2013 26th International Conference on VLSI Design and the 12th International Conference on Embedded Systems

Assertion-Based Functional Consistency Checking between TLM and RTL Models

Mingsong Chen
Shanghai Key Lab of Trustworthy Computing
East China Normal University, Shanghai, China
Email: mschen@sei.ecnu.edu.cn

Prabhat Mishra
Department of Computer & Information Sci. & Eng.
University of Florida, Gainesville, FL, USA
Email: prabhat@cise.ufl.edu

***Abstract*—Transaction Level Modeling (TLM) is promising for functional validation at an early stage of System-on-Chip (SoC) design. However, raising the abstraction level brings a major challenge - how to guarantee the functional consistency between TLM specifications and Register Transfer Level (RTL) implementations? This paper proposes an efficient mechanism for functional consistency checking using assertion observability. The experimental results using several industrial designs demonstrate that our method can automatically check the functional consistency between different abstraction levels.**

***Index Terms*—assertion, functional consistency, TLM, RTL**

I. Introduction

Maintaining the functional consistency between different abstraction layers is an important issue during System-on-Chip (SoC) design. The top-down SoC design flow starts from an executable SystemC TLM [1] specification. This TLM model should be thoroughly validated to ensure that it can be used as a "golden reference model" for the successive refinement [2]. High-level abstraction in TLM models significantly reduces functional validation effort compared to validation of RTL models. Once TLM models are verified, these models are refined into corresponding hardware and software implementations. In this paper, we focus on the refinement of hardware, i.e., RTL designs. Traditional simulation based methods can only guarantee the refinement correctness by enumerating input tests and comparing primary output results. Significant difference of internal structure of TLM and RTL designs is often eclipsed. Therefore, no correlation of the internal structure can be assumed during the simulation. It is also not feasible to perform formal consistency checking using conventional equivalence checkers due to significant difference between the TLM and RTL models as well as capacity restrictions of the checkers.

Assertion based validation (ABV) [3] is a promising alternative. It can not only increase the design observability during simulation, but also take advantage of formal techniques for improving the overall verification quality. An assertion can be viewed as an observation point in a TLM or RTL design to monitor the specified functional scenario. For simplicity, we use the term assertion to indicate both assertion and property.

In this paper, we propose a methodology to check the functional consistency between TLM and RTL models based on the observability of assertions. The basic idea is that in a TLM specification, if a test can exercise a specified functional scenario monitored by some assertions, then in its RTL implementation, the counterpart of the TLM test can also activate the counterparts of the TLM assertions. During the TLM-to-RTL functional consistency checking, we need to address the following three challenges:

1) *How to determine a set of TLM assertions for observing functional scenarios?* We propose promising transaction-level models that can lead to generation of assertions. The goal is to cover all possible faults in the fault models using the generated assertions.
2) *How to reuse TLM validation effort?* We develop the validation refinement rules which can convert TLM assertions to their RTL counterparts.
3) *How to use the correlation between TLM and RTL assertions for consistency checking?* We propose a method to verify the TLM-to-RTL consistency based on the criteria of assertion coverage and assertion ordering.

Our proposed approach addresses above challenges and makes two major contributions: i) presents a framework for automatic TLM-to-RTL assertion refinement, and ii) proposes a method that utilizes the assertion observability for functional consistency checking between TLM and RTL models. Since our work is based on the reuse of TLM validation effort, there is no extra cost (excludes defining the refinement rules) for this improved validation methodology. Furthermore, our method can be fully automated and can be easily scaled for large designs. Our approach has two synergistic advantages. It is expected to improve design quality due to functional consistency checking. It can also reduce the overall validation cost since TLM validation effort (tests and assertions) is reused for RTL validation.

The rest of the paper is organized as follows. Section II presents related works. Section III proposes our consistency checking framework based on validation reuse. Section IV presents the experimental results. Finally, Section V concludes the paper.

