Towards Fully Automated TLM-to-RTL Property Refinement*

Vladimir Herdt[1] Hoang M. Le[1] Daniel Große[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
{vherdt,hle,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—An ESL design flow starts with a TLM description, which is thoroughly verified and then refined to a RTL description in subsequent steps. The properties used for TLM verification are refined alongside the TLM description to serve as starting point for RTL property checking. However, a manual transformation of properties from TLM to RTL is error prone and time consuming. Therefore, in this paper we propose a fully automated TLM-to-RTL property refinement based on a symbolic analysis of transactors. We demonstrate the applicability of our property refinement approach using a case study.**

I. Introduction

In the recent years, the emergence of the *Electronic System Level* (ESL) [1] can be witnessed. It has become an industry common practice to model hardware designs at this new level of abstraction using the IEEE standard language SystemC [2], [3]. At ESL, functional behaviors of a design are described using the full descriptive power of C++ together with the added layer of concurrency by SystemC, while its communication interfaces are abstracted to function calls and parameters. The synchronization between functional behaviors and communication interfaces is realized using events. These modeling techniques are termed as *Transaction Level Modeling* (TLM) and standardized as a set of interfaces as TLM-2.0 [2].

In the top-down ESL design flow, SystemC TLM models are available very early and get successively refined to RTL. TLM models also serve as *reference models* to verify the corresponding RTL models later. The verification is commonly performed by a TLM/RTL co-simulation, where the same input stimuli are applied to both models and their outputs are checked for equivalence. This step requires an additional executable verification component, known as *transactor*, to bridge the function calls at TLM with the signal-based interfaces at RTL and vice versa, as shown in the upper half of Fig. 1.

Obviously, the correctness of TLM models is also of great importance. In the past few years, a wide body of verification techniques at TLM has been developed ranging from simulation-based (e.g. [4]–[6]) to formal verification (e.g. [7]–[14]). Please note that we do not try to be comprehensive but only mention a few representative approaches. Most of these approaches are based on the principle of *Assertion-Based Verification* (ABV) [15], which has been very successful for RTL verification. Assertions, also known as properties, provide a mean to formally capture the functional specification of the design. They specify conditions on design inputs, outputs and internal variables that are supposed to hold at all times. These conditions can also be temporal, i.e. referring to a sequence of values over time. Assertions can be automatically translated into executable monitors to be co-simulated with the design to check the specified conditions. For the specification of TLM properties, extensions to standardized language such as IEEE-1850 PSL or IEEE-1800 SVA have been proposed [4], [16].

While the verification step based on TLM/RTL co-simulation is indispensable, the obtained results are far from complete and cannot ensure the absence of bugs at RTL. For some special cases when the TLM models are high-level synthesizable (e.g. when all used constructs are in the *SystemC Synthesizable Subset*), formal sequential equivalence checking is supported by existing EDA tools such as Calypto SLEC or Synopsys Hector. For general TLM models, it is highly desirable that co-simulation is complemented with other (preferably formal) approaches. A promising direction is to reuse properties that have been proven on the TLM model. Due to the semantic differences of the involved abstraction levels, straight-forward reuse is not possible. The translation process, i.e. TLM-to-RTL property refinement, is mostly manual, therefore error-prone and time-consuming.

Related Work: Several improvements to the manual TLM-to-RTL property refinement have been proposed. Ecker et al. [17] formulated a set of requirements for the refinement process. In a follow-up work [18], an automated refinement framework has been introduced. Still, a set of refinement rules have to be defined by verification engineers before the automated translation can start. Chen and Mishra [19] proposed a similar approach that requires a formal (temporal) semantic mapping between TLM functions and clocked RTL signals. Pierre and Amor [20] developed a set of pre-defined "pattern-matching" rules to ensure that the refined RTL properties belong to the *simple subset* of PSL, which is generally easier to verify. However, the need for a manual semantic mapping is still not circumvented. Bombieri et al. [21] proposed to reuse TLM properties in a TLM/RTL co-simulation. Thanks to the availability of transactors, RTL signals over time are converted back to TLM transactions and the TLM properties can thus be checked on the RTL design. This is, however, not a semantic approach, i.e. no corresponding RTL properties can be derived to apply, for example, RTL property checking.

