Automated Design Understanding of SystemC-based Virtual Prototypes: Data Extraction, Analysis and Visualization

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*—The ever-increasing functionality of modern electronic systems and reduced time-to-market constraints have significantly altered the typical design flow. One possible solution to deal with this rising complexity is to increase the level of abstraction toward the *Electronic System Level* (ESL). At the ESL, modeling system as a *Virtual Prototype* (VP) using SystemC and its *Transaction Level Modeling* (TLM) framework has become an industry-accepted solution in the last decade. VP design exploration, analysis, debugging, and integration of ever-changing functional requirements can be made faster, more accurate, and errorless with the help of a strong design understanding method.**

This paper presents a comprehensive automated design understanding methodology that enables designers to trace detailed information related to the VPs structure and behavior. The proposed methodology includes three main phases which are data extraction, analysis, and visualization. Experimental results including a real-world VP-based system show the advantages of our methodology such as its accuracy and applicability.

I. Introduction

Modern embedded systems consist of many different functional blocks, including multiple (third-party) *Intellectual Property* (IP) cores, various on-chip interconnects, and memories. The exponential increase in the functionality of *System-on-Chips* (SoCs) and reduced *Time-to-Market* (TTM) constraints have significantly altered the design process to meet the high market demand.

One possible solution to handle the complexity of the electronic systems is to raise the level of abstraction towards the *Electronic System Level* (ESL) [1]. At the ESL, among the existing hardware modeling languages, SystemC [2] has become the de-facto standard modeling language that is used to describe systems as a *Virtual Prototype* (VP). A VP is an abstract and executable software model that is typically implemented using SystemC and its *Transaction Level Modeling* (TLM) [3] framework. By this means, a system can be prototyped quickly and used as a reference model for lower levels of abstraction. Moreover, before the actual hardware is manufactured, designers can test or evaluate the software parts of an embedded system. Thus, it acts as the standard communication platform among system designers, embedded software developers, and hardware engineers along the design process [4].

However, this modern VP-based design flow still has weaknesses, in particular, due to the significant manual effort involved for analysis (e.g., perform design space exploration for the next generation of SoCs, debug, verify or synthesize the existing SoCs) as well as modeling tasks (e.g., test new features and validate the capabilities of SoCs) which is both time consuming and error-prone. Although the simulation-based technique is still the predominant way to handle the tasks mentioned above (due to VP complexity), using them requires accurate knowledge about VPs' structure and behavior. This initial step in the design process is called *Design Understanding*. However, analyzing a given SystemC-based VP for design understanding goal is a non-trivial task. Mainly because C++ (and thus also SystemC, which is a library for the former) is inherently hard to analyze due to 1) the countless compiler-specific dialects that a source code may be written in, and 2) the executable binary format which may be heavily optimized.

In this paper, a comprehensive automated design understanding methodology is presented, enabling designers to accurately trace the structure and behavior of a given SystemC-based VP. The proposed methodology consists of three main phases: data extraction, analysis, and visualization. In the first phase, the static and run-time information of a given VP is extracted from two perspectives: the debugger-based and the compiler-based approaches. The debugger-based approach takes advantage of the *GNU Debugger* (GDB) as its underlying infrastructure. It provides designers with a non-intrusive analysis solution that only requires the executable model of VPs. Thus, in case of legacy or third-party IPs where the source code may not be available at all, it is the only applicable solution. The compiler-based approach is based on the flexible Clang compiler. It introduces a fast analysis technique that scales very well with an arbitrary complexity of VPs or their running software (or application). In the second phase, the retrieved information is translated into a set of *Intermediate Representation* (IR) models, presenting the whole system's behavior (which is based on simulation) and structure in well-structured formats. Finally, to enhance the understanding process of a given VP's intricacy, in the last phase, these IR models are used to visualize the VP's structure in an XML format, and its behavior as *Unified Modeling Language* (UML) diagrams.

II. Background and Motivation

In this section, we first give a brief introduction to the SystemC TLM-2.0 framework. Then, state-of-the-art VPs design understanding methods are discussed. Finally, with a motivating example, we show the necessity of needing a strong design understanding approach at the ESL.

