Automated Nonintrusive Analysis of Electronic System Level Designs

Mehran Goli, *Student Member, IEEE*, Jannis Stoppe, and Rolf Drechsler, *Fellow, IEEE*

***Abstract*—Due to the ever increasing complexity of hardware systems, designers strive for higher levels of abstractions in the early stages of the design process. Modeling hardware at the electronic system level (ESL) is one way to address this demand, with the C++-based system modeling framework SystemC and its abstract communication library transaction level modeling (TLM) having become de-facto standards for ESL system design. While the C++ compiler is sufficient to compile and simulate a given ESL design, for tasks of design understanding, debugging, or validation (where access to the details of design's structure and behavior is necessarily required), design needs to be processed by an appropriate tool. This problem is often solved by adding instrumentation code to either the design or the library, usually resulting in incomplete logs, work overhead and/or incompatibilities. This paper introduces an approach that automatically extracts information about both, structure and behavior of SystemC designs and TLM transactions, nonintrusively. The information is retrieved from a given design by running it in debug mode while being connected to a preprogrammed debugger, thus leaving the existing sources and workflows untouched while collecting a vast amount of data without user intervention. Illustrating use cases, value change dump files of the SystemC models' behavior and unified modeling language activity diagrams of transaction protocols are created automatically from simulation runs.**

***Index Terms*—Data extraction, GNU debugger (GDB), nonintrusive, SystemC, transaction level modeling (TLM), transaction.**

I. Introduction

THE EVER-INCREASING complexity of circuits and tight time-to-market constraints make designers work on raising levels of abstraction. System design at the electronic system level [25] (ESL) is one way to work on more abstract layers, allowing designers to implement executable mixed hardware-software simulations in high-level programming languages. The system-level library SystemC [5] has become the de-facto standard [36] to specify HW–SW designs at the ESL. SystemC includes a framework for transaction level modeling (TLM), providing the designer with standardized interfaces to model communication channels that at the same time provide fast simulation times. TLM focuses on functionality of data transfers among computational components and differentiates between the details of communication and the computation [7]. To model the communication, TLM provides designers with a set of TLM-2.0 components which are the base protocol (which provides a common infrastructure for all use cases), the generic payload (which represents a standardized way to model data that should be transferred) and the communication interfaces (which provide readily available methods to implement the required protocols).

Analyzing a given SystemC design in order to know the respective components of the design (e.g., structure) as well as their relation to each other (e.g., behavior) is a crucial as C++ (and thus also SystemC, which is a library for the former) is inherently hard to analyze due to the following.

1) The lack of proper analysis methods for C++ runtime behavior (e.g., the lack of a modern reflection framework).
2) The countless compiler-specific dialects that a source code may have.
3) The executable binary format which not only is stripped of any information that is not needed to execute the simulation but may also be heavily optimized.

These make the information retrieval from SystemC models a nontrivial task.

In addition to the aforementioned obstacles to analyze an ESL model, in case of SystemC TLM-2.0 designs this analysis can be even more difficult. The complete TLM communications represent a complex process that passes several phases and may be spread out far (concerning both, time and code). A complete system model may contain a vast amount of the TLM-2.0 base protocols—so understanding them becomes a complex (but still crucial) task whenever, e.g., new developers enter a team or are assigned new issues to work on.

While a well-written and up-to-date documentation should be a primary source for these cases, there may be situations where these documents have become outdated or are unavailable. For these or other situations (e.g., to validate that a given documentation still corresponds to the current state of the implementation), a way to generate a proper model from a given implementation would be desirable. Moreover, a part from the fact that knowing the properties of a given design (both structure and behavior) is necessary to understand it, this information can facilitate other steps of the design process such as debugging, validation, verification, and synthesis of the model [15], [16].

Manuscript received April 20, 2018; revised August 9, 2018 and October 19, 2018; accepted November 30, 2018. Date of publication December 25, 2018; date of current version January 18, 2020. This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the Project SecRec under Grant 16K1S0606K, in part by the German Research Foundation within the Subproject P01 "Predictive function" of the Collaborative Research Center under Grant SFB1232, and in part by the University of Bremen's Graduate School SyDe through the German Excellence Initiative. This paper was recommended by Associate Editor S. Kim. *(Corresponding author: Mehran Goli.)*

M. Goli and R. Drechsler are with the Institute of Computer Science, German Research Center for Artificial Intelligence (DFKI GmbH), 28359 Bremen, Germany, and also with the Department of Mathematics and Computer Science, University of Bremen, 28359 Bremen, Germany (e-mail: mehran.goli@dfki.de).

J. Stoppe is with the Institute of Computer Science, German Research Center for Artificial Intelligence (DFKI GmbH), 28359 Bremen, Germany, and also with the Maritime Security Technologies, DLR Institute for the Protection of Maritime Infrastructures, 27572 Bremerhaven, Germany.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2889665

To this end, this paper introduces a new method that focuses on retrieving run-time information of ESL models and presenting them in a structured format. The extracted information describes both structure (architecture) of designs (i.e., the modules, attributes, member functions, binding information of the modules' signals, and parameters of functions) and their simulation behavior (i.e., the sequence of the modules' activations, value changes and function calls during the execution). To illustrate how this behavioral information can be processed to facilitate design understanding, it is automatically transformed into value change dump (VCD) file and a set of unified modeling language (UML) [32] models for SystemC and TLM-2.0 designs, respectively. The UML models specify the behavior of transactions, allowing designers to use a familiar, abstract design language to understand the behavior that is actually occurring during simulation. In order to show how the extracted information assists the design process, this information is used to validate that a given ESL design adheres to its formal specification.

In summary, the main contributions of this paper are as follows.

1) The automatic and nonintrusive extraction of detailed information of a given ESL design which reflects its structure and behavior.
2) The presentation of the architectural information of ESL models in structured formats such as XML that can be used during the design process with minimal effort to be set up by designers.
3) The presentation of the extracted run-time information of SystemC models in form of VCDs.
4) The automatic creation of proper logs of the transactions of TLM-2.0 models—and their presentation as a UML diagram to enable the designer to easily trace the transactions' behavior.
5) The determination of the transactions' types that are implemented in a given TLM-2.0 design.
6) The validation of ESL designs.
7) The application of the proposed method to several ESL benchmarks in various domains to evaluate its effectiveness.

The rest of this paper is organized as follows. Section II outlines existing approaches in this area and compares them to the proposed method. Section III presents the methodology in detail. Section IV includes the use of the proposed method to facilitate the design process, specifically validation of ESL design. The evaluation and experimental results are presented in Section V. Finally, this paper is concluded in Section VI.

II. Related Work

Analyzing SystemC designs (e.g., for design understanding) is an active field of research. Several methods have been developed to satisfy this goal, each of them with its own features and issues. A common denominator for all these methods is whether they are based on static or hybrid techniques. Static approaches rely on extracting information from the original source code or the compiled binary model of them using parsers [12], [17], [30], [37] or existing C++ front-ends [8], [9], [35]. They do not (by definition) analyze the execution of the models. Their results can only describe some information related to the structure of a model and in the best case can be represented in an abstract syntax tree (AST). Besides the static data extraction, hybrid approaches retrieve additional information of a given SystemC design during its execution. While this is required to retrieve a model's behavior, it is often also the only reliable method to retrieve the structure of a design as SystemC creates all its modules during the elaboration phase (during which the design's modules are created) at run-time.

As the extraction of dynamic information is necessary to describe both structure and behavior of a given ESL design, the hybrid methods have received the most attention lately. In this section, we thus give an overview of hybrid methods based on whether they support TLM or only analyze plain SystemC, illustrating their features and issues.

A. Methods That Do Not Support TLM

The analysis methods of SystemC designs that do not support TLM constructs are introduced in this section. These approaches extract either the model structure or dynamic behavior of the model, or both.

In [13], the AST of a SystemC model is retrieved by parsing the model using a PCCTS-based parser. To extract dynamic information an instrumented version of the original source code is generated by adding some *recorder function*. The state of all variables of the model is recorded by executing its elaboration phase. Using the PCCTS-based parser limits the available SystemC constructs as it does not fully support the entire instruction set of C++. The recorder function makes the method an intrusive solution, modifying a design's original sources.

PinaVM [24] extracts the structure of a SystemC model from the translated version of the source code into LLVM bitcode by executing its elaboration phase. To extract the dynamic information, it specifies the parts of the source code which contain the parameters of interest (e.g., the address of ports or events in SystemC constructs). Afterwards, new functions are constructed to be added to the model during its compilation. Those parameters are retrieved using the generated functions during the model's execution. PinaVM takes advantage of the LLVM project to analyze SystemC models which limits it to setups that are built using LLVM.

SHaBE [10] retrieves the static data by utilizing the GNU debugger (GDB) [38] and extracts the dynamic information using a GCC plugin. In the next step, the dynamic information is linked with the hierarchical information and stored as an intermediate representation. The method has limitations to extract some static and dynamic information of SystemC constructs (e.g., SystemC primitive channels or processes sensitive to certain events).

The method presented in [39] uses aspect-oriented programming (AOP) to extract the behavioral data. AOP is a paradigm that allows the designer to write refactoring rules that are applied before compiling a program (a process called weaving). This approach comes with several pitfalls. Debugging AOP setups is a complex task, just like setting up a working AOP environment. Furthermore, the current implementation of AspectC++ which can be used in tandem with SystemC does not support, e.g., join points for field access (i.e., field variable assignments cannot be tracked), privileged aspects, templates, or macros, which limits the goal of arbitrary behavior tracing.

B. Methods That Support TLM

The methods that can analyze TLM models are introduced in this section. These approaches extract either the model structure or transaction behavior, or both.

Pinapa [27] retrieves the information of SystemC models in two steps. First, the AST of the models is extracted using a C++ front-end. Second, the elaboration phase of the models is executed to extract their dynamic information. The extracted information is linked to the AST to create the final result. However, the method describes the structure of a given SystemC model, it does not provide any information to reflect the design's behavior such as the order of function calls and process activation.

The SystemC verification (SCV) library by the open systemC initiative (OSCI) provides designers with a set of APIs to record transactions into a database. The APIs are divided into three different transaction collection classes that can be instantiated during the execution of a SystemC TLM-2.0 model. The results obtained by this method can be analyzed by some commercial tools such as Cadence Incisive [2] or Novas Verdi [4] with modifications to the result-file format. SCV introduces some overhead in execution time. The method is an intrusive solution to extract the behavior of a SystemC TLM-2.0 model as the original source code needs to be manipulated. For a complex design, this manual process is a nontrivial task.