Paper Contribution and Organization: To the best of our knowledge, we propose in this paper the first *fully automated* TLM-to-RTL property refinement approach. The lower half of Fig. 1 shows an overview. The main idea lies in a better reuse of the readily available transactors. At the core of our approach is a static transactor analysis based on symbolic execution (Section III). Essentially, the analysis reverse-engineers the executable transactors to create

* This work was supported in part by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1, by the German Federal Ministry of Education and Research (BMBF) within the project CONFIRM under contract no. 16ES0565 and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

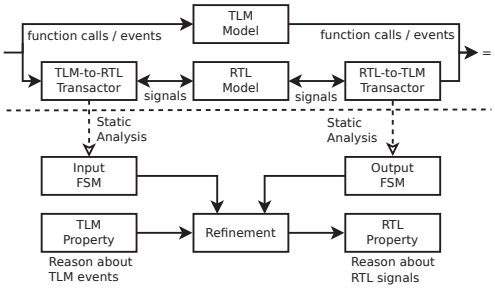

Fig. 1. TLM-to-RTL Property Refinement Overview

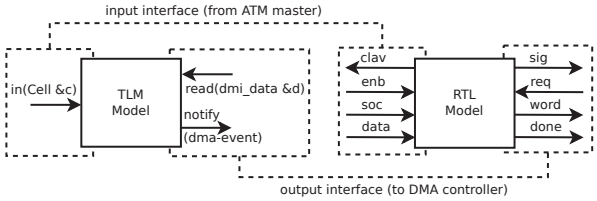

Fig. 2. UTOPIA controller TLM and RTL model IO interface (simplified)

a formal specification of the underlying protocol as *Finite State Machines* (FSM). Then, TLM properties are refined by relating high-level TLM events with RTL signal combinations at different clock cycles based on the FSM (Section IV). We believe that the proposed approach is of great practical interest. While it is possible to generate transactors automatically from a formal specification [22], in practice, verification engineers still develop transactors manually from scratch based on a textual specification. As the proposed approach can be best explained on concrete examples, we introduce in Section II a realistic case study for illustration and also feasibility demonstration.

II. UTOPIA Case Study

As case study we consider an UTOPIA-based (*universal test and operations PHY interface for ATM)* controller acting as receiver slave device. It communicates with an *Asynchrounous Transfer Mode* (ATM) master device using ATM cells, provides an internal buffer to store up to two cells, and a DMA interface to access the buffer. For illustrative purposes, we use a simplified data model, where a cell consists of 3 bytes and the DMA controller word size is 16 bits. DMA access is expected to be word-aligned thus the internal buffer cells are zero-padded (one byte here). Fig. 2 shows the TLM and RTL IO interface:

1) TLM Interface: The *in* function accepts a whole cell and writes it into the internal buffer. It blocks in case the buffer is full. The (dma-) read function provides a direct memory interface for reading a buffer cell (through a char pointer in *dmi_data*). The (dma-) event signalizes that a cell is available for reading (condition checked after every *in* or *read* call). For this interface, we have five TLM events for property specification: *in:begin*, *in:end*, *read:begin*, *read:end* and (dma-) *event:notified* (abbreviated *notified* as it is the only event).

2) RTL Interface: On the input side *clav* (cell available) signals the controller is able to receive a whole data cell, i.e. buffer not full. The master then starts the transfer by activating the *enb* (enable) signal, sets *soc* (start of cell) to 1 and data to the first byte of the cell. In subsequent clock cycles, the remaining cell bytes are transferred, one after another through *data* signal. During transfer, soc stays 0 and enb stays 1.

On the output side, *sig* (read signal) is active iff a whole cell is available in the internal buffer. DMA controller enables *req* (request) when ready to receive. Then, the data is provided sequentially through the *word* signal (2 byte steps). Finally, *done* is enabled one clock cycle after data transfer has finished.