II. Related Work

Although ABV is a promising approach for functional validation in RTL level, it is still a challenging domain for system level design. To address the issues when incorporating Property Specification Language (PSL) [4] within SystemC environments, Lahbib et al. [5] proposed an automated solution

This work was partially supported by NSF CAREER award 0746261, Natural Science Foundation of China 61202103, Open Project of SW/HW Co-design Engineering Research Center of MoE 2012002, and State High-Tech Development Plan of China 2011AA010101.

1063-9667/13 $26.00 © 2013 IEEE
DOI 10.1109/VLSID.2013.208

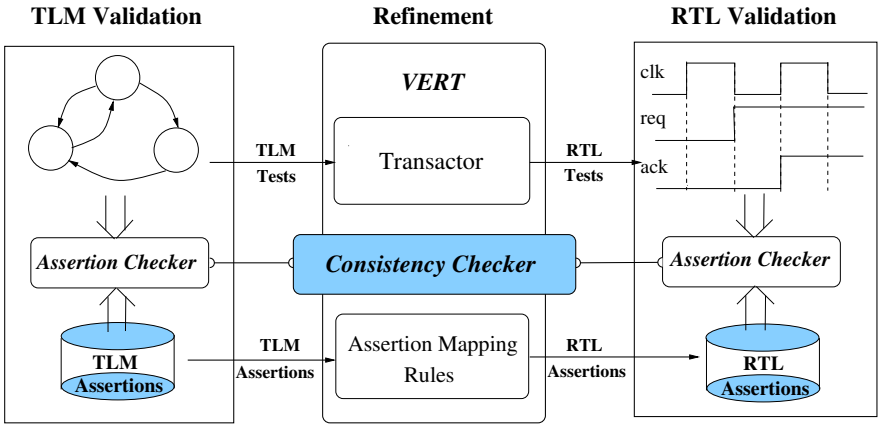

Fig. 1. Our consistency checking framework

which can embed PSL assertions in a SystemC design. Based on static code analysis and genetic algorithms, Habibi et al. [6] presented an efficient method to optimize test generation in order to increase the assertion coverage. Ecker et al. [7] proposed a transaction-level assertion framework using a new specialized language. In [8], Pierre described an efficient and tractable solution for verifying the PSL based properties of TLM designs during the simulation. However, most researches are focused on implementing PSL assertions in SystemC framework, and none of them use assertions for checking the TLM-to-RTL functional consistency.

To enable the consistency checking, various approaches [17], [21] are proposed. However, these methods assume strict timing information to be available, which is not suitable for TLM designs. The framework in [16] allows the equivalence checking for Electronic System Level (ESL) design flow from TLM by analyzing the accesses of user selected variables. In [9], Kasuya and Tesfaye presented a mechanism to construct and reuse temporal assertions in various TLM abstraction levels. Bombieri et al. [10] proposed a transactor-based dynamic verification method. By using transactors, the TLM testbenches can be reused during the TLM-RTL co-simulation. In [11], Bombieri et al. defined functional consistency based on event order without timing information. However, their method applies assertions on TLM specifications only. It monitors primary input and output signals without investigating RTL implementation details.

III. Assertion-based Consistency Checking

Figure 1 shows our consistency checking framework. First, TLM assertions are derived based on specified fault models by analyzing TLM specifications. To activate an assertion, users can use random tests, automated directed tests [12], [13] or the tests which are manually written by experts. Next, the refinement process translates the TLM assertions for RTL validation using our proposed mapping rules. The refined assertions are instrumented in RTL implementation. Through a user-defined transactor, the refined TLM tests are applied on the RTL implementation. The output of the tests and the activated assertions are monitored by an RTL assertion checker. Finally, by comparing simulation traces recorded by TLM and RTL assertion checkers, the consistency checker reports the results. Our methodology has three important steps: i) automatic TLM assertion generation, ii) TLM to RTL assertion/test refinement, and iii) assertion-based consistency checking. It is important to note that these three steps are independent of each other. The following subsections discuss each of these steps in detail.