DUST [23] is a SystemC TLM-2.0 analysis framework that extracts both structural and behavioral information of a TLM-2.0 model. It retrieves the model's hierarchy at the end of the execution of its elaboration phase and presents it in an XML format. It describes the behavior of the model by recording transactions at run-time. DUST works by enhancing some SystemC objects (e.g., *sc_port* to *sc_dust_port*) and using SCV constructs which are added to the original source code. Due to this intrusive solution, its application may thus be limited if the SystemC library is updated but the given framework is not. Additionally, existing sources need to be updated to use DUST's types instead of the standard SystemC types—a solution that may require considerable work.

The method presented in [40] takes advantage of debug symbols to extract static information and SystemC API calls to retrieve dynamic data during the execution of a SystemC model. The dynamic information extracted by this method only reflects the structure of the model. The simulation behavior is not captured at all though, leaving designers with static model descriptions. Moreover, the method only supports the debug symbols that are generated by Microsoft VC++ and not GCC or Clang-LLVM.

Recently, an automated approach was introduced to automatically retrieve the structure and behavior of a given ESL model [34]. It uses the ROSE compiler to generate an AST model of the design which is then used to generate an intermediate representation (IR). The IR is analyzed to extract both architectural information of the model and behavior as graphical multithread communication charts. However, solely relying on a static analysis, the method shares the limitations of previous static approaches. It cannot consider parameters which are set at run-time (and may affect the design's behavior). The behavioral information is restricted to only describe high level interaction of modules—thus, behavior such as value changes of a module's ports and function's variables during execution are not traceable. Moreover, the method does not support designs including pointers or array indices in port mappings.

In summary, existing solutions have two major limitations in terms of precise behavior extraction of a given ESL model and custom code annotations or language constructs. The first limitation is that most of them extract a limited set of information that only describes the structure of the model and does not reflect its run-time behavior. The second limitation is that most of them can only be applied to a restricted range of SystemC designs and do not support TLM constructs. Those approaches that analyze the SystemC TLM-2.0 models mostly rely on manipulating the original source code (which is either expensive for manual processes or may have compatibility problems for automated approaches) or the SystemC library and interfaces (which may be an issue for the application of several approaches in parallel, future updates, or restrictive environments).

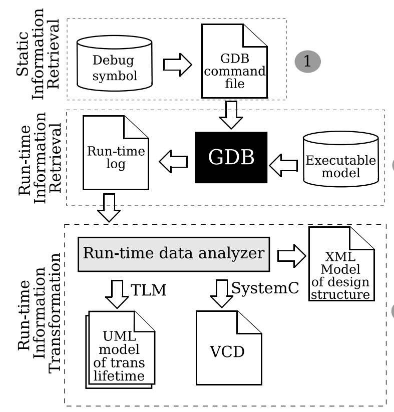

Static Information Retrieval
Debug symbol
GDB command file
1
Run-time Information Retrieval
Run-time log
GDB
Executable model
2
Run-time Information Transformation
Run-time data analyzer
XML Model of design structure
TLM
SystemC
UML model of trans lifetime
VCD
3

Fig. 1. Architecture of the proposed method.

III. Methodology

As illustrated in Fig. 1, the core idea of the proposed method to specify a given ESL design in terms of its structure and simulation behavior consists of three steps.

1) The static information of the compiled model is retrieved by analyzing its debug symbols to satisfy two goals.
   a) Identifying all components and their attributes and member functions which are required to describe structure of the model (e.g., the name and type of the modules' variables).
   b) Automatically generating a set of GDB instructions tailored to be used in the next step to extract the model's structure (dynamic data) and trace its behavior.
2) The model is executed via GDB using the previously generated instructions. The model's structure is retrieved when the execution reaches the objects for which the corresponding instructions to extract their information were generated. The execution of the model is paused at certain events (such as function calls) to record the run-time information.
3) The extracted information is automatically transformed into structured formats that can be used during the design process with minimal effort to setup by designers. The architecture of ESL models is presented in XML format, while the behavior of them as a VCD file.

Although this idea (executing a given ESL program in debug mode to access its run-time behavior using a preprogrammed debugger) properly works for analyzing SystemC designs [14], to support TLM constructs it requires some enhancements.

As stated in [11] and [23] to properly understand a given TLM model, its structure and its transactions' behavior need to be recorded and analyzed. The former refers to the object instances that are created and describe the structure of a TLM design (such as modules, functions and signals). The latter refers to the run-time information which specifies the flow of modules' communication with respect to transaction's data including transaction creation and manipulation.

The main difficulty to be overcome here is to nonintrusively trace each TLM transaction's payload at run-time. To deal with this issue, we take advantage of the TLM-2.0

rule stated in [7]—a transaction object is passed as a function argument to a method implementing one of the given communication-interfaces (*b-transport* or *nb-transport*) with a reference address (call by reference). The reference address of a transaction object remains constant from its creation until its destruction (i.e., during its lifetime). For transactions that are generated by different TLM initiator modules, the reference address of transactions can thus be used to isolate information related to each of them. This reference address is used as a *transaction ID* which is the main key to trace a transaction (related information to build its lifetime) among other transactions within the simulation log of a TLM design.

Therefore, in order to make the debugger properly trace and log the TLM payloads and function calls, the instructions must be generated specifically with regard to this case in step one, thus altering the execution in step two to extract the desired information about TLM transactions.

The enhancements that need to be performed in each step to extract and present the required TLM information are as follows.

1) Extract the structure from the TLM model to recognize the transaction object and its related parameters (e.g., phase, delay) for each TLM module.
2) Record the transaction's flow (i.e., the information of caller and callee object that communicate) and the transaction's data (i.e., the transaction's attribute).
3) Reflect the behavioral information in a big scale view to empower the designer to easily trace transactions' behavior. The extracted information is translated to a set of UML models once the execution finishes. Using an UML model to reflect the behavioral information enables designers to easily trace both transaction flow and data at the same time.

A. Static Information Retrieval

The model's static information is retrieved by analyzing its debug symbols (which is generated by the GDB) to extract all SystemC (e.g., name of modules and their member functions and attributes) and TLM constructs (e.g., the generic payload data type, initiator and target sockets, and the TLM utilities). This information is required to reflect the model's structure and trace its transactions (in case of SystemC TLM-2.0 designs) at run-time. For instance, consider the *AT-example* (which is explained in detail in Section V-A). A part of the static information related to the design's structure is presented in its generated debug symbols (Fig. 2, lines 8–15). It shows that the design contains a module `AT-typeA-initiator` with (among others) an initiator socket `socket` and a member function `nb_transport_bw` with return type `tlm_sync_enum`.

The retrieved information is translated into a more manageable data format in which each module is described in a hierarchical structure based on its member functions and attributes. Unlike other static approaches that consider this type of information as the result, this information is used as the foundation to extract additional run-time information. A GDB command file (GCF) which is used to program GDB is automatically generated based on this static data. It controls the execution of the ESL executable model running on GDB to extract the desired information in the simulation run.

For example, to extract all transactions related to the *AT-example*'s initiator module `AT-typeA-initiator`, we need to trace all functions of the module in which a transaction object is referenced. This is performed by finding blocks that contain the information of module's function from the debug symbols. The function's input arguments and local variables are also in the block (Fig. 2, lines 16–21). Based on the extracted data, the `nb_transport_bw` function of the module must be traced as well as the transaction object `trans` (Fig. 2, line 18) which is its input argument. To trace the function, a breakpoint is set at the beginning of the function body (Fig. 3, line 4). For this breakpoint, a set of commands (Fig. 3, lines 5–10) is defined that is executed whenever the breakpoint is triggered at run-time. The information that needs to be extracted can be defined within these commands and by default contains information such as the instance name of the module (Fig. 3, lines 6 and 7) or the transaction phase (Fig. 3, lines 24 and 25).

```
1  Symtab for file at_example.cpp
2  . . .
3  Blockvector:
4  . . .
5  block #169, object at 0x370e870 under 0x39d6930, 4\
6  syms/buckets in 0x407176..0x40755c,function\
7  AT_typeA_initiator::nb_transport_bw (tlm::
       tlm_generic_payload&,...)
8   struct AT_typeA_initiator : public sc_core::sc_module {
9    tlm_utils::simple_initiator_socket<AT_typeA_initiator\
10    ,32u, tlm::tlm_base_protocol_types> socket;
11     . . .
12   public:
13    virtual tlm::tlm_sync_enum nb_transport_bw(...);
14    . . .
15   } * const this; computed at runtime
16   class tlm::tlm_generic_payload {
17    . . .
18   } &trans; computed at runtime
19   class tlm::tlm_phase {
20    . . .
21   } &phase; computed at runtime
22 . . .
```

Fig. 2. Part of the debug symbol of the *AT-example* design generated by GDB.

```
1  set logging on run-time_traces_log.txt
2  ...
3   ##pre-defined breakpoit##
4  break AT_typeA_initiator::nb_transport_bw
5  commands
6   printf "instance_name is : "
7   print this->m_name
8   ...
9   gdb_AT_typeA_initiator_nb_transport_bw
10 end
11  ##pre-defined GDB function##
12 def gdb_AT_typeA_initiator_nb_transport_bw
13  info line *$pc
14  set $end_func_line_num= $_
15  ...
16  break +1    ##local breakpoint##
17  commands
18   gdb_AT_typeA_initiator_nb_transport_bw
19   ...
20  end
21  ...
22  printf "trans_ID AT_typeA_initiator.nb_transport_bw_-
        _trans_ID is : "
23  print &trans
24  printf "trans_phase AT_typeA_initiator.nb_transport_bw_
        -_phase.m_id i : "
25  print phase.m_id
26  ...
27 end
```

Fig. 3. Part of the *GCF* of the *AT-example*.