3) Transactors: Fig. 3 shows the relevant implementation parts of the input transactor (TLM-to-RTL). Essentially, the transactor has two functions: a TLM input function to receive a cell and a RTL interface function to send it out. The TLM input function receives a cell and stores it internally. It will block (Line 8) in case a cell has already been received but not send out (*data_available=true*). The RTL interface function is sampled at every RTL clock cycle. It mimics the behavior of an ATM master device. Therefore, the transactor keeps track of the internal protocol status (Line 18). It starts in *status=WAIT* until the clav signal is observed. Then (*status=START*) it initiates data transfer once a TLM cell is available. In the next clock cycles (all *status=SEND*), the remaining data bytes are transferred (Line 41). The variable i tracks the next byte to send. Once the transfer is finished, the internal protocol data (status and i) are reset and the TLM input function is unblocked (Line 44-47), i.e. can receive the next cell. Asserts are used in the transactor to detect illegal signal combinations (e.g. clav is set to low before data is sent). The implementation of the output transactor (RTL-to-TLM) is similar to the input transactor, and therefore omitted.

III. Static Analysis of Transactors

This section describes our static analysis based on symbolic execution to extract the underlying transactor protocol between TLM and RTL as FSMs.

A. Symbolic Execution

Starting with the initial symbolic execution state, the transactor function is repeatedly executed, until all reachable states are explored. Fig. 4 shows the complete symbolic state space of the transactor. In total we have 7 distinct states ($\{S0, ..., S6\}$), which make up the state space, and 6 additionally observed states ($\{*S7, ..., *S12\}$) that are equivalent to one of the first 7 and thus are discarded.

The initial execution state S0 is defined as follows: RTL signals and TLM input are externally provided and therefore initialized with symbolic values. The protocol data (*status* and index i) are initialized with concrete values. Beside the variable environment, a symbolic execution state has a path condition (*PC*, initialized to *True*, i.e. no constraints in S0). Constraints are added when executing an *assert(C)*, i.e. $PC = PC \wedge C$, or branch with symbolic condition C, e.g. starting from S0 execution will fork due to symbolic signal *clav* resulting in the states S1 and S6 (*PC* updated as $PC = PC \wedge C$ and $PC = PC \wedge \neg C$, respectively). Between subsequent executions protocol data is preserved, whereas the RTL signal values are reset to fresh symbolic values each time (as they can be modified in each clock cycle). The TLM input data is preserved between executions where the TLM input function cannot be called. This is the case for S3 and S4, since data_available is set to *true*, which will block the input function. This information is collected by performing a preliminary static analysis on the input function. S5 unblocks the input function again by setting *data_available* to *false* and notifying the awaited event *empty*. We annotate

```
1  struct TLMToRTLTransactor {
2    bool data_available = false;
3    char data[CELL_SIZE];
4    event empty;
5
6    void tlm_input(const Cell &c) {
7      if (data_available) {
8        wait(empty);
9        assert (!data_available);
10     } else {
11       data_available = true;
12       for (i=0; i<CELL_SIZE; ++i)
13         data[i] = c.data[i];
14     }
15   }
16
17   enum State {WAIT, START, SEND};
18   State state = WAIT;
19   i = 1;
20   RTLSignals rtl;
21
22   void rtl_interface() {
23     if (state == WAIT) {
24       rtl.enb = 0;
25       if (rtl.clav) {
26         state = START;
27       }
28     } else if (state == START) {
29       assert (rtl.clav);
30       if (data_available) {
31         rtl.enb = 1;
32         rtl.soc = 1;
33         rtl.data = data[0];
34         state = SEND;
35       } else {
36         rtl.enb = 0;
37       }
38     } else if (state == SEND) {
39       rtl.enb = 1;
40       rtl.soc = 0;
41       rtl.data = data[i];
42       ++i;
43       if (i == CELL_SIZE) {
44         i = 1;
45         state = WAIT;
46         data_available = false;
47         notify(empty);
48       }
49     }
50   }
51 };
```

Fig. 3. TLM-to-RTL transactor relevant implementation pseudocode

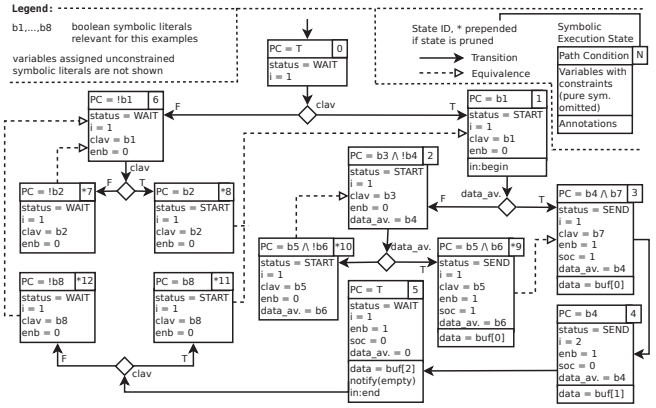

Fig. 4. Input transactor symbolic execution state space for the RTL interface function for the input transactor in Fig. 3, path conditions have been optimized by unused constraint elimination

