Automatic Protocol Compliance Checking of SystemC TLM-2.0 Simulation Behavior Using Timed Automata

Mehran Goli[1] Jannis Stoppe[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{mehran jstoppe drechsle}@informatik.uni-bremen.de

***Abstract*—The increasing complexity of todays digital circuit designs led to the increased usage of abstract models. In particular, the *Electronic System Level* (ESL) has emerged as an area of active research. For ESL design, SystemC and its *Transaction Level Modeling* (TLM) framework have become the standard tools for abstract modeling. The resulting models represent both, an executable specification and a reference model for the hardware design. The correctness of these designs is important as undetected errors may propagate to less abstract levels in the design process, increasing the potential amount of work required to fix them.**

To quickly ensure that implementations and reference emit the same behavior, the comparison between these abstractions needs to be both, flexible and automated. This paper presents a method to verify the simulation behavior of a given System TLM-2.0 design against TLM-2.0 protocols. The system's structural description and its run-time behavior are translated to a single, consistent formal model. This is then used to verify that a simulation run adheres to a given protocol. The protocol compliance checks are performed using the UPPAAL model checker and applied to several TLM models.

I. Introduction

One possible solution to cope with the ever increasing complexity of electronic circuits (and their design) is designing them on higher levels of abstraction, e.g. via the *Electronic System Level* (ESL). In ESL, the C++-based, standardized *SystemC* [1] library has become the de-facto standard [20] modeling language.

One important advantage of SystemC is the support of different levels of abstraction via *Transaction-Level Modeling* (TLM). The TLM-2.0 standard is widely used for early system validation by supplying reference models for less abstract levels of the design process (such as Register Transfer Level (RTL) implementations). It allows designers to describe a model in terms of abstract communications using a set of rules (the base protocol) and standard interfaces. Ensuring the validity of these abstract TLM models is vital as the cost of fixing undetected protocol violations are increasingly expensive in later design steps.

The TLM-2.0 standard comes with more than 150 rules that must be adhered to when a TLM model is implemented [5] and which define the expected behavior of a TLM-2.0 compliant model. A small part of these rules specify limitations concerning the structure of TLM models (e.g. an initiator socket must be connected to a target socket). This set is called the *static rules* and is checked during compilation. A larger part of these rules (the *dynamic rules*) determines restrictions on the TLM communication and transaction attributes – and as such define a set of protocols TLM designs should adhere to.

Neither the SystemC compiler nor the TLM library detect TLM protocol violations that occur during execution. Manually verifying all rules and detecting the source of any given error is error-prone and expensive even for simple models and thus practically impossible for complex designs. Therefore, automated verification techniques that verify the compliance of a given ESL model with at least the base protocol are needed.

Generally, SystemC verification has been studied in two scopes: formal verification and simulation-based verification. Formal methods refer to model checking techniques [6]. They require the designers to specify the model in formal semantics (e.g. abstract state machines). As SystemC lacks formal semantics [25], translating SystemC models to a formal language is not a trivial task. It restricts formal methods to verify a limited range of SystemC models as several assumptions need to be imposed on a given input design (as e.g. function pointers, recursion or templates cannot easily be described formally). Moreover, state space explosion is another well-known technical restriction of model checking, preventing it to verify complex SystemC models. In simulation based verification, the behavior of SystemC models is verified during simulation. In order to trace transactions during simulation time, they usually rely on intrusive techniques that either manipulate the original source code (by inserting additional modules to the original model) [7], [4], the SystemC kernel [23] or the SystemC library [21]. Intrusive techniques either rely on expensive manual processes or have compatibility issues that reduce the degree of automation.

This paper presents a hybrid automated approach to formally verify the simulation behavior of a given SystemC TLM-2.0 model against TLM-2.0's dynamic rules. The simulation behavior of the model is automatically extracted using the GNU debugger (GDB) [22] and transformed into a finite state machine both non-intrusively and without any restrictions concerning the original source code. This formal model is checked against most standardized TLM-2.0 rules by a model checker. To illustrate the effectiveness of the proposed methodology, an approximately-timed (AT) model, based on the TLM-2.0 standard, is used as an application example and several designs are tested with the given approach.

II. Related Work

As mentioned in the previous section, the approaches for verifying SystemC models are mostly divided in two main categories: formal and simulation based methods.

Formal Methods: Many approaches [15], [14], [16], [17] have been introduced to formally verify SystemC models. They rely on translating the models to formal semantics such as state machines and verify them by checking safety properties. In [15], SystemC models are transformed into the *Abstract State Machine Language* (AsmL) and properties are formulated using the *Property Specification Language* (PSL). The translated model is verified using a model checker. In [14], a SystemC-TLM design is translated to a sequential C model and the properties are defined using PSL. Then, a monitoring logic which is based on C assertions and finite state machines is utilized to verify the properties. In [16], [17], a transformation of a given SystemC design to a timed automata model is created. The translated model is then checked by model checkers such as UPPAAL and BLAST. The method proposed in [8] formalizes the semantics of SystemC designs in terms of Kripke structures and verifies it using symbolic model checking. The authors of [19] present an approach to translate SystemC models to an *Intermediate Verification Language* (IVL). Then, they provide a symbolic simulator to verify the IVL.

For all of the aforementioned methods, translation of SystemC designs to formal semantics is the main challenge confining them to verify only a sub-set of SystemC models.

Simulation-based Methods: Several works [18], [10], [24] have been proposed to verify SystemC models using *Aspect Oriented Programming* (AOP). The idea of AOP is based on a source-to-source translation enabling designers to add additional code (aspects) at specific points of the original source code. In [18] the AOP technique is utilized to instrument the source code of SystemC-TLM models to trace its simulation. The traced information is checked against the properties implemented as a C++ class. In [10] SystemC source code and properties written as aspect description in XML format by users are parsed. A weaver module is used to reassemble the parsed source code and aspects. Then, a SystemC formatter module is utilized to generate SystemC source code from the woven syntax tree to check the properties during the simulation time. In [24] user code primitives are defined in property specifications by users. Then, AOP is used to instrument the SystemC source code by generating a monitor for each property to be check during the execution.

