Scalable Simulation-based Verification of SystemC-based Virtual Prototypes

Mehran Goli Rolf Drechsler
Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{mehran.goli, rolf.drechsler}@dfki.de {mehran, drechsler}@uni-bremen.de

***Abstract*—*Virtual Prototypes* (VPs) at the *Electronic System Level* (ESL) written in SystemC language using its *Transaction Level Modeling* (TLM) framework are increasingly adopted by the semiconductor industry. The main reason is that VPs are much earlier available, and their simulation is orders of magnitude faster in comparison to the hardware models implemented at lower levels of abstraction (e.g. RTL). This leads designers to use VPs as reference models for an early design verification. Hence, the correctness assurance of these reference models (VPs) is critical as undetected faults may propagate to less abstract levels in the design process, increasing the fixing cost and effort.**

In this paper, we propose a novel simulation-based verification approach to automatically validate the simulation behavior of a given SystemC VP against both the TLM-2.0 rules and its specifications (i.e. functional and timing behavior of communications in the VP). The scalability and the efficiency of the proposed approach are demonstrated using an extensive set of experiments including a real-word VP.

I. Introduction

The increasing functionality of digital systems, and reduced time-to-market constraints push designers to model systems at higher levels of abstraction. Particularly, hardware modeling at the *Electronic System Level* (ESL) received strong attention in the last decades. Among high-level hardware description languages, SystemC [1] has become a de-facto standard at the ESL due to its fast simulation and strong support for mixed hardware-software systems. This has led to the rapidly-growing adoption of *Virtual Prototypes* (VPs) written in SystemC using its *Transaction Level Modeling* (TLM) framework [2]. The much earlier availability and the significantly faster simulation speed of the VP in comparison to the *Register Transfer Level* (RTL) hardware model motivate designers to use it as a reference model for an early system verification in the design process. Hence, ensuring the correctness of VPs is of the utmost importance, as undetected faults may propagate to lower levels of abstraction and become very costly to fix.

In the ESL design, TLM-2.0 (as the current standard) provides designers with a set of standard interfaces and rules (TLM-2.0 base protocol) to model a VP based on abstract communications (i.e. transactions). It allows designers to abstract away the implementation details related to the computation of *Intellectual Properties* (IPs) and only focus on the communications. Thus, communications (among different IP blocks) are the main part of a VP model that must be verified. The first step to verify the communications in a given VP is to check whether or not they adhere the TLM-2.0 rules. In addition, as a VP is the first executable model of the design specifications describing its functionality and timing behavior in terms of abstract communications, a functional assurance of the VP against its specifications is necessarily required. Especially, if the VP under development represents a safety-critical system. Therefore, to ensure the correctness of communications in a given VP, apart from validating the VP against TLM-2.0 rules (protocol validation), the functional and timing behavior of the VP must be verified as well.

In general, the SystemC-based VP correctness can be ensured by two different approaches: formal verification and simulation-based verification (also called validation). Formal approaches usually require to specify the model in formal semantics such as abstract state machines [3]–[6] or *Intermediate Representation* (IR) models [7]. However, due to the object-oriented nature and event-driven simulation semantics of SystemC, it is very challenging to formally verify a given SystemC VP. Moreover, state space explosion is another well-known problem for this category. Due to these restrictions, formal approaches are not able to verify complex systems as several assumptions need to be imposed on a given VP (e.g. function pointers, recursion or templates cannot easily be described formally). In contrast, simulation-based verification approaches [8]–[13] are still the predominant techniques to verify systems at ESL as they scale very well with an arbitrary complexity of VPs. In this scope, assertion-based techniques [10]–[12], [14] are particularly well-suited for validation purposes. However, the major drawback of them is that deriving assertions (i.e. properties) from the design specifications or TLM-2.0 rules usually requires manual effort by designers. Moreover, mostly the generated assertions need to be inserted manually into the VP.

In this paper, we propose a simulation based verification approach which automatically validates a given SystemC VP against the TLM-2.0 rules and its specifications. The main challenges to be overcome here are to deal with the SystemC specific language constructs, the TLM-2.0 semantics and timing models. We take advantage of the flexible compiler infrastructure Clang [15] provided by Clang-LLVM [16] to analyze the *Abstract Syntax Tree* (AST) of the VP model for run-time information extraction by automatically generating an instrumented version of the VP. A post-execution analysis is applied to the extracted information to build the simulation behavior of the VP and describe it in terms of abstract communications (i.e. transactions). Properties are automatically generated from the design specifications and TLM-2.0 rules. Finally, the simulation behavior of the VP is validated against the generated properties. The violated properties and the corresponding transactions are reported back to the designers for further analysis. The focus of the proposed approach is to detect the errors related to the most common and important fault types of communications in a VP at ESL; i.e. dynamic

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SecRec under grant no. 16K1S0606K, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

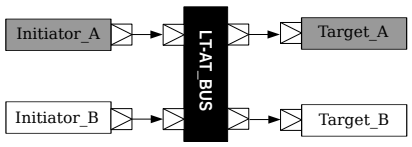

Fig. 1: The Architecture of *LT-AT_BUS* VP.

rules (that cannot be checked statically e.g. during compilation time) related to the TLM-2.0 base protocol transactions (and its attributes), functionality and timing behavior. The approach is applied to several case studies including a real-world VP to demonstrate its precision, scalability, and advantages.

II. Related Work

SystemC-based VPs verification includes both formal and simulation-based approaches. Section I already studied notable approaches of the former category. In this section, we focus only on simulation-based approaches.

In *Assertion-based Verification* (ABV) approaches [10]–[12], [14] VP's specifications or protocol rules that need to be verified are expressed as assertions. An assertion is a statement written usually in *Property Specification Language* (PSL) or System Verilog and checked either statically using model checkers or dynamically during simulation time. The main drawback of these approaches is that they require complex specification to formulate e.g. TLM-2.0 rules in terms of assertions. This needs to be performed manually by designers which heavily reduces the degree of automation. Furthermore, most of them specifically focus on validating communications in the VPs against the TLM-2.0 rules and do not support the correctness of their functional and timing behavior.

