Towards RTL Test Generation from SystemC TLM Specifications *

Mingsong Chen Prabhat Mishra
Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611
{*mchen, prabhat*}*@cise.ufl.edu*

Dhrubajyoti Kalita
Intel Corporation
1900 Prairie City Road, Folsom, CA 95630
dhrubajyoti.kalita@intel.com

Abstract

SystemC Transaction Level Modeling (TLM) is widely used to reduce the overall design and validation effort of complex System-on-Chip (SOC) architectures. Due to lack of efficient techniques, the amount of reuse between abstraction levels is limited in many scenarios such as reuse of TLM level tests for RTL validation. This paper presents a top-down methodology for generation of RTL tests from SystemC TLM specifications. This paper makes two important contributions: automatic test generation from TLM specification using a transition-based coverage metric and automatic translation of TLM tests into RTL tests using a set of transformation rules. Our initial results using a router design demonstrate the usefulness of our approach by capturing various functional errors as well as inconsistencies in the implementation.

1 Introduction

Increasing complexity of System-on-Chip (SOC) architectures coupled with time-to-market pressure create a critical need to raise the level of abstraction for SOC designs. SystemC has been used both in industry and academia for system level design. SystemC Transaction Level Modeling (TLM) is very promising for early exploration, hardware-software co-design, and platform-based design and verification. Due to recent TLM standardization efforts, transaction-level modeling also offers extensive design and verification reuse between projects as well as between abstraction levels in the same project [1].

Due to significant differences between TLM and RTL models, the degree of reuse is very limited. In the absence of significant reuse of design and validation efforts between different abstraction levels, the overall functional validation effort may increase since the designer has to verify TLM as well as RTL models. Furthermore, it is hard to guarantee the consistency between the abstraction levels. Simulation is the most widely used form of validation for both TLM and RTL designs using random and directed-random tests. Certain heuristics are used to generate directed random tests. However, due to the bottom-up nature and localized view of these heuristics, the generated tests may not yield a good coverage. A major challenge is to develop an efficient coverage metric at TLM level that enables coverage-driven directed test generation. The goal is to generate a small set of directed tests that will cover all the functionalities of the TLM design and reduce the validation effort at the TLM level. Complete reuse of such TLM tests will lead to a drastic reduction of RTL validation effort as well.

This paper presents a top-down methodology for generation of RTL tests from SystemC TLM specifications. The basic idea is to use TLM specification to perform coverage-based TLM test generation and TLM-to-RTL test translation using a set of transformation rules. The rest of the paper is organized as follows. Section 2 describes related work addressing TLM-based validation approaches. Section 3 presents our test generation methodology followed by a case study in Section 4. Finally, Section 5 concludes the paper.

2 Related Work

There are several existing validation approaches in the TLM-based design flow. Jindal et al. [2] present a method to reduce the verification time by reusing the earlier RTL testbenches. Wang et al. [3] describe a coverage-directed method that is suitable for transaction level verification. The approach is based on random test generation and the coverage is increased by using fault insertion method. There are various researches on validation reuse between TLM and RTL levels. For example, Bombieri et al. [4] shows that transactor-based verification is at least as efficient as a fully RTL verification methodology which converts TLM assertions into RTL properties and creates new RTL testbenches.

The existing simulation-based approaches cannot fully reuse the SystemC TLM level validation effort due to lack of a golden formal reference model. Various researchers have tried to translate SystemC TLM to a formal representation to enable automated analysis and test generation. Abdi et al. [5] introduce Model Algebra, a formalism for representing SOC designs at system level. The work by Kroening et al. [6] formalize the semantics of SystemC by means of labeled Kripke structures. Moy et al. [7] provide a compiler front-end that can extract architecture and synchronization information from SystemC TLM design using HPIOM. Karlsson et al. [8] translate SystemC models into a Petri-Net based representation PRES+. This model can be used for model checking of properties expressed in a timed temporal logic. To the best of our knowledge, there are no previous approaches that can automatically generate RTL tests from the TLM specification to enable implementation validation.

*This work was partially supported by grants from Intel Corporation.

3 RTL Test Generation from TLM Specification

Figure 1 shows our RTL test generation methodology. It assumes that the TLM specification as well as the RTL implementation is developed manually (hand-written) and both TLM and RTL models need to be validated.