The AOP-based methods mostly depend on user interaction to define the aspects and design primitives. Additionally, the difficulty of defining and debugging AOP setups, this renders the method rather hard to use.

There are also several attempts [21], [11], [9] to verify SystemC models by *Assertion-based Verification* (ABV). In ABV, properties are specified as assertions (written in languages like PSL or System Verilog) and checked during simulation time. However, the methods have several disadvantages. Property formulation requires complex specification – which is done manually by the user and is challenging to be applied during simulation time. Most require the design's original sources to be changed, making ABV as an intrusive solution.

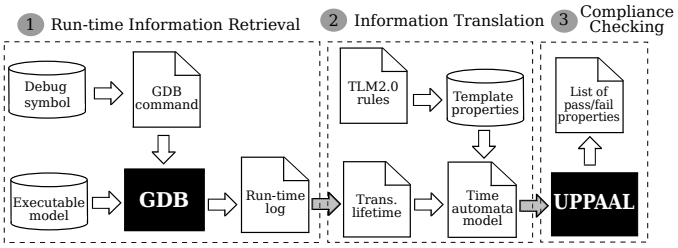

Fig. 1. The architecture of proposed methodology.

In [7], [4] SystemC-TLM models are verified by adding some TLM protocol checkers as an external SystemC module to monitor transactions. Transactions are monitored during simulation by inserting a copy of the protocol checker between every pair of TLM modules. They check whether or not each transaction satisfies the TLM protocols. Using these protocol checkers makes the method intrusive as the original source code is manipulated.

Recently, the idea of utilizing GDB to access the run-time data of SystemC-based models was introduced [13], [12]. While GDB is originally intended to provide programmers a tool to manually modify the execution behavior of a given program, it also provides an interface to control its behavior using predefined command files. In order to extract the information automatically, such a command file is generated automatically from the debug symbols that are present in the precompiled binary. The extracted information in [13] is used to functionally verify a SystemC design against its formal specification. However, the extracted information is limited to the order of function calls, disregarding the actual values that are accepted or returned. E.g., while it is able to check whether a multiplication was executed, it cannot verify that the result is indeed the product of the two parameters. Therefore, the method only verifies the sequence activity of a given SystemC TLM-2.0 model against its formal specification and does not verify it against the TLM-2.0 protocol. The process of the run-time information extraction is presented in Fig. 1 phase 1.

III. Proposed Methodology

As shown in Fig. 1, the proposed method is performed in three main phases:

1) extracting the simulation behavior,
2) transforming both the extracted information and TLM rules into formal semantics and
3) checking the transformed model of the simulation behavior against a set of properties of TLM rules.

A. Run-time Information Retrieval

The approach is to build upon the core idea of utilizing GDB as a readily available virtual execution environment for SystemC designs [13]. The unique traits of this method, allowing designers to leave their compilation workflow untouched, simply providing a compiled file to an application that does both, a static analysis and a run-time behavior extraction, makes it the most promising candidate for the required analysis.

```
transID_number :0x7054f0_1

seq1 -> (['AT_typeA_initiator::thread_process', 'initiator0', 0ns
,'NULL',initiator],['0x6bc660', '17764','tlm::TLM_READ_COMMAND'
,'4','tlm::TLM_INCOMPLETE_RESPONSE','BEGIN_REQ', '4565ps'])

seq2 -> (['AT_interconnect::nb_transport_fw', 'interconnect', 0ns
,'0x7054f0', 'interconnect'],['0x6bc660', '100', 'tlm::TLM_READ_COMMAND'
,'4','tlm::TLM_INCOMPLETE_RESPONSE', 'BEGIN_REQ', '4565ps', 'NULL'])

seq3 -> ([AT_typeE_target::nb_transport_fw, target4, 0ns
,'0x7054f0','target'],['0x6bc660', '100', 'tlm::TLM_READ_COMMAND'
,4,'tlm::TLM_OK_RESPONSE', 'BEGIN_REQ', '4565ps' ,'tlm::TLM_COMPLETED'])

seq4 -> ['AT_interconnect::nb_transport_fw', 'interconnect', 0ns
,'0x7054f0', 'interconnect'],['0x6bc660', '100', 'tlm::TLM_READ_COMMAND'
,'4','tlm::TLM_OK_RESPONSE', 'BEGIN_REQ', '4598ps', 'tlm::TLM_COMPLETED']

seq5 -> (['AT_typeA_initiator::thread_process', 'initiator0', 0ns
,'NULL', 'initiator'],['0x6bc660', '100', 'tlm::TLM_READ_COMMAND'
,'4','tlm::TLM_OK_RESPONSE', 'BEGIN_REQ', '4598ps'])
```

Fig. 2. A part of the *Trans lifetime* of the *AT-example*.

Note that the retrieved information must additionally include detailed data of the transactions' attributes and the related parameters (such as transition phase or timing annotations) to describe their lifetime (which consists of an initial and a final state as well as transitions – which are invoked by method calls – and intermediate states that connect the former two). This information is required to formally model a transaction lifetime and verify it against the TLM-2.0 protocol. Therefore, the instructions in *GDB command* are generated specifically with regard to this case, thus altering the execution to extract the desired information about TLM transactions.

The method is extended to extract

- the role of each module taking part in the transaction. It is extracted by analyzing the type of socket which is defined for each TLM module. Only initiator socket(s) (initiator module), only target socket(s) (target module) or both initiator and target socket(s) (interconnect module) and
- the name of executed functions, their arguments' values (e.g. transition phase, timing annotation) and return values.