In addition to the ABV approaches, [8], [9] use *Aspect Oriented Programming* (AOP) to access simulation behavior of VPs and then validate it against the VPs' properties. Essentially, AOP provides designers with a source-to-source translation where additional code (aspects) can be inserted to specific points of the VP's source code. In [8], the simulation behavior of VPs is traced using AOP technique and checked against the properties implemented as a C++ class. In [9] user code primitives are defined manually in property specifications by designers. Then, AOP is used to instrument the SystemC source code by generating a monitor for each property to be check during the execution. For both the aforementioned methods, user interaction is heavily required to define the aspects and design primitives. Additionally, defining and debugging AOP setups are very challenging and require further manual efforts by designers.

The recently published method [13] takes advantage of the *GNU debugger* (GDB) to retrieve the simulation behavior of VP models (introduced in [17]). The extracted VP's behavior and TLM-2.0 rules are translated into a set of time automata models, and properties in TCL, respectively. Although it has shown to be a promising approach, it has two major limitations. As the method uses the GDB breakpoints feature to extract the run-time data, the required time for the analysis can be really huge especially when the size of running software or VPs complexity increases. Moreover, the method can only validate the VPs' behavior against TLM-2.0 rules and not the functional and timing behavior of their IP blocks communications.

TABLE I: Different Types of the TLM-2.0 Transaction.

TM	TT	Communication Interface Call	Return Status	Phase Transition
LT	T_0	b_transport	TC	-
AT	T_1	nb_transport_fw	TC	BRQ
	T_2	nb_transport_fw	TU→TC	BRQ→BRP→ERP
	T_3	nb_transport_fw	TU→TA	BRQ→BRP→ERP
	T_4	nb_transport_fw/nb_transport_bw	TU→TA→TA	BRQ→ERQ→BRP→ERP
	T_5	nb_transport_fw/nb_transport_bw	TU→TC	BRQ→ERQ→BRP
	T_6	nb_transport_fw/nb_transport_bw	TU→TC	BRQ→BRP
	T_7	nb_transport_fw/nb_transport_bw	TU→TU	BRQ→ERQ→BRP→ERP
	T_8	nb_transport_fw/nb_transport_bw	TA→TC	BRQ→BRP
	T_9	nb_transport_fw/nb_transport_bw	TA→TA→TC	BRQ→BRP→ERP
	T_{10}	nb_transport_fw/nb_transport_bw	TA→TU	BRQ→BRP→ERP
	T_{11}	nb_transport_fw/nb_transport_bw	TA→TA→TC→TC	BRQ→ERQ→BRP→ERP
	T_{12}	nb_transport_fw/nb_transport_bw	TA→TA→TC	BRQ→ERQ→BRP
	T_{13}	nb_transport_fw/nb_transport_bw	TA→TA→TU	BRQ→BRP→ERP

TM: Timing Model **TT:** Transaction Type **TC:** TLM_COMPLETED **TA:** TLM_ACCEPTED **TU:** TLM_UPDATED
BRQ: BEGIN_REQUEST **BRP:** BEGIN_RESPONSE **ERQ:** END_REQUEST **ERP:** END_RESPONSE

III. Background and Motivation

In this section, we first give a brief introduction to the SystemC TLM-2.0 framework. Then, with a motivating example, we explain the necessity of the SystemC VP validation in the design process and different type of faults related to the IPs' communications in a given VP.

A. Background

SystemC is a C++ based system level design language providing an event-driven simulation kernel. TLM-2.0 framework (as the current standard of SystemC TLM framework) introduces the transaction concept allowing designers to describe a model in terms of abstract communications using the base protocol and standard interfaces (e.g. *b_transport* and *nb_transport*). A transaction is a data structure (i.e. a C++ object) passed through TLM modules using function calls. A TLM module may include initiators (generating transactions), interconnects (acts as a transaction router), and targets (responds to the incoming transactions). A communication between two TLM modules in a VP can be performed based on two timing models, *Loosely-timed* (LT), and *Approximately-timed* (AT). The former is appropriate for the use case of software development while the latter for architectural exploration and performance analysis. The LT model is implemented using the blocking transport interface (*b_transport*) allowing only two timing points to be associated with each transaction. The first timing point is the request, while the second is the response. The AT model is implemented using the non-blocking transport interface (*nb_transport*) providing multiple phases and timing points for a transaction. Due to the combination of these phases and timing points, 13 unique transaction types are defined in the base protocol. In summary, Table I shows different transaction types (column *TT*) of the TLM-2.0 base protocol and describes them based on the communication interface call, return status of the interface call and the transaction's phase transitions.

B. Motivating Example

Consider the *LT-AT_BUS* VP which is inspired by [18] and implemented in SystemC TLM-2.0. The VP includes five modules i.e. two initiator modules, an LT-AT interconnect and two target modules.

The design team implements the VP base on different TLM-2.0 base protocol transactions and the following specifications. *Initiator_A* communicates with target modules through *LT-AT_BUS* by generating two types of AT transactions. The transaction type T_1 and T_2 (w.r.t Table. I) to access *Target_A* (each type for different memory address ranges), and type

T_3 to access *Target_B*. The *Initiator_B* module generates AT transactions of type T_4 to access target modules *Target_A*, and the LT transactions type (T_0) to access *Target_B*. For example, consider the communication between *Initiator_A* and *Target_A* (the gray components in Fig. 1). The *Initiator_A* module generates transactions of types T_1 (including less transition phases to gain performance) and T_2 to access memory address range ($0x00$ to $0x0A$) and ($0x0B$ to $0xFF$) of the *Target_A* module, respectively. Now consider three possible *Fault Types* (FTs) that designers may be faced during the design process.

FT1: Implementing an incorrect TLM-2.0 base protocol transaction. After implementing the VP, it might happen that some TLM-2.0 based protocol rules are implemented incorrectly by designers. For example, the generated transactions by *Initiator_A* to access address range (0x0B-0xFF) of target module *Target_A* have wrong transition phase orders. However, this type of faults cannot be detected by either SystemC compiler or TLM-2.0 library. Moreover, manually checking the correctness of these rules even for a simple design is very difficult as TLM-2.0 includes many rules.

FT2: Transporting incorrect transaction data can also cause malfunction for the VP model. Transaction's attributes such as data, address or data length can be assign to a wrong value by each of the TLM modules, resulting in transporting transaction to an incorrect destination, or receiving a wrong data by the initiator or target modules. For example, some transactions of type T_1 generated by the *Initiator_A* access the memory addresses $0xB2$ and $0xB3$ which are against the VP specifications. In this case although the VP implementation adheres the TLM-2.0 rules, it violates the VP specifications. Thus, the traditional verification approaches such as [13] and [10] that only focus on verifying the VP against the TLM-2.0 base protocol rules fail to detect this type of faults.