B. Run-Time Information Retrieval

In order to retrieve the run-time information with respect to the execution's flow, the GCF contains a set of breakpoints to pause the execution, store the detailed information of the execution's state and resume it afterwards. These breakpoints are set for each function of the design's modules and the global functions of the model. They are referred to as *predefined*

```
<TLM_Design_Architecture Design_name = "AT—example">
 . . .
 <Module_name name = "AT_typeA_target" type="target">
 <Function name = "send_response" type="void">
  <Local_var name="trans" type="tlm_generic_payload"/>
  <Local_var name="status" type="tlm::tlm_sync_enum"/>
  <Local_var name="bw_phase" type="tlm_phase"/>
  <Local_var name="delay" type="sc_time"/>
 </Function_name>
  . . .
 <Global_var name="response_in_progress" type="bool"/>
 <Global_var name="n_trans" type="int"/>
 <Global_var name="socket" \
   type="tlm_utils::simple_target_socket"/>
 . . .
</TLM_Design_Architecture>
```

Fig. 4. Part of the structural presentation of the architecture of the *AT-example*.

breakpoints (line 4 in Fig. 3) as they are statically defined based on a function's name before running the GDB script. The *predefined breakpoints* (which are set up before execution) are placed at any relevant function's first line and thus triggered when their corresponding functions are called. Due to limitations concerning the amount of breakpoints, successive lines within these functions cannot all be prepared with breakpoints before the execution as well. Instead, to record any changes within a function, setting a new breakpoint for the next instruction of the function is part of the set of commands that are executed for the first breakpoint, just like it is part of this newly set breakpoint. This recursive process is performed repeatedly until the execution reaches the end of the function's body. The goal of a *predefined breakpoint* is to halt the execution at beginning of a module's function while successive *local breakpoints* are used to step through the body of module's function line by line.

C. Information Transformation

1) Architecture Presentation: The structure of a given ESL design is extracted in two steps.

1) During the static analysis in the first phase of the proposed method including:
   a) the root name and type of each module;
   b) the name and type of each function;
   c) the variables of each module;
   d) local variables of each function.
2) During the execution of the model where each *predefined breakpoint* contains instructions for GDB to retrieve the structural information that cannot be retrieved during the first phase including:
   a) the instance name of each module;
   b) binding information of signals and sockets.

The extracted information from both steps is bound together to create a complete structure of the design.

To present the extracted architectural information in a structured, standardized format, the *run-time data analyzer* module stores it in an XML formatted file. The root element of this XML model is the name of ESL design. The structure of the SystemC design is hierarchical itself, with the first child elements being modules and global functions. The child elements of these are their respective member functions and attributes. Fig. 4 shows a part of the XML presentation of the *AT-example* design's structure.

2) Behavior Presentation for SystemC Designs: One of the common solutions to trace the simulation behavior of a SystemC design is the utilization of VCD files. It is usually generated by the standard SystemC API. Although this method works well for SystemC data types which are defined as signals of modules, it comes with some drawbacks as follows.

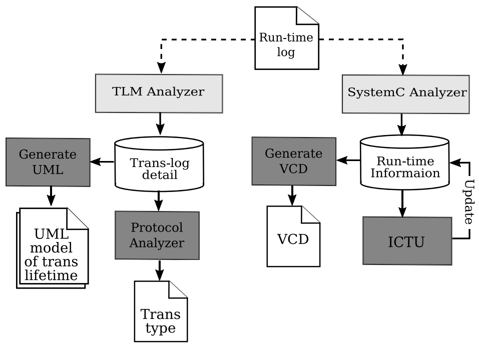


Fig. 5. Architecture of the *run-time data analyzer* module analyzing the extracted run-time information of ESL designs.

1) It lacks precision for base type variables (e.g., C++ data type) that may change several times during a single SystemC-δ-cycle (the smallest amount of time that may pass concerning the simulation kernel).
2) It fails for user-defined datatypes that are not supported at all unless the designers alter their code.
3) It cannot trace the values of local variables of modules and functions.

To present the value of SystemC signals, the standard SystemC API uses SystemC δ cycles as the smallest timestep to differentiate changes on each signal in VCD file. This may result of less precision for base type variables. As we take advantage of GDB as an execution environment, the smallest timestep to differentiate assignments is C++ statements. This (potentially) increases the precision of tracking value changes in the order they occur in, while still setting them in context of the current simulation time. We call this behavior of values within a single δ cycle the *intracycle behavior*.

The *intracycle behavior* analysis can facilitate the design process such as debugging where a C++ data type is altered by some wrong computations and then used in the right hand side to be assigned to a SystemC signal. While the conventional VCD file only illustrates the final value of the signal at each δ cycle, the *intracycle behavior* analysis enables designers to trace all local variables' intermediate changes within the δ cycle. Thus, the exact point of the bug can be easily detected.

In order to present the extracted information in the shape of VCD file, the following information is extracted during the execution time.

1) The simulation time stamp.
2) All variables' value of modules and local variables of functions.
3) The name of each port.
4) Binding information of each port.
5) The instance name of modules.
6) Clock information (if available).

As illustrated in Fig. 5, first all retrieved information is stored in an internal data structure by the *SystemC analyzer* module where each simulation time stamp is considered as a time unit in which the state of variables that was extracted at this time is stored. Second, to differentiate values of variables within a single point in simulation time, the *run-time information* is processed by the intracycle time unit (ICTU) module.

To present all events on a single timeline, the assignments of each single point in simulation time are sorted by their execution order o and the value $o \cdot \mu_t$ is added to their

Algorithm 1: ICTU Update Process

```
Input: simulation time scale $stc$, Run-time Information $RI$
Output: updated $RI$
foreach time unit $t$ in $RI$ do
    foreach variable $v$ in $t$ do
        $vc[t][v] \leftarrow sum(uinq\ value\ changes\ v)$;
    end
    $tvc[t] \leftarrow sum(vc[t])$;
end
$max_{tvc} \leftarrow max(tvc)$;
$\mu_t \leftarrow (1/max_{tvc}) * stc$;
foreach time unit $t$ in $RI$ do
    foreach variable $v$ in $t$ do
        if $vc[t][v] > 1$ then
            foreach value assignment do
                $t_{new} \leftarrow t + \mu_t$;
                store ($t_{new}$, value assignment);
                $t \leftarrow t_{new}$;
            end
        end
    end
end
update ($RI$);
```

original timestamps. The value μ_t is thus used to differentiate the particular assignments. It should be much smaller than the smallest step in simulation time in order to have all assignments being displayed before the next "large" simulation timestep. μ_t is therefore related to the maximum sum of the number of value changes of variables in a time unit among all time units as illustrated in Algorithm 1 and is calculated automatically.

Finally, the updated version of the *run-time information* is used to generate a VCD file of the design's behavior via the *generate VCD* module.

3) Behavior Presentation for TLM Designs: To present the behavior of a given SystemC TLM-2.0 model, both transaction flow and transaction data are retrieved and stored in the *run-time log* file. To extract the transaction flow, each *predefined GDB function* contains instructions for GDB to extract the following.

1) The sequence number of objects' activation.
2) The root name of each module taking part in the transaction.
3) The role of each module taking part in the transaction (for TLM modules can be initiator, interconnect or target and for others is global).
4) The instance name of each module.
5) The name of the current function, its arguments' values and its return value (if available).
6) Source code information (i.e., line of code and source file name).
7) The simulation time.
8) The transaction reference address.

In order to retrieve the transaction data, the GDB instructions extract the attributes of a transaction object which are the following.

1) Data value.
2) Address.
3) Command.
4) Data length.
5) Response status.

In the next step, the extracted information in the *run-time log* file is translated to a structured format. To do this, the information related to each single transaction within its lifetime is distinguished from other transactions. In order to trace

```
transID_number :0x7054f0_1
...
([AT_typeE_target::nb_transport_fw, target4\
,at_typee_target.h:74, 0 ns, 0x7054f0\
,target], [0x6bc660, 100, tlm::TLM_READ_COMMAND\
,4, tlm::TLM_OK_RESPONSE, BEGIN_REQ, 4565 ps\
,NULL], [])
...
transID_number :0x7078f0_1
...
([AT_interconnect::nb_transport_fw, interconnect\
,at_interconnect.h:196, 6397 ns, 0x7078f0\
,interconnect], [0x6c9dc4, 232, tlm::TLM_WRITE_COMMAND\
,4, tlm::TLM_OK_RESPONSE, END_RESP, 243 ps\
,tlm::TLM_COMPLETED], [])
...
```

Fig. 6. Part of the *trans log details* of the *AT-example*.

a single transaction in the *run-time log* file, transactions are separated based on some unique elements. In addition to the *transaction ID*, some attributes of a transaction object (e.g., response status) as well as other elements related to it (e.g., the value of the phase argument on call to and return from the *nb-transport* function and the return value of the function) are used to determine the start and end point of the transaction. The phase argument represents the current state of a module with respect to the TLM-2.0 base protocol state machine of phase transition. This information is referred to as the *transaction related information*.

As demonstrated in Fig. 5, the *TLM analyzer* module gets the *run-time log* as an input. It extracts the required information for each single transaction to describe the activity within its lifetime and stores it in the *trans-log detail* file. This data is an accurate trace of each transaction's behavior, covering all changes of transaction data that occurred during the execution of the model. Fig. 6 shows a part of generated *trans-log detail* of the *AT-example* design.

As all information of a single transaction to describes its lifetime is stored in the *trans-log detail*, the next step is to understand the type of transactions. By identifying the types of communication interface (blocking or nonblocking) in each transaction lifetime, the type of timing model is extracted either loosely or accurate time model or a combination of both. While the loosely time model can only be implemented in one way due to the TLM-2.0 base protocol, the accurate time model comes in 13 unique ways. In case of the accurate time model, we take advantage of the *transaction related information* to distinguish different types of base protocol transactions from each other. As shown in Fig. 5, this analysis is preformed by *protocol analyzer* module and the result is stored in the *trans type* file.

Finally, in order to reduce the complexity of understanding the extracted information stored in the *trans-log detail*, the *generate UML* module generates a UML sequence diagram for each single transaction stored in *trans-log detail*. The generated UML diagram is a message sequence chart introduced by the OSCI TLM-2.0 reference manual in [7] but it provides the designer with more detailed information. It describes the transaction flow among modules' communication for each single transaction within the design. It also includes the transaction data (i.e., the last changes of the transaction data and not all temporary changes) which is passed among modules during their interactions. In particular, the UML model includes a set of sequence numbers indicating both transaction flow and transaction data within its lifetime.

4) Designer Interaction for Optimized Translation: In order to provide a better view of a given ESL design's behavior and structure, the current implementation offers several configuration options available for designers. The extracted information of the design can be filtered based on the following parameters.

1) *Kinds of Variables:* The generated VCD file is optimized to only show the signals of modules, local variables of modules' functions or variables of global functions.
2) *Depth of Information:* The generated VCD file may exclude the *intracycle behavior* (resulting in a more "classic" SystemC trace that only tracks values when the simulation time advances).
3) *Time Window:* The generated VCD shows only the information from a specific period of simulation time.
4) *Depth of Hierarchy:* The generated XML model is optimized to only show the information of modules (and also their member functions and their variables).