In order to retrieve the transaction attributes, the GDB instructions are tailored to extract the attributes of a transaction object. These are

- data value,
- address,
- command,
- data length,
- byte enable length,
- streaming width,
- DMI allowed and
- response status.

The extracted information is stored in a log file called *run-time log*.

B. Information Translation

In order to formally verify the extracted simulation behavior of a given TLM model against TLM-2.0 rules, three steps are needed:

1) mapping pieces of information to transactions. As the information is stored in the order of execution, transactions overlap in this log file (as method calls related to a particular transaction may be carried out at different points in time). In order to verify each particular transaction against the given protocol, this large set of data must be separated into sets that each refer to a single given transaction.
2) transforming the transaction information into timed automata to be able to formally verify it using a model checker and
3) transforming the TLM-2.0 rules into formal properties that can be used to verify the generated timed automata.

1) Mapping Pieces of Information to Transactions: The run-time log file contains unordered information about the transactions' data and flow. The former refers to a transaction's attributes and the related parameters to describe its lifetime such as transition phases, timing annotation and return values of communication interfaces. The latter refers to the order of TLM modules taking part in the transaction's lifetime. To verify the simulation behavior of the model, each transaction needs to be checked against the TLM rules. These state that 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 unique reference address (call by reference). This address can be used as a *transaction ID* which is the main key to trace the transaction in its lifetime. However, the IDs may be re-used for new transactions as soon as an old one is discarded (as the objects are not deleted from memory but instead are kept in a pool for performance reasons). Therefore, information about the role of the modules taking part in the transaction and the return value of the communication-interfaces functions are used to detect the start and end point of the transactions' usage to differentiate distinct transaction instances using the same ID. By using this information, the *run-time log* is translated to a structural format (called *Trans lifetime* in Fig. 1) where for each transaction, the information of data and flow are defined in relation to its lifetime.

2) Transforming the Transaction Information into Timed Automata: In order to formally verify the simulation behavior of TLM models, it is necessary to transform the retrieved information into a formal model. As the extracted transactions' lifetimes have finite steps to implement the TLM protocol and include timing information (e.g. timing phase and timing annotations), they can be transformed to a timed automata model. A timed automaton is a finite state machine controlled by clock variables.

For specifying a transaction lifetime into a timed automata model the following definitions are used.

Definition 1. A timed automaton TA is a tuple (L, l_0, C, A, E, I), where L is a set of locations , $l_0 \in L$ is the initial location l_0 , C is a set of clock variables, and A is a set of actions, $E \subseteq L \times A \times B\ (C) \times 2C \times L$ is a set of edges, where $B(C)$ denotes a set of clock constraints, and $I : L \to B(C)$ assigns invariants to locations. The transition $l \xrightarrow{(a,g,r)} l'$ is valid when $(l, a, g, r, l') \in E$.

Definition 2. The semantics of a TA is defined as a transition system $(S, s_0, \to)$, where $S \subseteq L \times R_{\geq 0}^{|C|}$ is a set of states $s_0 = (l_0, u_0)$ the initial state and $\to \subseteq S \times (R_{\geq 0} \cup A) \times S$ the transition relation. A clock valuation is a function $u : C \to R_{\geq 0}$ that maps a non-negative real value to each clock. A semantic step of a timed automaton to model the simulation behavior is defined as

$$(l, u) \xrightarrow{a} (l', u') \text{ iff } l \xrightarrow{(a,g,r)} l' \text{ such that } u \in g \wedge u' = [r \to 0]u \wedge u' \in I(l').$$

A transaction lifetime includes several timing steps which present the transaction creation and manipulation by TLM modules. In order to transform the transaction lifetime into a timed automaton, each step is defined as a state (location) and the set of data exchanged between modules is defined as the transition of states (edges). A timing step is defined as a location by the information of the module and the sequence number of the step. In this model, a clock variable is used to control the sequence of transitions.

A simple example illustrates how the extracted simulation behavior of a TLM model is transformed into a timed automata model: Fig. 2 illustrates a part of simulation behavior of the *AT-example* (gray components in Fig. 3) which is the lifetime of a single transaction. This case study is the running example presenting one of the 13 possible ways of modeling an approximate timed model of the TLM-2.0 based protocol. A transaction is created by the initiator module *AT_typeA_initiator* with phase *BEGIN_REQ* and passed through the interconnect module *AT_interconnect* to reach the target module *AT_typeE_target*. The target module returns *TLM_COMPLETED* when it receives the transaction.

The timed automaton *TA_0x7054f0_1* formally denotes the simulation behavior of Fig. 2 based on definition 1:

$L = \{l_0 : seq0\_AT\_typeA\_initiator,$
$l_1 : seq1\_AT\_interconnect,\ \ l_2 : seq2\_AT\_typeE\_target,$
$l_3 : seq3\_AT\_interconnect,\ \ l_4 : seq4\_AT\_typeA\_initiator\}$
$l_0 = seq0\_AT\_typeA\_initiator$
$C = \{clk\}$
$A = \phi$
$E = \{l0, l1),\ (l1, l2),\ (l2, l3),\ (l3, l4)\}$
$I : l_1 \rightarrow clk \leq 1,\ l_2 \rightarrow clk \leq 2,\ l_3 \rightarrow clk \leq 3,\ l4 \rightarrow clk \leq 4$

The clock variable clk is initialized to zero and then used in two clock conditions. First, the invariant $clk \leq$ *maxtime* indicates that the corresponding location must be left before clk becomes greater than *maxtime*, and the guard $clk ==$ *maxtime* enables the corresponding edge at *mintime*. $A = \phi$ denotes that all transitions between locations are internal transition. It means that a transition is taken if only the guard condition of the edge is satisfied.