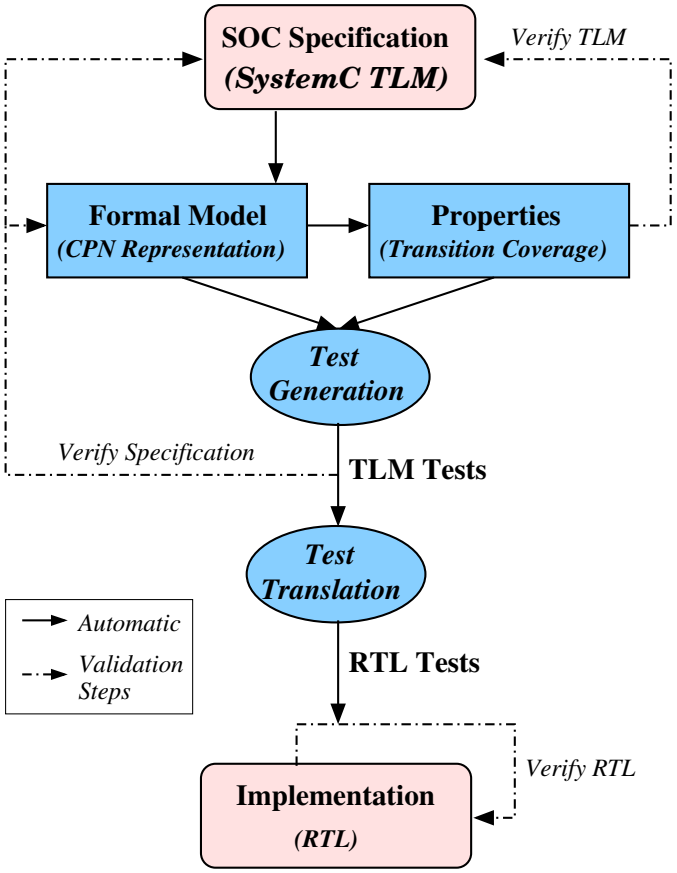

Figure 1. Proposed RTL test generation methodology

This methodology has three important steps: i) formal representation of the TLM specification, ii) coverage-based generation of TLM tests, and iii) translation of TLM tests into RTL tests using transformation rules. It is necessary to identify (or develop) a suitable formal representation that can capture a wide variety of TLM specifications. We have used Colored Petri Nets (CPN) [9] in our framework. The next step is to define a procedure that can generate the CPN representation from the TLM specification. To enable test generation, we need to define a suitable coverage metric. We have used the CPN transition coverage in our framework to generate necessary properties and tests. Finally, we define a set of transformation rules using timing and input/output mapping information to convert TLM tests into RTL tests.

The remainder of this section is organized as follows. Section 3.1 describes the suitability of CPN for formal representation. Section 3.2 presents the procedure for converting TLM specification into CPN representation. Section 3.3 outlines the TLM test generation approach using model checking. Finally, Section 3.4 presents the transformation rules for converting TLM tests into RTL tests.

3.1 Formal Modeling of TLM Specifications

The SystemC TLM specification itself is not formal enough to enable automated analysis and test generation. Therefore, it is necessary to convert the given TLM specification into a formal representation. There are three primary requirements for the formal model.

- It should be simple enough to allow correlation (one-to-one correspondence) between the model and the TLM specification.
- It should be powerful enough to capture a wide variety of TLM specifications.
- Most importantly, it should be formal enough to enable automated analysis and test generation.

We have evaluated the suitability of various formal models including finite state machines, Petri nets, graphs and their variations (such as CFSM, CDFG etc.) for representing TLM specifications. We have identified Colored Petri Nets (CPN) as suitable for our work. CPN is a variation of Petri net [10] which allows the use of tokens that carry data values - in contrast to the tokens of low-level Petri nets which by convention are drawn as black, "uncolored" dots. CPN provides a framework for construction and analysis of distributed and concurrent systems. Moreover, CPN offers hierarchical descriptions. Such properties make the CPN well suited for formal modeling of TLM specifications.

