Automatic RTL Test Generation from SystemC TLM Specifications

MINGSONG CHEN - East China Normal University
PRABHAT MISHRA - University of Florida
DHRUBAJYOTI KALITA - Intel Corporation

SystemC Transaction Level Modeling (TLM) is widely used to enable early exploration for both hardware and software designs. It can reduce the overall design and validation effort of complex System-on-Chip (SOC) architectures. However, due to lack of automated techniques coupled with limited reuse of validation efforts between abstraction levels, SOC validation is becoming a major bottleneck. This article presents a novel top-down methodology for automatically generating Register Transfer Level (RTL) tests from SystemC TLM specifications. It makes two important contributions: i) proposes a method that can automatically generate TLM tests using various coverage metrics, ii) develops a test refinement specification to automatically convert TLM tests to RTL tests to reduce overall validation effort. We have developed a tool which incorporates these activities to enable automated RTL test generation from SystemC TLM specifications. Case studies using a router example and a 64-bit Alpha AXP pipelined processor demonstrate that our approach can achieve intended functional coverage of the RTL designs as well as can capture various functional errors and inconsistencies between specifications and implementations.

Categories and Subject Descriptors: B.5.3 [**REGISTER-TRANSFER-LEVEL IMPLEMENTATION**]: Reliability and Testing—*Test generation*; B.7.3 [**INTEGRATED CIRCUITS**]: Reliability and Testing—*Test generation*; D.2.4 [**SOFTWARE ENGINEERING**]: Software/Program Verification —*Model checking*

General Terms: Algorithms, Verification

Additional Key Words and Phrases: Transaction level modeling, model checking, test generation

1. INTRODUCTION

Increasing complexity of SOC architectures makes the demand of the high level abstractions and analysis of SOC designs [Abrar and Thimmapuram 2010]. The functional errors of high level specifications may result in inevitable malfunctions in low level implementations. Therefore it is a major challenge to guarantee the correctness of different abstractions [Bombieri et al. 2007; Bruce et al. 2006]. Validating each abstraction is necessary but time-

Author's address: M. Chen, Software Engineering Institute, East China Normal University, Shanghai 200062, email: mschen@sei.ecnu.edu.cn. P. Mishra, Department of Computer and Information Science and Engineering, University of Florida, Gainesville FL 32611, email: prabhat@cise.ufl.edu. D. Kalita, Intel Corporation, Folsom CA 95630, email: dhrubajyoti.kalita@intel.com

consuming because it requires the profound understanding of the design. In addition, the inconsistency between different abstraction levels and the lack of automation techniques in each level aggravate the overall validation difficulty and workload. It is necessary to develop an approach that can automate the validation of high level abstractions and reuse the validation effort among abstraction levels.

In SOC design, the top-down SOC design process starts from Transaction Level Modeling (TLM) [Cai and Gajski 2003] to Register Transfer Level (RTL) implementation. As a system level modeling specification, SystemC TLM [Rose et al. 2005] establishes a standard to enable fast simulation speed and easy model interoperability for hardware/software co-design. It mainly focuses on the communication between different functional components of a system and data processing in each component. Unlike TLM, RTL contains detailed information (such as interface and timing information) to describe the hardware behaviors. These differences limit the degree of validation reuse between TLM and RTL models. In the absence of significant reuse of design and validation efforts between different abstraction levels, the overall functional validation effort will increase since designers have to verify TLM as well as RTL models. Furthermore, the consistency between different abstraction levels should be guaranteed.

Existing SOC validation techniques for both TLM and RTL designs widely employ a combination of simulation based techniques and formal methods. Simulation based validation uses random or directed test vectors to check the correctness of the design. Certain heuristics are used to generate random tests. However, due to the bottom-up nature and localized view of these heuristics, the generated tests may not yield a good coverage. The directed tests can exactly focus on testing targets thus reducing validation effort since fewer tests can achieve the same coverage goal compared to random tests. A major challenge to enable directed test generation is to automatically extract a formal representation from the TLM specifications and develop an efficient coverage metric that allows coverage-driven directed test generation. The goal of directed testing 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. Furthermore, complete reuse of such TLM tests will lead to a drastic reduction of RTL validation effort as well.

In this article, we propose a top-down directed test generation methodology for both TLM and RTL designs. The basic idea is to use TLM specifications to perform coverage based TLM test generation and generate RTL tests from TLM tests using a set of transformation rules. A tool which incorporates this flow has been developed to enable automated directed RTL test generation from SystemC TLM specification. Our framework first automatically extracts a formal model from the TLM specification. Then based on the coverage of the fault models, a set of properties indicating the validation adequacy are derived. The counterexamples generated from derived properties can be used as the tests for the TLM validation. Finally, the generated TLM tests can be converted to RTL tests by applying our proposed Test Refinement Specification (TRS) which defines the rules to construct the relation between the TLM and RTL models. Because the generated TLM and RTL tests check the same functionality of the system, so essentially they can ensure the consistency between TLM specifications and RTL designs.

This article makes two major contributions: First, we propose a method which enables automatic TLM test generation using various coverage criteria. Second, we develop the TRS that can convert TLM tests to RTL tests. The rest of the article is organized as follows.

Section 2 describes related work addressing TLM-based validation approaches. Section 3 presents our test generation methodology followed by case studies in Section 4. Finally, Section 5 concludes the article.

2. RELATED WORK

Compared to RTL designs, TLM provides a rapid prototyping platform for the architecture exploration and hardware/software integration [Ghenassia 2005]. To guarantee the correctness of SystemC TLMs in the top-down design flow, there are two kinds of techniques: i) simulation based techniques using random and constrained random tests, ii) formal verification based techniques such as model checking, and iii) hybrid techniques that combine the simulation and formal verification techniques.

Simulation based methods validate system using test vectors. They terminate when the required testing adequacy is achieved. Wang et al. [2005] described a coverage directed method for transaction level verification. The approach is based on random test generation and the coverage is increased by using fault insertion method. Although simulation is fast, it is difficult to automate the test generation process. To enable automated analysis, various researchers have tried to extract formal representations from SystemC TLM specifications. Abdi et al. [2005] introduced *Model Algebra*, a formalism for representing SOC designs at system level. The work by Kroening et al. [2005] formalized the semantics of SystemC by means of labeled Kripke structures. Moy et al. [2005] provided a compiler front-end that can extract architecture and synchronization information from SystemC TLM design using HPIOM. Karlsson et al. [2006] translated 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. Habibi et al. [2006] proposed a method that adopts the formal model AsmL. A state machine generated from AsmL can be verified, and then can be translated to both SystemC code and properties for low level verification. All these modeling techniques focus on the formal modeling and translation of SystemC specifications rather than directed test generation. It is hard to guarantee the correctness of the given specifications. Our test case generation approach is different from above verification techniques since it is based on property falsification. The assumption of our method is that the given specification is correct. Therefore, we can get one directed test for each false property.

Completeness is an important issue during the validation. Since formal approaches are good at handling corner cases, hybrid techniques can converge to the required coverage quickly. As a hybrid method, Assertion Based Verification (ABV) using Property Specification Language (PSL) [IEEE-1850] is accepted as a promising approach for functional validation. Lahbib et al. [2005] discussed the issues faced within SystemC environments to incorporate PSL assertions. It also proposed an automatic solution that enhances SOC system level design flow with PSL assertions embedded into SystemC designs. Habibi et al. [2004] presented a method to efficiently verify SystemC assertions. It is based on both static code analysis and genetic algorithms to optimize test generation to get more efficient coverage of assertions. However, due to lacking of coverage metrics, hybrid methods often fails to guarantee the overall correctness of system implementations because of incomplete properties. To address this problem, Fin et al. [2003] proposed a SystemC framework that can evaluate property validation incompleteness in a completely automatic way. For a large complex system, it is necessary to efficiently handle coverage metrics. Fedeli et al. [2007] presented a methodology based on a combination of static and dynamic verification

which can reduce the property evaluation time. However, such coverage metrics are for RTL designs only. Inspired by their work, we propose two fault models for the transaction coverage evaluation in our method.