The operational semantics of the timed automaton *TA_0x7054f0_1* is thus formulated (based on definition 2):

$$(l_0, clk \leq 0) \xrightarrow{T_0, clk==0} (l_1, clk \leq 1)$$
$$(l_1, clk \leq 1) \xrightarrow{T_1, clk==1} (l_2, clk \leq 2)$$
$$(l_2, clk \leq 2) \xrightarrow{T_2, clk==2} (l_3, clk \leq 3)$$
$$(l_3, clk \leq 3) \xrightarrow{T_3, clk==3} (l_4, clk \leq 4)$$

The parameter ${}^{3}_{i:0}T_i$ is defined as an assignment statement when a transition is taken from l_i to l_{i+1}. It sets the value of transaction attributes and the value of corresponding variables describing the transaction flow.

3) Transforming TLM Rules into Formal Properties: Like the simulation behavior of TLM models, the TLM-2.0 rules need to be formally expressed in a well-defined language in order to compare the former to the latter. Properties are specified using a subset of *Timed Computation Tree Logic* (TCTL) [2] and described in the language of temporal logic. The language includes state formulas and path formulas. State formulas are expressions that are checked for a state, while path formulas evaluate whether a given state formula is satisfied over paths by any reachable state.

As illustrated in Fig. 1 phase 2, TLM rules are transformed from the textbook specification written in the TLM documentation into a set of template properties. This process is done manually (once) to create a database of predefined temporal properties. TLM rules are defined formally using pre-defined symbols which are taken from the field of temporal logic using the following definitions.

Definition 3. If p and q are a property of states, then the temporal logic formula:

- **Exists eventually** p ($E <> p$) describes that there is a path that leads to a state in which p holds.
- **Exists globally** p ($E [\ ] p$) describes that there is a path in which p holds for all the states of the path.
- **Always globally** p ($A [\ ] p$) describes that p holds for all states of all paths.
- **Always eventually** p ($A <> p$) describes that all paths p hold for at least one state of the path.
- q **always leads to** p ($q \rightarrow p$) describes any path that starts with a state in which q holds later reaches a state in which p holds.

The *Template properties* database contains the properties to verify both the transaction semantics and the functionality of TLM communication. There are 40 of template properties, of which 15 are defined to verify the semantics of a transaction and 25 are used to check the compliance of the communications semantics against TLM-2.0 rules.

As an example of *transaction semantics* rules, the TLM rules related to the default value of a transaction address attribute is considered. The textbook specification of this rule in TLM-2.0 documentation is as follow.

"The default value of a transaction response status must be equal to "TLM_INCOMPLETE_RESPONSE".

Due to definition 3, the formal definition of this statement is as follow:

$$(l_i\ \&\&\ transact.tstatus\ ==\ 0) \rightarrow\ (l_{i+1}\ \&\&\ transact.tstatus\ ==\ 0) \text{ where } \{l_i, l_{i+1} \in\ E\},\ i = 0$$

As the temporal logic does not support enumerate types, the possible values of all TLM-2.0 enumeration types are denoted by integer values. In case of the transaction response status attribute $transact.tstatus$ the enumeration value *TLM_INCOMPLETE_RESPONSE* and *TLM_OK_RESPONSE* are defined by 0 and 1, respectively.

The TLM-2.0 *communication semantics* rules on the other hand define how the communication should be carried out. The textbook specification of this rule in TLM-2.0 documentation is as follow.

"A phase transition can only take place if the return-value of the non-blocking transport is TLM_UPDATED."

Based on definition 3, the formal presentation of this rule is as follow:

$$(l_i\ \&\&\ tlm\_retun\_stus\ !=\ 2\ \&\&\ transact.tphase\ ==\ 't') \rightarrow (l_{i+1}\ \&\&\ transact.tphase\ ==\ 't') \text{ where } \{l_i, l_{i+1} \in\ E\},\ i > 0$$

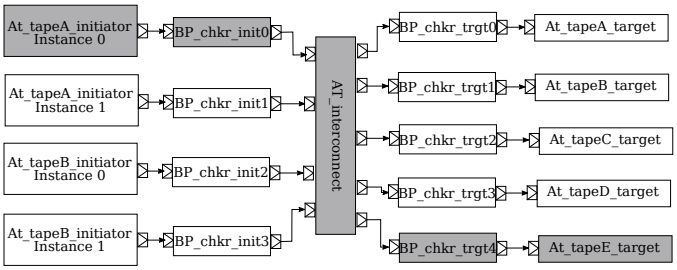

Fig. 3. The architecture of *AT-example*.

The tlm_retun_stus is an integer variable that denotes the return value of the non-blocking transport function. The enumeration value *NO_RETURN, TLM_ACCEPTED, TLM_UPDATED* and *TLM_COMPLETED* are specified by *0, 1, 2* and *3*, respectively. $transact.tphase$ refers to the transition phase of the TLM transaction (such as *BEGIN_REQ*).

C. TLM Protocol Compliance Checking

In order to verify the formal presentation of the simulation behavior of a given TLM design, UPPAAL [3], a model checker that supports the timed automata model, is used. In UPPAAL, a query is a property that may or may not hold for the system. The UPPAAL query language is a sub-set of TCTL, denoting properties using a temporal logic language.

To verify a transaction's behavior using UPPAAL,

1) each transaction lifetime is defined as a system,
2) the queries are automatically generated from the template properties stored in the *Template properties* database and
3) the generated properties are added to the *Timed automata model*.

The *Timed automata model* thus contains both, the formal presentation of the simulation behavior of TLM models and the required properties to check the compliance of TLM-2.0 rules. The model checker gets the *Timed automata model* as an input file and verifies the system. The results of this compliance check are reported by the model checker, including the satisfied properties and violated ones.