The CPN uses primarily places and transitions to describe a system. The ellipses or circles in CPN are called places. They describe the states of the system. The bars or rectangular boxes are called transitions which are used to describe actions or transitions. The transitions may have guards. A transition can only be enabled if the value of its guard is true. The arrows are called arcs. The arc expressions describe how the state of the CPN changes when the transition occurs. Each place contains a set of markers called tokens. In contrast to low-level Petri nets (such as Place/Transition Nets), each of these tokens carries a data value, which belongs to a given type.

3.2 Transformation of SystemC TLM to CPN

Each SystemC statement is represented by one transition in CPN and each SystemC variable can be represented by a place. The transition can perform actions of the statement. Each token carries the value of the corresponding variable (the token also has the clock information for timed CPN). The tokens in the input places enable the execution of the transition. The transition can be fired only when its condition is satisfied and all the input places have the tokens. When the transition is executed, it will distribute the resulting tokens according to the expression

and the color set of the outgoing arcs. Based on this semantics, CPN can be used to model the concurrent systems and the synchronization activities.

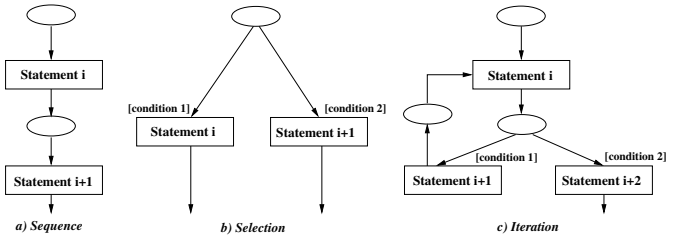

Figure 2. Three fundamental constructs

It is possible to establish one-to-one mapping from SystemC to CPN. Figure 2 shows how CPN models three key constructs in SystemC TLM description: sequence, selection and iteration. A complex CPN can be constructed by composing these three constructs. For example, Figure 3 shows a segment of SystemC code and corresponding CPN representation. At first, only places $p1$ and $p2$ have the tokens, so the transition $t1$ can be fired. If the loop condition is met, a new token consisting of variables n and i will be sent to place $p3$. Otherwise, the loop will finish and the token of n will be sent to the place $p4$. The transition $t2$ performs the operation $n = n + i$. In this figure, all transitions have time delay interval of 0, so the delays are omitted.

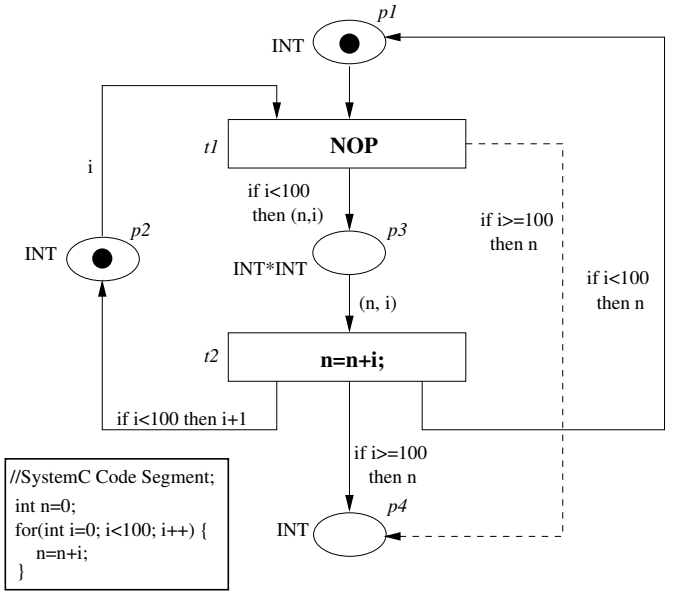

Figure 3. An example of generated CPN from SystemC code

It is important to note that each transition can be hierarchically defined. So it is natural to model a function call by using a transition. This transition can be refined by a CPN. During the refinement, it is important to guarantee that the input and output tokens are consistent with the input parameters and return values of the top level structure.

SystemC TLM descriptions consists of modules, interfaces and channels described in a hierarchical manner. Similarly, CPN can capture them in a hierarchical manner which can also be flattened into a complex CPN representation. The SystemC TLM use the *wait* and $notify$ to synchronize the system model. In CPN, the semantics of the transition can do the same job. The CPN transition can not only synchronize the dataflow and control flow, but also it can have a time delay constraint which is corresponding to $wait(time)$ in SystemC TLM.