A. SystemC TLM-2.0

SystemC is a C++ based system-level design language providing an event-driven simulation kernel. TLM-2.0 framework (as the current standard) introduces the transaction concept allowing designers to describe a model in terms of abstract

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	-
	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
AT	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

communication 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). Communication between two TLM modules in a VP can be performed based on two-timing models, *Loosely-timed* (LT), and *Approximately-timed* (AT). The LT model is appropriate for the use case of software development while the AT model for architectural exploration and performance analysis. The LT model is implemented using the blocking transport interface (*b_transport*) allowing only two-timing points (request and response) to be associated with each transaction. 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. Related Work

Analyzing SystemC-based VPs for design understanding goal is an active field of research. Several methods have been developed to achieve this goal, each of them with its features and issues. These methods can be divided into two main categories based on whether they rely on static or hybrid techniques.

Static approaches rely on extracting information from the source code or its compiled binary model using parsers [5]–[10] or existing C++ front-ends [11]–[13]. 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).

Hybrid approaches [14]–[19] use the best features of the static and dynamic methods to extract the designs' structure. The extracted structural information is linked using a post-process analysis. To monitor the VP model's behavior, they usually utilize the extracted structural information of the VP to access its run-time behavior. Current hybrid methods that can analyze SystemC VPs' behavior have been developed in different ways such as manually modifying the source code, altering the existing SystemC library, interfaces, kernel or compiler.

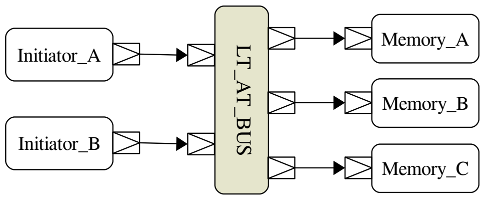

Fig. 1: Architecture of the *LT_AT_BUS* VP.

Overall, existing design understanding solutions mostly suffer from the following drawbacks which are in short 1) low degree of automation [16], 2) limiting language constructs [9]–[11], 3) lack of precise VPs' behavior extraction [10], [14], [15], [17], [18], and 4) lack of supporting TLM constructs [14], [15], [18].

C. Motivating Example

Consider the third-party *LT_AT_BUS* VP (inspired by [20]) shown in Fig. 1 that the documentation is not available (or poorly written). The VP includes six modules that differentiate based on underlying base protocol transactions: two initiators (*Initiator_A*, *Initiator_B*), one interconnect (*LT_AT_BUS*), and three targets (*Memory_A*, *Memory_B*, and *Memory_C*). The Initiator_A module communicates with target modules through *LT_AT_BUS* by generating four types of AT transactions. Two types T_0 and T_4 to access *Memory_A* (each type for different memory address ranges), and types T_1 and T_2 to access *Memory_B* and *Memory_C*, respectively. The *Initiator_B* module generates transactions of type T_3 to communicate with all target modules. For example, consider the communication between *Initiator_A* and *Memory_A* (the gray components in Fig. 1). The *Initiator_A* module generates transaction types T_0 and T_4 to access memory address range (0x00 to 0x0A) and (0x0B to 0xFF) of the *Memory_A* module based on the functions call and timing phases described in Table I, respectively. Now consider a scenario that may happen during the design process. Designers decide to reuse or revise some modules of the VP. For example, they want to modify the AT base protocol transaction type T_4 of the *Initiator_A* module and change it to T_1, including fewer transition phases to gain performance. This modification also needs to be applied to the *LT_AT_BUS* and *Memory_A* to properly build a communication path between the initiator and target modules through the interconnect. Before any changes can be applied to the VP, it needs to be adequately understood. However, lacking (proper) documentation makes the understanding process very complicated.

III. Design Understanding Methodology

In this section, we explain the three phases of our proposed methodology in detail.