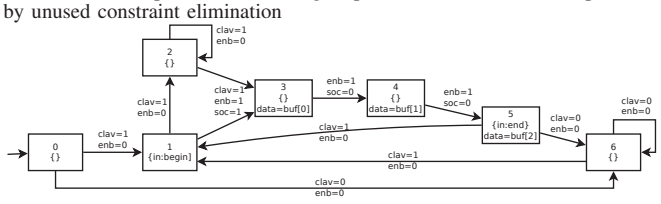

Fig. 5. FSM for the TLM-to-RTL transactor (input)

the input related TLM events, i.e. *in:b* and *in:e*, to S1 and S5, respectively. The reason is that the execution of S1 is the first one to access input related data, and S5 the last one. Furthermore, we annotate the data mapping between the internal buffer *buf* and the RTL *data* signal in S3,S4 and S5. Combining it with the data mapping of the input function, there exists a mapping between the RTL *data* signal and TLM input cell at different states.

B. FSM Construction

The symbolic state space is transformed into an FSM by abstracting away all data except the RTL signal values. The FSM contains the same states and edges as the symbolic state space, but the edges are annotated with RTL signal values. For example starting from S0 either S1 (with *clav=1* and *enb=0*) or S6 (with *clav=0* and *enb=0*) can be reached. Please note, that the value of clav is constrained to 1 (S1) and 0 (S6) due to the path condition. The other signal values are unconstrained and thus not mentioned. Signal combinations which are not covered, e.g. *clav=0* and *enb=1* in S0, are invalid based on the protocol. By repeating this process for all other edges, the transactor FSM is obtained. The input transactor FSM is shown in Fig. 5. We additionally preserve the TLM event annotations and data mappings in the FSM states. The output transactor FSM shown in Fig. 6 can be obtained similarly.

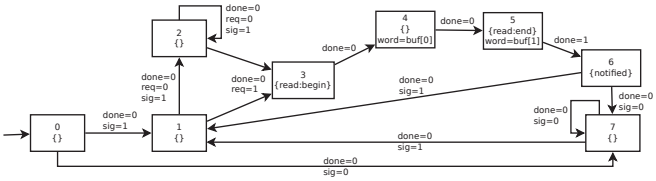

Fig. 6. FSM for the RTL-to-TLM transactor (output)

IV. Property Refinement

This section outlines the refinement process based on the extracted FSMs. Before going into details, we briefly describe the used property language as well as the actual TLM properties for refinement.

A. Property Specification Language

For property specification, we consider the *simple subset* of PSL with TLM extensions similar to [4], [16]. For TLM properties, we use TLM events as atomic boolean propositions. For every function f we define the events *f:begin* and *f:end* (abbreviated as *f:b* and *f:e*). Arguments can be captured to reason about the data, e.g. *in(x):e* denotes that the *in* function has finished with x being passed as parameter. Finally, *e:notified* denotes that event *e* has been notified. Boolean C++ expressions can be embedded to reason about data relations. We use SEREs (sequentialized regular expressions) to specify RTL traces (i.e. signal sequences sampled at clock cycles).

B. TLM Properties for Refinement

We consider two TLM properties P1 and P2, shown in upper part of Table I. P1 specifies that after a DMA read a DMA event notification needs to happen before the next DMA read is started. This ensures that data is available before it is attempted to read. P2 checks that no packets are lost. It captures the input packet in the variable x and when the input is completed, either the next or second next DMA read will read x. The *next_event(α, ϕ)* operator requires that ϕ is true at the next occurrence of α. The $CEQ(y, x)$ predicate returns true when the DMA result and input cell are equal. The predicate can be expressed in C++ syntax as shown in Table I (upper part).