3.3 Generation of TLM Tests using Model Checking

Algorithm 1 outlines our TLM test generation approach. The first step is to generate the design and the properties. The SMV description of the design is generated from the CPN representation. Properties are generated based on the CPN transition coverage. In other words, one property is generated for each CPN transition. Next, the design and the negated version of the property are applied to the model checker which produces a counterexample. The generated counterexample can be analyzed to obtain the TLM test.

Algorithm 1: *Test Generation*
Inputs: i) Model of the design, M
ii) Set of faults/interactions, F (based on CPN transition coverage)
Outputs: Test programs
Begin
  TestPrograms = ϕ
  for each fault F_i in the set F
    P_i = CreateProperty(F_i)
    $\overline{P_i}$ = Negate(P_i)
    $test_i$ = ModelChecking($\overline{P_i}$, M)
    TestPrograms = TestPrograms $\cup$ $test_i$
  endfor
  return TestPrograms
End

Section 4.3 presents a detailed example of TLM test generation using model checking. Clearly, model checking based approach is promising for automated and directed test generation but may lead to state space explosion in the presence of complex design and properties. In these circumstances, various design and property decomposition techniques [11] can be explored to reduce the complexity of test generation.

3.4 Translation of TLM Tests into RTL Tests

A major challenge in TLM-to-RTL test translation is how to bridge the abstraction gap. We use transformation rules to generate RTL tests from TLM tests. The transformation rules use the input/output mapping information as well as timing details between TLM and RTL. For example, "$p \rightarrow chan$" in TLM is mapped to "$data[0:1]$" in RTL. We have developed transformation rules to automatically convert TLM tests into RTL tests

using input/output mappings and timing specifications. We use the following five transformation rules.

- **Add Initialization:** This part includes initialization of various input variables at RTL which have no mappings in TLM.
- **Add Reset Sequence:** This part includes the sequence of assignments to the reset signal in RTL for enabling the normal operation.
- **Transformation of Names and Expressions:** This part describes the names and expressions in TLM that need to be replaced based on the mapping functions.
- **Add Delay:** Appropriate delays need to be inserted after each step based on the timing information.
- **Application Specific Transformations:** This part includes transformations specific to an application. For example, in case of a router the *packet_valid* signal needs to be initialized to **0** and **1** at specific time to enable data read and parity check.

4 A Case Study

We applied our methodology on an industrial router example. This section is organized as follows. First, we describe the TLM specification of the router. Next, we present the CPN representation as well as TLM test generation for the router. Finally, we discuss the TLM-to-RTL test translation for validation of the router implementation.

4.1 TLM Specification of The Router

Figure 4 shows an architecture graph of the router specification. The router has one input port and three output ports. Each port is connected to a first-in-first-out (FIFO) buffer (channel) which stores the packet.

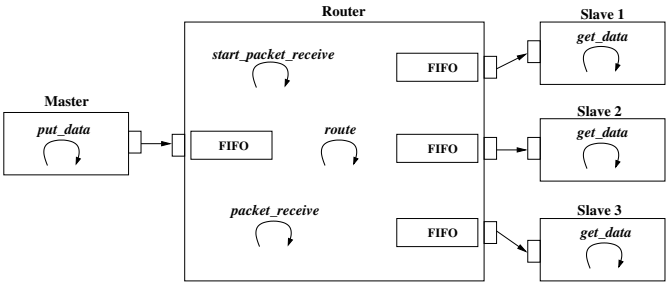

Figure 4. The architecture graph of a router

The master module sends the packet which is in the form as shown in Figure 7a. The packet consists of 3 parts: header, payload and parity. The header has 8 bits, bit 0 and bit 1 are used as the address of output port. The other 6 bits indicate the size of the payload. So the maximum payload size is 63. The last byte of the packet is the parity of both header and payload. The router has three processes which are implemented by *SC_METHOD*. Process *start_packet_receive* can be triggered if there are some packets in the input FIFO channel. Then after the time *PACKET_GAP*, *start_packet_receive* will trigger process *packet_received* to receive the packet. After time *RCV_DELAY*, it will trigger process *route* to distribute the packet to the respective output channel according to the address bits of the header. Finally, the slave modules will read the packets when data is available in the respective FIFOs.

