { END_DATA    } , { TLM_ACCEPTED              } },
10          { { UNGRANT_BUS} , { TLM_COMPLETED             } }
11      }
12    );
```

Fig. 4. Example for describing several protocol sequences

Before we can show the (optimized) graphical representation, the graph build from all protocol sequences as basic data structure for protocol compliance checking is introduced in the following.

C. Protocol Sequence Graph Generation

The *Protocol Sequence Graph* (PSG) represents all specified protocol sequences. Formally, a PSG is defined as follows:

Definition 1: A PSG is a rooted, directed graph $G = (V, E)$, where the vertex set V contains three types of vertices: A *phase-vertex* has the name of a TLM phase as attribute while a *return-value-vertex* holds the name of one of the three possible TLM return values $\{\mathit{TLM\_ACCEPTED}, \mathit{TLM\_UPDATED}, \mathit{TLM\_COMPLETED}\}$. The third type of vertices are nodes to represent the root of the PSG (termed start vertex $S \in V$) as well as the terminal of the PSG (final vertex $F \in V$). The edges define the dependencies between the TLM phases and TLM return-values.

Since we are interested in a compact representation of the PSG for the user-specified protocol sequences, we build the PSG such that no redundant successor vertices (or subgraphs) are created. For this task we have developed Algorithm 1.

The basic idea of the algorithm is to add each protocol sequence to the initially empty PSG G. Therefore, the current protocol sequence PS is processed top-down line by line while using two global maps to avoid the creation of redundant vertices (Line 17 - Line 32). We have specified the auxiliary function *create_or_lookup+* (Line 1 - Line 12) to make the core of the algorithm easier to follow. The key value used there for hashing is a tuple containing two elements: The first element describes either the name of a phase or the name of a return value read in the currently processed $Line$ of the protocol sequence. The second element stores a list of predecessor vertices V_{preds} specified by evaluating the previous $Line$ of the current protocol sequence (Line 24, 30, 32). If no entry for the key exists in the map, a new vertex for the appropriate phase or return value is created and connected to all its predecessors (stored in V_{preds}; Line 8). Finally, the

Algorithm 1: Build protocol sequence graph

```
   Input: TLM protocol sequence $PS$
   Output: Add $PS$ to PSG $G$
1  // $V_{preds}$ is a list of vertices
2  create_or_lookup+($(name, V_{preds}), map$)
3  begin
4      if $(name, V_{preds}) \in keys(map)$ then
5          $v$ =map[$(name, V_{preds})$];
6      else
7          $V = V \cup \{v\}$;
8          foreach $w \in V_{preds}$ do
9              $E = E \cup \{(w, v)\}$;
10         map[$(name, V_{preds})$]=$v$;
11     return($v$);
12 end
13
14 foreach $(ph_i, RetV_i) \in PS$ from first to last do
15     if $TLM\_UPDATED \in RetV_i$ and $|RetV_i| \geq 2$ then
16         $RetUP^+ = RetUP^+ \cup ph_{i+1}$;
17 $v_{retv} = v_{ph} = S$;
18 foreach $(ph, RetV) \in PS$ from first to last do
19     if $ph_{preds} == \emptyset$ then $ph_{preds} = \{v_{ph}\}$ ;
20     $v_{ph}$=create_or_look up+($(ph, ph_{preds}), map$);
21     $ph_{preds} = \emptyset$ ;
22     if $ph \in RetUP^+$ then
23         $v_{phSec}$=create_or_lookup+($(ph, \{ph_{uppred}\}), map\_up$);
24         $ph_{preds} = \{v_{phSec}\}$ ;
25     $ph_{uppred} = \emptyset$ ;
26     if $RetV \neq \emptyset$ then
27         foreach $retv \in RetV$ do
28             $v_{retv}$= create_or_lookup+($(retv, \{v_{ph}\}), map$) ;
29             if $retv == TLM\_UPDATED$ then
30                 $ph_{uppred} = \{v_{retv}\}$ ;
31             else
32                 $ph_{preds} = ph_{preds} \cup \{v_{retv}\}$ ;
```

key value is inserted and initialized with the new created vertex.