A. Data Extraction

The first phase of the proposed design understanding methodology is to access both the static and run-time information of a given VP describing its *structure* and *behavior* (which is defined in terms of transaction). The structure of a given VP refers to the data that is described in the VP's source code

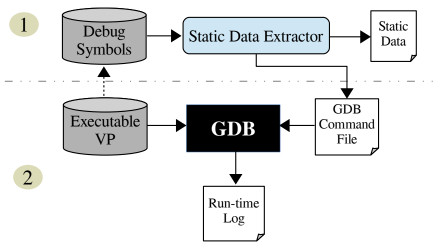

Fig. 2: Overview of the Debugger-based approach.

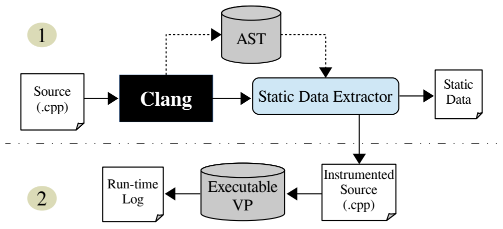

Fig. 3: Overview of the Compiler-based approach.

consisting of the modules' types (initiator, interconnect or target), binding information of modules' sockets, and transaction objects and their related variables such as timing annotation, phase and return status of transporting interfaces.

A large part of this information is static data, which can be accessed before the execution of VPs (e.g., in compile-time) including e.g., the root name and type of each module, and the name and type of each function. However, some parts of this information may be identified after the VP's execution, which is considered as dynamic data (e.g., dynamic variables and parameters).

The VPs' behavior refers to the run-time information of abstract communication among different IP cores. As communication is performed using the transaction concept, describing the TLM VP's behavior is connected to identify the behavior of its transactions. This identification requires to access three essential elements of transactions during the execution time, which are *flow*, *data*, and *type*.

The transaction's flow represents the order of TLM modules taking part in the transaction's lifetime (i.e., the period between transaction construction and destruction). For example, the transaction's flow of type T_1 generated by *Initiator_A* to access data in *Memory_B* is based on the sequence order (1)→(2)→(3)→(2)→(1) where (1), (2) and (3) are *Initiator_A*, *LT_AT_BUS*, and *Memory_B*, respectively. The information that must be extracted to describe the transaction's flow properly is

- the sequence number of objects' activation,
- the root and instance name, and the role of each module taking part in the transaction lifetime,
- the name of the current function, its arguments' values and its return value (if available),
- the simulation time, and
- the transaction reference address.

The transaction data denotes the transaction's attributes such as data value, address, command, data length, and response status. The transaction's type refers to the transaction's timing model (LT or AT). In the case of the AT model, it also requires to be specified which type of base protocol transactions is used.

1) Debugger-based approach: As illustrated in Fig. 2, the core idea of the proposed approach to extract the structure and simulation behavior of a given SystemC-based VP consists of two main phases. First, the static information of the compiled model is retrieved by analyzing its debug symbols to achieve two goals:

- identifying all components and their attributes and member functions which are required to describe the structure of the model, and
- 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.

Second, the model is executed under the control of 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. Due to the lack of space, we refer the reader to [21], [22] for the details of the approach.

2) Compiler-based approach: Fig. 3 provides an overview of the proposed approach consisting of two main phases. First, analyzing the *Abstract Syntax Tree* (AST) of a given VP to extract the static information of the model which is required to describe the VP's structure. Second, generating an instrumented version of the VP's source code using the extracted static information from the previous phase to retrieve the run-time information (i.e., behavior). This is performed by automatically compiling the instrumented source code with a standard C++ compiler (e.g., GCC or Clang) and executing it to log the run-time information. We refer the reader to [23] for the details of the approach.

B. Analysis

The generated run-time log file from the previous phase contains unordered information about the VPs' behavior. As the information is stored in the order of execution, transactions overlap in this log file (as process or function calls related to a particular task may be executed at different points in time). To present the behavior of each particular transaction (or value of a variable), this large set of data must be separated into sets that each refers to a single transaction (or variable). Thus, the goal of this section is to propose a post-execution analysis of the extracted run-time data to generate an intermediate representation of the VP's simulation behavior. This translation is performed as a transaction lifetime and access path.