FT3: Implementing an incorrect timing behavior can also cause an error. This specifically is related to the timing annotation of a transaction and defined as the delay that the transaction requires to be transferred between two TLM modules. For example, the delay parameter of transactions generated by *Initiator_A* to access the address range ($0x00$ to $0x0A$) of *Target_A* must be less than 80 ns. Hence, designers want to know whether or not the timing behavior of the generated transactions by *Initiator_A* to access this range of address in *Target_A* adheres the VP's timing specifications. For this type of faults, again traditional approaches such as [13] and [10] fail to validate the VP against its specifications.

Hence, to detect all the aforementioned types of fault, an automated validation process is required that not only checks the correctness of VPs against TLM-2.0 rules but also validates their functinal and timing behavior against their specifications.

IV. Methodology

A. Overall Workflow

Fig. 2 provides an overview of the proposed approach includes four main phases as below.

1) Extracting the run-time behavior of the SystemC VP by executing its instrumented version.
2) Analyzing the extracted information to
   a) retrieve each transaction lifetime from the VP's run-time behavior log and
   b) transform transactions' lifetime into a set of access paths.

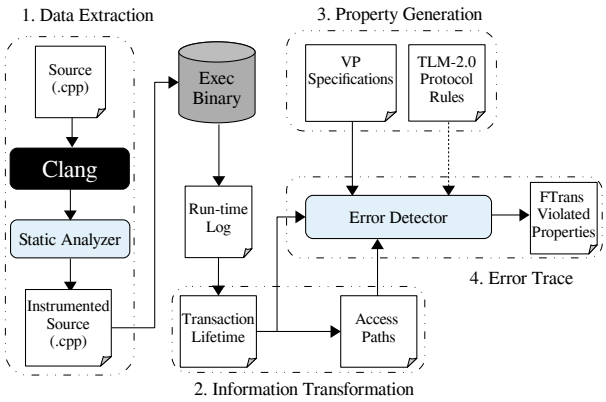

Fig. 2: The Proposed Methodology Overview.

3) Generating a set of properties from the following sources.
   a) TLM-2.0 based protocol rules and
   b) VP specifications related to its functional and timing behavior.
4) Validating the VP's behavior against the TLM-2.0 base protocol rules and the generated properties from the VP's specifications.

In the following, each phase of the proposed approach is explained in detail and illustrated using the motivating example *LT-AT_BUS* VP (Fig. 1).

B. Data Extraction

The first phase of the proposed validation approach is to access the run-time information of a given VP describing its behavior (which is defined in terms of transaction). This requires to trace all transactions of the VP generated by different initiator modules and transferred through the interconnect to access the corresponding target modules. To trace a given VP's transactions, we take advantage of the Clang compiler to generate an instrumented version of the VP source code (inspired by [19]). To do this, the *Static Analyzer* (Fig. 2-phase 1) module analyzes the AST of the VP which is generated by Clang from its source code. This analysis includes two steps. In the first step, the information related to the VP's structure is extracted by visiting the reverent node in the AST. This information consists of

- the name of TLM modules and their member functions and
- type of each TLM module which is defined by analyzing its socket(s) type. Due to the TLM-2.0 rules an initiator module only has initiator socket(s), a target module only has target socket(s) and an interconnect module has both types of socket.

In the second step, the extracted information is used to automatically generate an instrumented version of the existing source code by inserting the *Trans_Recorder* statements. The *Trans_Recorder* statement includes the instructions to extract the information which is necessary required to properly trace a transaction. This information is the transaction's reference address, the value of transaction's attributes and its related parameters such as timing annotation, phase (e.g. BEGIN_REQ) and functions' return status (e.g. TLM_COMPLETED). Moreover, the simulation time is retrieved to identify the exact time when the transactions' state changes. In order to trace

```
1  struct Target_A: sc_module {
2   tlm_utils::simple_target_socket<Target_A, 32> socket;
3   ...
4   void send_response(tlm::tlm_generic_payload& trans){
5    tlm::tlm_sync_enum status;
6    tlm::tlm_phase phase;
7    sc_time delay;
8    ...
9    status = socket->nb_transport_bw(trans, phase, delay);
10   Fout<<"Target_A::send_response::Ref_adrs="<<trans<<"Data="<<trans->
         get_data_ptr()<<"Cmd="<<trans->get_command()<<"Adrs="<<trans->
         get_address()<<"Rps="<<trans->get_response_status()<<"Dl="<<trans->
         get_data_length()<<"Delay="<<delay<<"Phase="<<phase<<"
         Instance_name_module="<<this->name()<<"ST="<<sc_time_stamp()<<endl;
11  ...}
```

Fig. 3: Part of the Instrumented Source Code of Module *Target_A* of the *LT-AT_BUS* VP.

a transaction after any possible changes, two locations need to be considered. First, the line of code where the transaction is defined (e.g. as a function arguments or a local variables within the function's body), is considered as DEF location. Second, the function call (e.g. transport interfaces *b_transport* or *nb_transport*) where the transaction object is used as an input argument, is considered as USED location. Thus, the *Trans_Recorder* statements are inserted to the source code after the aforementioned locations.

For example, consider the *Target_A* module of the *LT-AT_BUS* design (Fig. 3). Line 10 is not initially available. In order to retrieve all transactions related to this module, all functions of the module (e.g. *send_response*) in which a transaction object is referenced needs to be traced. This is performed by analyzing the VP's AST using the *Static Analyzer* module (Fig. 2, phase 1) to find DEF and USED locations in the source code. As an example of the USED location, consider Line 9, in Fig. 3 where the transaction object `trans` is used as a function argument of the `nb_transport_bw` interface. To properly trace the transaction, all information related to its flow and data needs to be extracted. The former refers to the module name (`Target_A`) and the parent function (`send_response`) to which this transaction belongs and the transaction's reference address. The later refers to all attributes of the transaction which are data, address, response status and data length, and its related parameters. The transaction's related parameters are the `phase` and `delay` arguments of the `nb_transport_bw` interface and its return status stored in the `status` variable. From the extracted information, the *Trans_Recorder* statement `Fout` (Line 10, Fig. 3) is automatically generated and inserted after the USED location in the new source code. To extract the instance name of the module, instruction `this->name()` is added to the *Trans_Recorder* statement. This requires to identify that the transaction `trans` belongs to which instance of the `Target_A` module. Moreover, instruction `sc_time_stamp()` is used to extract the simulation time when the transaction is sent.