A. Automatic TLM Assertion Generation

Assertions are used to specify the required functional behaviors of a system. To investigate the consistency between TLM and RTL models, we need to explore as many assertions as possible. In our method, we define a set of *fault models* to achieve a complete set of assertions. Each fault indicates a required "design behavior" which may be violated during the system design. For example, when validating a desired scenario described by a *sequence p* (*sequence* is a PSL term which indicates a sequential expression), we use the following PSL statement pairs to detect whether the sequence p will happen finally. The *Prop*1_1 asserts that the sequence p must "*eventually*!" hold strongly during the simulation, and *Prop*1_2 is used to record the assertion coverage during the simulation by using verification directive "cover".

```
Prop1_1: assert eventually! p;
Prop1_2: cover (p);
```

SystemC TLM emphasizes the functionality of data transfers instead of actual implementation. Essentially a SystemC TLM design interconnects a set of processes using transactions (i.e., C++ function calls) for communication. Each process does the following tasks: receiving data, processing data and sending data. Therefore the most important factors in TLM are transaction data and transaction flow. So during the TLM-to-RTL synthesis, these factors should be reflected. Inspired by the fault models based on *bit failures* and *condition failures* proposed in [15], we consider the following two fault models: *data fault model* and *flow fault model* which reflect both the TLM structure and behavior information.

Transaction data fault model deals with the possible value assignment for each part of a transaction data. However, for property generation, due to the large size of value space, trying all possible values of a data is infeasible. By checking each bit of a variable (data bit fault) separately, the data content coverage can be partially guaranteed. The following is an example of a data fault.

```
//The second bit of "packet.parity" can be 1.
assert eventually! (packet.parity==2);
cover (packet.parity==2);
```

Transaction flow fault model handles the controls along a transaction flow. To ensure transaction flow coverage, one can cover branch conditions which exist in *if-then-else* or *switch-case* statements. The goal is to check all possible transaction flows. The following is an example of a transaction flow fault.

```
//The condition packet.to_chan=1 can be true.
assert eventually! (packet.to_chan==1);
cover (packet.to_chan==1);
```

It is important to note that the proposed fault models are by no means "golden". It can be modified for improvement in our validation methodology. The order of the assertion activations plays a key role when verifying functional consistency as described in Section III-C.

B. Refinement of TLM Assertions/Tests

In our framework, we use SystemC for transaction level modeling and Verilog for RTL modeling. Since TLM design is significantly different from its RTL implementation in port definition, internal structure and timing details, it is necessary to provide the mapping information. There details are also needed during the manual or automatic TLM-to-RTL synthesis. As shown in Figure 1, the *Validation Effort Reuse Tool* (VERT) enables TLM-to-RTL refinement by specifying transformation rules. VERT supports transactor definition for test refinement. VERT also allows users to provide Assertion Refinement Specification (ARS) which contains the rules to guide the assertion refinement. Generally an ARS contains two parts as follows.

1) Symbol Mapping specifies the name and type mapping between TLM variables and RTL signals.
2) Assertion Refinement Rules specify patterns and timing information for RTL assertions.

Due to the naming convention inconsistency between TLM specification and RTL implementation, during the validation refinement, it is necessary to have a symbol table which specifies the name mappings. Each item in the symbol table defines the correspondence between TLM and RTL variables. Generally it provides the following information: i) name mapping, ii) data type mapping, and iii) bit mapping.