Reusing validation effort between abstraction levels can reduce the overall validation time. There are various researches on validation reuse between TLM and RTL levels. Bombieri et al. [2006] showed 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. They also presented an incremental ABV methodology [Bombieri et al. 2007] to check the correctness of TLM-to-RTL refinement by reusing assertions. Jindal et al. [2003] presented a method to reduce the verification time by reusing earlier RTL testbenches. Ara et al. [2003] proposed an approach which combines transaction level languages (e.g. SystemC) and the RTL level language (e.g. Verilog) based on Component Wrapper Language (CWL). By defining various test patterns using CWL, RTL verification suites from original specifications can be quickly generated. Thus it can yield much shorter verification periods versus conventional methods.

As described above, most existing TLM validation approaches are focusing on system level validation or validation effort reuse. To the best of our knowledge, our methodology is the first attempt to automatically bridge the validation gap between TLM and RTL designs.

3. RTL TEST GENERATION FROM TLM SPECIFICATIONS

Figure 1 shows the framework of our RTL test generation methodology. This methodology has three important steps: i) translating SystemC TLM to formal SMV specifications, ii) deriving properties based on proposed fault models to enable automated test generation, and iii) refining TLM tests to RTL tests using our proposed TRS. It is important to note that the test refinement is independent of how TLM tests are generated. In other words, test refinement can accept TLM tests generated by other approaches such as random test generation. The generated TLM tests can be used to validate TLM specifications. The refined RTL tests can be applied on the RTL implementation for functional validation.

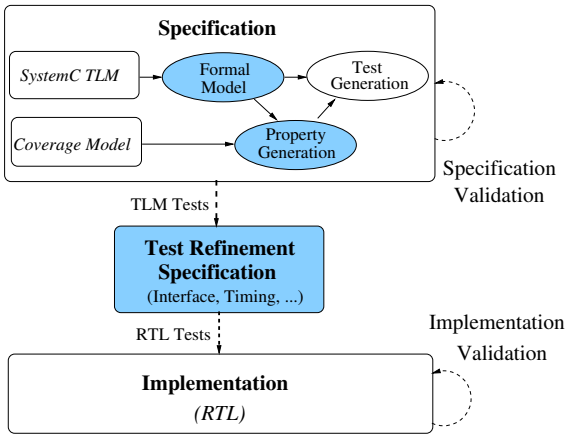

Fig. 1. Proposed RTL test generation methodology

In this section, we demonstrate the processes of TLM to SMV transformation as well as test refinement using a router example (details are described in Section 4.1). The remainder

of this section is organized as follows. Section 3.1 describes the formal model that is used during the TLM to SMV translation. Section 3.2 presents the procedure for converting TLM specifications into SMV descriptions. Section 3.3 outlines our automated property generation and TLM test generation approaches. Section 3.4 presents our TLM to RTL test translation using TRS. Finally, Section 3.5 gives a brief introduction to our prototype tool which incorporates our methodology.

3.1 Formal Modeling of SystemC TLMs

As a high level specification, SystemC TLM emphasizes the functionality of the data transfers instead of actual implementation. A SystemC TLM design interconnects a set of processes communicating with each other using transaction data token (i.e., C++ objects). The initial process starts a communication, and the target process passively responds to the communication. Similar to the producer/consumer models, each process does the following tasks: consuming data, processing data and producing data.

Since SystemC is based on C++, it supports various programming constructs (e.g., template, inheritance, etc.). Although the concept of some TLM components (signals, ports, etc.) is easy, their C++ implementation details are really complex. Therefore, directly translating their behaviors to enable automated validation is difficult. In our framework, we abstract such SystemC components and hide the implementation details using the pre-defined SMV constructs. Furthermore, the underlying complex SystemC scheduler aggravates the modeling complexity. For SystemC TLM, to mimic the parallel execution of processes, the SystemC scheduler activates the *ready-to-run* processes in a "nondeterministic" way. However, since SMV is parallel in essence, it is not necessary to model the SystemC scheduler explicitly.

For TLM, two most important factors are transaction data and transaction flow. So the extracted formal model of TLM specifications should reflect both information. In our test generation framework, it is required that the extracted models can not only guide the generation of SMV specifications, but also can be used to automatically derive the properties for TLM test generation. Definition 1 gives the formal model representation of SystemC TLM designs. It is based on a simplified version of Colored Petri-net [Jensen 1997].

DEFINITION 1. *The* **formal model** *of a SystemC TLM design is an eight-tuple (Σ, P, T, A, E, M, I, F) where*

(*1*) Σ *is a set of transaction data tokens.*

(*2*) $P = \{p_1, p_2, \ldots, p_m\}$ *is a set of places.*

(*3*) $T = \{t_1, t_2, \ldots, t_n\}$ *is a set of transitions.*

(*4*) $A \subseteq \{P \times T\} \cup \{T \times P\}$ *is a set of arcs between places and transitions.*

(*5*) $E = \{e_1, e_2, \ldots, e_k\}$ *is a set of arc expression. The mapping Expression(a_i) = e_i ($a_i \in A$, $1 \leq i \leq k$) gives the enable condition e_i for a_i. A token can pass arc a_i only when e_i is true.*

(*6*) $M : 2^{P \times \Sigma} \times T \rightarrow 2^{P \times \Sigma}$ *is a function that describes the internal operations on input transaction data and output transaction data of a transition.*

(*7*) $I \in 2^{P \times \Sigma}$ *specifies the initial state.*

(*8*) $F \subseteq 2^{P \times \Sigma}$ *specifies the final states.*

We use this formal model as an immediate form to capture the execution as well as interconnection of processes. In our framework, each TLM data is described by a transaction data token, each TLM module is described by a transition, and each interconnection (port, channel, etc.) between two TLM modules is described by a place. As an example, Figure 2a) shows an interconnection of six modules. Each arrow indicates a port binding between two modules. Figure 2b) shows the graph representation of its corresponding formal model. In the formal model, each circle is called a *place* that is used to indicate the input or output buffer of a module. It can temporarily hold the transaction data for later processing. The vertical bars are *transitions* which are used to indicate modules which contain processes to manipulate input and output transaction data tokens. The places without incoming arcs are *initial places* which start a transition. The places without outgoing arcs are *target places*. Transaction data tokens flow from the initial places to the target places and token values may change in transitions when necessary. The transaction flow is synchronized by the transition, and the internal logic of a transition determines the flow of the transaction.

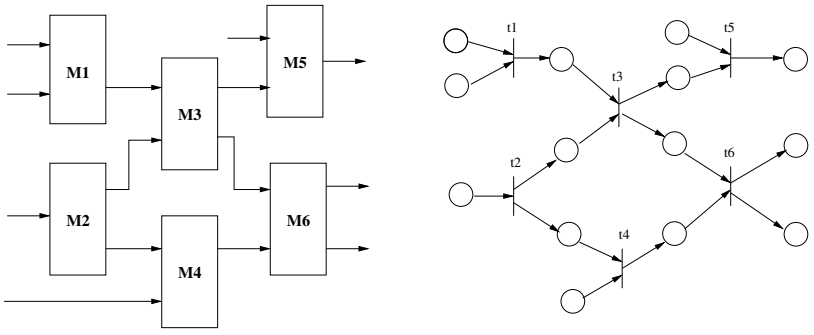

Fig. 2. Mapping from a SystemC structure to its corresponding formal model

3.2 Transformation from SystemC TLM to SMV

Model checking techniques are very promising for directed test generation in hardware and software domains [Ammann et al. 1998; Beyer et al. 2004; Kupferman et al. 1999; Mishra and Dutt 2008]. In our framework, we adopt SMV as the formal specification to describe both the structure and behavior information of SystemC TLMs because of the following reasons. First, the underlying semantics of SMV is similar to the semantics of SystemC scheduler. So we can mimic most TLM's behaviors using SMV without modeling complex scheduler behavior. Second, SMV and TLM have the similar structure hierarchy. Each processing unit encapsulated by a TLM module corresponds to a SMV module. The interconnections (e.g. channels, ports and sockets) between TLM modules can be abstracted by using module parameters in SMV. Third, like SystemC, SMV provides a rich set of programming language constructs such as *if-then-else*, *case-switch* and *for loop* statements. Fourth, SMV main module connects, similar to SystemC, each component of the system. Finally, SMV supports various kinds of data types and data operations. Especially users can define their own data type. All of these SMV features facilitate the translation from TLMs to SMV specification.