IV. Experimental Evaluation

The proposed method has been applied to several SystemC TLM-2.0 models provided by Doulos [4] that cover all TLM-2.0 core-interfaces (i.e transport interfaces and direct memory), the base protocol and coding styles (e.g. loosely-timed and approximately-timed). To demonstrate the benefits of the proposed method, the simulation behavior of the *AT-example* is altered to violate some TLM-2.0 rules.

The model checker UPPAAL 4.1 is used to verify the extracted timed automata of each case study. The analysis has been performed on a PC equipped with 8 GB RAM and an Intel core i7-2760QM CPU running at 2.4 GHz.

A. Case Study: AT-example

The *AT-example* design consists of multiple approximately-timed (AT) initiators and targets, as well as an AT interconnect. It includes nine TLM designs, implementing 9 of the 13 TLM-2.0 base protocol's specified transaction protocols. The unused transaction protocols are trivial cases such as transactions that are responded immediately after initiation.

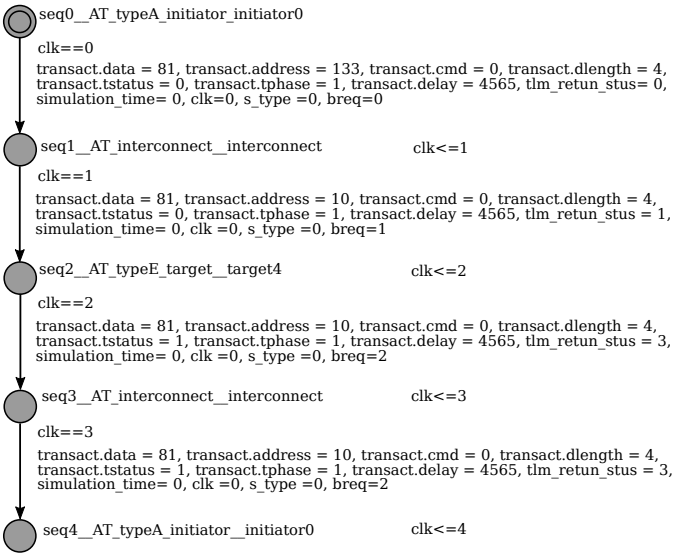

Fig. 4. The formal presentation of *AT-example* in UPPAAL timed automata format.

The traced transactions of each sub-design are automatically defined as a system in UPPAAL semantics. To model the lifetime of each transaction, it needs to be defined in terms of states (locations) and their transitions (edges).

For each transaction, the following steps are applied to map the transaction lifetime onto UPPAAL's timed automata with respect to definition 1 and 2:

- the name of each system is denoted by *TA_ID_number* showing a transaction with the ID and the number of reputation,
- each sequence in the transaction lifetime is mapped onto a location (which is specified by number of sequence, the root name of TLM module and its instance name),
- the transaction's attributes and its related parameters of each sequence are mapped onto assignments for each transition and
- the invariant clk is defined to control the sequence of transitions.

As an example, the first system in Table I shows the first transaction using ID *0x7054f0*. Fig. 4 illustrates the result of mapping the extracted transaction lifetime onto UPPAAL timed automata. The node with double circle specifies the initial state and each node is specified by a name and an invariant. The assignment statements on each edge express the value of transaction's attributes and all related parameters to describes its flow. The related parameters are the simulation time *simulation_time*, the return value of function call *tlm_retun_stus*, the type of socket *s_type* and flags to check the timing phase *breq, ereq, brsp* and *ersp*. The struct *transact* is defined to specify the value of transaction attributes (*data, address, cmd, dlength, tstatus*), the phase of the transition (*tphase*) and its timing annotations (*delay*) using integer values. As UPPAAL only supports integer values, all enumeration types of TLM-2.0 are mapped onto integer values.

TABLE I
EXPERIMENTAL RESULT FOR AT-EXAMPLE

System	Modules	#TSeq	#Queries	Original Model			FaultA			FaultB			R/VMU (MB)
				#Satisfied	#violated	VT(ms)	#Satisfied	#violated	VT(ms)	#Satisfied	#violated	VT(ms)	
0x7054f0_1	iA-intc-tE	5	31	31	0	45	29	2	43	26	5	43	6.8/42.1
0x7054f0_2	iA-intc-tD	9	44	44	0	61	42	2	59	32	12	57	7.1/42.1
0x705840_1	iA-intc-tC	15	72	72	0	89	70	2	87	59	13	81	7.3/43.2
0x706160_1	iB-intc-tE	6	34	34	0	51	32	2	49	26	8	48	6.9/42.1
0x706160_2	iB-intc-tB	10	49	49	0	73	47	2	71	33	16	68	7.1/42.4
0x706160_3	iA-intc-tA	10	52	52	0	76	50	2	74	36	16	70	7.4/42.4
0x706a50_1	iB-intc-tA	19	105	105	0	173	103	2	171	72	33	165	7.8/43.2
0x7078f0_1	iB-intc-tD	9	44	44	0	69	42	2	67	27	17	64	7.1/42.4
0x708f40_1	iB-intc-tB	10	49	49	0	73	47	2	71	33	16	68	7.1/42.4

iA: initiator-typeA, **inct**: interconnect, **tE**: target-typeE **TSeq**: Transaction Sequence **VT**: Verification Time **R/VMU**: Resident/Virtual Memory Usage Peaks