In case of SystemC TLM-2.0 designs, the generated UML model may only show the sequence of certain modules' activity or include the transactions' data as well. This allows the designer to easily focus on the points of interest in the design by filtering out any irrelevant information. Moreover, reducing the amount of translated information enhances the readability of generated UML, VCD, or XML models.

IV. Application: Validation of ESL Designs

In this section, the utility value of the run-time information analysis method is illustrated. The extracted information is used to assist the validation of an ESL design against its formal specification [15]. First, the importance of ESL design validation and the related work in this domain are presented in Section IV-A. Second, the proposed methodology is shown in Section IV-B.

A. Background

One solution to ensure that design constraints such as timing and phases are satisfied at ESL, is the utilization of formal specification languages such as the UML [32] early in the design process. UML has become a de-facto standard for the modeling phase in design processes as it supports object-oriented and concurrency features that are used by SystemC [28], [31]. This enables designers to define the structure and the behavior of the design in order to either create a reference model for ESL implementation or generate ESL code stubs or initial implementation. However, manual refinement steps of the generated ESL model are required. As the implementation of ESL designs based on a formal specification is either a partly (in case of generating the initial steps automatically from UML model) or fully manual process, errors may be introduced–resulting in the final ESL model potentially differing from the formal reference model. Therefore, the validation of ESL design against its formal specification is necessary.

Previous approaches (either formal or simulation-based) mostly focus on verifying SystemC TLM-2.0 designs against TLM-2.0 rules and do not support compliance checking of ESL models against their formal specification [18]–[20], [22], [26]. The method [41] employs SystemC data extraction approach [40] to extract the information of a SystemC model. It checks the equivalence of the retrieved data to the model's formal specification. Due to use of [40], it inherits the same limitations which are no support for behavior validation and TLM construct.

Overall, the existing methods have two major problems: in case of formal verification, methods are limited to support a wide range of ESL designs. Regarding simulation-based verification, they are usually using intrusive techniques, either altering the source code or modifying the SystemC library. This can reduce the degree of automation or create compatibility problems for the application of several approaches in parallel. Moreover, they are mostly restricted to verify SystemC TLM-2.0 designs against TLM-2.0 rules. Therefore, in this section we show how both, the behavior and the structure of a given ESL design, can automatically and nonintrusively be verified against its formal specification.

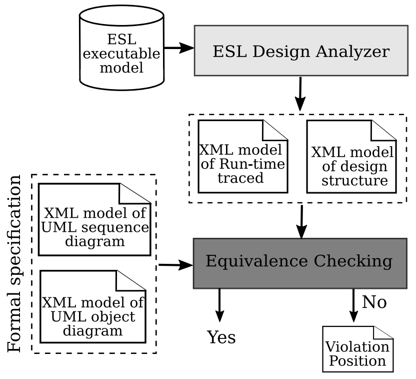

ESL executable model
ESL Design Analyzer
XML model of Run-time traced
XML model of design structure
Formal specification
XML model of UML sequence diagram
XML model of UML object diagram
Equivalence Checking
Yes
No
Violation Position

Fig. 7. Architecture of the validation method.

B. Validating ESL Designs Against Their Formal Specifications

To ensure the equality of a given ESL model against its formal specification, both, structure (e.g., modules, signals, or binding information) and behavior (the order in which methods are invoked, must be checked. The structure of the model is usually defined using block definition diagrams including components, their attributes and member functions, binding information of modules' signals, and parameters of functions. The behavior of the model is usually specified using sequence diagrams, determining the expected order of the modules' activations (the information for each sequence includes the name of module, its instance name, and function name). For a design implementing different tasks or communication protocols (in case of SystemC TLM-2.0 design), each of them are presented as an independent sequence diagram. We consider each of this sequence diagrams a "scenario," presenting one different task or communication protocol of the design formally.

The architecture of the validation methodology itself is presented in Fig. 7. In order to validate the structure of an ESL design against its specification, the model of the design's structural information is compared to its formal presentation. We assume that the formal specification of the structure and the behavior of the design are provided by designers. As the structure of an ESL design is described using root elements (which are modules and global functions), a bijective (one-to-one) mapping algorithm is proposed to check the equality of two models. Based on this, a root element in the XML format of the ESL's formal specification is selected to be mapped to the corresponding root element in the XML model of its run-time traces. Whether an element of the specification matches an element from the implementation is determined by several criteria.

1) Their names must be equal.
2) Their child elements must be equal, i.e.:
    a) their variables' names and types;
    b) their ports' names and types;

c) their functions' signatures (name, return type, and parameters);
d) their local variables.

If no fitting root element is found in the XML model of run-time traces, the corresponding implementation element is reported as a violation. Otherwise, the algorithm continues with the next element. If a specification does not include detailed information (such as variable names or their types) this information can be easily filtered from the results in order to keep the extracted models on roughly the same level of abstraction as the given specification.

In order to check the compliance of a models' behavior, the extracted run-time information from the analysis step in Fig. 1 is translated to an XML model. We assume that each design is run using a testbench that includes a set of tests which provide a full coverage on the complete functionality and all communication protocols of the design. The ESL model's behavior compliance is tested by checking whether each transaction (implementing a unique communication protocol) or functionality (in case of SystemC design) matches the expected sequences of components' activation (i.e., function calls and corresponding responses).

In case of SystemC designs, in [15] the simulation run is used to generate only one sequence diagram from the *run-time log*. As the behavior of a design is rarely ever defined using only one single sequence diagram, comparing this to the given specification is difficult. Instead, each functionality is translated into its own sequence diagram. Functionalities are distinguished based on test cases, for each of which a unique sequence diagram is generated. Since several test cases can activate the same functionality, we only verified unique functionality to reduce the validation time. The instance name and the name of its function are concatenated for all modules taking part in each functionality to create a unique identifier, which is called the *functionality signature*. The validation algorithm then checks for each unique functionality in the XML presentation of the extracted run-time traces of the model whether or not it complies with the expected scenario.

Regarding SystemC TLM-2.0 designs, the algorithm checks the equivalence of each unique transaction with the expected scenario. Thus the first task is to create a list of unique transactions, each of them representing a unique communication protocol. This reduces the validation complexity as the design might contain several transaction of the same communication protocol. In [15] only the transactions' reference address (*transaction ID*) is used as key to filter redundant transactions. This may not be sufficient as an initiator module can generate two transactions with the same address implementing different communication protocols (e.g., the transactions are sent to two different target modules through an interconnect). Therefore, some scenarios might not verified in this case. To solve this problem, in addition to the *transaction ID*, we add another key to distinguish transactions that share the same address: the instance name of each module taking part in transaction lifetime is concatenated to the transaction ID to create a *transaction signature*.

As illustrated in Algorithm 2, a list of unique transactions and functionalities is stored, allowing a matching scenario to be selected from the formal specification part if available. If a fitting scenario is found, the success is indicated to the user and, for each extracted transaction or functionality, a corresponding (XML formatted) log is generated from the transaction's run-time traces. Otherwise, while the algorithm continues for all transactions and scenarios, the scenarios that could not be matched are reported as violations.

Algorithm 2: Behavior Validation Algorithm

```
Input: SystemC_trace ST, TLM_trace TT, scenario_list SL
Output: Equivalence_list EL
functionality_unique_list ful ← ∅;
transaction_unique_list tul ← ∅;
foreach functionality f in ST do
    fs ← functionality_signature(f);
    if fs not in ful then
        add (fs, ful);
foreach transaction t in TLM-2.0_trace do
    ts ← trans_signature(t);
    if ts not in tul then
        add (ts, tul);
foreach transaction/functionality t/f in ful/tul do
    foreach scenario s in SL do
        if s in ful/tul then
            EL[s] ← true;
        else
            EL[s] ← false;
```

V. Experimental Evaluation

Several standard ESL designs from various domains have been used to evaluate the quality of the proposed method. The experimental evaluation is presented in two steps. First, two case studies—RISC-CPU (implemented in SystemC) [21] and *AT-example* (implemented in SystemC TLM-2.0) [6]—are discussed in Section V-A. Second, we give a brief discussion based on the obtained results to evaluate the quality of the proposed method in Section V-B. All algorithms are implemented in python. The experiments are carried out on a PC equipped with 8-GB RAM and an Intel core i7 CPU running at 2.4 GHz.

A. Case Studies

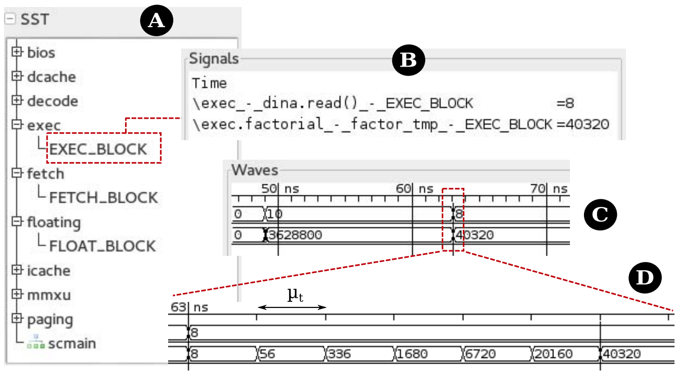
The experimental results of applying the proposed method to all benchmarks are presented in Tables III and IV for SystemC and SystemC TLM-2.0, respectively.

The SystemC models that are presented in Table III are taken from the standard examples which provided by OSCI [21], S2CBench [33], GitHub [3], [29], and the University of Edinburgh [1]. In this table, column *test size* illustrates the size of application running on each design. columns *SystemC Model*, *LoC*, *Comp*, and *Test* show the name of designs, the lines of code, number of modules and the number of test applying to each design, respectively. We compare the results of our methods to the SystemC trace file API in terms of the number of retrieved variables *#Var*, number of extracted time units *#TU* and execution time *Exec*. Columns *ExecD* and *ExecM* show the execution time of the proposed method when data is stored on disk and memory, respectively. As demonstrated in this table, the amount of both, retrieved variables and time units for all case studies of the proposed method are much higher than those retrieved via the SystemC trace API. The parameter *#TU* represents value changes of variables. These parameters illustrate the accuracy of our approach to extract the detailed behavior of a SystemC design. This difference in the amount of retrieved data reflects one major advantage of the proposed method. As it is able to trace value changes even for both, global and local native variables (that have the SystemC primitive data types) or compound data types (which can be constructed using the programming language's primitive data types) regardless of whether they are placed on the stack or the heap not only when the simulation time advances but also of any assignments in-between. Moreover, using the SystemC trace API to analyze the behavior of a given SystemC model is an intrusive solution. It required further programming effort by the designer to modify the original source code to include all variables that need to be traced. Thus, for a complex design with lots of variables, may be a time consuming task.