C. Information Transformation

To validate that a transaction adheres the TLM protocol rules and the VP specifications, building the transaction lifetime is necessary. This requires to analyze the *Run-time Log* file to retrieve all information related to a transaction from the time that it is created by an initiator module until its completion. This time is considered as the transaction lifetime. Due to the TLM-2.0 rules [2] a transaction object is passed as a function argument to a communication interface (e.g. *b_transport* or *nb_transport*) with a reference address (call by reference). This address can be used as the transaction identifier to isolate the information related to a transaction within its lifetime from others in the *Run-time Log* file. However, this reference address is not a unique identifier as it may be re-used for new transactions when an old one is completed. Thus, to handle this issue, the information related to the type of the modules (i.e. initiator, interconnect or target) taking part in the transaction lifetime and the return value of the communication interfaces are used to identify the start and end points of the transactions with the same reference address. By this, the *Run-time Log* file is transformed into a structural format (*Transaction lifetime* in Fig. 2-phase 2) where each transaction is described within its lifetime. A transaction lifetime includes several sequences (timing steps) illustrating the transaction creation, manipulation by TLM modules and its completion. Each sequence in the transaction lifetime is defined based on the following definition.

Definition 1. A sequence *SQ* is a tuple (F, D) where F is all information related to the transaction's flow and D denotes the information describing the transaction's attributes and its related parameters.

$$SQ = \{(F, D) \mid F = (M, I, Func, ST, TID, MT), \\ D = (data, adrs, cmd, dl, rps, phase, delay, rs)\} \text{ where}$$

- M and I are the root and instance names of a module, respectively.
- $Func$ is the function name that a transaction object is referred.
- ST shows the simulation time.
- TID is the transaction reference address.
- MT illustrates the type of a TLM module.
- $data$, $adrs$, cmd, dl and rps are the transaction's attributes denoting the data, address, command, data length and the response statues, respectively.
- $phase$, $delay$ and rs present the transaction's phase, timing annotation and return status of the communication interfaces, respectively.

Please note that the $phase$ parameter in LT model is set to $NULL$ as it is only relevant to the AT model. Base on Definition 1, transaction lifetime is defined as the following.

Definition 2. A transaction lifetime *TL* is a set of sequences SQ where

$$TL = \{SQ_i \mid 1 \leq i \leq n_T\}$$

and n_T is defined based on which base protocol transaction is used as different types have disparate number of sequences.

Although the generated transaction lifetime TL has a proper structure to validate each transaction against the TLM-2.0 base protocol rules, to verify functional (FT2) and timing (FT3) fault types further translation on the transaction lifetime is required. Since the validation of a given VP's transactions against FT2 or FT3 requires to check whether or not the transactions are sent to the right target module (e.g. a right memory address) with the expected transaction type or delay w.r.t the VP specifications, we transform each transaction lifetime into an access path based on the following definition.

Definition 3. A complete simulation behavior of a given SystemC VP can be defined as a set of access paths SAP where each path AP shows a connection between an initiator module IM and a target module TM as below

$$SAP = \{AP_i \mid AP_i = \{IM \rightarrow TM :: (TID, TT, Tadrs, cmd, TD)\}, \\ 1 \leq i \leq n_{seq}\} \text{ where}$$

```
SQ1: ([Initiator_A, initA, process_1, 50ns ,NULL, initiator],
      [0x76ab561, 0x06, WRITE, 4, TLM_INCOMPLETE_RESPONSE, BEGIN_REQ, 5ns, NULL])

SQ2: ([LT–AT_BUS, bus_0, nb_transport_fw, 55ns, 0x683c10, interconnect],
      [0x76ab561, 0x06, WRITE, 4, TLM_INCOMPLETE_RESPONSE, BEGIN_REQ, 5ns, NULL])

SQ3: ([Target_A, trgA, nb_transport_fw, 60ns, 0x683c10, target],
      [0x76ab561, 0x03, WRITE, 4, TLM_OK_RESPONSE, BEGIN_REQ, 5ns, TLM_COMPLETED])

SQ4: ([LT–AT_BUS, bus_0, nb_transport_fw, 65ns, 0x683c10, interconnect],
      [0x76ab561, 0x03, WRITE, 4, TLM_OK_RESPONSE, BEGIN_REQ, 5ns, TLM_COMPLETED])

SQ5: ([Initiator_A, initA, process_1, 70ns ,NULL, initiator],
      [0x76ab561, 0x03, WRITE, 4, TLM_OK_RESPONSE, BEGIN_REQ, 5ns, NULL])
```

Fig. 4: A single transaction lifetime of the *LT-AT_BUS* VP.

- IM and TM are initiator and target modules (their root and instance names), respectively.
- TT is the transaction type illustrating which timing model (LT or AT) is used. In case of the AT model, it shows which type of the based protocol transaction is implemented w.r.t the Table I. To identify the transaction type, a unique type signature is generated by concatenating three parameters from the transaction lifetime i.e. *communication interface call*, *return status*, and *phase transitions*.
- $Tadrs$ shows the address of the transaction in the target module TM.
- cmd is the transaction command attribute. It shows the type of the transaction access (e.g. read or write).
- TD is the total delay of all sequences within the transaction lifetime. This is obtained by differentiating the simulation time ST of the first and last sequences.
- n_{seq} is the number of sequence in a transaction lifetime.

For example, Fig. 4 shows a part of single transaction lifetime of the *LT-AT_BUS* VP. It shows that the transaction lifetime has five sequences and implement type T_1 of the based protocol transaction as its type signature is "*nb_transport_fw+BRQ+TC*". The access path representation of this transaction lifetime based on Definition 3 is as below.

$$AP = \{Initiator\_A : initA \rightarrow Target\_A : trgA :: (0x683c10, T_1, 0x03, WRITE, 20ns)\} \quad (1)$$

It shows that the instance *initA* of initiator module *Initiator_A* created a transaction with reference address *0x683c10* to write in memory address *0x03* of the instance *trgA* of target module *Target_A*. It also indicates that the overall delay for this transaction is *20* ns as its first (SQ_1) and last (SQ_5) sequences are started at simulation time 50 ns and 70 ns, respectively.

D. Property Generation

The design rules are usually written in a textbook specifications and designers use them to implement the design. To model a SystemC VP, a part of these specifications is defined by the TLM-2.0 base protocol describing e.g. how communications between TLM modules must be implemented. This type of constraint is implemented as a part of the *Error Detector* module (Fig. 2-phase 3) and explained in Section IV-E. The other parts of these specifications related to the functional and timing behavior of the VP are defined by designers and considered as *User constraints*. The focus of this section is to show how the *User constraints* are translated into the corresponding proprieties.