TABLE II
EXPERIMENTAL RESULTS FOR ALL CASE STUDIES

	System	LOC	#Comps	#Trans	#UTrans	TM	#Seq	#Queries	#SatQ	#VioQ	R/VMU (MB)	VT (s)	ET (m:s)	TotalT (m:s)	CExeT (s)
Original Model	LT-example	175	2	16	1	LT	35	96	96	0	6.7/41.5	0.134	0:09	0:09.134	1.6
	Routing-model	456	6	2	1	LT	68	24	24	0	6.7/41.5	0.036	0:21	0:21.036	1.7
	Example-4	547	2	348	1	AT	14060	13703	13703	0	7.8/43.2	20.562	66:27	66:47.562	1.8
	Example-5	650	7	69	2	LT	1147	828	828	0	6.7/41.5	1.275	31:13	31:14.175	2.1
	Example-6	713	9	245	2	AT	14354	15267	15267	0	7.6/42.6	23.900	53:03	53:26.900	2.2
	AT-example	2942	19	9	9	AT	1008	480	480	0	7.8/43.2	0.710	7:41	7:41.710	21
	Locking-two	3831	23	371	10	LT/AT	16379	8765	8765	0	7.6/42.6	13.147	79:15	79:28.147	24.3
FaultA	LT-example	175	2	16	1	LT	35	96	64	32	6.7/41.5	0.134	0:09	0:09.134	1.6
	Routing-model	456	6	2	1	LT	68	24	20	4	6.7/41.5	0.036	0:21	0:21.036	1.7
	Example-4	547	2	348	1	AT	14060	13703	13007	696	7.8/43.2	18.110	66:27	66:45.110	1.8
	Example-5	650	7	69	2	LT	1147	828	690	138	6.7/41.5	1.015	31:13	31:14.015	2.1
	Example-6	713	9	245	2	AT	14354	15267	14777	490	7.6/42.6	21.511	53:03	53:24.511	2.2
	AT-example	1950	10	9	9	AT	611	480	462	18	7.8/43.2	0.692	4:22	4:22.692	19.7
	Locking-two	2907	13	371	10	LT/AT	10923	8765	8023	742	7.6/42.6	12.410	41:38	41:50.410	22
FaultB	LT-example	175	2	16	1	AT	35	96	64	32	6.7/41.5	0.124	0:09	0:09.124	1.6
	Routing-model	456	6	2	1	AT	68	24	20	4	6.7/41.5	0.034	0:21	0:21.034	1.7
	Example-4	547	2	348	1	LT	14060	13703	11615	2088	7.8/43.2	17.406	66:27	66:44.406	1.8
	Example-5	650	7	69	2	LT	1147	828	690	138	6.7/41.5	1.101	31:13	31:14.101	2.1
	Example-6	713	9	245	2	AT	14354	15267	11299	3968	7.6/42.6	20.109	53:03	53:23.109	2.2
	AT-example	1950	10	9	9	AT	611	480	345	135	7.8/43.2	0.653	4:22	4:22.653	19.7
	Locking-two	2907	13	371	10	LT/AT	10923	8765	5805	2960	7.6/42.6	11.764	41:38	41:49.764	22

LOC: lines of Code **UTrans**: Unique Base Protocol Transaction **TM**: Timing Model **SatQ**: Satisfied Queries **VioQ**: Violated Queries
R/VMU: Resident/Virtual Memory Usage Peaks **VT**: Verification Time **CExeT**: Compilation and Execution Time **ET**: Extraction Time

TABLE III
FAULT MODELS

Number	Type	TLM-2.0 Protocol Description	Fault Model
1	FaultA	Default value of transaction's response status shall be set to TLM_INCOMPLETE_RESPONSE	TLM_INCOMPLETE_RESPONSE $\downarrow$ $(change\ to)$ TLM_OK_RESPONSE
2	FaultA	Transaction data length must be greater than 0	data_length $\xrightarrow{(set\ to)}$ -4
3	FaultB	The data length attribute shall be set by the initiator, and shall not be overwritten by any interconnect component	transaction data_length modified by interconnect component
4	FaultB	The response status attribute shall not be modified by interconnect component	response status modified by interconnect component
5	FaultB	A target component can send END_REQ when it received BEGIN_REQ	target component send END_RESP after receiving BEGIN_REQ

In order to verify the *AT-example* design, several queries are automatically generated for each system using the proposed method. To evaluate the generated queries, two types of possible faults have been defined and injected to the original design. *FaultA* and *FaultB* which are related to transaction attributes and communication semantics respectively.

1) FaultA: transaction semantics: This fault changes the value of a transaction's attribute and the related parameters to describe its lifetime. It might change the right value either to the wrong value in acceptable range of the TLM-2.0 enumeration data type (i.e. predefined protocol value) or an out of range value. As illustrated in Table III first row, the default value of transaction response status attribute must be set to *TLM_INCOMPLETE_RESPONSED*. A transaction semantic fault might change it to *TLM_OK_RESPONSE* that is one of the TLM-2.0 predefined protocol values but may be invalid in the given context. Fault number 2 in Table III describes the instances of *FaultA* which change the value of the transaction attribute to a non-predefined protocol value. The query that detects these is: $A[\,](transact.dlength > 0)$ where $dlength$ specifies the data length attribute of the struct data type $transact$. It means that the data length attribute $dlength$ must contain a positive value for all states.

2) FaultB: communication semantics: This fault refers to the modifiability of transaction attributes by different types of modules, base protocol rules concerning TLM-2.0 core-interfaces, phase sequences and timing annotations. As shown in Table III, the data length attribute shall be set by the initiator module and shall not be overwritten by the interconnect module – fault number 3 refers to a violation of this rule. Fault number 4 is a similar case, where an interconnect module modifies the $response\_status$ value. Fault number 5 refers to a case that corrupts the TLM-2.0 core interfaces and the transaction phase sequences by using the wrong order of sent requests. As an example the following query is defined to detect *FaultB-4*:

$$(system\_0x7054f0\_1.seq1\_\_AT\_interconnect\_\_interconnect \;\&\&\; transact.tstatus == 0) \;\rightarrow\; (system\_0x7054f0\_1.seq2\_\_AT\_typeE\_target\_\_target4 \;\&\&\; transact.tstatus == 0)$$

The query means that for the system $system\_0x7054f0\_1$ in state $seq1\_\_AT\_interconnect\_\_interconnect$ where the status value $tstatus$ of the transaction struct $transact$ is equal zero, the transition to the next state $seq2\_\_AT\_typeE\_target\_\_target4$ should leave the value of $transact.tstatus$ unchanged.