The SystemC TLM-2.0 models in Table IV are the standard examples providing by Doulos [6]. Columns *TLM model* shows the name of SystemC TLM-2.0 designs. The *LoC* and *#Comp*, *#TM*, and *#Trans* illustrate the complexity and difference of each design in terms of lines of code, number of components, number of transaction and the timing model, respectively. Columns *#UTrans* presents how many different base protocol transactions are implemented in each design which is performed automatically by analyzing each transaction lifetime. The *#Seq* column shows the number of lines related to the unique behavioral information that has been extracted during the execution of each design. Column *CExeT* presents the compilation and execution time of each design without any modification. The TP-GDB column shows for each SystemC TLM-2.0 design the required time to analyze it using GDB trace point. The term "GDB trace point" refers to the watchpoints feature of GDB that can be used to trace an expression such as the value of a variable or an operation during the execution of a program. Whenever the value of aforementioned expression changes, the corresponding watchpoint automatically stops the execution and reports its old and new values. This requires that the designer first sets a breakpoint at a specific location (e.g., a function of a design) and then adds a watchpoint to trace the desired expressions (e.g., variables or signals).

In both tables the OSDS column shows the size of generated output data sets followed by size of the *run-time log* (column *LogF*) and generated XML model of each design's structure (column *XML*). Column *VCD* in Table III presents size of generated VCD for each SystemC design. The UML size reported in Table IV is the average size of the generated UML diagram over all transactions of each TLM design. In both tables, the largest output file is the *run-time log* (which is generated in the first phase of the proposed method) as it includes the detailed data and the complete history of the execution. A large part of the information in the *run-time log* is used to track the useful data related to the behavior and structure of the design and thus filtered out during the translation process (in the second phase). The XML and UML files for all case studies remain in the order of KB. Using the filter options can reduce the size of output data sets (VCD, UML, and XML) even more, allowing the designer to have a better readability, focusing the resulting logs at whatever issue is currently at hand.

1) RISC-CPU (SystemC Design): The RISC-CPU design is a standard OSCI example implementing a CPU in SystemC using ten different modules. The instruction set is based on commercial RISC processors together with MMX-like instruction for DSP programs. It consists of more than 39 instructions such as arithmetic, logical, branch, floating point and SIMD (MMX-like). In order to show the intracycle behavior, we modified the `exec` module by adding a combinational function calculating the `factorial` of the `dina` input. The factorial function is added as an exemplary hardware accelerator, computing the factorial computation in a single clock cycle. Integrating such accelerators is a common approach to gain performance for a specific task. Fig. 8 illustrates a part of the generated VCD file of the RISC-CPU system. The



```
Time
\exec_-_dina.read()_-_EXEC_BLOCK          =8
\exec.factorial_-_factor_tmp_-_EXEC_BLOCK =40320
```

Fig. 8. Part of the generated VCD file of the RISC-CPU example.

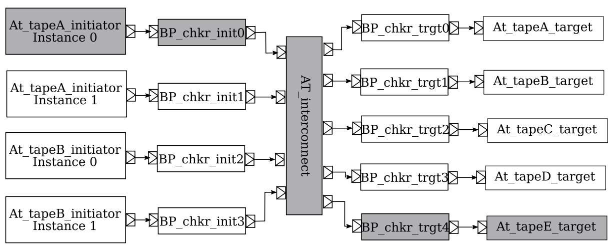


Fig. 9. Architecture of the *AT-example*.

retrieved structural and behavioral information of the design is shown in four parts in this figure.

1) (A) shows a basic hierarchy of the RISC-CPU design in form of the signal search tree (SST). It includes the name of modules and their instances as well as the global functions (e.g., *scmain* function in this example). E.g., the `exec` module has only one instance in the RISC-CPU design which is `EXEC_BLOCK`.
2) (B) illustrates the state of the design's variables after the simulation. Each variable is identified based on its hierarchical structure which consists of the name of the root module, the name of function (for local variables), the name of variable and the instance name of the root module. The expression `exec_-_dina.read()_-_EXEC_BLOCK` shows that the instance `EXEC_BLOCK` of module `exec` has a signal `dina`. The value of this signal (here is 8) is access by member function `read()` of its type. The expression `exec.factorial_-_factor_tmp_-_EXEC_BLOCK` illustrates that the function `factorial` of instance `EXEC_BLOCK` of module `exec` has a local variable `factor_tmp`. The value of this signal is 40 320.
3) (C) shows the value of each variable which is assigned during run-time in the shape of a waveform with respect to the simulation time.
4) (D) finally illustrates the intracyclic information, showing how new, small timesteps illustrate a variable being increased within a single SystemC-δ-cycle. In this example, at time unit $t = 63$ ns the input signal `dina` is assigned to 8 thus the combinational function `factorial` is called to calculate 8! It is computed by a *for-loop* statement where the local variable `factor_tmp` is defined to store the partial values of the factorial of `dina` input in each iteration. As the entire calculation is performed in a single SystemC-δ-cycle, these temporary changes are covered by the intracycle behavior analysis. To cover these temporary

changes in a single time unit, the *Dynamic Information* is analyzed by ICTU module based on Algorithm 1. As the maximum amount of assignments within each of the simulation timesteps is max $= 100$ and the simulation time scale is $ts = 1$ ns, the smallest step within a single time unit is $\mu_t = (ts/\max) = 10$ ps. This allows the method to store all value changes (even temporary ones).

2) AT-Example (TLM-2.0 Design): The *AT-example* uses eight of the 13 TLM-2.0 base protocol's specified transaction protocols, with the unused parts being small cases such as transactions that are canceled immediately after initiation. The design includes multiple approximately timed (hence the name of the model) initiators and targets, as well as an AT interconnect. The architecture of the *AT-example* design is shown in Fig. 9. It includes 19 modules, namely four initiators, one interconnect, five targets, and nine checkers. It consists of two different type of initiators named `AT-typeA-initiator` and `AT-typeB-initiator`, and five different targets named `AT-typeA-target` through `AT-typeE-target`. Initiators (types `A` and `B`) and targets (types `A-E`) each implement different cases of a TLM communication's phase transitions (from the TLM-2.0 standard). For each communication that is done by each TLM module, the transaction packet is checked by the checker modules (which are `BP-chkr-init0` to `BP-chkr-trgt4`).

The proposed method uses the suggested approach and the extracted data to generate a sequence diagram, shown in Fig. 10. For each interaction between two modules, the corresponding arrow indicates both.

1) Operations that are called on a module instance.
2) Transaction data.

Fig. 10 illustrates a part of behavioral information of the *AT-example* (i.e., the gray component in Fig. 9) for a single transaction in the shape of an UML model. In this figure, the black stadium shapes illustrate the root name and instance name of modules and global functions within the design. For global functions, the instance name is NULL. The role of each component is presented on top of the components' name. It separates TLM modules and global functions or modules (e.g., for monitoring a transaction) which are not supposed to be a main part of transaction flow with respect to TLM-2.0 protocol but still necessary to describe the model's behavior. For each interaction between two modules, the information on each arrow indicates operations that are called on an instance and are drawn from the caller to the callee with respect to the simulation time. In particular, for a call from a TLM module or related modules (e.g., for monitoring a transaction), it shows the number of sequence, the name of the caller function, the timing phase, the delay time related to the phase transition, and the return value of the callee (if available). Regarding a call from a global function, it shows the sequence number and the name of the caller function.

In addition to the transaction flow, the generated UML model shows detailed transaction data. The box under each arrow contains the transaction's attributes, which are passed as an argument from caller to callee. The white boxes indicate a transaction object locally created by an initiator module and passed as a function argument while the blue boxes show a transaction object reached the callee through a function call. As an example, `seq-18` in Fig. 10 shows the information of the response of the target module `AT-typeE-target`. It includes the name of called function (`nb_transport_fw`), timing phase of the transition (`BEGIN_REQ`), timing annotation (`4598 ps`), and the return value from *nb_transport_fw* (`tlm::TLM_COMPLETED`). It also shows the transaction data passed to the callee module including the reference to the transaction data (`0x6bc660`), address (`100`), command (`tlm::TLM_READ_COMMAND`), length (`4`), and response status (`tlm::TLM_OK_RESPONSE`). The obtained result in this example also demonstrates that the timing annotation that is used for the internal process does not have an effect on the simulation time, as it is expected from TLM transactions. Notice that the information also includes both, class names and instance names for the given modules, seen on top of Fig. 10.

TABLE I
ANALYSIS OF THE *AT-Example* TRANSACTIONS TIMING MODEL

Call	Return	Phase Transition	#Seq
fw	TC	BRQ	45
fw	TU/TC	BRQ/BRP/ERP	80
fw/bw	TU/TC	BRQ/ERQ/BRP	82 / 82[1]
fw/bw	TA/TC	BRQ/BRP	59
fw/bw	TA/TA/TA	BRQ/BRP/ERP	103
fw/bw	TA/TU	BRQ/BRP/ERP	81
fw/bw	TA/TA/TA/TA	BRQ/ERQ/BRP/ERP	127
fw/bw	TA/TA/TU	BRQ/ERQ/BRP	82

[1]Example contains two implementations, both being a sequence of 82 calls.

Table I illustrates the result of analyzing the *AT-example* by the proposed method. It shows the different ways of modeling communication-interfaces (implementing different base protocol transactions) of the design, all of which are automatically extracted by analyzing the transactions' lifetime. Therefore, designers can easily understand which types of communication interfaces and base protocol transactions are implemented in a given TLM-2.0 model. The *Call* column presents the type of call to transport the transaction which are *nb_transport_fw* (*fw*) and *nb_transport_bw* (*bw*), i.e., forward and backward paths of the nonblocking transport interface. The *return* column shows the return value of these methods based on which protocol is used. Eligible values are *TLM_ACCEPTED* (*TA*), *TLM_COMPLETED* (*TC*), and *TLM_UPDATED* (*TU*). *Phase transition* presents the possible transitions of the timing phase for each transaction protocol which are *BGIN_REQ* (*BRQ*), *END_REQ* (*ERQ*), *BGIN_RESP* (*BRP*), and *END_RESP* (*ERP*). Finally, *#seq* shows the number of function calls related to the given transaction protocol that were extracted by the proposed method.

B. Integration and Discussion

The proposed method is able to automatically retrieve a huge amount of information to describe the structure and behavior of a given ESL implementation.