1) User constraints – (Functional Properties): To validate the behavior of a given VP against the VP specifications related to the functional constraints, these rules need to be translated into a set of properties. To do this, the VP specifications are given as inputs based on the following definition.

Algorithm 1: Functional Property (FP) Generation.

Data: Design specification VP_{fs}
Result: Functional properties FP to validate connection paths in SAP

```
1  L_trg ← extracting all target modules of VP from VP_fs;
2  i ← 0;
3  FP ← ∅;
4  foreach target TM of initiator module IM in VP_fs do
5      FTL_IM ← L_trg − TM;
6      foreach target T in FTL_IM do
7          foreach transaction type TT in L_TT do
8              p_i ← (IM, T, TT);
9              add(FP, p_i);
10             i ← i + 1;
11         end
12     end
13     FTT_IM ← L_TT − TT_IM;
14     foreach target TM in TL do
15         foreach transaction type tt in FTT_IM do
16             p_i ← (IM, TM, tt);
17             add(FP, p_i);
18             i ← i + 1;
19         end
20     end
21 end
```

Definition 4. The VP functional specifications VP_{fs}, for each initiator module IM include the list of all target modules TM that IM is allowed to access with a specific transaction type TT as below.

$$VP_{fs} = \{IM_i \mid IM_i \rightarrow \{(TM_j(address\_range),\ TT_n)\}, \\ 0 \leq i \leq n_{init}, 0 \leq j \leq n_{trg}, 0 \leq n \leq 13\}$$

Where n_{init} and n_{trg} indicate the number of initiator and target modules, respectively.

The given VP specifications are analyzed by the *Error Detector* module (Fig. 2-phase 3) to automatically generate all properties related to the invalid connection paths of an initiator and a target module either based on a wrong address or transaction types. As illustrated in Algorithm 1, the VP_{fs} is analyzed to extract all target modules of the VP and store them in the L_{trg} (Line 1). For each initiator module IM of the VP a *Forbidden Target List* FTL_{IM} is generated containing the target modules that the initiator is not allowed to access (Lines 4 and 5). Then, properties are generated to indicate the following two important facts.

1) For the initiator module IM there must be no connection path to the targets in its FTL_{IM} for all types of based protocol transactions specified in L_{TT} (Lines 5 to 12).
2) For the initiator module IM there must be no connection path to its target list TL with the transaction types specified in its *Forbidden Transaction Type* list FTT_{IM} (Lines 13 to 20).

For example, consider the *LT-AT_BUS* VP (Fig. 1). The VP specifications related to the *initiator_A* is defined as below.

$$VP_{fs} = \{Initiator\_A \rightarrow (Target\_A(0x00 - 0x0A), T_1), \\ (Target\_A(0x0B - 0xFF), T_2), \\ (Target\_B(0x00 - 0xFF), T_3)\} \quad (2)$$

Base on the aforementioned VP_{fs}, (3) shows a part of the generated functional properties FP related to the communication path between $Initiator\_A$ and $Target\_A(0x00 - 0x0A)$.

$$FP = \{p_1 = (Initiator\_A, Target\_A(0x0B - 0xFF), T_2), ... \\ p_{12} = (Initiator\_A, Target\_A(0x0B - 0xFF), T_{13}), ...\} \quad (3)$$

Algorithm 2: Timing Property (TP) Generation.

Data: Design specification VP_{ts}
Result: Timing properties TP to validate connection paths in SAP

```
1 VP_ts;
2 i ← 0;
3 TP ← ∅;
4 foreach target TM of initiator module IM in VP_ts do
5 |   p_i ← (IM, T, TT, != DT);
6 |   add(TP, p_i);
7 |   i ← i + 1;
8 end
```

The generated properties p_1 and p_{12} in (3) show that $Initiator\_A$ must not access the address range $(0x00 - 0x0A)$ of $Target\_A$ by transaction types T_2 and T_{13}, respectively.

2) User constraints – (Timing Properties): In order to validate the timing behavior of a given VP's transactions generated by different initiator modules against the VP's specifications, the timing specifications of the VP are required to be defined and given as inputs. This specification is defined in the same way as the functional specification. The only difference is that in addition to the information in Definition 4, the required time of a communication between an initiator module and its corresponding target (total transaction delay) needs to be identified in the VP's specifications. Thus, the VP_{ts} is defined as the following.

$$VP_{ts} = \{IM_i \mid IM_i \rightarrow \{(TM_j(address\_range),\ TT_n, TD)\}, \\ 0 \leq i \leq n_{init}, 0 \leq j \leq n_{trg}, 0 \leq n \leq 13\} \quad (4)$$

Where TD denotes the total delay of the generated transaction type TT by the initiator module IM to access the target module TM.

The given VP's specifications are analyzed by the *Error Detector* module (Fig. 2-phase 3) to automatically generate timing properties. As shown in Algorithm 2, for each target TM of the initiator IM, the property p_i indicates that there must not be a communication with the transaction type TT that requires less or more time than TD to perform the communication (Lines 4 to 8). For example, a part of the generated timing properties TP related to the transactions generated by $Initiator\_A$ to access $Target\_A(0x00 - 0x0A)$ is as the following.

$$TP = \{p_1 = (Initiator\_A, Target\_A(0x00 - 0x0A), T_1, != 20)\} \quad (5)$$

E. Error Trace

The validation process of a given VP is performed in two main steps. First, the TLM-2.0 rules are checked indicating whether or not the VP behavior adheres the TLM.2.0 based protocols. In this case any violation is reported to designers to be overcome before the user constraints (defined based on the VP specifications) are verified. In the second step, user constraints are verified including both the functional and timing properties.

1) TLM-2.0 Rules Validation: This type of constraint is directly generated from the TLM-2.0 base protocol including all rules related to the **transaction types** (e.g. the generated transaction of a given VP describes one of the valid based protocol transactions), **transaction attributes** (e.g. the data length attribute of a transaction must be a positive integer number) and the expected **TLM modules behavior** (e.g. an interconnect module must not modify the data attribute of a transaction).