4.2 CPN Representation of the Router

Figure 5 shows the CPN representation of the router. There are 17 places and 11 transitions in the CPN. Each transition is associated with one action of the system. For the conditional branch in the CPN, we consider all the possibilities. For example, although there are only 3 output channels, we add the "*channel*=3" in the CPN because there are 2 bits to decide the output address. The mapping between CPN transitions and TLM actions is presented in Table 1.

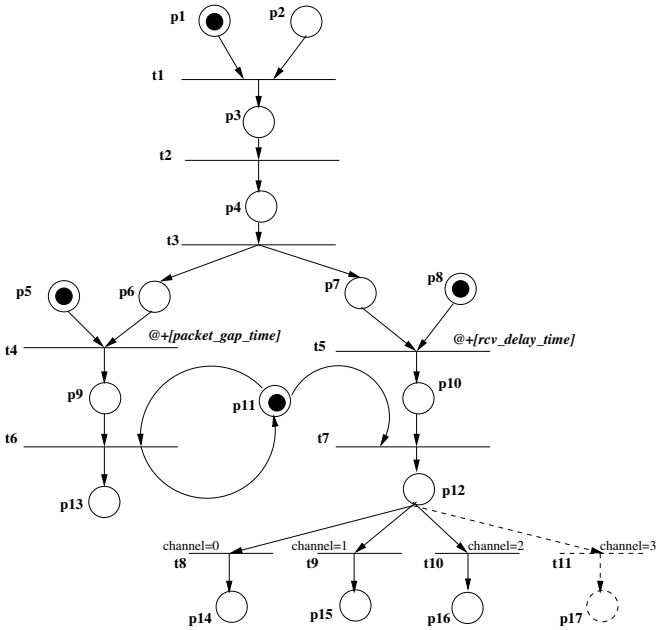

Figure 5. CPN model of the router

4.3 Generation of TLM Tests

There are 11 transitions in the CPN model of the router as shown in Table 1. These transitions are very important since they either affect the control flow or represent a transaction between two components. Our test generation framework generates one temporal logic property for each transition. We use SMV [12] model checker in our framework. The SMV description of the router is generated from the CPN model. The design and the negated version of the properties are applied using SMV. The generated counterexample is analyzed (using

Table 1. Mapping of CPN transitions and TLM actions

$t1$	$input.peek().to\_chan$;
$t2$	Find $packet\_time$, rcv_delay_time and $packet\_gap\_time$
$t3$	fork
$t4$	$receive\_complete\_event.notify(packet\_time+ packet\_gap\_time, SC\_NS)$;
$t5$	$transmit\_complete\_event.notify(packet\_time+ rcv\_delay\_time, SC\_NS)$;
$t6$	$input.nb\_get(tmp\_packet)$;
$t7$	join
$t8$	$chanX.nb\_put(tmp\_packet)$;
$t9$	$slaveX.get\_packet.ok\_to\_get()$;
$t10$	X=0, 1, 2, 3
$t11$	

scripts) to produce the TLM tests. For example, a property corresponding to transition *t9* and the generated TLM test are shown below. The test generation time is in the order of a few seconds using a 1 GHz Sun UltraSparc with 8G RAM.

```
// Property for t9 and its negated version
 t9-property: assert G((pkt.to_chan = 1)
            -> F(slave1.receivedPkt = pkt))
 negate-t9: assert F((pkt.to_chan = 1)
            & G(slave1.receivedPkt ~= pkt))

// TLM Test
 pkt->to_chan = 1;
 pkt->payload_sz = 2;
 pkt->payload[0] = 15;
 pkt->payload[1] = 240;
 pkt->parity = 246;
```

4.4 RTL Interface Specifications of the Router

TLM provides the transaction level information of the design. Therefore, it is necessary to obtain RTL-to-TLM mapping information and RTL input/output timing information to enable TLM-to-RTL test translation. Figure 6 shows the input/output information of the router. This level of information and other details (such as packet description in Figure 7) is used to perform name mapping. For example, "$pkt \rightarrow to\_chan$" in TLM is mapped to "$data[0:1]$" in RTL.