Currently, there are many complex TLM constructs such as FIFO channels, direct memory interface, etc., which cannot be mapped to their SMV counterparts directly. This is mainly due to the limitation of the description power of SMV model checker. To the best of our knowledge, so far only the synthesizable code can be automatically translated to its corresponding SMV code by some verification tools such as $vl2smv$ (the translator from Synchronous Verilog to SMV) [Cadence Berkeley Labs]. Not all the TLM constructs can be easily synthesized to low level implementations (i.e., RTL). Therefore our TLM-to-SMV translation framework only supports the translation of a small sub-set of TLM constructs which are synthesizable. It is important to note that our method for automated test generation is based on model checking techniques. To scale down the complexity, it is necessary to apply abstraction on complex SystemC TLM components. Since the standard TLM components are pre-defined and assumed to be correct, we also pre-define the corresponding SMV counterparts and use them directly during the translation. Furthermore, due to the expressiveness of the SMV language, currently our framework just supports loosely timed modeling. We are planning to use the timed automata checker (such as UPPAAL [Larsen et al. 1997]) in our framework to enable the timing verification of transactions.

As an intermediate form for TLM to SMV translation, the formal model provides both structure and behavior information. Such information need to be collected for a translation to a SMV representation to enable automated directed test generation. The structure information includes the data type definition and connectivity between modules. It corresponds to the description of transaction data token as well as interconnection of transitions and places in the formal model. The behavior information contains token processing and token routing. In the formal model, it represents the internal processing of a transition. This section discusses how to extract both structural and behavioral information and transform it to a SMV specification.

3.2.1 *Structure Extraction.* Before simulation, SystemC does the $elaboration$ which creates required data structures to support the simulation. During elaboration, all the parts of system hierarchy (modules, ports, channels, processes and etc.) are created, and ports and exports are bound to channels or to each other. When parsing TLM specifications, our transformation procedures will extract all such information to construct the skeleton of SMV specifications.

```
class packet{                    typedef packet struct{
public:
  sc_uint<2> to_chan;              to_chan : 0..3;
  sc_uint<6> payload_sz;           payload_sz : 0..63;
  sc_uint<8> payload[4];           payload : array 3..0 of 0..255;
  sc_uint<8> parity;               parity : 0..255;
};                               }
 a)packet in SystemC TLM           b) packet in SMV
```

Fig. 3. An example of data type transformation

In TLM, the content of a transaction data token indicates the transaction flow and the output of each component. So it is the key part of TLM tests. Generally a transaction token consists of several attributes with different data types. Because data type determines the size of the specified variable which in turn affects the model checking performance, it is necessary to figure out the data type of a token. Besides all native C++ types, SystemC defines a set of data type classes within the namespace sc_dt to represent values with

application-specific word lengths applicable to digital hardware. SMV also supports various data types such as array, Boolean, integer, struct and so on. Such data type definitions facilitate the mapping of data types between SystemC TLM and SMV specification. During the transformation, the word length of user-defined data type need to be considered. For example, $sc\_uint < 2 >$ has 2 bits and will be transformed to a range $0..3$ in SMV. Figure 3 gives an example of the router $packet$ in the form of SystemC TLM and SMV respectively.

Derived from the base class *sc_module*, TLM modules are the main processing units for the transaction data. Generally each *sc_module* contains the definitions of processes whose types are *SC_METHOD* or *SC_THREAD*. Modules communicate with each other by sending and receiving transaction data tokens via output and input ports. SystemC provides a communication wrapper for the system components (modules). In SystemC, there exists various binding mechanism (e.g. port to export binding, export to export binding and port to channel binding) to establish interconnection between modules. Usually each binding corresponds to a channel such as a first-in-first-out (FIFO) channel to temporarily hold transaction tokens.

```
class router : public sc_module{
 public:
     sc_export<tlm_put_if<packet> > packet_in;
     sc_export<tlm_fifo_get_if<packet> > packet_out0;
     sc_export<tlm_fifo_get_if<packet> > packet_out1;
     sc_export<tlm_fifo_get_if<packet> > packet_out2;

     router(sc_module_name module_name);
     void route();
 private:
      tlm_fifo<packet> chan0, chan1, chan2, input_;
      packet tmp_packet;
};
```

Fig. 4. An example of SystemC TLM module

Figure 4 shows the TLM module structure of a router. The class sc_export can be used as a port to communicate with other modules. Because the interface type of port $packet\_in$ is $tlm\_put\_if<packet>$, it is an input port. In contrast, $packet\_outx$ (x=0,1,2) have the interface $tlm\_fifo\_get\_if<packet>$, so they are output ports. During the router communication, each connection between a port and an export uses a FIFO channel to temporarily hold a packet.

Structurally similar to SystemC TLMs, SMV specification is also modularized and hierarchically organized. So the extraction of structure information needs to map the TLM constructs into the right place of the SMV specification. Figure 5 shows the SMV module skeleton corresponding to example in Figure 4 after the structure extraction. In SMV, a module uses the parameters as the input and output ports to both communicate with other modules and configure the system status defined in the $main$ module. In the example of Figure 5, the SMV module has one input port and three output ports. The type of the input and output ports is packet. All the declarations of member variables except for the FIFO channels are declared in the SMV specification. Because a FIFO channel together with its port pairs are abstracted as a SMV parameter, it is not necessary to create a variable in SMV explicitly. Based on context during the elaboration, some of the declared variables

will be initialized. In SMV specification, each output ports and local variables need to be initialized. For example, $packet\_out0$ is a parameter which refers to an output port, so it will be initialized with a value "0". In our framework, it is required that all such module connections should be defined in the module sc_top.

```
module router(packet_in, packet_out0, packet_out1, packet_out2){
    input   packet_in : packet;
    output  packet_out0, packet_out1, packet_out2: packet;
    tmp_packet : packet;

    init(packet_out0):=0;
    init(packet_out1):=0;
    init(packet_out2):=0;
    init(tmp_packet):=0;
    ......
}
```

Fig. 5. An example of SMV module

3.2.2 *Behavior Extraction.* TLM behavior describes the run-time information of TLMs including transaction creation, transaction manipulation and module communication. Transaction creation initializes a transaction by creating a data token (i.e., a C++ object) with proper values. Transaction execution describes the transaction flow among the modules. A module is a container which has a cluster of relevant processes. Such processes will handle the incoming transaction tokens and decide where to send them according to the specified conditions. Thus different value of a token will lead to different transaction flows. In our current prototype release, there are two kind of process communication supported in transaction flows: 1) direct procedure call from one process to another process, and 2) channel-based events triggered by the procedure call. For example, in the blocking mode, a process can fetch a transaction data token from the specified input port only when the corresponding channel is not empty. Otherwise, the operation "get" will be blocked until there is an event triggered by the "put" operation by other processes.

Figure 6 gives the module process $route$ of the router example. The process receives a packet from the driver via channel $input\_$, and then it decides where to send data based on the packet header information to_chan.

```
router::router( sc_module_name mname ): sc_module(mname){
  packet_in(input_);   packet_out0(chan0);
  packet_out1(chan1);  packet_out2(chan2);
  SC_METHOD(route);
  sensitive << input_.ok_to_get();
  dont_initialize();
}
void router::route() {
  input_.nb_get(tmp_packet);
  if(tmp_packet.to_chan == (sc_uint<2>)0)
     chan0.nb_put(tmp_packet);
  else if(tmp_packet.to_chan == (sc_uint<2>)1)
     chan1.nb_put(tmp_packet);
  else chan2.nb_put(tmp_packet);
}
```

Fig. 6. An example of TLM process

TLM modeling provides some synchronization mechanism for the communications between modules. As shown in Figure 6, the router can fetch the data from the FIFO queue $input\_$ only when the driver put a package and the FIFO channel event ok_to_get is triggered. Thus the synchronization between two modules is implicitly achieved.