In our framework, we use SystemVerilog Assertion (SVA) [14] for RTL validation. The generated TLM assertions are in the simple syntax like "assert eventually! p". Most of them are temporal assertions involving transaction data only without any clock and control signal information. However, RTL assertions generally have such lower level details. Therefore, during the assertion refinement, we also need to consider clock expression and control signals. Once these details are available, the assertion refinement can be done by inserting the timing and control information as well as by substituting symbols.

```
SYMBOL_MAPPING
 bit[1:0] addr = tmp_packet.to_chan;
 ......
END_SYMBOL_MAPPING

ASSERTION_SPEC
 `set_clock (posedge clock);
 ......
 `control
   tmp_packet.to_chan
   @ $rose(packet_valid);
 ......
END_ASSERTION_SPEC
```

An example of ARS specification is shown above for a packet router with one master node and several slave nodes. Here, *tmp_packet.to_chan* is a TLM variable that denotes the target address of a packet. From the symbol mapping, we can figure out the corresponding RTL internal signal is *addr* which is a 2-bit register. In the *ASSERTION_SPEC* block, the directive $\backtick set\_clock$ sets the clock expression for the refined assertions. Because in RTL different value of control signals may specify different meaning to input data signals, we use the directive $\backtick control$ to set the RTL control signals during the TLM data refinement. The first parameter of $\backtick control$ is a TLM variable that appears in the TLM assertion. The second parameter is the corresponding RTL control signal expression for the TLM variable. In this example, only when the RTL signal *packet_valid* rises, the RTL signal *addr* can indicate the target slave address. The following example shows the TLM-to-RTL assertion translation using the above ARS. We can find that the RTL assertion includes the clock expression. The VERT substitutes the TLM variable *tmp_packet.to_chan* for its RTL signal $addr[1:0]$ accompanied by its control signal $\$rose(packet\_valid)$.

```
TLM assertion: cover(tmp_packet.to_chan==1);
RTL assertion: cover property
(@(posedge clock) ($rose(packet_valid)&&
addr[1:0]==2'd1));
```

C. Assertion-Based Functional Consistency

We define functional consistency based on the assumption that if a TLM test can trigger a TLM assertion, then its RTL counterpart will also trigger the corresponding RTL assertion. It is important to note that our method relies on the same concept of classical consistency checking (or equivalence checking) - if two designs are consistent, when same inputs are used, they will produce the same outputs. However, like white-box testing, our method also investigates the internal assertion activation events. Our goal is to increase the confidence of TLM-to-RTL functional consistency checking under the monitoring of assertion activations.

1) Assertion-Based Functional Coverage: Since an assertion activation means that a specific functional scenario is covered, the coverage of the assertions indicates the adequacy of the functional validation. Let T be a TLM design and R be an RTL design of T. We generate a set of TLM assertions $T_{assertion}$ according to the specified fault models of T, and we obtain a set of TLM tests T_{test} to activate such assertions. Assume that RTL test set R_{test} is a refinement of T_{test}, and RTL assertion set $R_{assertion}$ is a refinement of $T_{assertion}$. When running T_{test} and R_{test} on T and R individually, we can get the assertion coverage defined as follows.

Definition 1: Given a TLM specification T and its RTL implementation R, by applying T_{test} on T and R_{test} on R, the assertion coverage can be calculated as:

$$T_{coverage} = \frac{\#\ of\ exercised\ TLM\ assertions}{|T_{assertion}|}$$
$$R_{coverage} = \frac{\#\ of\ exercised\ RTL\ assertions}{|R_{assertion}|} \quad \blacksquare$$

2) Assertion Ordering: For a TLM or RTL design which is instrumented with a large number of assertions, during the simulation, a test may exercise a sequence of assertions. Simulation using an input test leads to an *assertion trace* which reveals the temporal order of checked functions in a system behavior. For a TLM test and its refined RTL version, when applying them on the TLM and RTL designs individually, it is required that the TLM and RTL execution behaviors are consistent. In other words, the TLM assertions and corresponding RTL assertions should happen in their traces in the same order.

Since several assertions may be activated simultaneously, it is difficult to determine the order of the assertions in an assertion trace. In addition, the loop structure may further increase the difficulty in assertion matching. Inspired by the algorithm proposed by Lamport [18], in our framework, each assertion activation in a trace is associated with a "timestamp" to indicate the *happens before* (marked by $\prec$) relation. We use the timed assertion in the form of (a,t) to denote that the assertion a happens at clock cycle t.