To distinguish between several protocol sequence paths described in a single function call to the checker interface, specific phases have to identified depending on the return values to create two vertices for the same phase during adding the PS (see Line 22 - Line 24). This is the case if the size of the return value list $RetV$ in the processed $Line$ is at least two and contains the return value *TLM_UPDATED* (Line 14 - Line 16), then the following phase has to be inserted to $RetUP^+$. The vertices representing the return values of the current protocol sequence $Line$ are created from Line 26 - Line 32. Since the proposed algorithm avoids creating redundant vertices until the names of two vertices differ, we additionally employ an optimization step (not shown) in which all remaining redundant paths are identified and merged in bottom-up manner. This will increase the efficiency of the protocol compliance checking algorithm during the simulation.

D. Protocol Compliance Checking

In this section, we introduce the *Protocol Compliance* (PC) checking mechanism of our approach. Basically, when a transaction is started we have to traverse the PSG from the root and proceed according to current status of the transaction. If the transaction behaves according to the protocol specification we finally reach the terminal vertex of the PSG while always performing valid TLM phase transitions and obtaining legal TLM responses, respectively.

Algorithm 2: Protocol sequence compliance checking

```
   Input: transaction $trans$, name of phase or return value $name$,
          PSG $G$
   Output: $complete$, $violate$ or $pending$ stating compliant
           transaction, failure or no violation so far
1  check_transaction($trans, name, G$)
2  begin
3      $P = mapt[trans].P$ ;
4      $name\_matched = false$ ;
5      $(v_0, v_1, \ldots, v_k) = P$ ;
6      foreach $(v_k, v_{succ}) \in E$ do
7          if $name(v_{succ}) == name$ then
8              $P.add(v_{succ})$ ;
9              $name\_matched = true$ ;
10             break;
11     if $name\_matched == true$ then
12         $mapt[trans].P = P$;
13         $(v_0, v_1, \ldots, v_l) = P$;
14         if $v_l == F$ then
15             mark_path_as_visited($P$);
16             $status = complete$;
17         else
18             $status = pending$;
19     else
20         $status = violate$;
21     $mapt[trans].status = status$;
22     return $status$;
23 end
```

When simulating the TLM design a transaction is observed with monitors between the TLM modules. The monitors are inserted during binding and provide the current transaction to the centralized PC checker. By doing this, the PC checker can analyze the current transaction before forwarding the non-blocking transport call to the destination module and also before the backward function is executed (see also Fig. 1). To distinguish between different transactions we use a map hashing the address of the transaction (denoted later $mapt$). This address remains the same during the lifetime of a transaction while updating the transaction elements and the related phase. The associated hash values for a transaction consists of:

- $status$: The result when checking the current transaction. The status can be *complete* if the transaction conforms to the protocol which means that for the transaction a protocol path from the root to the terminal has been successfully traversed. In case of *pending* the final result can not be decided yet. The last value is *violate*, i.e. a protocol violation has just been detected.
- P: This is the path (list of vertices) already traversed for the current transaction.

The main part for protocol compliance checking is shown in Algorithm 2. When the PC checker is called for the first time for a transaction, the path P for this transaction is initialized to the start vertex of the PSG (not shown in the algorithm). Now, during simulation $check\_transaction$ is called by the PC checker whenever a new phase transaction or TLM response should be carried out. Thereby, the name of the new phase or return value is passed as argument $name$. The algorithm checks whether the end vertex of the already traversed transaction path P has a successor matching to $name$ (see Line 7). If this is the case, the path is extended by this vertex (Line 8). If this new vertex is the terminal vertex (Line 14), the current transaction is protocol compliant and the status becomes *complete*. In addition, we mark the path

P of the compliant transaction as visited during simulation for the coverage analysis later. Otherwise, the result can not be determined yet and the status is set to $pending$ (Line 18). If no matching successor vertex could be found (illegal phase transition or illegal return value with respect to the user-specified protocol sequences), the status is set to $violate$ (Line 20). Finally, the status of the current transaction is updated in the hash table.