SMV supports many constructs similar to the common programming language such as *if-then-else*, *switch-case* and *for loop*. So these constructs facilitate the behavior modeling of processes from TLM to SMV specification. Figure 7 is the translated SMV specification of the TLM example presented in Figure 6. During the translation from TLM to SMV, we abstract a channel as an implicit buffer between two ports. So a SMV module will get the input data from its input ports. There is no mapping of the channel in transformed SMV specification. For example, the tmp_packet is assigned with the value of $packet\_in$ instead of the value of $input\_$ shown in the TLM example of Figure 6.

```
module router(packet_in, packet_out0, packet_out1, packet_out2){
    ......
    next(tmp_packet) := packet_in;
    if(tmp_packet.to_chan = 0){
        next(packet_out0) := tmp_packet;
        next(packet_out1) := 0;
        next(packet_out2) := 0;
    }else if(tmp_packet.to_chan = 1){
        next(packet_out0) := 0;
        next(packet_out1) := tmp_packet;
        next(packet_out2) := 0;
    }else{
        next(packet_out0) := 0;
        next(packet_out1) := 0;
        next(packet_out2) := tmp_packet;
    }
}
```

Fig. 7. An example of SMV process

3.3 Automatic TLM Test Generation

For model checking based testing, a test is derived from the counterexample of a false safety property. A safety property in the temporal logic form *!F(p)* asserts that a specified scenario cannot happen (i.e. property *p* cannot be true). Otherwise a counterexample which explains the reason of the error will be reported by a model checker. In other words, such counterexample can then be extracted to a test to validate the specified scenario. In our method, the quality of the generated TLM tests is determined by the corresponding properties. So during the property generation, it is required to guarantee that the generated properties can sufficiently validate the system. This sub-section first proposes fault models to enable the automatic property generation. Then we introduce the TLM test generation method using model checking.

3.3.1 *Property Generation Based on Fault Models.* The coverage metrics play an important role in testing to indicate the testing adequacy. Test generation using model checking techniques requires that the automatically generated properties can cover as many desired scenarios in the design as possible. In our framework, properties are derived from a *fault model* which represents a complete set of specific errors. Each *fault* in the fault model indicates a potential "design error" which can be described by a temporal logic property.

The test generated from such property can be applied on the design to check the specific scenario (negation of the fault). For example, when validating a desired scenario described by a LTL formula p, we use the negation $!\ p$ as a fault. By checking the property $!\ p$, we can derive a test to check the scenario where property p holds.

The properties generated from a fault model can guarantee the specific fault coverage in a design. Therefore the testing coverage can be assured. In other words, a proper fault model with a good fault coverage determines the success of TLM test generation. Our TLM fault models are inspired by the fault model based on *bit failures* and *condition failures* proposed in [Ferrandi et al. 1999]. All such fault models are simple but effective. They can easily obtained by anlyzing the syntax of TLM models. In our method, we did not consider complex functional scenarios like "if communication C1 occurs before communication C2, then condition C3 will hold until communication C4 is asserted". This is because the major concern of our method is the automation for directed test generation. By parsing the syntax of OSCI TLM models, it is difficult to figure out the complex dynamic semantics of a design automatically. However, our framework does not exclude any properties written manually. The verification engineer can insert their properties after the SMV file generation and the corresponding TLM and RTL tests can be generated automatically as well.

In TLM, transaction data and transaction flow are two most important aspects. They indicate both the structure and behavior information. So in our framework we defined two fault models based on them as follows.

(1) **Transaction data fault model** investigates the content of the variables relevant to the transaction. For each variable, it is assumed that a specific value cannot be assigned in a faulty scenario.

(2) **Transaction flow fault model** investigates the controls along the path where the transaction flows. For each transaction path, it is assumed that it cannot be activated in a faulty scenario.

Transaction data fault model deals with the possible variable assignments for each part of transaction data. However, in the property generation, due to the large size of value space, trying all the possible values of a data is time-consuming and impossible. In our experiment, we use the data bit fault model which checks each bit of a variable respectively. These model can not only partially guarantee the TLM data content coverage, but also increase the toggle coverage for the corresponding RTL designs. Since a transaction flow is a sequence of transactions, it can be used to reason the transaction ordering indirectly. Transaction flow fault model deals with the controls along the transaction flow. To ensure transaction flow coverage, all the branch conditions like *if-then-else*, *switch-case* statements along the transaction flow should be investigate. The goal is to check all possible transaction flows. It is important to note that the above models are not golden models. It is allowed that users can provide their own fault models to derive false properties for test generation. Based on the router example shown in Section 4.1, Figure 8 presents two examples for these two fault models.

```
P1: For variable temp4, the first bit of parity cannot be 1.
  LTL formula: !F (temp4.parity = 1 & temp4.to_chan != 0
        & temp4.payload_sz != 0);
P2:The condition ``tmp_packet.to_chan=1'' cannot be true.
  LTL formula: ! F(my_router.tmp_packet.to_chan = 1);
```

Fig. 8. Examples of two kinds of faults and corresponding properties

3.3.2 *TLM Test Generation using Model Checking.* Model checking falsification algorithm is promising for automated generation of directed test [Kupferman et al. 1999; Mishra and Dutt 2008]. The algorithm has two inputs: i) model of the design in SMV specification and ii) a set of properties derived from the specified fault models described in Section 3.3.1. During test generation, the model checker will generate one counterexample for each property. The generated counterexample is a sequence of variable assignments which can be transformed to a TLM test. Figure 9 shows an example of generated TLM test based on a transaction flow fault of the router example. It is derived from the condition of an "if-then-else" statement of the router example shown in Section 4.1. By applying this test on the TLM specification of the router example, the specified condition will be activated.

```
// The property of a transaction flow fault
assert  ! F (my_router.tmp_packet.to_chan = 1);
// TLM Test
 p->to_chan = 1;
 p->payload_sz = 4;
 p->payload[0] = 128;
 p->payload[1] = 0;
 p->payload[2] = 0;
 p->payload[3] = 0;
 p->parity = 132;
```

Fig. 9. The TLM test for a transaction control fault

Clearly, model checking based approach may be time-consuming in the presence of complex designs and properties. In these circumstances, various learning [Strichman 2001; Chen and Mishra 2010; Chen et al. 2010] and decomposition [Koo et al. 2009] based optimization approaches can be used to reduce the overall complexity of test generation.

3.4 Translation from TLM Tests to RTL Tests

A major challenge in test translation is how to bridge abstraction gap between TLM and RTL. For the same TLM specification, RTL designs may differ because of input/output definitions, timing details, programming styles and so on. So when converting TLM tests to RTL tests, it is required to provide necessary information such as the input/output mappings between TLM and RTL as well as timing details of RTL input signals. For example, "$p \rightarrow to\_chan$" in TLM is mapped to an input signal for "$DATA[1:0]$" in RTL.

In our framework, we developed the language TRS which allows specifying rules for TLM to RTL test transformation. Since TLM tests only reflect the transaction data information, our RTS can analyze the transaction data in TLM tests and generate the corresponding RTL tests which are consistent to the interface protocol. One might argue that it may be easier to write RTL tests than writing TRSs. However, for the TLM tests which will be refined to the same RTL components, they share the same RTL input/output interface protocol. Generally, for each testing component, we will generate a large set of TLM tests. Most of them are only different with transaction data values. In other words, a large cluster of TLM tests can share one TRS. Therefore we just need to write several TRSs to cover all the testing scenarios which is time-efficient. In addition, the repeated sub-scenarios can be reused across TRSs. The overall automatic RTL test generation time can be significantly reduced. Generally, the TRS contains the following three parts:

Input/Output Mappings specify the correspondence between TLM I/O variables and RTL I/O signals.

Patterns are templates which define small segments of the test behavior. It can be used to compose various testing scenarios.

Timing Sequence describes a complete scenario of input signals with timing information.

In this section, we discuss each part of TRS in details with illustrative examples. All these examples are based on the router example shown in Section 4.1.

3.4.1 *Input/Output Mappings.* During the TLM to RTL test translation, one important step is how to map TLM test data to its corresponding RTL test stimulus. Because of the difference between TLM data and RTL data, in the mapping, we need to give the size information of each RTL signal as well as the bit correspondence between TLM data and RTL data.