From the block diagram, it is impossible to figure out the clock information of the router. So during the test generation, we need the timing specification of the input and output signals. The timing specification for the router is as follows. All input/output signals are active high and are synchronized to the falling edge of the clock. The *packet_valid* signal has to be asserted on the same clock when the first byte of the packet (the header byte) is driven onto the data bus. Each subsequent byte of data should be driven on the data bus with each new falling clock. After the last payload byte has been driven, on the next falling clock, the *packet_valid* signal must be deasserted (before the parity byte is driven). The packet parity byte should be driven on the next falling clock edge. The router asserts the vld_chan_x ($x \in \{0,1,2\}$) signal when valid data appears on the *channelx* output. The $read\_enb\_x$ input signal must then be asserted on the falling clock edge in which data is read from the *channelx* bus. As long as the $read\_enb\_x$ signal remains active, the *channelx* bus drives a valid byte on each rising clock edge.

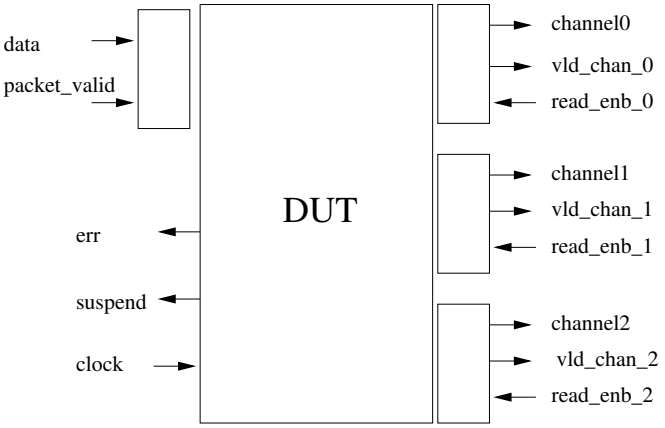

Figure 6. Block diagram of design under test

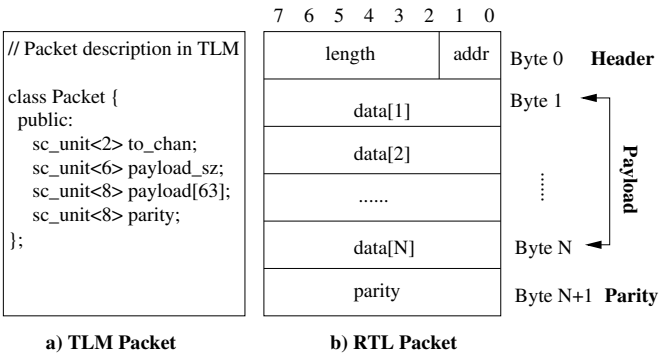

Figure 7. The packet format of the router in TLM and RTL

4.5 Rule based Transformation of TLM-to-RTL Tests

We use the mapping and interface information (described in the previous section) and the transformation rules (described in Section 3.4) to perform TLM-to-RTL test translation. For example, Figure 8 shows the TLM test corresponding to transition t9 (in Table 1) and its RTL counterpart. The first part of the RTL test contains the initialization of the RTL input variables. The second part contains the reset sequence. The third part contains the assignment to *packet_valid* signal. The subsequent entries in the RTL test is generated by transforming corresponding TLM entry by using a combination of name mapping, delay insertion and composition of values (used in one case). Finally, the *packet_valid* signal needs to be low before sending the parity followed by assignment of read enable signals for four time steps (to read four entries: header, two data elements and parity) so that the slave can read the packet.