At the end of the simulation we also check whether there is any pending transaction left which is also reported as error.

Note that a complex TLM design consists typically of multiple interconnects and hence a transaction is transported through several TLM modules via successive calls to the non-blocking transport methods. In this case, if the protocol compliance checker is used as a central module for verifying the entire protocol specification, the same transaction phase or return value would be verified against the PSG each time a successive call to the non-blocking transport interfaces is made. This would lead to a protocol violation, since the PSG is constructed depending on the phase transitions of the specified protocol and not according to the components which basically forward the transaction. To overcome this problem, we call Algorithm 2 only once at the beginning of the successive call sequence, but store the phase and the return value. This allows us to check if a protocol violation occurs if the transaction is not passed correctly to the destination.

E. Protocol Sequence Coverage

Before detailed protocol implementations can be further refined to lower levels and finally to RTL it is very important to thoroughly verify the communication. Hence, the effectiveness of the test generation needs to be guaranteed.

Thus, we analyze the protocol sequence coverage using the PSG to detect protocol sequences that have never been executed during the simulation. This points the design team to either a weakness of the test generator or to incorrect behavior of the TLM model with respect to the protocol specification.

In order to identify uncovered protocol sequences on the PSG, we firstly have to check for each particular vertex how many times the outgoing edges had been traversed from each of its incoming edges during the simulation. For this purpose, we assign to each in- and outgoing edge several counters such that each counter uniquely defines for an outgoing edge which incoming edge has been traversed during simulation. Once a given transaction successfully reache the final node and describes a particular path, the appropriate counters of all related edges will be incremented. This step is performed by calling the function shown in Algorithm 2, Line 15.

After the simulation has finished, the PSG is traversed using a depth-first search algorithm. For each visited vertex, all counters of any associated outgoing edge are automatically computed and compared against the counters of the incoming edges. In this way, any uncovered protocol sequence as well as all sequences which are not completely traversed can be found. Furthermore, an outgoing edge will be marked in PSG if all associated counters of this edge are zero. Finally, the graphical representation of PSG is generated to visualize the uncovered protocol paths.

V. Experimental Evaluation

This section presents the experimental evaluation of the proposed approach. At first the SoC model used for evaluation is described. Then, the results obtained by our approach are presented.

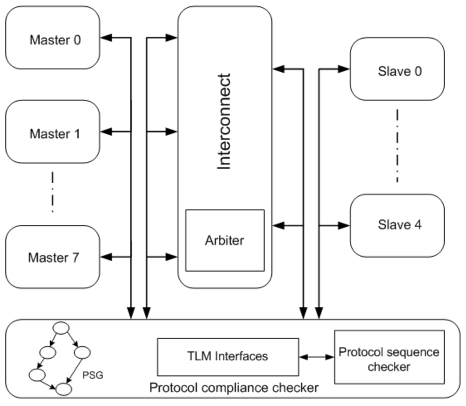

Fig. 5. SoC model

A. Model Description

The proposed protocol compliance checking approach has been applied during the design of a SoC model running a bus protocol based on AMBA AHB specification [17]. The components have been implemented at high level of abstraction using the OSCI TLM-2.0 *approximately-timed* coding style. The SoC model consists of a bus-arbitration module and 8 masters generating the required test traffic and 5 slaves representing SRAM and ROM memories with different response time (see Fig. 5). The target architecture is configurable and allows several parameters such as the number of processing units, delay times and memory parameters to be configured for each simulation run.

The data transfer for read or write access is executed in the SoC model as a single transfer (also termed single burst in the AHB terminology). Therefore, a master has to request access to the bus each time it wants to exchange data with other modules. In addition, the communication delays for transactions and bus arbitration policy as well as the computation times have been modeled. To ensure synchronization between the involved masters, we use the round-robin arbitration policy to control the access on the bus. In the following we describe the protocol sequences using the TLM phases of the implemented parts of the AMBA AHB protocol:

- Each master requests access to the bus with the phase *BUS_REQ* resulting in a query of the arbiter.