C. Refinement Process

Our refinement works recursively. The operators *always*, *next*, *before*, |, and *next_event(α, ϕ)* are preserved. The implication *L* −> *R* is mapped to *L* |−> *R*, i.e. first *L* needs to be matched and matching of *R* starts at the clock cycle *L* has been matched at. This is a valid refinement thanks to the monotonic advancement of time required by the simple subset.

A TLM event E is mapped to a sequence of RTL signals (i.e. traces) by following back all edges starting from FSM state S, event E is associated with, and collecting the signal assignments until the resulting traces T uniquely identify S,

TABLE I
MAPPING OF TLM TO RTL PROPERTIES WITH REFINEMENT STEPS

CEQ(y,x)	`(y.ptr[0] & 0xff == x.data[0]) ∧ (y.ptr[0] >> 8 == x.data[1]) ∧ (y.ptr[1] & 0xff == x.data[2])`
P1	`always(read:e -> notify before read:b)`
P2	`always(in(x):e -> next_event(read(y):e, CEQ(y,x) | next(next_event(read(z):e, CEQ(z,x)))))`
RTL(notified)	`done`
_H1	`(!done&&!sig | done ; !done&&!sig) | done ; !done&&sig ; !done&&req`
_H2	`(!done&&!sig | done ; !done&&sig) | !done&&!req&&sig ; !done&&!req&&sig`
RTL(read:b)	`_H1 | _H2 ; !done&&req`
RTL(read:e)	`!done&&sig | !done&&sig&&!req ; !done&&req ; !done ; !done`
RTL(in(x):e)	`(enb&&clav&&soc):NEW(`x_0`:=data) ; (enb&&!soc):NEW(`x_1`:=data) ; (enb&&!soc):NEW(`x_2`:=data)`
_S4	`(done | !done&&!sig ; !done&&sig) | !done&&!req&&sig ; !done&&req ; !done`
RTL(read(y):e)	`_S4:NEW(`y_0`:=word) ; !done:NEW(`y_1`:=word)`
RTL(CEQ(y,x))	$(y_0$ `&` $0xff == x_0)$ $\wedge$ $(y_0$ `>>` $8 == x_1)$ $\wedge$ $(y_1$ `&` $0xff == x_2)$
RTL(P_1)	`always(RTL(read:e) |-> RTL(notify) before RTL(read:b))`
RTL(P_2)	`always(RTL(in(x):e) |-> next_event(RTL(read(y):e), RTL(CEQ(y,x))) | next(next_event(RTL(read(z):e), RTL(CEQ(z,x)))))`

Upper half shows the TLM properties and equality predicate CEQ, middle half shows mapping of TLM events to RTL signal sequences, and lower half shows the refined results.

i.e. starting from any state Q in the FSM and following any $t \in T$, only state S can be reached (otherwise there has been an invalid transition). We call T the distinguishing traces of state S, and can straightforwardly express T as a SERE by sequencing (operator `;`) all signal assignments within a trace and then union all traces (operator `|`). Table I (middle part) shows the results for the relevant TLM events. For example *notified* (associated with state S6 in output FSM) is uniquely identified by *done* (i.e. *done=1*), as S5 is the only state where *done* is a valid signal assignment. On the other hand to identify S5 (associated with *read:e*) going back one step is not enough, as many states can be reached with *!done*. In fact, T for S5 contains two traces of length 4.

Function call events capturing input/output arguments require an extended refinement procedure. It is not enough to just visit the FSM state associated with the function begin/end, but also go through all states where a data mapping to the arguments exists. The event *in(x):end* requires to visit the state sequence *[S3,S4,S5]* in the input FSM. Therefore, we generate T for the first state in the sequence and extend it with signal assignments to reach the subsequent states by following along the FSM edges. At each state we capture the current data signals in a new local variable for S3, e.g. *NEW(x_0:=data)* (see [20]). Based on the data mapping from the FSMs, we obtain that RTL signal *data* in S3 (input FSM) is mapped to *buf[0]* and following along the data mapping of the TLM input function, that *buf[0]* maps to *c.data[0]* of the input cell. Similarly, we obtain mappings for the output side. With the captured data signals and mappings, we can automatically refine the *CEQ(y,x)* predicate (Table I lower part).