1) Transaction Lifetime: The first step of this analysis is to describe each extracted transaction based on its flow, data, and type within its lifetime. This requires to isolate for every single transaction its corresponding information from other transactions. In order to trace a single transaction in the *Run-time log* file, transactions are separated based on some unique elements. The key element for this isolation is the transaction reference address. We take advantage of the TLM-2.0 rule stated in [3] – 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

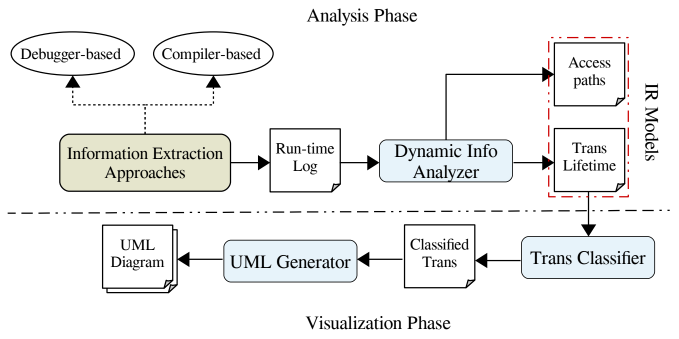

Fig. 4: Overview of the analysis and visualization phases.

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). Moreover, 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 endpoint of the transaction.

As demonstrated in Fig. 4, the *Dynamic Info Analyzer* module receives the *Run-time log* as an input. It extracts the information of every single transaction to describe the transaction's lifetime. The transaction's lifetime is stored in the *Trans lifetime* file. This information is an accurate trace of each transaction's behavior, covering all changes in transaction data that occurred during the execution of the model. Fig. 5 shows a part of the generated *Trans lifetime* of the *LT_AT_BUS* VP. It includes five sequences (timing steps), illustrating the transaction creation, manipulation by TLM modules and its completion.

2) Transaction Access Path: The generated transaction lifetime TL in the previous step provides designers with detailed information about the VPs' behavior. In this section, we perform further analysis on the extracted transactions to provide an abstract representation of the whole VPs' behavior called *Access Path* (AP). This empowers designers to know the relation of different IP cores in a given system in a big-picture view. A complete simulation behavior of a given VP can be defined as a set of access paths S_{AP} where each path AP shows a connection between an initiator module IM and a target module TM as below:

$$S_{AP} = \{AP_i \mid AP_i = \{IM \rightarrow TM :: (TID, TT, Tadrs, cmd, TD)\}, \quad 1 \leq i \leq n_{seq}\} \tag{1}$$

where IM and TM are initiator and target modules, respectively. TT is the transaction type illustrating which timing model (LT or AT) is used. $Tadrs$ shows the address of the transaction in the target module TM. cmd is the transaction command attribute (e.g., read or write). TD is the total delay of all sequences within the transaction lifetime. Finally, n_{seq} is the number of sequence in a transaction lifetime.

For example, the access path representation of the transaction lifetime of the *LT-AT_BUS* VP (in Fig. 5) based on (1) is as below.

$$AP = \{Initiator\_A : initA \rightarrow Memory\_B : trgB :: (0x76ab561, T_1, 0x04, READ, 20\ ns)\} \tag{2}$$

```
SQ1: ([Initiator_A, initA, process_A, 40ns, 0x6e47f0, initiator],
        [0x76ab561, 0x09, READ, 4, TLM_INCOMPLETE_RESPONSE, BEGIN_REQ, 5ns, NULL])

SQ2: ([LT_AT_BUS, bus_0, nb_transport_fw, 45ns, 0x6e47f0, interconnect],
        [0x76ab561, 0x04, READ, 4, TLM_INCOMPLETE_RESPONSE, BEGIN_REQ, 5ns, NULL])

SQ3: ([Memory_B, trgB, nb_transport_fw, 50ns, 0x6e47f0, target],
        [0x76ab561, 0x04, READ, 4, TLM_OK_RESPONSE, BEGIN_REQ, 5ns, TLM_COMPLETED])

SQ4: ([LT_AT_BUS, bus_0, nb_transport_fw, 55ns, 0x6e47f0, interconnect],
        [0x76ab561, 0x04, READ, 4, TLM_OK_RESPONSE, BEGIN_REQ, 5ns, TLM_COMPLETED])

SQ5: ([Initiator_A, initA, process_A, 60ns, 0x6e47f0, initiator],
        [0x76ab561, 0x09, READ, 4, TLM_OK_RESPONSE, BEGIN_REQ, 5ns, NULL])
```

Fig. 5: A single transaction lifetime of the *LT-AT_BUS* VP.