According to [27] the following criteria should be considered by ESL analysis approaches.

1) Avoiding any preconditions concerning the source code or workflow in order to be as compatible as possible.
2) Applying as little intrusion to the existing sources and workflow as possible.
3) Extracting detailed information of a design which reflects its structure and behavior.
4) Supporting TLM-based models.

We evaluate the characteristics and the applicability of the proposed method based on how it does meet the aforementioned criteria.

1) Concerning the First and Second Criteria: In comparison to approaches that rely on manipulating the original source code or modifying the SystemC library or interfaces to extract the run-time information, the proposed method extracts

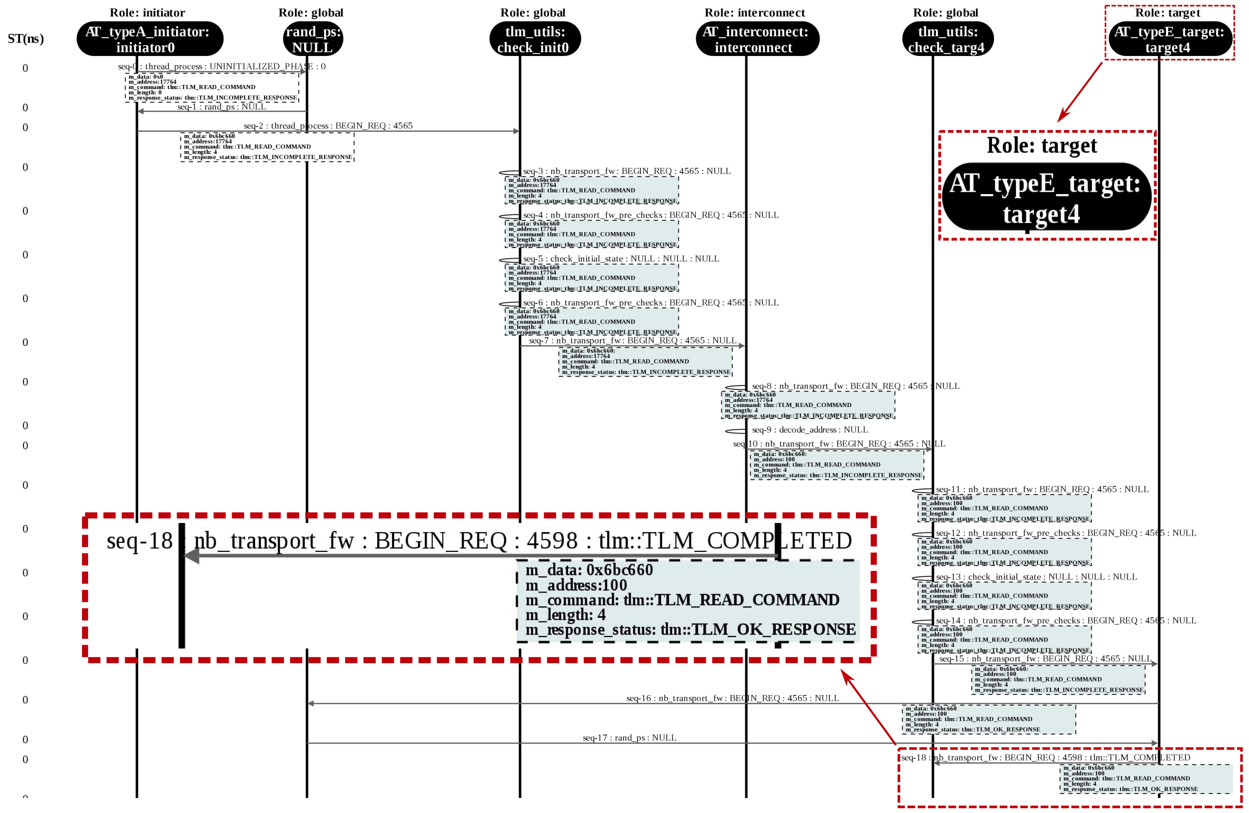


Fig. 10. Part of the generated UML model of the *AT-example* behavior presenting detailed transaction data.

the detailed run-time information without any modification of the user's implementation and the standard tool flow. The information extraction process by the proposed method can be perform even without the original source code as it only needs the binary executable of the design and its debug symbols. It means that the proposed method is a nonintrusive and code-independent approach. The main problems of using intrusive techniques are the compatibility issues (that arise from modifying the SystemC library) and the reduction of the automation degree (due to expensive manual processes to manipulate the original source code). Moreover, it can be combined with setups that already rely on a modified SystemC library which makes the approach applicable for a wide range of ESL designs. This makes the proposed method as an easy-to-use ESL analysis solution.

2) Regarding the Third and Fourth Criteria: Unlike the methods that focus on extracting the static aspect of an ESL implementation, the proposed method extracts the models' behavior as well as their structure. It presents the retrieved information of a given ESL model in two different categories.

1) An XML model of the design architecture.
2) A structured trace file of design's run-time information including:
    a) intracycle analysis of SystemC designs' behavior in the standard VCD format;
    b) all details of transaction data and flow (UML presentation for each single transaction in its lifetime) by precision of an instruction execution and with respect to design's execution flow.

In case of analyzing SystemC TLM-2.0 models, the developed method meets the requirements stated in [11], [23, Sec. III] as it provides designers with behavioral analysis of the models' transactions.

TABLE II
COMPARING THE PROPOSED METHOD WITH EXISTING APPROACHES

Method	Automated	Non-intrusive	Struct. info.	Behav. info.	TLM-2.0
Ours	✓	✓	✓	✓	✓
[13]	✓	✗	✓	✗	✗
Pinapa [27]	✓	✓	✓	✗	✓
PinaVM [24]	✓	✓	✓	≈	✗
SHaBE [10]	✓	✓	✓	✗	✗
DUST [23]	≈	✗	✓	≈	✓
[40]	✓	✓	✓	✗	✓
[34]	✓	✓	✓	≈	✓

✓support ≈ not fully support ✗not support
Struct. info.: Structural information **Behav. info.:** Behavioral information

Overall, Table II demonstrates the characteristics of the proposed method to analyze a given ESL design in comparison with the existing approaches. It shows that our method not only satisfies all criteria stated in [27], but also provides designers with precise run-time information describing the behavior of an ESL design.

3) Performance: The performance of the proposed method depends on the time that is spent extracting the information of a model. The extraction of the model's run-time information is the major time consuming part of the method. The time that is consumed to extract the run-time information depends on three parameters.

1) The complexity of each ESL design.
2) The size of running application (testbench).
3) The amount of information that needs to be extracted.

Concerning the first parameter, as the program is executed on GDB to store the state of the program during its simulation time on disk (or memory), the execution has to be halted repeatedly. The number and duration of halting a program are very related to the instruction types such as loop, wait

TABLE III
EXPERIMENTAL RESULTS FOR ALL SYSTEMC CASE STUDIES

Test size	SystemC Model	LoC	#Comp	#Test	SystemC Trace #Var	#TU	Exec(s)	Ours #Var	#TU	ExecD(m:s)	ExecM(m:s)	CExeT(s)	TP-GDB Exec(h)	OSDS(MB) LogF	VCD	XML
Small	4-bit shift-register[1]	135	2	50	3	29	0.01	12	54	0:03	0:02	2.2	> 1	0.27	0.012	0.004
	FIR-filter[2]	233	5	30	6	24	0.01	23	504	0:05	0:04	4.1	> 3	1.41	0.063	0.007
	3-stages-pipe[2]	290	5	10	8	23	0.01	25	34	0:06	0:03	4.9	> 3	1.17	0.022	0.007
	FFT_flpt[2]	484	2	10	8	81	0.01	34	129	0:51	0:43	3.9	> 5	1.53	0.041	0.011
	Cholesky[3]	522	2	27	4	11	0.01	28	46	0:17	0:13	5.4	> 5	1.70	0.032	0.010
	Hamming-code[4]	563	6	16	13	32	0.02	55	64	1:09	0:54	6.9	> 5	2.13	0.045	0.016
	Interpolation[3]	596	2	10	10	21	0.02	32	30	0:47	0:41	6.1	> 5	1.55	0.039	0.006
	IDCT[3]	815	2	100	9	205	0.02	27	206	0:41	0:32	3.6	> 5	4.72	0.058	0.004
	VGA[3]	856	6	100	9	59	0.02	37	239	4:23	3:49	3.8	> 5	12.25	0.072	0.010
	Decimation[3]	883	2	20	10	47	0.02	35	48	0:33	0:24	5.4	> 5	1.45	0.054	0.008
	pkt-switch[2]	1020	20	50	14	93	0.05	332	5606	0:20	0:17	8.5	> 10	4.11	0.651	0.036
	MD5-hash[3]	1111	2	5	18	23	0.06	33	115	10:12	8:57	6.3	> 10	18.43	0.041	0.011
	RISC CPU[2]	1960	10	10	89	37	0.03	299	121	0:13	0:11	11.1	> 10	1.06	0.103	0.026
	Simple-bus[2]	2100	7	10	10	359	0.01	32	852	3:43	3.07	5.1	> 10	13.49	0.213	0.011
	AES128lowarea[3]	3280	13	5	10	18	1.39	86	89	12:23	10:04	17.2	> 10	42.67	0.061	0.021
	LZW-encoder[5]	5132	22	5	20	30	2.42	398	139	45:48	37:43	22.3	> 10	91.71	0.239	0.052
Large	4-bit shift-register[1]	135	2	150	3	78	0.02	12	142	0:07	0:05	2.2	> 1	0.76	0.031	0.004
	FIR-filter[2]	233	5	150	6	66	0.02	23	2291	0:19	0:16	4.1	> 3	6.23	0.257	0.007
	3-stages-pipe[2]	290	5	150	8	300	0.02	25	341	0:21	0:11	4.9	> 3	16.26	0.314	0.007
	FFT_flpt[2]	484	3	150	8	1465	0.02	34	5841	4:10	3.29	3.9	> 10	20.12	0.542	0.011
	Cholesky[3]	522	2	108	4	29	0.02	28	429	1:52	1.25	5.5	> 10	6.17	0.102	0.010
	Hamming-code[4]	563	6	128	13	256	0.02	55	512	5:06	4:04	6.9	> 10	16.71	0.311	0.016
	Interpolation[3]	596	2	104	10	205	0.02	32	295	2:31	2:09	6.1	> 10	15.10	0.345	0.006
	IDCT[3]	815	2	996	9	1997	0.02	27	1998	4:11	3.07	3.7	> 10	43.40	0.513	0.004
	VGA[3]	856	6	1000	9	619	0.02	37	2005	15:28	13.11	3.8	> 10	110.37	0.640	0.010
	Decimation[3]	883	2	215	10	437	0.02	35	438	6:06	4:41	5.4	> 10	12.17	0.409	0.008
	pkt-switch[2]	1020	10	350	55	1723	0.05	332	19559	3:39	3:05	8.7	> 10	26.05	3.620	0.036
	MD5-hash[4]	1111	2	50	18	66	0.06	33	307	41:20	35:32	6.3	> 10	165.27	0.192	0.011
	RISC CPU[2]	1960	10	150	89	137	0.03	299	368	9:52	8:16	11.2	> 10	13.39	0.268	0.026
	Simple-bus[2]	2100	7	100	10	4506	0.02	32	9767	24:05	20:09	5.2	> 10	129.96	2.039	0.011
	AES128lowarea[3]	3280	13	50	10	110	3.39	86	397	51:38	41:48	19.2	> 24	419.11	0.219	0.021
	LZW-encoder[5]	5132	22	50	20	290	5.16	398	529	193:41	159:03	25	> 24	904.27	0.722	0.052