In each mapping rule, the left hand side is the RTL data declaration, and the right hand side is the bit mapping from TLM data to RTL data. Our TRS language allows the user to specify the RTL data using the concatenation of several TLM data. Also it supports the mapping from an array of TLM data to an array of RTL data. Figure 10 gives an example of the data mappings. In the example, $parity$ is a RTL data with 8 bits. It maps to the TLM variable $packet.parity$. The $header$ is a RTL data whose most significant six bits correspond to the TLM data $payload\_sz$ and the least significant two bits correspond to the TLM data to_chan. The RTL data $payload$ is an array where the width of each element is 8 bits. The $i'th$ element $payload[i]$ corresponds to the $i'th$ element of the TLM data $packet.payload[i]$.

```
mapping_def:
   bit[7:0] parity = packet.parity;
   bit[7:0] header = {packet.payload_sz[7:2], packet.to_chan[1:0]};
   bit[7:0] payload[0..packet.payload_sz-1]
                = packet.payload[0..packet.payload_sz-1];
end_mapping_def
```

Fig. 10. An example of mapping between TLM data and RTL data

3.4.2 *Patterns.* When writing tests, some sub-scenarios may occur several times. To enable the reuse of segments of a scenario, TRS introduces the construct $pattern$ to group several statements together. Essentially, like a macro, the content of a pattern will substitute for the pattern statements in the timing sequence. Thus the usage of the pattern can reduce the programming time as well as increase the programming flexibility.

```
pattern reset()
    #5  RST = 1;
    #20 RST = 0;
end_pattern

pattern slave_read(int slave_no, int enable)
    #10 ENB%slave_no = %enable;
end_pattern
```

Fig. 11. Two examples of patterns

In TRS, a pattern can have parameters. During pattern text substitution, the tags defined in patterns will be replaced with the given value of parameters. Figure 11 presents two examples of patterns *reset* and *slave_read*. The pattern *reset* has no parameters. So its content will be directly embedded at the place of the pattern statement. The pattern *slave_read* has two parameters to indicate which slave will be enabled.

3.4.3 *Timing Sequence.* The timing sequence in TRS composes a sequence of statements and pattern instances to describe a testing scenario. According to the definition of input/output mappings and patterns, the compiler will translate testing scenarios described in timing sequence to corresponding RTL tests. Figure 12 presents an example of a timing sequence. It describes a testing scenario of the packet delivering for a router as follows: i) a master sends a packet to a router, ii) the router holds the packet and notifies the corresponding slave to fetch the packet, and iii) the slave receives the packet.

```
SPEC router(packet)
    ......
    main:
     begin
       initialize();
       reset();
       -- the master sends a packet
       #5 PKT_VALID = 1'b 1;
       DATA = header;
       for(int i=0; i<packet.payload_sz; i++){
           #10 DATA = parity[i];
       }
       #10 PKT_VALID = 1'b 0;
       DATA = parity ;
       -- a slave receives the packet
       slave_read(packet.to_chan, 1);
       FINISH();
     end
END_SPEC
```

Fig. 12. An example of a timing sequence

3.5 *ARTEST*: A Prototype Tool for TLM-to-RTL Test Generation

We developed a prototype tool *Automatic RTL Test gEnerator from SystemC TLM* (*ARTEST*) which incorporates the proposed methods. Figure 13 shows both the structure and workflow of our tool. The following sub-sections will present its three key components: i) *TLM2SMV* for SMV model and property generation, ii) TLM test generation using model checking, and iii) *TLM2RTL* for RTL test generation.

3.5.1 *TLM2SMV.* Implemented based on the C++ parser Elsa [McPeak], *TLM2SMV* can automatically translate the SystemC TLM to a SMV specification and derive properties based on the fault models. Due to the complex data type definition and complex constructs defined in SystemC TLM library files, direct translation to SMV will cause the state space explosion. So in our tool, we simplify such definition and predefine them for SMV transformation. For example, we restrict the queue size for TLM FIFO channels. In SystemC, an integer is 32-bit (with $2^{32}states$). However, we reduce its size to 8 bits (with $2^{8}states$) during the SMV transformation.

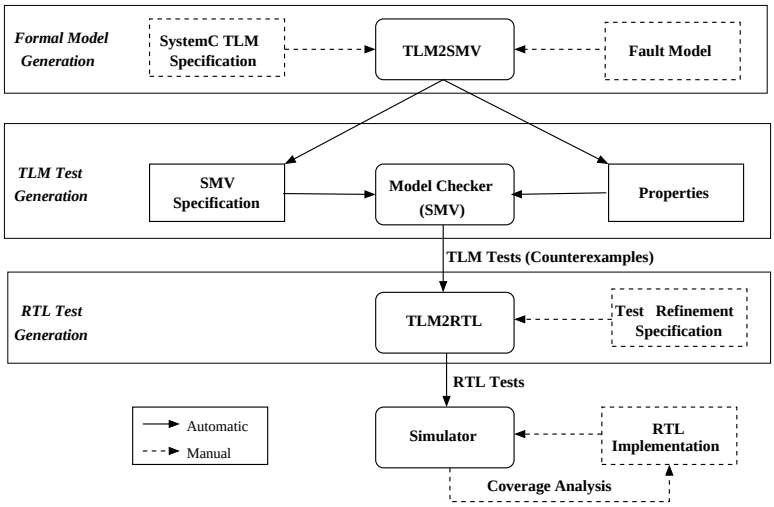

Fig. 13. The structure of our prototype tool

Before the TLM to SMV translation, preprocessing procedure of *TLM2SMV* will do the following three tasks: i) eliminate the header files and the comments, ii) add the necessary predefine constructs and iii) convert the data type if necessary. Then *TLM2SMV* will start to transform the TLM specification. As described in Section 3.2, *TLM2SMV* will extract both static and dynamic information. At the mean time, it also explores the information such as transaction relevant data, branch conditions for the property generation. Finally based on the collected information, we can get both a formal specification in SMV and properties derived by specified fault models. By using Cadence SMV verifier [Cadence Berkeley Labs], we can get a set of counterexamples. The TLM tests are extracted from these counterexamples.

3.5.2 *TLM Test Generation.* When a specified safety property is false, SMV model checker will generate a counterexample to falsify it. A generated TLM counterexample is in the form of a sequence of state assignments. This sequence starts from first state (initial state) and ends at the error state which violates the property. If the Cone of Influence (COI) is enabled during the property checking, each state will only contain the variables which are relevant to the specified property. The generated counterexample is refined to produce the TLM test.

3.5.3 *TLM2RTL.* Because SystemC TLM focuses on the system level modeling, the generate TLM tests lack the implementation level knowledge. So the generated TLM tests are different with RTL tests and cannot be directly used to validate RTL implementation. For example, most loosely timed TLM models are too abstract and assume that a transaction happened in one or a sequence of function calls. However, a RTL design has much more pins and it needs the detailed timing information for each signal. In our framework, the user should provide a RTS which provides the mapping rules for the TLM to RTL test translation. With the generated TLM tests and the TRS as inputs, the *TLM2RTL* can translate the TLM tests to RTL tests. Finally, the coverage of the TLM implementation will be reported when simulating the generated RTL tests on the RTL design.

4. CASE STUDY

We applied our method on various practical examples. In this section, two case studies are presented to show the effectiveness of our method. The results are obtained while running our tool on a 2 GHz AMD Opetron Processor with 8G RAM using Linux operating system.

4.1 A Router Example

Figure 14 shows the TLM structure of the router. The router consists of five modules: one master, one router and three slaves. It consists of 4 classes, 8 functions, and 143 lines of code. The main function of the router is to analyze and distribute the packets received from the master to target slaves.