V. DISCUSSION AND FUTURE WORK

Our proposed *fully automated* TLM-to-RTL property refinement approach is still a work-in-progress. The monotonic advancement of time through the formula allows for an intuitive recursive refinement algorithm, where evaluation of subformulas starts at the clock cycle the previous one finished at. Although the feasibility of our refinement approach has been demonstrated on a realistic case study with two representative TLM properties, a formal characterization of supported properties is still to be defined in a future work. One limitation is that we assume the next (or SEREs in general) operator is not used to specify an absolute order of TLM events. The reason is, that it is unclear how to automatically refine it (especially when combined with other operators), as there can be (arbitrary) many clock cycles between TLM events at RTL. Moreover, it has been suggested that using the next operator in this way at TLM is not well suited (up to the point of being unsound) due to its restrictiveness [20]. Rather, the *before* and *next_event* operators, which specify precedence, should be used. However, using a single next operator in combination with *−>* and *next_event* makes sense to avoid matching the subformula at the current instant (e.g. as used in our property P2 in Section IV-C to avoid matching the same *read*). This issue needs further investigation.

For property refinement, we compute distinguishing traces for FSM states. However, this is not possible for FSMs that have multiple state cycles, which accept the same input sequence. For a protocol description such an FSM implies that it is not possible to detect the protocol status by just observing the IO signals, i.e. there is no synchronization or handshaking in place. While we believe it is a reasonable assumption to have distinguishing states, our refinement algorithm can be extended to handle non-distinguishing states as well by: 1) accessing internal variables of the model, or 2) generating additional logic between model and monitor to simulate the FSM (can be done automatically). We plan to evaluate these extensions in future work.

REFERENCES

[1] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.
[2] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.
[3] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.
[4] L. Ferro and L. Pierre, "Formal semantics for PSL modeling layer and application to the verification of transactional models," in *DATE*, 2010, pp. 1207–1212.
[5] D. Tabakov and M. Y. Vardi, "Monitoring temporal SystemC properties," in *MEMOCODE*, 2010, pp. 123–132.
[6] M. F. S. Oliveira et al., "The system verification methodology for advanced TLM verification," in *CODES+ISSS*, 2012, pp. 313–322.
[7] P. Herber, J. Fellmuth, and S. Glesner, "Model checking SystemC designs using timed automata," in *CODES+ISSS*, 2008, pp. 131–136.
[8] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *MEMOCODE*, 2010, pp. 113–122.
[9] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking SystemC," *TCAD*, vol. 32, no. 5, pp. 774–787, 2013.
[10] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *DAC*, 2013, pp. 116:1–116:6.
[11] V. Herdt, H. M. Le, and R. Drechsler, "Verifying SystemC using stateful symbolic simulation," in *DAC*, 2015, pp. 49:1–49:6.
[12] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "ParCoSS: efficient parallelized compiled symbolic simulation," in *CAV*, 2016, pp. 177–183.
[13] ——, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, pp. 52:1–52:8.
[14] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Towards formal verification of real-world SystemC TLM peripheral models – a case study," in *DATE*, 2016, pp. 1160–1163.
[15] H. Foster, "Applied assertion-based verification: An industry perspective," *Found. Trends Electron. Des. Autom.*, vol. 3, no. 1, pp. 1–95, 2009.
[16] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman, "A temporal language for SystemC," in *FMCAD*, 2008, pp. 1–9.
[17] W. Ecker, V. Esen, T. Steininger, and M. Velten, "Requirements and concepts for transaction level assertion refinement," in *IESS*, 2007, pp. 1–14.
[18] T. Steininger, "Automated assertion transformation across multiple abstraction levels," Ph.D. dissertation, TU Munich, Germany, 2009.
[19] M. Chen and P. Mishra, "Assertion-based functional consistency checking between TLM and RTL models," in *VLSID*, 2013, pp. 320–325.
[20] L. Pierre and Z. B. H. Amor, "Automatic refinement of requirements for verification throughout the SoC design flow," in *CODES+ISSS*, 2013, pp. 29:1–29:10.
[21] N. Bombieri, F. Fummi, and G. Pravadelli, "Incremental abv for functional validation of tl-to-rtl design refinement," in *DATE*, 2007, pp. 882–887.
[22] F. Balarin and R. Passerone, "Specification, synthesis, and simulation of transactor processes," *TCAD*, vol. 26, no. 10, pp. 1749–1762, 2007.