Constraints related to the transaction types are generated by translating the base protocol transactions into the corresponding type signature as illustrated in Table. I. This covers 25 rules of the TLM-2.0 based protocols. In order to check the correctness of each transaction lifetime against the transaction types fault, the followings two steps are performed. First, the transaction type signature is generated by analyzing each transaction lifetime in TL. Then, an string matching algorithm is performed to identify unmatched transaction type signature that is not match the reference model. The lifetime of the violated transactions is analyzed to indicate the first faulty sequence. This sequence is reported to designers.

Concerning the transaction attributes rules, for each sequence SQ in the transaction lifetime TL based on Definition 2, the *Error Detector* module (Fig. 2-phase 3) checks its data D to find an attributes' rule violation. Overall, 10 TLM-2.0 rules in this regard are implemented.

To validate TLM-2.0 rules related to the TLM modules behavior, the *Error Detector* module (Fig. 2-phase 3) first, identifies the type of modules in a transaction lifetime TL. This is performed by analyzing the transaction's flow F in each sequence SQ w.r.t Definition 1. Then, the behavior validation process is performed based on their types. For example, to validate that an interconnect module does not change the data attribute of a transaction object in its lifetime, first the sequences that the interconnect module takes part are identified. Then, the data attribute of the transaction in this sequence is compared to the previous and next sequences to indicate whether or not the data is changed. In case of violation, this sequence is reported to designers. Overall, 15 TLM-2.0 rules are implemented in the *error detector* module regarding this type of faults.

2) User constraints Validation: Verifying a VP against both functional and timing properties is performed by analyzing the access paths in SAP. For each property in FP or TP the SAP is traversed in order to find property violations related to the functional or timing behavior of the VP, respectively. For both cases, the violated paths are reported to designers.

V. Experimental Results

The *Static Analyzer* module is implemented using the LibTooling library of Clang compiler [16]. The *Error Detector* module is implemented in C++ based on the explanation in section IV-E and Algorithm 1 and 2.

The proposed approach is applied to several standard VPs provided by Doulos [18] and [20]. The experimental results (Table II) are described in two parts. First, a real-word case study – the LEON3-based VP SoCRocket (implemented in SystemC TLM-2.0) [20] is illustrated in detail in Section V-A. Second, we give a brief discussion on the quality of obtained experimental results in Section V-B.

All the experiments have been carried out on a PC equipped with 8 GB RAM and an Intel core i7 CPU running at 2.4 GHz.

A. Case Studies

To evaluate the quality of the proposed approach, we have injected faults into the VPs based on FT1, FT2 and FT3 introduced in Section III. The proposed approach was applied to validate the correctness of each VP against the TLM-2.0 rules and the VP's specifications. These faults are injected

TABLE II: Experimental Results for all Case Studies.

	SystemC VP	LoC	#Trans	#TT	TM	#Properties - Ours				#Properties - [13]				ET (s) - Ours					ET (s) - [13]	CET (s)		
						Total	Pass	Fail	FTrans	Total	Pass	Fail	FTrans	P1	P2	P3	P4	Total		Compile	Exe	Total
Original	LT-example[1]	175	1,000	1	LT	37	37	0	0	5,886	5,886	0	0	1.4	0.9	0.6	2.2	5.1	566.7	1.3	0.1	1.4
	Routing-model[1]	456	1,000	1	LT	84	84	0	0	11,960	11,960	0	0	1.9	1.0	0.6	2.3	5.8	10,579.2	1.7	0.1	1.8
	AT-example[1]	2,942	2,000	8	AT	245	245	0	0	TO	TO	TO	TO	27.5	1.7	1	3.3	33.5	TO	21.0	0.2	21.2
	Locking-two[1]	3,831	4,000	10	LT/AT	233	233	0	0	TO	TO	TO	TO	29.2	2.9	1.1	6.1	39.3	TO	23.9	0.3	24.2
	SoCRocket[2]	50,000	12,000	8	LT/AT	773	773	0	0	TO	TO	TO	TO	53.8	5.8	1.6	18.6	79.8	TO	27.6	2.3	29.9
FT1	LT-example	175	1,000	1	LT	22	21	1	275	5,886	4,415	1,471	275	1.4	0.9	0	1.6	3.9	515.4	1.3	0.1	1.4
	Routing-model	456	1,000	1	LT	22	20	2	388	11,960	7,416	4,544	388	1.9	1.0	0	1.6	4.5	9,421.3	1.7	0.1	1.8
	AT-example*	1,950	2,000	8	AT	35	32	3	719	TO	TO	TO	TO	27.5	1.7	0	2.1	31.3	TO	19.8	0.2	20.0
	Locking-two*	2,907	4,000	10	LT/AT	55	52	3	1,091	TO	TO	TO	TO	29.2	2.9	0	3.9	36.0	TO	21.2	0.3	21.5
	SoCRocket	50,000	12,000	8	LT/AT	55	52	3	2,168	TO	TO	TO	TO	53.8	5.8	0	10.7	70.3	TO	27.6	2.3	29.9
FT2	LT-example	175	1,000	1	LT	13	12	1	317	5,886	5,886	0	0	1.4	0.9	0.4	0.4	3.1	566.7	1.3	0.1	1.4
	Routing-model	456	1,000	1	LT	54	44	10	207	11,960	11,960	0	0	1.9	1.0	0.4	0.4	4.7	10,579.2	1.7	0.1	1.8
	AT-example	2,942	2,000	8	AT	195	180	15	492	TO	TO	TO	TO	27.5	1.7	0.6	0.7	30.5	TO	21.0	0.2	21.2
	Locking-two	3,831	4,000	10	LT/AT	156	135	21	723	TO	TO	TO	TO	29.2	2.9	0.6	1.3	34.0	TO	23.9	0.3	24.2
	SoCRocket	50,000	12,000	8	LT/AT	676	642	34	3,391	TO	TO	TO	TO	53.8	5.8	0.9	5.2	65.7	TO	27.6	2.3	29.9
FT3	LT-example	175	1,000	1	LT	2	1	1	480	5,886	5,886	0	0	1.4	0.9	0.2	0.2	2.7	566.7	1.3	0.1	1.4
	Routing-model	456	1,000	1	LT	8	6	2	95	11,960	11,960	0	0	1.9	1.0	0.2	0.3	3.4	10,579.2	1.7	0.1	1.8
	AT-example	2,942	2,000	8	AT	15	11	4	471	TO	TO	TO	TO	27.5	1.7	0.4	0.5	30.1	TO	21.0	0.2	21.2
	Locking-two	3,831	4,000	10	LT/AT	22	17	5	811	TO	TO	TO	TO	29.2	2.9	0.5	0.9	33.5	TO	23.9	0.3	24.2
	SoCRocket	50,000	12,000	8	LT/AT	42	31	11	2,392	TO	TO	TO	TO	53.8	5.8	0.7	2.7	63.0	TO	27.6	2.3	29.9

1 and **2** provided by [18] and [20], respectively **LoC:** Lines of Code **#Trans:** Number of Transactions **#TT:** Number of Transactions' Type **TM:** Timing Model **FTrans:** Number of Faulty Transactions **CET:** Compilation and Execution Time **ET:** Extraction Time **TO:** Time Out (it has been set to three hours) *The *AT-example* and *Locking-two* VPs includes custom base protocol checkers in their original source codes and were removed to create faulty model FT1.

to the VPs based on the aforementioned fault types as the followings.