Definition 2: Given two timed assertions $(a,t1)$ and $(b,t2)$ in an assertion trace. The relation between them are as follows.

- $(a,t1)$ happens before $(b,t2)$ iff $t1 < t2$.
- if $t1 == t2$, then the two assertions are concurrent, written $(a,t1)\ ||\ (b,t2)$ $\blacksquare$

Definition 2 describes the relation between the timed assertions. The key issue in determining the order is to figure out the timestamp for an assertion. For RTL design, we can monitor the simulation at each clock cycle. Therefore, we can define the timestamp using the clock cycle number. However, figuring out the assertion order for TLM designs is not trivial due to multiple abstraction levels.

According to the definitions in [1], there are three TLM abstraction layers: Programmer's View (PV), Programmer's View with Time (PVT), and Cycle Accurate (CA). The model in CA abstraction level is cycle accurate. It is quite similar to the corresponding RTL model with respect to the notion of time. For the PVT abstraction level, the model simulates in non-zero simulation time. In spite of the timing inaccuracy, we can still judge the assertion order according to the simulation time. During the simulation, if an TLM assertion is exercised, we can use the SystemC function *sc_time_stamp()* to record the current simulation time. Such *sc_time* information can be used to order assertions. The PV abstraction level is untimed. To determine the order of the interaction between communicating processes, SystemC provides the *delta cycle* concept which adopts the *evaluate-update* paradigm to interpret *zero-delay* semantics. Each tiny delta cycle consists of these *evaluate* and *update* phases without advancing the simulation time. Therefore the delta cycle can be utilized for ordering the assertions. In this case, we need to use a global variable as a counter of delta cycles, which can be used as the timestamp for assertions. If two assertions happen in the same delta cycle, then they are concurrent. Otherwise there is a "happen before" relation between them.

3) Functional Consistency Checking Using Assertions: The refinement process is described by two functions - AR for assertion refinement and TR for test refinement as follows.

$$AR : T_{assertion} \rightarrow R_{assertion}$$
$$TR : T_{test} \rightarrow R_{test}$$

We also define the function M_{TLM} and function M_{RTL} to indicate the relation between tests and assertions, i.e., the set of assertions that are activated during the simulation by a given test.

$$M_{TLM} : T_{test} \rightarrow 2^{T_{assertion}}$$
$$M_{RTL} : R_{test} \rightarrow 2^{R_{assertion}}$$

M_{TLM} indicates the set of TLM assertions that are covered by a given TLM test. Similarly, M_{RTL} indicates the set of RTL assertions that are covered by a given RTL test. Based on the above definitions, the definition of TLM-to-RTL consistency is given as follows.

Definition 3: Given a TLM specification T and its RTL implementation R, T and R are assertion consistent iff T_{test} can achieve 100% TLM assertion coverage and

$$\forall t \in T_{test}.\ \ M_{RTL}(TR(t)) \supseteq \{AR(a_1), AR(a_2), \ldots, AR(a_n)\}$$
$$where\ \ M_{TLM}(t) = \{a_1, a_2, \ldots, a_n\}. \quad \blacksquare$$

The assertion consistency only defines the assertion coverage for each test. In fact, there is a temporal relation between assertions. If the assertion consistency considers the event order, we call it *strongly assertion consistent*.

Definition 4: Given a TLM specification T and its RTL implementation R, T and R are strongly assertion consistent iff

- T and R are assertion consistent; and
- $\forall t \in T_{test}$, the TLM assertions covered by t and the RTL assertions covered by $TR(t)$ are activated in the same order. $\blacksquare$

Figure 2 illustrates an example of assertion consistency. Assuming that the TLM specification and RTL implementation

are assertion equivalent and t is a TLM test and $t' = TR(t)$, we can get $M_{TLM}(t)$ = $\{a1,a2,a3\}$ and $AR(M_{TLM}(t))$ = $\{b1,b2,b3\}$, where $AR(M_{TLM}(t))$ is a subset of $M_{RTL}(t')$. It shows that the assertion activation order is not consistent ($a2$ happens before $a1$, but $b1$ happens before $b2$). Therefore, in this case, the TLM design and RTL design are assertion consistent but not strongly assertion consistent.