[1]Provided by University of Edinburgh [1] [2]Provided by OSCI [21] [3]Provided by [33] [4]Provided by GitHub [3] [5]Provided by [29] **LoC**: Line of Code **#Comp**: Number of components **#Var**: Number of Variables **#TU**: Number of extracted Time unit **ExecD**: Execution Time on Disk **ExecM**: Execution Time on Memory **CExeT**: Compilation and Execution Time **TP-GDB**: Trace Point by GDB **OSDS**: Output Size of Data Set

statements and function call. Therefore, more complexity leads to an increased execution time. Experimental results (columns *ExecD* and *ExecM* in Table III and columns *ETD* and *ETM* in Table IV) show that storing data in the main memory instead of writing it to disk eliminates I/O bottlenecks. This provides designers in average with 21% and 19% performance gain for SystemC and SystemC TLM-2.0 designs, respectively. As long as new inputs are applied to the design, the extraction time is increased linearly. Reducing the amount of data to be extracted (e.g., by limiting the extraction to certain data fields of particular transactions) also reduces the overhead of the extraction method.

In order to show how the performance of the proposed method is related to the second and third parameters, we evaluate it by defining two depths of data extraction specified by *abstract* and *detailed* and two size of applications to be run on each design which are *small* and *large*. The *abstract* data extraction depth includes all structural information and the sequence of all component activations. In addition to the information retrieved in *abstract* level, the *detailed* depth of data extraction contains all value changes of modules and functions variables during execution. Figs. 11 and 12 illustrate the execution time of the proposed method for two depths of data extraction of each design where small and large set of tests are applied to them, respectively. This provides designers with four corners of the ESL design analysis based on the amount of information to be extracted which are small-abstract (SA), small-details (SD), large-abstract (LA), and large-details (LD). The SA corner shows the fastest analysis and the least amount of information about the design while the LD corner represents the slowest and most precise analysis on a given ESL design.

In addition to the *abstract* and *detailed* depth of behavior analysis, the proposed method provides designers with a set of configuration options and parameters in the first phase to generate the GCF script based on the designers' optimization. This allows them to filter out the information that is not required for their analysis and focus on the points of interest. These interactive options are as follows.

1) Filtering some elements of the design, e.g., only trace certain modules (and their signals).
2) Filtering based on depth of information, e.g., trace only modules (and their member functions and variables).
3) Filtering some specific types of function, variables or signals, e.g., do not trace global functions, local variables or signals of type *sc_in*.

The proposed method thus provides a designer-chosen tradeoff between the amount of information to be extracted and the time that is needed to do this.

Regarding SystemC TLM-2.0 designs, Table IV shows the execution time for extracting the static and run-time information and storing them in structured models to a hard drive (column *ETD*) and memory (column *ETM*). It shows that

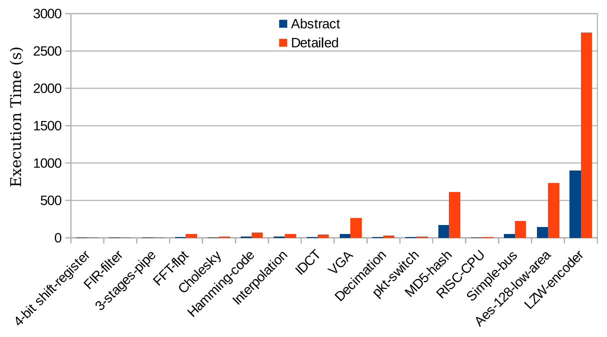


Fig. 11. Run-time analysis of the proposed method based on two depths of information extraction for small applications running on each SystemC design.

TABLE IV
EXPERIMENTAL RESULTS FOR ALL TLM-2.0 CASE STUDIES

Test size	TLM Model	LoC	#Comps	#Trans	#UTrans	TM	#Seq	ETD(m:s)	ETM(m:s)	CExeT(s)	TP-GDB(h)	OSDS(MB)		
												LogF	UML	XML
Small	LT-example	175	2	16	1	LT	35	0:09	0:06	1.4	> 1	0.72	0.007	0.002
	Routing-model	456	6	2	1	LT	68	0:21	0:15	1.5	> 1	1.30	0.014	0.004
	Example-4	547	2	4	4	AT	488	1:29	1:12	1.4	> 1	4.73	0.019	0.003
	Example-5	650	7	2	2	LT	49	1:25	1:09	2.1	> 1	1.72	0.014	0.005
	Snooping-sticky	676	5	64	2	LT/AT	568	2:51	2:12	6.2	> 1	8.24	0.016	0.009
	Example-6	713	9	10	10	AT	683	1:53	1:31	2.2	> 1	8.92	0.038	0.017
	AT-example	2942	19	8	8	AT	1008	7:41	6:19	21	> 1	8.61	0.042	0.036
	Locking-two	3831	23	10	10	LT/AT	523	9:06	7:54	22.6	> 5	10.32	0.038	0.047
Large	LT-example	175	2	160	1	LT	350	2:11	1.39	1.6	> 1	6.39	0.007	0.002
	Routing-model	456	6	100	1	LT	1402	4:15	3:04	1.7	> 1	61.42	0.014	0.004
	Example-4	547	2	348	4	AT	14060	66:27	53:47	1.8	> 24	406.22	0.019	0.003
	Example-5	650	7	69	2	LT	47	31:13	25:53	2.1	> 10	57.14	0.014	0.005
	Snooping-sticky	676	5	512	2	LT/AT	4544	11:33	8:56	6.8	> 5	62.89	0.016	0.009
	Example-6	713	9	245	10	AT	14354	53:03	43:21	2.2	> 24	213.27	0.038	0.017
	AT-example	2942	19	49	8	AT	6181	27:33	23:41	23.2	> 10	51.63	0.042	0.036
	Locking-two	3831	23	371	10	LT/AT	16379	79:15	69:05	24.3	> 24	379.40	0.038	0.047

LoC: Line of Code **UTrans**: Unique Transaction **TM**: Timing Model **ETD**: Execution Time on Disk **ETM**: Execution Time on Memory **CExeT**: Compilation and Execution Time **TP-GDB**: Trace Point by GDB **OSDS**: Output Size of Data Set

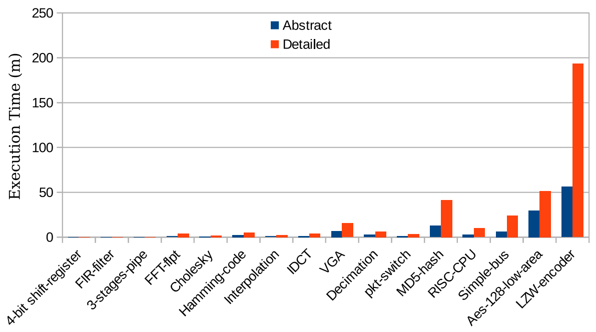


Fig. 12. Run-time analysis of the proposed method based on two depths of information extraction for large applications running on each SystemC design.

by increasing the numbers of transactions (from small test size to large), the execution time increases. The first half of the table (small test size) illustrates that designs with more lines of code and AT timing model (which including more complex base protocol transactions) have larger execution time. Tables III and IV compare the execution time of our method to analyze a given ESL design with both trace point functionality by GDB and its pure compilation and execution time using GCC. Although GDB's trace point feature gives a detailed access to analyze the ESL model, the required execution time even for a simple ESL design lies in an order of hours and for a complex design, days. The method also needs to be set up manually which makes the solution inapplicable even for a simple model. The required time to analyze an ESL design using our method in the SD corner is in most cases within a reasonable boundary in comparison to its pure compilation and execution time using GCC. Even for complex designs, it is still in order of minutes.

Although the performance impact of the proposed method can be considerable for analyzing a large given ESL design (concerning the LD corner) in comparison to its pure compilation and simulation time (worst case the *LZW-encoder* running for about 3 h versus half a minute), there is a wide variety of use cases that justify the application of the given approach.

1) The method is able to retrieve detailed traces from, e.g., native variables, the information which has so far been limited to manual debugging.
2) The build pipeline remains untouched. This keeps the setup time low and lets designers quickly evaluate especially short, questionable test cases. Moreover, any results obtained using the given method are sure to be identical to those that are shown for the production system.
3) In case of third party intellectual property for which only an executable binary is available, the proposed method remains operable. It is the only solution that works on an executable model of the design without availability of source code—retrieving the information that is available via debug symbols and omitting the unavailable data without further setup.

The method can be used at any stage of the design cycle such as helping in understanding the designs' complexity, validation, verification, or synthesis.

4) Memory: As illustrated in Tables III and IV, the largest portion of the generated output data sets (column *OSDS*) belongs to the *run-time log* file (column *LogF*). The reported size for all case studies is in order of MB with the worst case *LZW-encoder* in LD corner which is about 1 GB. This allows designers to usually store the extracted data in the first phase of the proposed method (creating the *run-time log*) in the main memory (instead of the hard disk) to reduce I/O overhead and improve the performance.

5) Limitation: The only precondition of the suggested approach is that the executable ESL design must contain debug information that is compatible with GDB. While GCC and Clang-LLVM are thus supported, Microsoft Visual Studio is currently not. Although our methodology is an overall sound analysis, it does share the inherent performance limitations that come with most other run-time-analysis tools build on GDB.