Table I demonstrates the experimental results of verifying the *AT-example* using the proposed method. The *System* column presents the name of each transaction of the *AT-example* which is defined as a system in UPPAAL. The *Modules* column demonstrates which type of modules belong to each sub-design of the *AT-example*. The *TSeq* column shows the number of sequences related to each transaction lifetime. The *Queries* column presents the number of queries generated by the proposed method to verify each system. The generated queries have been applied to three copies of the *AT-example* design. Three columns *Original Model*, *FaultA* and *FaultB* show the result of applying the proposed verification method to the *AT-example* without any manipulation and with injected fault types *FaultA* and *FaultB* respectively. For each of these columns, the *Satisfied*, *Violated* and *VT* sub-columns illustrate the number of satisfied queries, violated queries and the time to verify all queries by UPPAAL respectively.

For the original model, all queries are satisfied, meaning the design follows all TLM-2.0 rules. In case of the *AT-example* with *FaultA*, two models of this fault are defined and injected to the original source code of the *AT-example*. As it shown in Table I, these faults cause two violations over the generated queries of each system. *FaultA-2* changes the transaction data length attribute to a negative value. In case of the *AT-example* with FaultB, the number of violated queries for each system is different depending on which TLM-2.0 base protocol transaction is used. This is because each transaction may contain a different number of states, with individual queries (all inheriting the same "overall" parent query) verifying different transactions concerning the same rule. E.g. a rule violation that alters the payload size between states 1 and 2 is detected in a different query than the one that alters the payload size between states 2 and 3.

The *P/VMU* column shows the value of resident memory over virtual memory usage peaks when checking a property using the UPPAAL model checker. As UPPAAL considers each query separately, for each system the maximum *P/VMU* over all queries has been reported.

The complete results of applying the proposed method to verify different TLM-2.0 designs are presented in Table II using the same scenarios (*Original Model*, *FaultA* and *FaultB*). The *LOC*, *Comp*, *Trans* and *TM* columns present the complexity and difference of each design in terms of lines of code, number of components, number of transaction and the timing model, respectively. The *UTrans* column shows how many different base protocol transactions are implemented in each design. The *Seq* column shows the number of lines related to the unique behavioral information that has been extracted during the execution of each design. The *TotalT* column illustrates the total time required to verify each design using the proposed method. It consists of two parts: the time required to check the whole queries for each design (*VT*) and the time spent to extract its simulation behavior (*ET*). The *CExeT* column is the time required to compile and execute each design using GCC without applying the proposed method.

In Table II, the faulty *AT-example* and *Locking-two* designs have different *LOC* and *Comp* values than the original models. These differences result from the custom base protocol checkers that were included in both designs and that needed to be removed to create faulty models. As the complexity of the faulty models in both cases decreased, the value of parameters *Seq*, *ET* and *CExeT* related to this complexity reduced as well.

B. Integration and Discussion

The proposed method is able to automatically retrieve the simulation behavior of a given SystemC TLM-2.0 implementation and formally verify it against TLM-2.0 rules.

In comparison to the formal verification methods that are restricted to a sub-set of SystemC models because of several pre-conditions and pre-assumptions on SystemC implementations, the proposed method has no limitation on how and which SystemC constructs are used to implement an ESL design. Although the proposed method does not assure the validity of the model like formal methods (as it verifies the model's simulation behavior instead of its formal specification), it makes a trade-off between the accuracy of the model to check the compliance of the model and its applicability to verify a wide range of SystemC designs.

Unlike simulation based verification approaches that rely on manipulating the original source code or modifying the SystemC library and/or interfaces to verify the run-time information, the proposed method verifies the detailed simulation behavior of a given TLM model without any modification of the user's implementation and the standard tool flow. It means that the proposed method provides the designers with a non-intrusive and an easy-to-use TLM protocol compliance checking solution. Moreover, it can be combined with setups that already rely on a modified SystemC library which makes the approach applicable to a wide range of TLM models.

The only precondition of the suggested approach is that the executable SystemC TLM-2.0 model contains debug information that is compatible with GDB. Hence, while GCC and Clang-LLVM are thus supported, Microsoft Visual Studio is not.

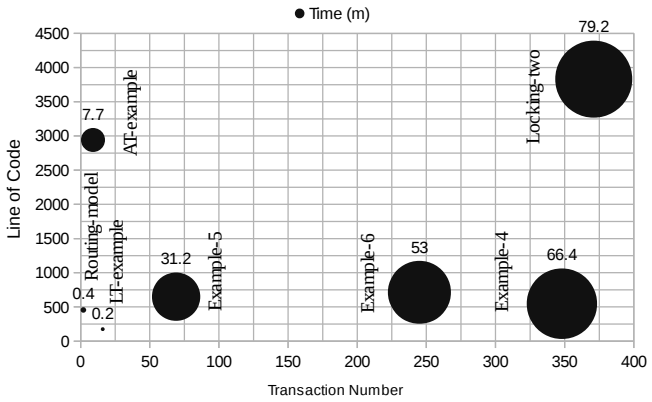

Fig. 5. Analysis of data extraction time

The performance of the proposed method is considered in two phases which is the time that is spent to

- retrieve the run-time information of a given TLM design and transform it into the timed automata model and
- verify the transformed model against set of properties using the UPPAAL model checker

As illustrated in Table II, the extraction of a TLM-2.0 design's run-time information is the major time consuming part of the method. The information extraction process is expensive as the program is executed on GDB. To store the state of the program during its simulation time on disk, the execution has to be halted repeatedly. Therefore, this time is related to the complexity of an application and the amount of information to be extracted depending on the model's timing parameters. This complexity can be related to the lines of code and the number of transactions. Fig. 5 illustrates an analysis on run-time information extraction based on the lines of code and the number of transactions. For low numbers of transactions, the designs with more lines of code and AT timing model exhibit an increased execution time. By increasing the number of transactions and lines of code, the required time increases. The maximum extraction is used by *Locking-two* (about 80 minutes).