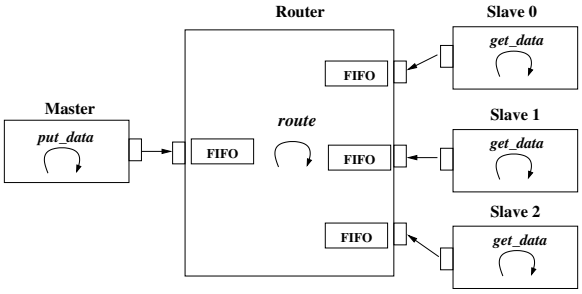

Fig. 14. The TLM structure of the router

At the beginning of a transaction, the master module creates a packet which is in the form as shown in Figure 15a). The packet consists of three 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. Then, the driver sends the packet to the router for package distribution. The router has one input port and three output ports. Each port is connected to a FIFO buffer (channel) which temporarily stores packets. The router has one process $route$ which is implemented as a *SC_METHOD*. The $route$ first collects a packet from the channel connected to the driver, decodes the packet header to get the target address of a slave, and then sends the packet to the channel connected to the target slave. Finally, the slave modules will read the packets when data is available in the respective FIFOs. The transaction data (i.e. packet) flows from the master to its target slave via the router. The flow is determined by the address to_chan in the packet header.

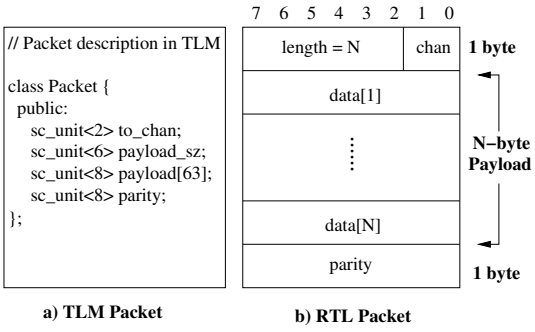

Fig. 15. The packet format of the router in TLM and RTL

In the following sub-sections, we present the workflow of the RTL test generation and provide the validation result of the router implementation.

4.1.1 *RTL Tests Generation.* By using the tool *ARTEST*, we can get the SMV input from the SystemC TLM specification described in Section 3.2. Also according to different fault models defined in Section 3.3.1, we can generate a set of properties. For each property, we can derive a test from its counterexample. As a high level modeling language, SystemC TLM lacks the information of pins and timing in low level implementation. The generated TLM tests are not appropriate as the inputs of RTL designs. Therefore, it is necessary to provide an interface mapping to enable TLM-to-RTL test translation.

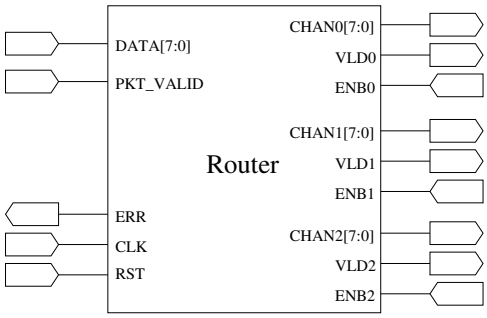

Fig. 16. Block diagram of design under test

Figure 16 shows the input/output interfaces of the router. This RTL information and other TLM details (such as packet description in Figure 15) are used to perform mapping between TLM variables and RTL signals. For example, "$packet.to\_chan$" in TLM corresponds to the RTL data "$header[1 : 0]$" and "$packet.payload\_sz$"corresponds to "$header[7 : 2]$". And the array of TLM data $packet.payload$ will be mapped to RTL data "$payload$". Such information should be defined in the *mapping definition* of TRS as shown in Figure 10.

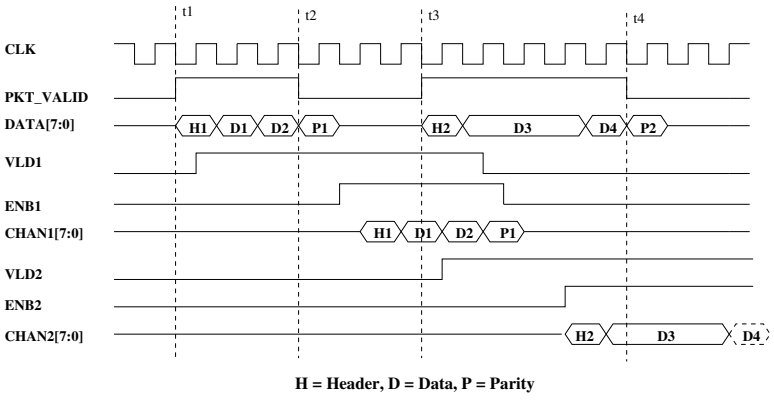

Fig. 17. Timing chart for the router example

The TRS of the RTL tests are derived from the timing specification of the RTL implementation. Figure 17 presents the timing chart of a testing scenario. From this chart, we can extract the timing specification for the router as follows. All input/output signals are

active high and are synchronized to the falling edge of the clock. The *PKT_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 *PKT_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 *VLDx* ($x \in \{0, 1, 2\}$) signal when valid data appears on the *CHANx* output. The *ENBx* input signal must then be asserted on the falling clock edge in which data is read from the *CHANx* bus. As long as the *ENBx* signal remains active, the *CHANx* bus drives a valid byte on each rising clock edge. Such timing information need to be extracted and described in the timing sequence of TRS.

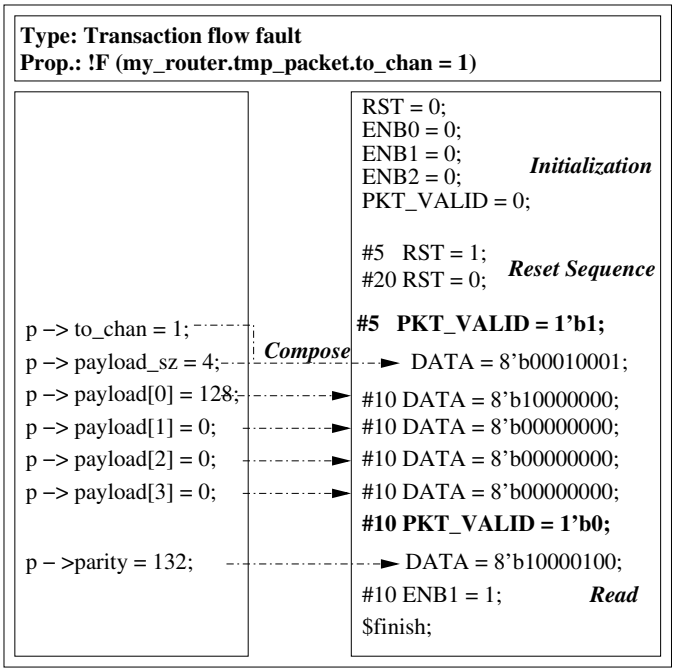

Fig. 18. TLM to RTL Test Transformation in the Router Example

Section 3.4 gives out the details of the router TRS. The RTL tests can be obtained by using this specification. Figure 18 shows both TLM and RTL tests corresponding to the transaction flow fault shown in Figure 8. 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 *PKT_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 *PKT_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 slaves can read the packet.

To increase the RTL coverage, we also manually generated several RTL tests which are not related to the proposed fault models. These tests are required to cover the additional

functionalities in RTL that are not available in TLM. For example, TLM does not have notion of *reset* signal. Therefore, we needed to generate RTL tests related to reset check operations and so on.

4.1.2 *RTL Validation and Analysis.* We generated a total of 92 TLM tests: 4 based on transaction flow fault model and 88 for transaction data fault model. It is important to note that the TLM test generation and RTL test translation are independent. In other words, TLM tests can come from multiple sources. However, our tool will automatically convert TLM tests to RTL tests. Due to the lack of FIFO channel information, we manually created 4 RTL tests based on FIFO overflow, reset check, and asynchronous read. Finally we got 92 TLM tests and 96 RTL tests for validation.

To show the effectiveness of our directed tests, we applied both random tests and directed tests on the RTL implementation of the router and measured various coverage metrics using Synopsys VCS cmView [Synopsys]. Table I shows the coverage results. The first row indicates the RTL coverage metrics. The second to fourth rows show various coverage using 100, 1000 and 10000 random tests respectively. Although the number of random tests increases exponentially, there is no drastic improvement on the coverage ratio. The fifth row shows the coverage results using our directed tests. It shows that the method using directed tests can achieve better RTL coverage with significantly fewer tests. The sixth to eighth rows present the coverage result that combines both random tests and our directed tests. The results indicate that our method can activate the functional scenarios that are difficult to be activated by the random method. For example, in the third row and seventh row, we can find that coverage using the random method can be further improved by adding our directed tests. This is because our directed tests are derived from TLM designs and carry the system level information. To further improve the coverage result using directed method in the fifth row, the last row gives the coverage with four extended manual tests. It can achieve the best coverage (except the line coverage and toggle coverage) with much shorter simulation time.