- FT1: an incorrect initialization of the transaction's response status (fault related to the transaction attributes rules), modification of the transaction data length by an interconnect module (fault related to the TLM modules behavior) and a wrong sequences order of transactions' phase transitions (fault related to the transaction type).
- FT2: initiating transactions with an incorrect address computation or an incorrect initialization of the VP memory configuration file.
- FT3: altering the timing annotation of transactions with an incorrect computation.

For a real-word experiment, we applied the proposed approach to validate the LEON3-based VP SoCRocket [20]. The VP is implemented in SystemC TLM-2.0 including more than 50,000 lines of code. It consists of several IPs working together in master (e.g. initiator modules *LEON3* processor, *ahbin1* and *ahbin2*) or slave (e.g. target modules *AHBMem1* and *AHBMem2*) mode connecting to the on-chip bus AMBA-2.0 AHB (Advanced High-performance Bus). The communication uses a 32-bit address mode where the 12 most significant bits are used to specify the memory address. For brevity, we refrain from giving a detailed introduction to the whole VP. To show how different fault types are injected to the VP and the validation process was performed, consider a part of the VP including the initiator modules *ahbin1* and *ahbin2* and the target modules *AHBMem1* and *AHBMem2* connected to the AMBA-2.0 AHB.

Regarding FT1, we injected the transaction attributes fault into the *ahbin2* to generate transactions with an incorrect default value of the response status attribute (i.e. TLM_OK_RESPONSE instead of TLM_INCOMPLETE_RESPONSE). We injected the TLM modules behavior fault into the AMBA-2.0 AHB to change the data length of receiving transactions (i.e. 2 instead of 4 Bytes). Moreover, we injected the transaction type fault into the *ahbin1* to generate transactions with initial phase BEGIN_RESP instead of BEGIN_REQ for transaction type T_1. The first fault was detected by checking the first sequence of each transaction lifetime where the initial response statues must be TLM_INCOMPLETE_RESPONSE. The *Error Detector* module was able to find the second fault by comparing the data length attribute of each transaction of the interconnect sequences to the previous and next sequences in the transaction lifetime. The third fault was detected by comparing the type signature of each transaction lifetime to the pre-defined reference type signature (based on Table I). The faulty type signature "*nb_transport_fw+TC+BRP*" was not matched any of the pre-defined signatures. Overall, 1000 transactions are generated by both initiator modules which 273 of them were against the TLM 2.0 rules.

Concerning FT2, we consider the expected functional specifications of the VP as the following.

$$\begin{aligned} VP_{fs} = \{ & ahbin1 \rightarrow (AHBMem1(0xA0000000 - 0xA0000CC4), T_1), \\ & (AHBMem1(0xA0000CC5 - 0xA0000FFF), T_2), \\ & (AHBMem2(0xB0000000 - 0xB0000FFF), T_8), \\ & ahbin2 \rightarrow AHBMem2(0xB0000000 - 0xB0000FFF), T_0)\} \end{aligned} \quad (6)$$

The functional faults injected into the lines of code of the *ahbin1* where transactions address are generated. These lines of code were altered to generate random values for the 12 less significant bits of the transactions' address (i.e. 000 to FFF) for both transaction types T_1 and T_2. Due to the VP_{fs}, 52 functional properties were generated by the *Error Detector* which a part of them are as the following.

$$\begin{aligned} FP = \{ & p_1 \rightarrow (ahbin1, (AHBMem1(0xA0000000 - 0xA0000CC4), T_2)) \\ & \ldots, p_{52} \rightarrow (ahbin2, (AHBMem2(0xB0000000 - 0xB0000FFF), T_{13}))\} \end{aligned} \quad (7)$$

Overall, 12 properties were violated indicating the initiator modules *ahbin1* and *ahbin2* had 12 invalid access to the target modules *AHBMem1* and *AHBMem2* w.r.t the functional specifications of the VP.

Regarding FT3, we consider the expected timing specifications of the VP related to the *ahbin1* and *ahbin2* as below.

$$\begin{aligned} VP_{ts} = \{ & ahbin1 \rightarrow (AHBMem1(0xA0000000 - 0xA0000CC4), T_1, 50), \\ & (AHBMem1(0xA0000CC5 - 0xA0000FFF), T_2, 100), \\ & (AHBMem2(0xB0000000 - 0xB0000FFF), T_8, 200), \\ & ahbin2 \rightarrow (AHBMem2(0xB0000000 - 0xB0000FFF), T_0, 50)\} \end{aligned} \quad (8)$$

We changed the lines of code where the timing annotation of the transactions type T_1 and T_2 generated by *ahbin1*

are defined. Due to the VP_{ts}, four timing properties were generated by the *Error Detector* as the following.

$$\begin{aligned} TP = \{ & p_1 \rightarrow (ahbin1, (AHBMem1(0xA0000CC5 - 0xA0000FFF), T_1, != 50), \\ & p_2 \rightarrow (ahbin1, (AHBMem1(0xA0000CC5 - 0xA0000FFF), T_2, != 100), \\ & p_3 \rightarrow (ahbin1, (AHBMem2(0xB0000000 - 0xB0000FFF), T_8, != 200), \\ & p_4 \rightarrow (ahbin2, (AHBMem2(0xB0000000 - 0xB0000FFF), T_0, != 50)\} \end{aligned} \quad (9)$$

Our validation approach could detect two properties violation which are p_1 and p_2. From 1000 generated transactions by the initiator modules *ahbin1* and *ahbin1*, 37 transactions had incorrect timing behavior w.r.t the VP_{ts}.