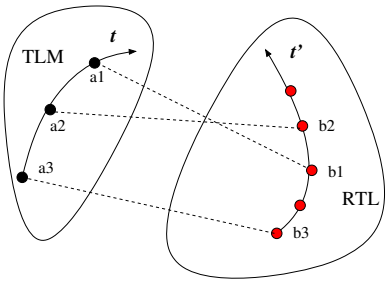

Fig. 2. An example of assertion consistency

IV. Experiments

This section presents two case studies: a router system and a simplified version of the pipelined Alpha AXP processor [19]. We developed a prototype tool which can: 1) parse SystemC designs and automatically generate TLM assertions from the specified fault models; 2) translate TLM assertions to RTL assertions according to the provided ARS; and 3) report the consistency checking result. Currently, we derive one ARS manually for each design. Although this process needs expert knowledge of the design, we believe it can be automated when synthesizable description of TLM models are available in near future. The experimental results are obtained on a 2.0 GHz Intel i7 server with 4G RAM using Linux operation system.

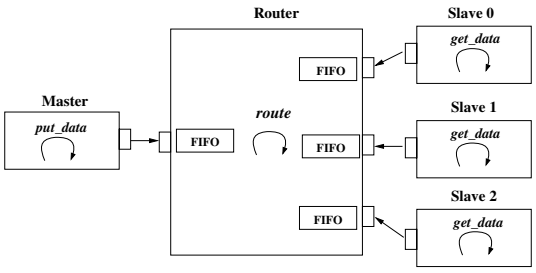

Fig. 3. The TLM structure of the router

A. A Router Example

Figure 3 shows the structure of the TLM specification of a router example. It consists of five modules: one master, one router and three slaves. The main function of the router is to parse incoming packets and distribute them to target slaves. By using our tool VERT, 59 TLM assertions and 59 RTL assertions (55 from data fault model and 4 from flow fault model) were automatically generated. To achieve 100% TLM assertion coverage, 1000 random tests were applied on the TLM design that took less than 3 seconds. For each assertion, we selected one random test to exercise it. Therefore we selected 59 random TLM tests. To improve the RTL code coverage, in the router example we manually created 2 directed TLM tests (1 tests for FIFO overflow, and 1 test for reset check). Finally we got 61 TLM tests and 61 RTL tests for validation purposes.

It is important to note that the generation of TLM assertions/tests and the validation refinement process are independent. In other words, TLM assertions and tests can come from multiple sources. Under the guidance of transactors and assertion mapping rules, the tool VERT can translate the TLM tests and TLM assertions to the corresponding RTL tests as well as RTL assertions in the form of SVA.

TABLE I
RTL Coverage for the Router Example

Module	Line	Toggle	FSM	Condition	Assertion
fifo	100%	100%	NA	100%	NA
port_fsm	95.92%	100%	87.5%	71.88%	100%
router	100%	100%	NA	NA	100%

* NA stands for "Not Available". DVE [20] does not provide any data in these scenarios.

We applied the TLM and RTL tests on the TLM and RTL levels independently. For the TLM design, we can get 100% coverage on both code and assertions. The RTL implementation of the router primarily consists of three kinds of components: FIFO buffers (*fifo*), controller (*port_fsm*) and datapath (*router*). During the simulation of the RTL design, we measured various coverage metrics using Synopsys VCS Discovery Visualization Environment (DVE) tool [20]. Table I shows the coverage obtained using the refined tests. Due to some unreachable code and missing "else" statements in RTL implementation, we cannot obtain 100% coverage in all the categories. It is important to note that initially the directed tests can only have 98.3% assertion coverage on the refined RTL assertions. We investigated the uncovered assertions and found the reason is that the generated assertions and tests try to activate the functional scenario $to\_chan = 3$ (error scenario), which is not implemented in the RTL design. So we corrected the RTL implementation and finally we can get 100% assertion coverage.