Table I. RTL coverage of the router

Tests (%)	Line (%)	Condition (%)	FSM (%) State / Transition	Toggle (%) Regs / Nets	Path (%)	Time (minute)
$Rand_{100}$	98.63	51.06	75.0/37.5	56.76/48.91	51.39	0.07
$Rand_{1000}$	99.92	53.19	100/62.5	56.76/57.61	56.94	1.23
$Rand_{10000}$	99.99	46.81	75/37.5	64.86/70.65	58.33	17.63
$Directed$	98.89	72.34	75.0/37.5	59.46/68.48	68.06	0.08
$Rand_{100} + Directed$	99.55	72.34	75.0/37.5	78.38/81.52	68.08	0.08
$Rand_{1000} + Directed$	99.97	78.72	100/62.5	78.38/84.78	73.61	1.35
$Rand_{10000} + Directed$	99.99	78.72	100/62.5	81.08/85.87	73.61	17.43
$Extended + Directed$	99.48	78.72	100/75	79.97/80.43	73.61	0.10

We have identified several fatal errors during validation of the RTL implementation using our generated tests. 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! This is due to the incorrect implementation of FIFO size which is always decremented without zero check. 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.

4.2 A Pipelined Processor Example

In this sub-section, we first present the TLM model of the pipelined processor and associated TLM test generation. Next, we present the TRS specification for RTL test generation. Finally, we discuss the results of the RTL design validation using generated tests.

4.2.1 *TLM Test Generation.* Figure 19 shows a simplified version 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 reads the values of the operands if necessary. EX module does ALU operations, also it will notify whether the conditional or unconditional branch happens. Memory module reads and writes data to the data memory. Writeback module stores the result to specified registers. The communication between two modules uses the port binding associated with a blocking FIFO channel with one slot. For example, there is a binding from the *port* of IF module to the *export* of ID module, and the *export* of ID module binds to a blocking FIFO channel for holding incoming instructions. So each time, the IF module can only issue one instruction to ID module; otherwise it will be blocked. The whole TLM design contains 6 classes, 11 functions and 797 lines of code.

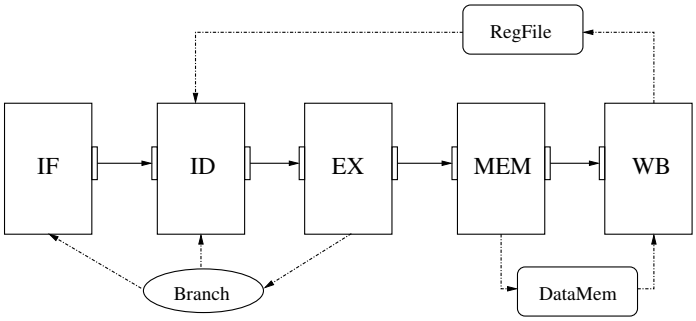

Fig. 19. The TLM Structure of an Alpha AXP Pipeline Processor

During the TLM to SMV translation, the global data structure (such as register file, data memory etc.) is defined in the SMV main function, and they are used as the input and output parameters by each modules. Initially, the program counter (PC) starts from 0 and the value of registers and memories are all 0. In this example, we use both transaction data fault model and transaction flow fault model to derive properties. Figure 20 presents an example for a transaction data fault.

```
// Property derived from a transaction data fault
assert !F(data_memory[3]=2);
//TLM Test
LDQ R0, 2(R0)       // register[0] = 2;
STQ R0, 3(R1)       // register[3] = 2;
```

Fig. 20. TLM tests for the Alpha AXP processor

4.2.2 *RTL Test Generation.* In Alpha AXP processor, there are four types of instructions: *CALL_PAL*, *OPERATE*, *BRANCH* and *MEMORY*. For *OPERATE* instructions, there are three different instruction formats. Figure 21 shows a partial TRS description that is used to translate the processor TLM tests to RTL tests. There are two input ports for the RTL design of the processor: *RESET* for resetting all five stages, and 64-bit signal *Imem2proc_bus* which contains two 32-bit instructions. In every two clock cycles, the processor fetches one 32-bit instruction from the instruction memory through the bus connected to the instruction memory. Because there are 4 different types of instructions, in the *mapping_def* part, it is necessary to list four different instruction formats. And in the timing sequence part, the input signals to *Imem2proc_bus* will be determined by the instruction class information included in TLM tests.

```
SPEC Alpha_AXP(inst1, inst2)
mapping_def:
    bit[31:0] memory_1 =   {inst1.op[31:26], inst1.ra[25:21],
          inst1.rb[20:16], inst1.mem_disp[15:0]};
    bit[31:0] memory_2 =   {inst2.op[31:26], inst2.ra[25:21],
          inst2.rb[20:16], inst2.mem_disp[15:0]};
    ....
end_mapping_def

pattern initialize()
    Imem2proc_bus = 64'h 0000_0000_0000_0000;
    #2 RESET_=0;
end_pattern

pattern reset()
    RESET_ = 1;
    #2 RESET=0;
end_pattern

main: begin
    initialize();
    reset();
    if (inst1.class == MEMORY)
       Imem2proc_bus[31:0] = memory_1;
    ....
    if (inst2.class == MEMORY)
       #2 Imem2proc_bus[63:32] = memory_2;
    ....
    #2 FINISH();
end
END_SPEC
```

Fig. 21. A Test Refinement Specification for the Alpha AXP Pipeline processor

Figure 22 shows the mapping from a TLM instruction to a RTL instruction. Because the given TLM test is of memory type, according to the TRS mapping information defined in Figure 21, the 32-bit instruction contains four segments: opcode, register $rega$, register $regb$ and memory address displacement. The mapping rules provides both value and place information for the transformation.

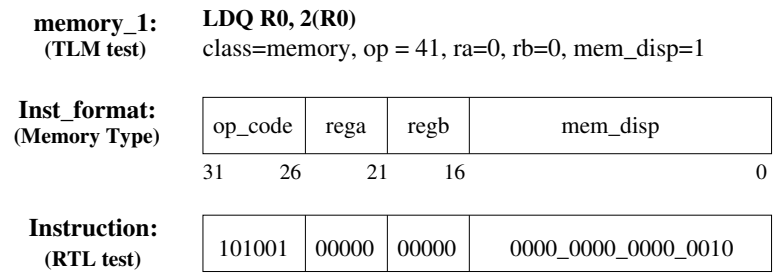

Fig. 22. The TLM to RTL instruction mapping of Alpha AXP Pipeline Processor

We apply the *Alpha_AXP* TRS on the TLM tests generated from the SMV counterexamples. Figure 23 shows an example of the transformation from a TLM test to a RTL test. The left part shows a TLM test with two TLM instructions, and the right part presents its corresponding RTL test. During the test transformation, each TLM instruction in the left part data will be composed and mapped to a 64-bit input RTL signal.