It shows that the instance *initA* of initiator module *Initiator_A* created a transaction with reference address $0x6e47f0$ to read from memory address $0x04$ of the instance *trgB* of target module *Memory_B*. 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 $40\ ns$ and $60\ ns$, respectively.

C. Visualization

After analyzing the extracted run-time information and generating a set of IR models of the VPs' behavior, the next step is to visualize this information in such a way that helps designers in understanding the VPs' intricacies.

1) Transaction Classification: Since it is possible that many of the extracted transactions in *Trans lifetime* have the same flow and type (only their data is different), a further analysis step is required to only visualize those which present a unique behavior. This effectively reduces the number of generated UML diagrams, allowing designers to quickly understand the behavior of a given VP-based embedded system.

The transactions' classification is performed in two levels as the following.

- First, based on the transactions' flow, providing designers with an abstract view of IPs communication in the VP. This is done by distinguishing transactions based on different communication patterns.
- Second, based on transactions' type providing designers with an accurate analysis of the transactions' type that different TLM modules used to communicate.

For example, Table. II shows the transaction classification results of the *LT_AT_BUS* VP. It illustrates that the VP has six different communication flows, four different transaction types and overall seven unique patterns of flow and type.

2) VP's Structure: The preferred format to present the VPs' structure depends on the purpose of designers or the back-end tools which may use the results for further analysis. As we want to have a generic presentation of this information, the extracted information from the model is stored in an XML document. The root element of the generated XML document

TABLE II: Transaction classification of the *LT_AT_BUS* VP.

Number	Flow	Modules	Transaction Type
1	F_1	Initiator_A→LT_AT_BUS→Memory_A	T_0
2	F_1	Initiator_A→LT_AT_BUS→Memory_A	T_4
3	F_2	Initiator_A→LT_AT_BUS→Memory_B	T_1
4	F_3	Initiator_A→LT_AT_BUS→Memory_C	T_2
5	F_4	Initiator_B→LT_AT_BUS→Memory_A	T_3
6	F_5	Initiator_B→LT_AT_BUS→Memory_B	T_3
7	F_6	Initiator_B→LT_AT_BUS→Memory_C	T_3

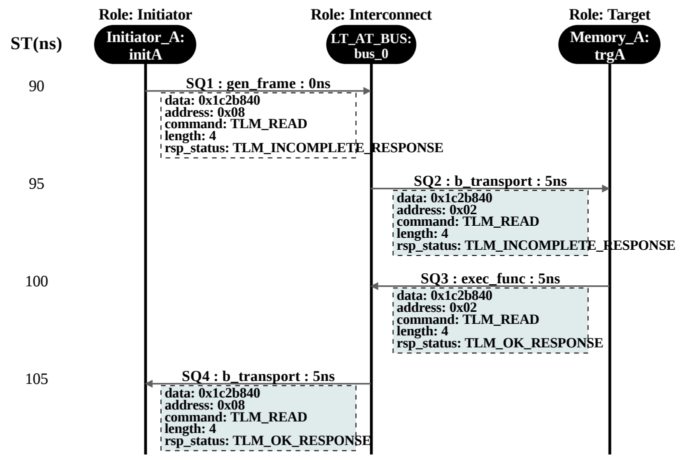

Fig. 6: UML diagram of a single LT transaction.

is the name of the VP model. The structure of the VP 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.

3) VP's Behavior: To reduce the complexity of understanding the extracted information related to the VP's behavior, the UML diagram is generated by the *UML Generator* module (Fig. 4) for a transaction that has a unique flow and type. The generated UML diagram is a message sequence chart introduced by the OSCI TLM-2.0 reference manual in [3] but it provides more detailed information. It describes the transaction flow among modules' communication for every 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 diagram includes a set of sequence numbers indicating both transaction flow and transaction data within its lifetime.