Our result shows that the TLM and RTL designs of the router example are assertion consistent. Since the TLM design is timed, by matching the timed assertions on the assertion trace of each test, it shows that the TLM and RTL designs of the router example is also strongly assertion consistent. Moreover, the result shows that our method drastically reduces the RTL validation effort. By applying the 61 refined random RTL tests, it only took 4 seconds to achieve 100% RTL assertion coverage. However, for RTL validation using random method individually, achieving 100% assertion coverage needs more than 10000 random RTL tests which took 1057 seconds.

B. A Pipelined Processor Example

Figure 4 shows the structure of the TLM specification of the Alpha AXP processor. It consists of five stages: Fetch (IF), Decode (ID), Execute (EX), Memory (MEM) and Writeback (WB). IF module fetches instructions from the instruction memory. ID module decodes instructions and fetches the

operand data if necessary. EX module does ALU operations as well as asserts whether the conditional or unconditional branch happens. Memory module reads and writes data from (to) the data memory. Writeback module stores the result in specified registers.

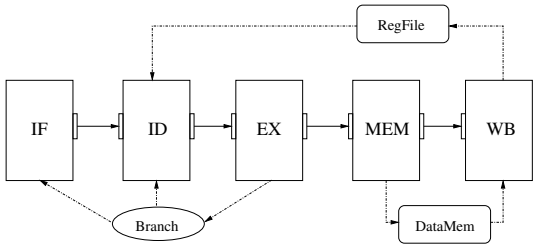

Fig. 4. The TLM structure of the Alpha AXP processor

Our tool generated 163 TLM assertions (117 from data fault model and 46 from flow fault model). To obtain 100% TLM assertion coverage, 3000 random TLM tests were needed. We selected 163 TLM tests (from the 3000 tests) that can activate all the 163 TLM assertions. Both tests and assertions were refined for consistency checking using our VERT tool. The result shows that assertion consistency can be achieved. By comparing the assertion activation sequence, the result indicates that we can also achieve a strong assertion consistency by using the derived TLM and RTL tests.

TABLE II
RTL COVERAGE FOR THE ALPHA AXP PROCESSOR

Module	Line	Toggle	FSM	Condition	Assertion
IF_stage	100%	68.82%	NA	100%	100%
ID_stage	100%	80.00%	60.00%	100%	100%
EX_stage	100%	52.94%	NA	100%	100%
MEM_stage	100%	74.19%	NA	100%	100%
WB_stage	100%	78.52%	NA	100%	100%
regfile	100%	71.29%	NA	55.56%	100%

* NA stands for "Not Available". DVE [20] does not provide any data in these scenarios.

Table II shows various RTL coverage results using the refined 163 RTL tests, which only required 15 seconds for simulation. Note that without refinement information from the TLM counterpart, it is hard to achieve 100% coverage of refined assertions in the RTL design. For this example, 50000 RTL tests (took 1390 seconds) were needed to achieve 100% assertion coverage. Besides consistency checking, in this example, our method can efficiently achieve promising RTL coverage in all categories except for the toggle coverage. This is because only a small set of directed tests is involved during the coverage generation. By increasing the test number, the toggle coverage can be improved. For the *regfile* module, since TLM does not consider all kinds of internal forwarding, the transformed 163 RTL tests can only achieve a 55.56% condition coverage.