- If the bus is not busy, it assigns the appropriate master access with the phase *GRANT_BUS*.
- When a master gets access to the bus, it starts requesting the target slave with the phase *BEGIN_REQ* through the bus. At this point, all required information like address, byte length and access type can be set in the given transaction. The target slave sets the phase to *END_REQ* if it is ready to handle the master request.
- For write transfer the master sends the data through the phase *BEGIN_DATA*, for read transfer the slave answers with the phase *BEGIN_RESP*. The completion of the transfer is notified through the phase *END_DATA* or *END_RESP*, respectively. In these cases, the master cannot access the bus anymore which is communicated by the phase *UNGRANT_BUS*.

These protocol sequences have been realized in TLM-2.0 and hence a complex protocol implementation results; it handles the different protocol states in the respective components according to the TLM return values. Note, that a phase transition can only take place if the return-value of the non-blocking transport is *TLM_UPDATED*. Otherwise, a TLM module has to return the value *TLM_ACCEPTED* if it doesn't change any of the transaction parameters or if the data transfer can not be finished now. When returning *TLM_COMPLETED*, the lifetime of the current transaction has to be terminated by the initiator.

B. Results

In the elaboration phase, i.e. just before the simulation starts, all modules are instantiated and bound according to the configuration (given by a configuration file). Moreover, the PSG is build for the specified protocol sequences. By running the simulation, the protocol compliance checker indicated a protocol violation and aborted the simulation immediately. By analyzing the reported error it was clear that the SRAM updated the phase from *BEGIN_REQ* to *BEGIN_RESP* and loaded the required data to the master in response to the master read request. This phase transition is valid in the base protocol, i.e. the target can annotate directly the response delay time and can start data transfer. However, the AHB protocol allows data transfer only if the target acknowledges explicitly that it has executed the master request correctly by returning the value *TLM_ACCEPTED* or updating the transaction phase to *END_REQ*. When implementing this protocol functionality the designer assumed the same behavior as in the base protocol and hence caused this bug.

We simulated the SoC model with the same testbench configuration again after fixing the bug in the SRAM. The simulation finished successfully without any protocol violation. However, the protocol sequence coverage approach identified untested protocol flow. Fig. 6 depicts the relevant protocol sequences. The beginning of the unexecuted protocol path can be seen in the upper right part of the figure (edges are shown dashed red). This problem can be due to the two following reasons: The testbench is inadequate or there is a bug in the design. In our case, the test generator provides the required test traffic for each module with respect to the configured delay times to stimulate each protocol path. Thus, after analyzing the bus implementation, we recognized that a specific case has not been implemented, namely, if only one master requests access to the bus at a certain time point where also the bus is not busy and no request is stored in the request list, then it shall update the transaction phase to *GRANT_BUS* by returning the value *TLM_UPDATED* without triggering the arbitration process. In summary, our proposed approach found two major bugs in the non-trivial protocol implementation of our SoC model.

VI. Conclusions

In this paper we have presented an approach for TLM protocol compliance checking. From user-specified protocol sequences a protocol sequence graph is built. During simulation the graph is traversed for each transaction to determine the protocol compliance. Furthermore, we identify unverified protocol paths to improve the testbench or revealing additional design bugs.

The experimental results have clearly shown the advantages of our approach. During the design of a SoC model at approximately-timed level using parts of the AMBA AHB protocol for communication two protocol implementation bugs have been found.

For future work, we plan to extend our method for supporting a pipelined burst transfer mode for on-chip communication protocols. Moreover, we also want to check user-defined extensions for a given transaction.