As shown in Table II, the time consumed for the first step by the proposed method is within a reasonable boundary for the simple designs like *LT-example* and *Routing Model* or the complex design *AT-example* with low numbers of transactions in comparison to their compilation and execution time using GCC. For complex designs such as *Locking-two* with large numbers of transactions, the results in this table show the time is still in order of minutes and hours meaning that while the method may not be applicable for the figurative coffee break of a developer, it still provides designers with a simple solution to analyze a design's behavior within reasonable time frames.

As the proposed method streams all information directly to disk, the memory overhead for the first step is negligible.

V. Conclusion

In this paper a protocol compliance checking methodology of SystemC TLM-2.0 designs has been proposed. The method is based on formally verifying the simulation behavior of a given SystemC TLM model by UPPAAL model checker. The simulation behavior which describes the transactions of the model is extracted automatically and non-intrusively using an automated debugger. The transactions and TLM-2.0 protocol rules are defined formally in terms of timed automata and property language used by UPPAAL respectively. A formal description of a large set of all TLM-2.0 protocol rules has been provided in TCTL for the first time. Two types of potential violations related to the transaction attributes and communication semantics are defined and injected to the original case studies to evaluate the proposed method. The experimental results confirm the applicability and quality of the proposed method to verify several TLM-2.0 designs with different complexity. For future work, we plan to extend the method to support both user-defined protocol and the TLM model including transaction extension.

VI. Acknowledgments

Financial support of subproject P02 "Heuristic, Statistical and Analytical Experimental Design" of the Collaborative Research Center SFB 1232 "Farbige Zustände" by the German Research Foundation (DFG), the Reinhart Koselleck project DR 287/23-1 (DFG), University of Bremen's graduate school SyDe, funded by the German Excellence Initiative and BMBF grant SELFIE, no. 01IW16001 is gratefully acknowledged.

References

[1] IEEE Standard SystemC Language Reference Manual. *IEEE Std 1666-2005*, pages 1–423, 2006.
[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, 1993.
[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, pages 183 – 235, 1994.
[4] J. Aynsley. TLM-2.0 base protocol checker. https://www.doulos.com/knowhow/systemc/tlm2/at_example. Accessed: 2016-01-30.
[5] J. Aynsley, editor. *OSCI TLM-2.0 Language Reference Manual*. Open SystemC Initiative (OSCI)., 2009.
[6] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
[7] M. Bawadekji, D. Große, and R. Drechsler. TLM protocol compliance checking at the electronic system level. In *DDECS*, pages 435–440, 2011.
[8] C. N. Chou, Y. S. Ho, C. Hsieh, and C. Y. Huang. Symbolic model checking on SystemC designs. In *DAC*, pages 327–333, 2012.
[9] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull. Interactive presentation: Implementation of a transaction level assertion framework in SystemC. In *DATE*, pages 894–899, 2007.
[10] Y. Endoh. ASystemC: An AOP extension for hardware description language. In *AOSD*, pages 19–28, 2011.
[11] L. Ferro and L. Pierre. Isis: Runtime verification of TLM platforms. In *FDL*, pages 1–6, 2009.
[12] M. Goli, J. Stoppe, and R. Drechsler. AIBA: an Automated Intra-cycle Behavioral Analysis for SystemC-based design exploration. In *ICCD*, 2016.
[13] M. Goli, J. Stoppe, and R. Drechsler. Automatic equivalence checking for SystemC-TLM 2.0 models against their formal specifications. In *DATE*, 2017.
[14] D. Große, H. M. Le, and R. Drechsler. Proving transaction and system-level properties of untimed SystemC TLM designs. In *MEMOCODE*, pages 113–122, 2010.
[15] A. Habibi and S. Tahar. Design and verification of SystemC transaction-level models. *VLSI*, 14(1):57–68, 2006.
[16] P. Herber, J. Fellmuth, and S. Glesner. Model checking SystemC designs using timed automata. In *CODES+ISSS*, 2008.
[17] P. Herber and S. Glesner. A HW/SW co-verification framework for SystemC. *TECS*, 12:61:1–61:23, 2013.
[18] M. Kallel, Y. Lahbib, R. Tourki, and A. Baganne. Verification of SystemC transaction level models using an aspect-oriented and generic approach. In *DTIS*, pages 1–6, 2010.
[19] H. M. Le, D. Große, V. Herdt, and R. Drechsler. Verifying SystemC using an intermediate verification language and symbolic simulation. In *DAC*, pages 1–6, 2013.
[20] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosentiel. Object-oriented modeling and synthesis of SystemC specifications. In *ASP-DAC*, pages 238–243, 2004.
[21] H. Sohofi and Z. Navabi. Assertion-based verification for system-level designs. In *SQED*, pages 582–588, 2014.
[22] R. Stallman and C. Support. *Debugging with GDB: The GNU Source-level Debugger*. Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA, ninth edition, 2010.
[23] D. Tabakov and M. Y. Vardi. Monitoring temporal SystemC properties. In *MEMOCODE*, pages 123–132, 2010.
[24] D. Tabakov and M. Y. Vardi. Automatic aspectization of SystemC. In *MISS*, pages 9–14, 2012.
[25] T. L. Veldhuizen. C++ templates are turing complete. Technical report, 2003.