V. CONCLUSIONS

Raising the abstraction level in SoC design flow can significantly reduce the overall design effort but introduce two challenges: i) how to guarantee the functional consistency between system-level design and low-level implementation, and ii) how to manage the increasing overall validation effort among different abstraction levels? To address both problems, this paper proposed a methodology which reuses TLM-level validation effort to enable RTL validation as well as assertion-based functional consistency checking between TLM and RTL models. Our framework can generate a set of assertions and corresponding tests to validate all the specified TLM "faults". The assertions and tests can be translated to their RTL counterparts using our proposed framework. During the simulation, the TLM-to-RTL functional consistency can be verified based on the assertion coverage and assertion ordering. The experimental results using several industrial designs demonstrated that our approach can simultaneously improve the design quality and reduce the overall validation effort by several orders of magnitude.

REFERENCES

[1] OSCI TLM WG Whitepaper. Transaction Level Modeling in SystemC. *http://www.systemc.org*.
[2] L. Cai and D.Gajski. Transaction Level Modeling: An Overview. In *Proc. of CODES + ISSS*, pages 19–24, 2003.
[3] H. D. Foster, A. C. Krolnik, and D. Lacey. *Assertion-Based Design, 2nd Edition*. Kluwer Academic Publishers, Boston, MA, 2004.
[4] Property Specification Language. *http://www.eda.org/ieee-1850/*.
[5] M. Lahbib, R. Kamdem, M. Benalycherif, and R. Tourki. An Automatic ABV Methodology Enabling PSL Assertions across SLD Flow for SOCs Modeled in SystemC. *Computers and Electrical Engineering*, 31(4):282–302, 2005.
[6] A. Habibi and S. Tahar. Towards An Efficient Assertion Based Verification of SystemC Designs. In *Proc. of HLDVT*, pages 19–22, 2004.
[7] W. Ecker, V. Esen, T. Teininger, M. Velten, and M. Hull. Interactive Presentation: Implementation of A Transaction Level Assertion Framework in SystemC. In *Proc. of DATE*, pages 894–899, 2007.
[8] L. Pierre and L. Ferro. A Tractable and Fast Method for Monitoring SystemC TLM Specifications. *IEEE Transactions on Computers*, 57(10):1346–1356, 2008.
[9] A. Kasuya and T. Tesfaye. Verification Methodologies in a TLM-to-RTL Design Flow. In *Proc. of DAC*, pages 199–204, 2007.
[10] N. Bombieri, F. Fummi, and G. Pravadelli. On the Evaluation of Transactor-Based Verification for Reusing TLM Assertions and Testbenches at RTL. In *Proc. of DATE*, pages 1–6, 2006.
[11] N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva. Towards Equivalence Checking Between TLM and RTL Models. In *Proc. of MEMOCODE*, pages 113–122, 2007.
[12] M. Chen and P. Mishra. Property Learning Techniques for Efficient Generation of Directed Tests. *IEEE Transactions on Computers*, 60(6):852–864, 2011.
[13] M. Chen, X. Qin, H. Koo and P. Mishra. *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*. Springer, 2012.
[14] SystemVerilog Assertion Homepage. *http://www.eda.org/sv-ac/*.
[15] F. Ferrandi, F. Fummi, L. Gerli and D. Sciuto. Symbolic Functional Vector Generation for VHDL Specifications. In *Proc. of DATE*, pages 442–446, 1999.
[16] D. Große, M. Groß, U. Kühne and R. Drechsler. Simulation-Based Equivalence Checking Between SystemC Models at Different Levels of Abstraction. In *Proc. of GLSVLSI*, pages 223–228, 2011.
[17] S. Vasudevan, J. A. Abraham, V. Viswanath, and J. Tu. Automatic Decomposition for Sequential Equivalence Checking of System Level and RTL Descriptions. In *Proc. of MEMOCODE*, pages 71–80, 2006.
[18] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communication of ACM*, 21(7):558–565, 1978.
[19] R. L. Sites. Alpha AXP Architecture. *Communication of ACM*, 36(2):33–44, 1993.
[20] SYNOPSYS VCS Verification Library. *http://www.synopsys.com*.
[21] M. Chen, P. Mishra, and D. Kalita. Automatic RTL Test Generation from SystemC TLM Specifications. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(2):38, 2012.