Consider the *LT_AT_BUS* VP, due to the classification analysis illustrated in Table II, the VP includes seven unique transactions (flow and type). Thus, seven UML diagrams are generated to present the simulation behavior of the VP. For example, Fig.6 illustrates the UML diagram of an extracted transaction of type T_0 generated by the initiator module *Initiator_A*. The black shapes present the root and instance name of modules within the design. The role (type) of each module is shown on top of the module's name. The information on each arrow demonstrates the interaction between two modules that are drawn from the caller to the callee w.r.t the simulation time. In particular, for a call from an instance of a TLM module, it presents the number of the sequence, the name of the caller function and timing annotation. Moreover, the generated UML model consists of the detailed transaction data. The box under each arrow shows the transaction's attributes. The white boxes illustrate a local transaction object while the blue boxes show a transaction object reached the callee through a function call.

IV. Experimental Results

The proposed methodology was applied to several standard VPs provided by Doulos [20] and to the real-world LEON3-based VP SoCRocket [24]. Table III shows the extracted data including number of transactions (*#Trans*), unique flows (*#UFlow*), unique types (*#UType*) and number of generated UML diagrams (*#UML*). The amount of extracted data for both debugger-based and compiler-based approaches is the same, meaning both approaches have the same accuracy. Table IV illustrates the required analysis time for both approaches in detail and compares it with the pure execution and compilation time (column *CET*) of each VP. While the execution time for all phases of the compiler-based approach is in the same boundary with CET, the required execution time for debugger-based approach might be large when the VP complexity increases. The major time-consuming part of the debugger-based approach is the second phase where the program is executed under control of GDB and the execution has to be halted repeatedly to trace transactions during simulation time. The difference of the analysis and visualization time (column *VAT*) between both approaches comes from the different amount of information that is extracted by each of them in the first phase and stored in the *Run-time Log* file. As the debugger-based approach extracts information by the precision of instruction execution, the generated *Run-time Log* file is larger than once generated by the compiler-based approach. Hence, it needs a larger analysis time.

The debugger-based approach requires only the executable version of the VP, thus the original source code and workflow (e.g., SystemC library or compiler) stay untouched. The main problem with intrusive approaches that rely on altering e.g., the SystemC library, interfaces, simulation kernel, or compiler is that these modifications may cause an issue for the application of several approaches in parallel, future updates or restrictive environments. Moreover, they mostly reduce the degree of automation as they require manual effort by designers. In the case of third-party IPs or legacy models where the source code may not be available at all, this approach is the only applicable solution. On the other hand, the compiler-based approach is very fast and scales well with an arbitrary complexity of VPs. However, it requires the availability of the VP's original source code. Since the proposed approach modifies neither the SystemC library nor the SystemC simulation kernel nor compiler, any results obtained using the approach are identical to the reference results in terms of VP's timing behavior and its functionality.

Overall, due to the designer's concerns and requirements, they have this option to choose either the debugger-based or the compiler-based approach. The retrieved information (IR models) using both proposed approaches can be effectively utilized for ESL designs space exploration [25]–[27], verification [28]–[30] and security validation [31]. This also pays off other advantages of the proposed design understanding methodology and proof of its usefulness in enhancing the aforementioned tasks in the design process.

V. Conclusion

In this paper, we presented a comprehensive automated design understanding methodology for SystemC-based VPs at the ESL from two perspectives: debugger-based and compiler-based approaches. The debugger-based approach provides designers with a non-intrusive analysis solution that only requires the executable model of VPs. The compiler-based approach

TABLE III: Experimental results related to the amount of extracted information for all SystemC TLM-2.0 VPs