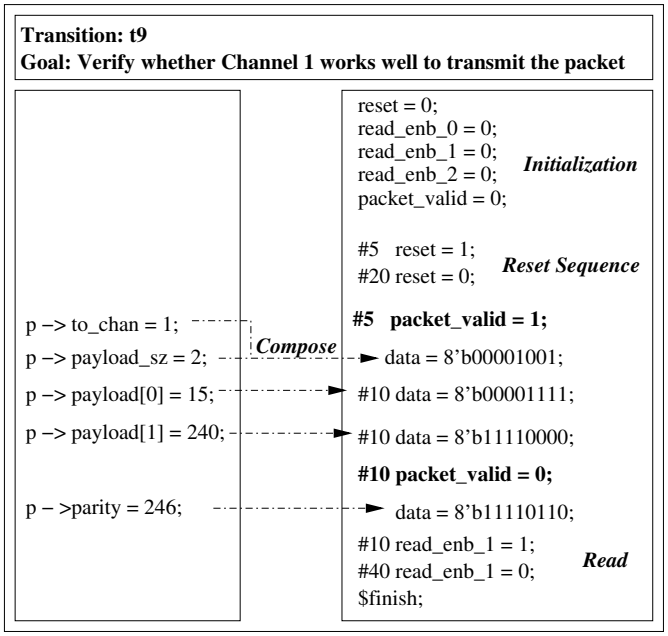

Figure 8. TLM and equivalent RTL tests for transition $t9$

We also generated various other tests (not related to transitions) to increase RTL coverage. These tests are required to cover the additional functionality in RTL that are not available in TLM. For example, TLM does not have explicit notion of FIFO queue and its capacity. Similarly, TLM does not have any notion of *reset* signal. Therefore, we needed to generate tests related to FIFO overflow, reset check, and so on.

4.6 RTL Validation and Analysis

We generated total 15 tests: 11 based on transition coverage and remaining four based on FIFO overflow, reset check, and asynchronous read. We applied all the generated tests on the RTL implementation of the router and measured various coverage metrics using Synopsys VCS tool [13]. Table 2 shows the coverage obtained using the generated tests. Due to some unreachable code and missing "else" statements it was not possible to obtain 100% coverage in all the categories. It is important to note that only 15 tests were able to give the highest possible coverage.

Table 2. RTL coverage using transformed TLM tests

Source	Condition	FSM State (Transition)	Toggle	Path
99.4%	85.1%	100% (100%)	100%	66.7%

We have identified several fatal errors in the RTL implementation. The first error is encountered when a FIFO buffer is empty and slave tries to read the corresponding channel, the empty FIFO buffer becomes full! The second one occurred if the destination of packet is "channel 3". In this case the packet should be discarded, but in RTL the data is written to the "channel 0". Also, one of the test identified an inconsistency between TLM and RTL FIFO implementations. The overflow in TLM level is 16 packets whereas the overflow in RTL is 16 bytes.

5 Conclusions

This paper presented an efficient top-down RTL test generation methodology based on TLM specification. This approach has various advantages. First, the validation result of the SystemC TLM design can be reused in the RTL validation and thereby significantly reduce the RTL validation effort. Second, the generated test contains the information of the system level requirement which is hard to capture in the RTL level without ad-hoc reverse engineering efforts. Third, the test and its accompanying transformation rules enable consistency checking between different abstraction levels. Finally, the RTL test can be ready before the RTL implementation is available.

Clearly, the model checking based approach will not be suitable for test generation of complex SOC examples due to state space explosion. We plan to investigate two complementary directions to address this issue: development of efficient decomposition techniques in model checking based test generation and development of test generation techniques without using model checking.

References

[1] OSCI TLM WG Whitepaper. Transaction Level Modeling in SystemC. Available at http://www.systemc.org.

[2] R. Jindal, and K. Jain. Verification of Transaction-Level SystemC models using RTL Testbenches. *MEMOCODE*, 199–203, 2003.

[3] Z. Wang and Y. Ye. The improvement for transaction level verification functional coverage. *ISCAS*, 5850–5853, 2005.

[4] N. Bombieri et al. Transactor-based Verification for Reusing TLM Assertion and Testbenches at RTL. *DATE*, 1–6, 2006.

[5] S. Abdi, D. Gajski. A formalism for functionality preserving system level transformations. *ASPDAC*, 139–144, 2005.

[6] D. Kroening, N. Sharygina. Formal Verification of SystemC by Automatic Hardware/Software Partitioning. *MEMOCODE*, 101–110, 2005.

[7] M. Moy et al. LusSy: A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level. *Application of Concurrency to System Design*, 26–35, 2005.

[8] D. Karlsson, P. Eles, Z. Peng. Formal verification of SystemC designs using a Petri-net based representation. *DATE*, 1228–1233, 2006.

[9] K. Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Springer, 1995.

[10] J. Peterson. *Petri Nets Theory and the Modeling of Systems*. Prentice-Hall, N.J., 1981.

[11] H. Koo and P. Mishra. Functional Test Generation using Property Decompositions for Validation of Pipelined Processors. *DATE*, 1240–1245, 2006.

[12] Symbolic Model Verifier. Available at http://www-cad.eecs.berkeley.edu/~kenmcmil/smv.

[13] Synopsys. http://www.synopsys.com.