VI. CONCLUSION

In this paper, a new easy-to-use ESL analysis approach was introduced. The approach is able to analyze SystemC and SystemC TLM-2.0 implementations and translate the corresponding data to usable formats such as VCD and UML sequence diagrams, respectively. The proposed method automatically and nonintrusively extracts both structural and behavioral information of a given SystemC or SystemC TLM-2.0 model without any modification to the source code of the model or the library. Several ESL benchmarks were run to evaluate the effectiveness of the proposed method to analyze ESL designs. The retrieved information in this method can be effectively utilized for ESL designs validation, verification, and synthesis process.

REFERENCES

[1] *The University of Edinburgh School of Informatics-Computer System: LAB 2*. Accessed: Jan. 30, 2017. [Online]. Available: http://www.inf.ed.ac.uk/teaching/courses/inf2c-cs/13-14/labs/lab2.html
[2] *Cadence Incisive (SimVision), Information Available on Cadence Website*. Accessed: Jan. 30, 2016. [Online]. Available: http///www.cadence.com

[3] *Freecores*. Accessed: Jan. 30, 2017. [Online]. Available: https://github.com/freecores/
[4] *Novas Verdi, Information Available on Novas Website*. Accessed: Jan. 30, 2016. [Online]. Available: http///www.novas.com
[5] *IEEE Standard for Standard SystemC Language Reference Manual*, IEEE Standard 1666-2011, 2012, pp. 1–638.
[6] J. Aynsley. *TLM-2.0 Base Protocol Checker*. Accessed: Jan. 30, 2016. [Online]. Available: https://www.doulos.com/knowhow/systemc/tlm2/at_example
[7] J. Aynsley, Ed., *OSCI TLM-2.0 Language Reference Manual*, Open SystemC Initiative, 2009.
[8] D. Berner, J. P. Talpin, H. Patel, D. A. Mathaikuty, and S. K. Shukla, "SystemCXML: An extensible SystemC front end using XML," in *Proc. FDL*, 2005, pp. 405–409.
[9] N. Blanc, D. Kroening, and N. Sharygina, "SCOOT: A tool for the analysis of SystemC models," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Heidelberg, Germany: Springer, 2008, pp. 467–470.
[10] H. Broeders and R. Van Leuken, "Extracting behavior and dynamically generated hierarchy from SystemC models," in *Proc. Design Autom. Conf. (DAC)*, New York, NY, USA, 2011, pp. 357–362.
[11] F. Doucet, S. Shukla, and R. Gupta, "Introspection in system-level language frameworks: Meta-level vs. integrated," in *Proc. Design Autom. Test Europe (DATE)*, 2003, Art. no. 10382.
[12] G. Fey *et al.*, "ParSyC: An efficient SystemC parser," in *Proc. Workshop Synth. Syst. Integr. Mixed Inf. Technion. (SASIMI)*, 2004, pp. 148–154.
[13] C. Genz and R. Drechsler, "Overcoming limitations of the SystemC data introspection," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Apr. 2009, pp. 590–593.
[14] M. Goli, J. Stoppe, and R. Drechsler, "AIBA: An automated intra-cycle behavioral analysis for SystemC-based design exploration," in *Proc. ICCD*, Scottsdale, AZ, USA, 2016, pp. 360–363.
[15] M. Goli, J. Stoppe, and R. Drechsler, "Automatic equivalence checking for SystemC-TLM 2.0 models against their formal specifications," in *Proc. DATE*, Lausanne, Switzerland, 2017, pp. 630–633.
[16] M. Goli, J. Stoppe, and R. Drechsler, "Automatic protocol compliance checking of SystemC TLM-2.0 simulation behavior using timed automata," in *Proc. ICCD*, Boston, MA, USA, 2017, pp. 377–384.
[17] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient automatic visualization of SystemC designs," in *Proc. Forum Specification Design Lang. (FDL)*, 2003, pp. 646–658.
[18] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 1, pp. 57–68, Jan. 2006.
[19] P. Herber, J. Fellmuth, and S. Glesner, "Model checking SystemC designs using timed automata," in *Proc. CODES+ISSS*, 2008, pp. 131–136.
[20] P. Herber and S. Glesner, "A HW/SW co-verification framework for SystemC," *ACM Trans. Embedded Comput. Syst.*, vol. 12, pp. 1–61, Mar. 2013.
[21] (2016). *AS Initiative*. [Online]. Available: http://www.accellera.org/downloads/standards/systemc
[22] V. Jain, A. Kumar, and P. Panda, "Exploiting UML based validation for compliance checking of TLM 2 based models," *Design Autom. Embedded Syst.*, vol. 16, no. 2, pp. 93–113, 2012.
[23] W. Klingauf and M. Geffken, "Design structure analysis and transaction recording in SystemC designs: A minimal-intrusive approach," in *Proc. FDL*, 2006, pp. 169–177.
[24] K. Marquet and M. Moy, "PinaVM: A SystemC front-end based on an executable intermediate representation," in *Proc. Embedded Softw. (EMSOFT)*, 2010, pp. 79–88.
[25] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. San Francisco, CA, USA: Morgan Kaufmann, 2007.
[26] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level," in *Proc. Appl. Concurrency Syst. Design (ACSD)*, 2005, pp. 26–35.
[27] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: An extraction tool for SystemC descriptions of systems-on-a-chip," in *Proc. Embedded Softw. (EMSOFT)*, New York, NY, USA, 2005, pp. 317–324.
[28] W. Mueller *et al.*, "UML for ESL design: Basic principles, tools, and applications," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, San Jose, CA, USA, 2006, pp. 73–80.
[29] Orahyn. *LZW-Encoder*. Accessed: Jan. 30, 2017. [Online]. Available: https://github.com/arshadri/lzw_systemc/tree/master/systemc
[30] H. D. Patel, D. A. Mathaikutty, D. Berner, and S. K. Shukla, "SystemCXML: An extensible SystemC front end using XML," in *Proc. Forum Specification Design Lang.*, Lausanne, Switzerland, 2005, pp. 27–30.
[31] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, "A UML 2.0 profile for SystemC: Toward high-level SoC design," in *Proc. ACM Int. Conf. Embedded Softw. (EMSOFT)*, 2005, pp. 138–141.
[32] J. Rumbaugh, I. Jacobson, and G. Booch, Eds., *The Unified Modeling Language Reference Manual*. Boston, MA, USA: Addison-Wesley, 1999.
[33] B. C. Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC benchmark suite for high-level synthesis," *IEEE Embedded Syst. Lett.*, vol. 6, no. 3, pp. 53–56, Sep. 2014.
[34] T. Schmidt, G. Liu, and R. Dömer, "Automatic generation of thread communication graphs from SystemC source code," in *Proc. 19th Int. Workshop Softw. Compilers Embedded Syst. (SCOPES)*, 2016, pp. 108–115.
[35] T. Schubert and W. Nebel, "The quiny SystemCTM front end: Self-synthesising designs," in *Proc. Selected Contributions Forum Specification Design Lang. (FDL)*, 2007, pp. 93–109.
[36] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosentiel, "Object-oriented modeling and synthesis of SystemC specifications," in *Proc. ASP-DAC*, 2004, pp. 238–243.
[37] W. Snyder. *SystemPerl Homepage*. Accessed: Jan. 30, 2016. [Online]. Available: http://www.veripool.com/systemperl.html
[38] R. Stallman and C. Support, *Debugging With GDB: The GNU Source-Level Debugger*, 9th ed. Boston, MA, USA: Free Softw. Found., 2010.
[39] J. Stoppe and R. Drechsler, "Analyzing SystemC designs: SystemC analysis approaches for varying applications," *Sensors*, vol. 15, no. 5, 2015, Art. no. 10399.
[40] J. Stoppe, R. Wille, and R. Drechsler, "Data extraction from SystemC designs using debug symbols and the SystemC API," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, 2013, pp. 26–31.
[41] J. Stoppe, R. Wille, and R. Drechsler, "Validating SystemC implementations against their formal specifications," in *Proc. 27th Symp. Integr. Circuits Syst. Design (SBCCI)*, 2014, pp. 1–8.

Mehran Goli (S'16) received the B.Sc. degree from the University of Shahid Beheshti, Tehran, Iran, in 2012, and the M.Sc. degree from the University of Tehran, Tehran, in 2015, both in computer engineering. He is currently pursuing the Ph.D. degree with the Group of Computer Architecture, University of Bremen, Bremen, Germany.

His current research interests include system level design, verification, and validation.

Jannis Stoppe received the Dr.-Ing. degree in computer science from the University of Bremen, Bremen, Germany, in 2017.

He recently joined the German Aerospace Center DLR, Institute for the Protection of Maritime Infrastructures, Bremerhaven, Germany, where he currently leads the Situational Awareness Group, focusing on sensor fusion and data analysis. His current research interest includes development of new techniques for analyzing high-level hardware descriptions written in SystemC.

Rolf Drechsler (F'15) received the Diploma and Dr. Phil.Nat. degrees in computer science from J. W. Goethe University Frankfurt am Main, Frankfurt, Germany, in 1992 and 1995, respectively.

He was with the Institute of Computer Science, Albert-Ludwigs University, Freiburg im Breisgau, Germany, from 1995 to 2000, and the Corporate Technology Department, Siemens AG, Munich, Germany, from 2000 to 2001. Since 2001, he has been with the University of Bremen, Bremen, Germany, where he is currently a Full Professor and the Head of the Group for Computer Architecture, Institute of Computer Science. In 2011, he additionally became the Director of the Cyber-Physical Systems Group, German Research Center for Artificial Intelligence (DFKI), Bremen, where he is a Co-Founder of the Graduate School of Embedded Systems and the Coordinator of the Graduate School "System Design" funded within the German Excellence Initiative. His current research interest includes development and design of data structures and algorithms with a focus on circuit and system design.

Dr. Drechsler was a recipient of the Best Paper Award at HVC in 2006, FDL in 2007 and 2010, DDECS in 2010, and ICCAD in 2013 and 2018. He was the Symposium Chair of ISMVL 1999 and 2014 and ETS 2018, and the Topic Chair for "Formal Verification" DATE 2004, DATE 2005, DAC 2010, as well as DAC 2011. He is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, *IET Cyber-Physical Systems: Theory and Applications*, the *International Journal on Multiple-Valued Logic and Soft Computing*, and *ACM Journal on Emerging Technologies in Computing Systems*. He was a Program Committee Member of numerous conferences, including DAC, ICCAD, DATE, ASP-DAC, FDL, MEMOCODE, and FMCAD.