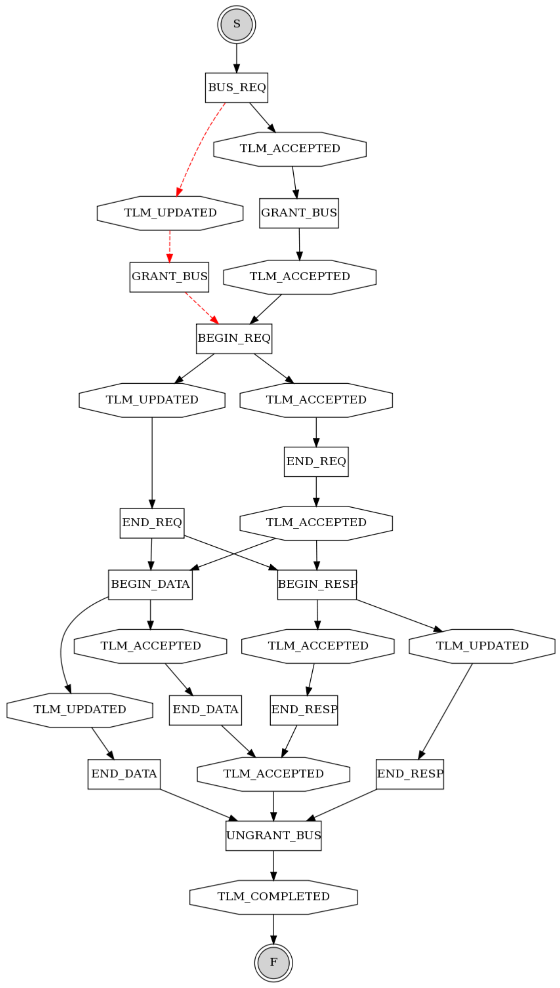

Fig. 6. Protocol sequence graph describing the protocol sequences for the write and read accesses

References

[1] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2003, pp. 19–24.
[2] "SystemC-AMS," http://www.systemc-ams.org.
[3] A. Vachoux, C. Grimm, and K. Einwich, "SystemC-AMS requirements, design objectives and rationale," in *Design, Automation and Test in Europe*, 2003, pp. 10 388–10 395.
[4] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.
[5] J. Aynsley, *OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL*. Open SystemC Initiative (OSCI), 2009.
[6] K. Shimizu, D. Dill, and A. Hu, "Monitor-based formal specification of PCI," in *Int'l Conf. on Formal Methods in CAD*, ser. LNCS, vol. 1954, 2000, pp. 335–353.
[7] M. Oliveira and A. Hu, "High-level specification and generation of IP monitors," in *Design Automation Conf.*, 2002, pp. 129–134.
[8] Y.-C. Yang, J.-D. Huang, C.-C. Yen, C.-H. Shih, and J.-Y. Jou, "Formal compliance verification of interface protocols," in *International Symposium on VLSI Design, Automation, and Test*, 2005, pp. 12 – 15.
[9] G. Fey, D. Große, and R. Drechsler, "Avoiding false negatives in formal verification for protocol-driven blocks," in *Design, Automation and Test in Europe*, 2006, pp. 1225–1226.
[10] M. D. Nguyen, M. Thalmaier, M. Wedler, D. Stoffel, and W. Kunz, "A re-use methodology for formal SoC protocol compliance," in *Forum on specification and Design Languages*, 2009.
[11] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull, "Interactive presentation: Implementation of a transaction level assertion framework in SystemC," in *Design, Automation and Test in Europe*, 2007, pp. 894–899.
[12] L. Ferro and L. Pierre, "ISIS: runtime verification of TLM platforms," in *Forum on specification and Design Languages*, 2009, pp. 1–6.
[13] J. Aynsley, "TLM-2.0 Base Protocol Checker." [Online]. Available: http://www.doulos.com/knowhow/systemc/tlm2/base_protocol_checker
[14] D. C. Black and J. Donovan, *SystemC: From the Ground Up*. Springer-Verlag New York, Inc., 2005.
[15] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2006.
[16] "Working draft, standard for programming language C++." [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3126.pdf
[17] ARM, *AMBA Specification (Rev. 2)*, 1999.