VP Model	LoC	#Comps	TM	#Trans	Compiler-based Approach			Debugger-based Approach		
					#UFlow	#UType	#UML	#UFlow	#UType	#UML
LT-example[1]	175	2	LT	160	1	1	1	1	1	1
Routing-model[1]	456	6	LT	100	4	1	4	4	1	4
Example-4[1]	547	2	AT	348	1	4	4	1	4	4
Example-5[1]	650	7	LT	69	7	1	7	7	1	7
Example-6[1]	713	9	AT	245	16	2	16	16	2	16
AT-example[1]	3,410	19	AT	49	12	9	14	12	9	14
Locking-two[1]	4,690	23	LT/AT	371	14	10	16	14	10	16
SoCRocket[2]	50,000	20	LT/AT	1,000	19	8	21	19	8	21

1 provided by [20] **2** provided by [24] **LoC**: Lines of Code **TM**: Timing Model **#Trans**: number of extracted Transaction **#UFlow**: Unique transaction Flow **#UType**: Unique transaction Type **#UML**: number of generated UML diagram

TABLE IV: Experimental results related to the required analysis time for all SystemC TLM-2.0 VPs

VP Model	LoC	#Comps	TM	#Trans	Compiler-based Approach (s)				Debugger-based Approach (s)				CET (s)		
					Phase1	Phase2	AVT	Total	Phase1	Phase2	AVT	Total	Cmp	Exe	Total
LT-example[1]	175	2	LT	160	1.12	0.11	0.39	1.62	1.93	61.18	1.66	64.77	1.02	0.10	1.12
Routing-model[1]	456	6	LT	100	2.11	0.12	0.31	2.54	2.76	107.29	2.81	112.86	1.42	0.01	1.53
Example-4[1]	547	2	AT	348	2.17	0.21	0.52	2.90	5.81	1,761.08	16.63	1,783.52	1.33	0.18	1.51
Example-5[1]	650	7	LT	69	3.21	0.10	0.19	3.50	6.11	778.92	2.69	787.72	2.01	0.09	2.09
Example-6[1]	713	9	AT	245	4.79	0.41	0.63	5.83	7.59	1,013.72	11.09	1,032.4	2.02	0.33	2.55
AT-example[1]	3,410	19	AT	49	19.05	0.24	0.39	19.68	22.41	791.07	4.96	818.44	20.03	0.34	17.19
Locking-two[1]	4,690	23	LT/AT	371	25.62	0.79	0.76	27.17	29.08	1,639.83	17.24	1,686.15	22.32	0.66	22.98
SoCRocket[2]	50,000	20	LT/AT	1,000	52.82	1.21	1.63	55.66	146.39	7,446.19	29.12	7,621.70	26.72	1.08	27.80

1 provided by [20] **2** provided by [24] **LoC**: Lines of Code **TM**: Timing Model **#Trans**: number of Transaction **AVT**: Analysis and Visualization Time **CET**: Compilation and Execution Time **Cmp**: Compilation Time **Exe**: Execution Time

introduces a fast analysis technique that scales very well with an arbitrary complexity of VPs or their running software. The proposed methodology is efficient, automated and significantly advance the current state-of-the-art of the VP analysis at the ESL and can be used in conjunction with an existing framework or any SystemC setup.

Acknowledgment

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SATiSFy under contract no. 16KIS0821K, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

References

[1] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[2] "IEEE Standard SystemC Language Reference Manual," *IEEE Std 1666-2005*, pp. 1–423, 2006.

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

[4] M. Goli and R. Drechsler, *Automated Analysis of Virtual Prototypes at the Electronic System Level: Design Understanding and Applications*. Springer International Publishing, 2020.

[5] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler, "ParSyC: An efficient SystemC parser," in *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, 2004, pp. 148–154.

[6] F. Karlsruhe, "Kascpar - karlsruhe SystemC parser suite," 2012.

[7] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient automatic visualization of SystemC designs." in *Forum on specification and Design Languages (FDL)*, 2003, pp. 646–658.

[8] H. D. Patel, D. A. Mathaikutty, D. Berner, and S. K. Shukla, "Systemcxml: An extensible SystemC front end using XML," Tech. Rep., 2005.

[9] W. Snyder, "SystemPerl homepage," http://www.veripool.com/systemperl.html, accessed: 2016-01-30.