The experimental results for different types of ESL benchmarks are shown in Table II. The first column shows four variants of SystemC VPs denoted as *Original*, *FT1*, *FT2* and *FT3* referring to the reference model of the VP and three faulty models, respectively. Columns *SystemC VP*, *Loc* and *#Trans* list name, lines of code and the number of extracted transactions for each VP, respectively. The *#TT* column illustrates the number of transaction types implemented in each VP. Column *TM* presents the timing model of each design. Column *#Properties* shows the number of generated properties to validate each VP against the TLM-2.0 rules or its specifications. For this column, *Total*, *Pass*, *Fail* and *FTrans* illustrate the number of generated, satisfied, violated properties and faulty transactions, respectively. For the original model of each VP, column *properties* shows the total number of all generated properties (including TLM-2.0 rules) that the VP was validated against them. The value zero for columns *Fail* and *FTrans* indicate that all transactions adhered the TLM-2.0 rules and the VP specifications. Please note that for *FT1* variant of each VP, column *#Properties-Ours* illustrates the number of TLM-2.0 rules that are covered by the proposed approach w.r.t the VP's timing model.

The execution time of the proposed approach is reported in column *ET* followed by the data extraction *P1*, information transformation *P2*, property generation *P3* and total execution time *Total*. Column *CET* shows the time of each VP's compilation and execution without any instrumentation.

B. Discussion

We evaluated the quality of the proposed approach to analyze a given SystemC VP by comparing it to [13]. The comparison is performed based on the ability of each approach to validate a given SystemC VP against different types of fault and the required time for this analysis.

The experimental results in Table II illustrate that the proposed approach not only can validate a given SystemC VP against the TLM-2.0 rules but also it is able to validate the VP's functional and timing behavior against its specifications. The proposed approach reports the same number of faulty transactions in case of FT1 to designers as [13]. Concerning FT2 and FT3, [13] does not support the validation of VPs against its specifications as the number of failed properties (column *Fail*) and faulty transactions (column *FTrans*) for the *FT2* and *FT3* variants of VPs are zero. The main reason is that, this method is not able to generate functional and timing properties. Thus, all its generated properties (which related to TLM-2.0 rules) are passed in case of fault types FT2 and FT3. In contrast, our proposed approach detects both the aforementioned types of fault. Moreover, Table II shows that the execution time of the proposed approach (column *ET*) for all case studies lies in an order of seconds, allowing it to be used in common development environments. In comparison to [13], this time for our validation approach is significantly lower (seconds versus hours). For complex designs with large number of transactions, the parameter *TO* (that is set to four hours) indicates that [13] is not applicable.

In summery, the proposed approach provides designers with a comprehensive validation technique that has negligible performance loss in comparison to its pure compilation and execution time in Table II, column *CET*. Since neither the SystemC library nor the SystemC simulation kernel are modified by the proposed approach, any results obtained using the approach are identical to the original results (both functionality and timing behavior).

As the proposed approach is based on run-time analysis, it inherits the same limitations. The validation process depends on the ability of the input stimulus (i.e. testbench or running software) to activate the possible types of fault in the first phase. However, this can be solved by manual or automated test generation techniques (which is out of scope of this paper).

VI. Conclusion

In this paper, we presented a fast and easy-to-use solution to validate a given SystemC-based VP against three main types of fault. These fault types are related to the TLM-2.0 rules and the VP's specifications (functionality and timing behavior of IPs' communications). The approach is based on analyzing the AST of the VP to extract static information and generate instrumented version of the VP's source code for run-time data extraction. The extracted information is translated into set of transactions' lifetime and access paths to be validated against the TLM-2.0 rules and the VP's specifications, respectively. We demonstrated the effectiveness and scalability of our approach on several standard VPs including a real-world system.

References

[1] "IEEE Standard SystemC Language Reference Manual," 2006, pp. 1–423.

[2] J. Aynsley, Ed., *OSCI TLM-2.0 Language Reference Manual*. Open SystemC Initiative (OSCI)., 2009.

[3] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *VLSI*, vol. 14, no. 1, pp. 57–68, 2006.

[4] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *MEMOCODE*, 2010, pp. 113–122.

[5] C. N. Chou, Y. S. Ho, C. Hsieh, and C. Y. Huang, "Symbolic model checking on SystemC designs," in *DAC*, 2012, pp. 327–333.

[6] P. Herber and S. Glesner, "A HW/SW co-verification framework for SystemC," *TECS*, vol. 12, pp. 61:1–61:23, 2013.

[7] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying systemc using intermediate verification language and stateful symbolic simulation," *TCAD*, accepted 2018.

[8] M. Kallel, Y. Lahbib, R. Tourki, and A. Baganne, "Verification of SystemC transaction level models using an aspect-oriented and generic approach," in *DTIS*, 2010, pp. 1–6.

[9] D. Tabakov and M. Y. Vardi, "Automatic aspectization of SystemC," in *MISS*, 2012, pp. 9–14.

[10] H. Sohofi and Z. Navabi, "Assertion-based verification for system-level designs," in *ISQED*, 2014, pp. 582–588.

[11] L. Ferro and L. Pierre, "Isis: Runtime verification of TLM platforms," in *FDL*, 2009, pp. 1–6.

[12] L. Pierre and M. Chabot, "Assertion-based verification for SoC models and identification of key events," in *DSD*, 2017, pp. 54–61.

[13] M. Goli, J. Stoppe, and R. Drechsler, "Automatic protocol compliance checking of SystemC TLM-2.0 simulation behavior using timed automata," in *ICCD*, 2017, pp. 377–384.

[14] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull, "Interactive presentation: Implementation of a transaction level assertion framework in SystemC," in *DATE*, 2007, pp. 894–899.

[15] T. C. Team, "Clang: a C language family frontend for LLVM," https://clang.llvm.org/, accessed: 2017-10-01.

[16] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *BSD*, 2008, pp. 1–2.

[17] M. Goli, J. Stoppe, and R. Drechsler, "Automated non-intrusive analysis of electronic system level designs," *TCAD*, accepted 2018.

[18] J. Aynsley, "TLM-2.0 base protocol checker," https://www.doulos.com/knowhow/systemc/tlm2, accessed: 2018-01-30.

[19] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Automated analysis of virtual prototypes at electronic system level," in *GLSVLSI*, 2019, pp. 307–310.

[20] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "Socrocket - A virtual platform for the european space agency's soc development," in *ReCoSoC*, 2014, pp. 1–7, http://github.com/socrocket.