[10] T. Schmidt, G. Liu, and R. Dömer, "Automatic generation of thread communication graphs from SystemC source code," in *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2016, pp. 108–115.

[11] 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)*. Springer, 2008, pp. 467–470.

[12] D. Berner, J. piere Talpin, H. Patel, D. A. Mathaikuty, and E. Shukla, "SystemCXML: An extensible SystemC front end using XML," in *Forum on specification and Design Languages (FDL)*, 2005, pp. 405–409.

[13] A. Kaushik and H. D. Patel, "SystemC-clang: An open-source framework for analyzing mixed-abstraction SystemC models," in *Forum on specification and Design Languages (FDL)*, 2013, pp. 1–8.

[14] C. Genz and R. Drechsler, "Overcoming limitations of the SystemC data introspection," in *Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2009, pp. 590–593.

[15] H. Broeders and R. Van Leuken, "Extracting behavior and dynamically generated hierarchy from SystemC models," in *Design Automation Conference (DAC)*. ACM, 2011, pp. 357–362.

[16] W. Klingauf and M. Geffken, "Design structure analysis and transaction recording in systemc designs: A minimal-intrusive approach," in *Forum on specification and Design Languages (FDL)*, 2006.

[17] J. Stoppe, R. Wille, and R. Drechsler, "Data extraction from SystemC designs using debug symbols and the SystemC API," in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI)*. IEEE, 2013, pp. 26–31.

[18] K. Marquet and M. Moy, "Pinavm: a SystemC front-end based on an executable intermediate representation," in *Embedded software (EMSOFT)*. ACM, 2010, pp. 79–88.

[19] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: An extraction tool for SystemC descriptions of systems-on-a-chip," in *Embedded Software (EMSOFT)*. New York, NY, USA: ACM, 2005, pp. 317–324.

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

[21] M. Goli, J. Stoppe, and R. Drechsler, "AIBA: an Automated Intra-cycle Behavioral Analysis for SystemC-based design exploration," in *IEEE International Conference on Computer Design (ICCD)*, 2016, pp. 360–363.

[22] M. Goli, J. Stoppe, and R. Drechsler, "Automated nonintrusive analysis of electronic system level designs," *IEEE Trans. on CAD of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 2, pp. 492–505, 2020.

[23] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Automated analysis of virtual prototypes at electronic system level," in *Great Lakes Symposium on VLSI (GLSVLSI)*, 2019, pp. 307–310.

[24] 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, available at http://github.com/socrocket, pp. 1–7.

[25] M. Goli, J. Stoppe, and R. Drechsler, "Resilience evaluation for approximating SystemC designs using machine learning techniques," in *International Symposium on Rapid System Prototyping (RSP)*, 2018, pp. 97–103.

[26] D. Lemma, M. Goli, D. Große, and R. Drechsler, "Power intent from initial ESL prototypes: Extracting power management parameters," in *IEEE Nordic Circuits and Systems Conference (NORCAS)*, 2018, pp. 1–6.

[27] M. Goli and R. Drechsler, "PREASC: Automatic portion resilience evaluation for approximating SystemC-Based designs using regression analysis techniques," *ACM Trans. Des. Autom. Electron. Syst. (TODAES)*, Accepted 2020 (doi: 10.1145/3388140).

[28] M. Goli, J. Stoppe, and R. Drechsler, "Automatic equivalence checking for SystemC-TLM 2.0 models against their formal specifications," in *Design, Automation and Test in Europe (DATE)*, 2017, pp. 630–633.

[29] M. Goli, J. Stoppe, and R. Drechsler, "Automatic protocol compliance checking of SystemC TLM-2.0 simulation behavior using timed automata," in *IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 377–384.

[30] M. Goli and R. Drechsler, "Scalable simulation-based verification of SystemC-based virtual prototypes," in *EUROMICRO Digital System Design Conference (DSD)*, 2019, pp. 522–529.

[31] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Security validation of VP-based SoCs using dynamic information flow tracking," *it - Information Technology*, vol. 61, no. 1, pp. 45–58, 2019.