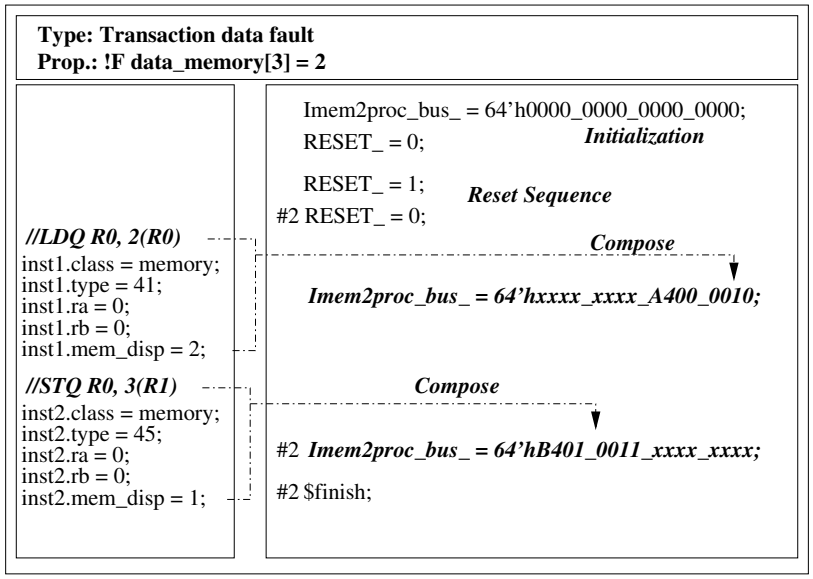

Fig. 23. TLM to RTL Test Transformation in the Alpha AXP Pipeline Processor

4.2.3 *Validation Results.* The test generation for the Alpha AXP processor is based on the transaction data and flow fault models. The transaction data faults mainly indicate the bit value change for each transaction variable and global variable such as data memory, register file, data forward and branch status. The transaction flow faults indicate the instruction category and instruction execution. Overall, there are 212 TLM tests generated, including 86 tests for condition faults and 126 tests for data bit faults. It costs 311.47 minutes to achieve all these tests using Cadence SMV verifier [Cadence Berkeley Labs]. We also use the Bounded Model Checker NuSMV [ITC-IRST and CMU] to optimize the test generation time. By using NuSMV, the test generation time just needs 3.23 minutes.

Since some of the generated tests are redundant (same test) and can be removed. So finally we get 112 TLM tests, including 50 tests for transaction flow faults and 62 tests for transaction data faults. We applied both random tests and directed tests on the RTL implementation to measure the effectiveness of our directed tests. We derived 100, 500, 5000 and 50000 random RTL tests respectively. The directed RTL tests are generated using the TRS presented in Section 4.2.2. The coverage results are shown in Table II. For the condition coverage, there is no improvement with more random tests. It is important to note that, in this example, the test generation of 50000 random tests costs 23.18 minutes, while our 212 directed tests derived using bounded model checker just needs 3.23 minutes. Moreover, our method can achieve better coverage result (except line coverage) than the random method with less time. We also combined both random tests and directed tests. The result shows that our directed method can activate the functional scenarios that are difficult for random methods to explore. For the example in the fifth row, when applied 50000 random tests, the path coverage ratio is 80.82%. However, by adding our directed tests incrementally in the tenth row, the path coverage ratio increases to 95.89%.

Table II. RTL coverage results for the Alpha AXP Processor

Tests Tests	Line (%)	Condition (%)	FSM FSM	Toggle Regs/Nets	Path (%)	Time (minutes)
$Random_{100}$	97.63	82.93	NA	67.69/65.89	60.27	0.20
$Random_{500}$	99.68	82.93	NA	69.23/66.36	72.60	0.35
$Random_{5000}$	99.98	82.93	NA	70.77/68.22	80.82	2.32
$Random_{50000}$	99.99	82.93	NA	70.77/68.22	80.82	23.18
$Directed$	98.94	95.73	NA	87.69/81.32	86.30	0.83
$Directed + Random_{100}$	99.92	96.34	NA	89.23/82.24	90.41	1.20
$Directed + Random_{500}$	99.98	96.34	NA	89.23/82.24	90.41	1.10
$Directed + Random_{5000}$	99.99	96.34	NA	89.23/82.24	90.41	3.10
$Directed + Random_{50000}$	99.99	97.56	NA	89.23/82.24	95.89	23.30

5. CONCLUSIONS

This article presented an automatic RTL test generation methodology based on TLM specifications. Our approach has various advantages. First, TLM validation efforts can be reused during RTL validation and thereby will significantly reduce the RTL validation effort. Next, the generated tests contain the information of the system level requirements which are hard to capture at the RTL level without ad-hoc reverse engineering efforts. Moreover, the tests and their accompanying transformation rules enable consistency checking between different abstraction levels. Finally, the RTL tests can be ready before the RTL implementation is available.

We implemented a prototype tool *ARTEST* which can be downloaded from the website http://www.cise.ufl.edu/∼prabhat/research/tlmValidation/. This tool can automatically generate TLM tests from TLM specifications. The generated TLM tests will be automatically transformed to RTL tests using our proposed test refinement specification. The case studies demonstrated that the RTL tests generated by our method can achieve the intended functional coverage.

Clearly, the model checking based approach will not be suitable for test generation of complex SOC designs due to state space explosion. We plan to investigate two complementary directions to address this issue: development of efficient automatic decomposition

techniques in model checking based test generation and development of test generation techniques using property clustering and learning techniques.

Acknowledgments

This work was partially supported by grants from Intel Corporation, National Science Foundation Faculty Early Career Development (CAREER) Award 0746261 and National Natural Science Foundation of China No. 61021004. A preliminary version [Chen and Mishra 2007] of this paper appeared in the proceedings of IEEE International High-Level Design Validation and Test Workshop (HLDVT) 2007.

REFERENCES

ABDI, S. AND GAJSKI, D. 2005. A formalism for functionality preserving system level transformations. In *Proceedings of Asia and South Pacific Design Automation Conference (ASPDAC)*. 139–144.

ABRAR, S. AND THIMMAPURAM, A. 2010. Functional refinement: a generic methodology for managing ESL abstractions. In *Proceedings of International Conference on VLSI Design (VLSID)*. 122–127.

AMMANN, P., BLACK, P., AND MAJURSKI, W. 1998. Using model checking to generate tests from specifications. In *Proceedings of International Conference on Formal Engineering Methods (ICFEM)*. 46–54.

ARA, K. AND SUZUKI, K. 2003. A proposal for transaction-level verification with component wrapper language. In *Proceedings of Design, Automation and Test in Europe: Designers' Forum*. 20082.

BEYER, D., CHLIPALA, A., HENZINGER, T., JHALA, R., AND MAJUMDAR, R. 2004. Generating tests from counterexamples. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 326–335.

BOMBIERI, N., FUMMI, F., AND PRAVADELLI, G. 2006. On the evaluation of transactor-based verification for reusing TLM assertions and testbenches at RTL. In *Proceedings of Design, Automation, and Test in Europe (DATE)*. 1–6.

BOMBIERI, N., FUMMI, F., AND PRAVADELLI, G. 2007. Incremental ABV for functional validation of TL-to-RTL design refinement. In *Proceedings of Design, Automation and Test in Europe (DATE)*. 882–887.

BOMBIERI, N., FUMMI, F., PRAVADELLI, G., AND MARQUES-SILVA, J. 2007. Towards equivalence checking between TLM and RTL models. In *Proceedings of International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. 113–122.

BRUCE, A., HASHMI, M., NIGHTINGALE, A., BEAVIS, S., ROMDHANE, N., AND LENNARD, C. 2006. Maintaining consistency between systemC and RTL system designs. In *Proceedings of Design Automation Conference (DAC)*. 85–89.

CADENCE BERKELEY LABS. *The Cadence SMV Model Checker*. Available at http://www.kenmcmil.com/.

CAI, L. AND GAJSKI, D. 2003. Transaction level modeling: an overview. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 19–24.

CHEN, M. AND MISHRA, P. 2010. Functional test gneration using efficient property clustering and learning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 29,* 3, 396–404.

CHEN, M., MISHRA, P. AND KALITA, D. 2007. Towards RTL test generation from SystemC TLM specifications. In *Proceedings of International High-Level Design Validation and Test Workshop (HLDVT)*. 91–96.

CHEN, M., QIN, X. AND MISHRA, P. 2010. Efficient Decision Ordering Techniques for SAT-based Test Generation. In *Proceedings of Design, Automation and Test in Europe (DATE)*. 490–495.

FEDELI, A., FUMMI, F., AND PRAVADELLI, G. 2007. Properties Incompleteness Evaluation by Functional Verification. *IEEE Transactions on Computers 56,* 4, 528–544.

FERRANDI, F., FUMMI, F., GERLI, L., AND SCIUTO, D. 1999. Symbolic functional vector generation for VHDL specifications. In *Proceedings of Design, Automation and Test in Europe (DATE)*. 442–446.

FIN, A., FUMMI, F., PONCINO, M., AND PRAVADELLI, G. 2003. A SystemC-based framework for properties incompleteness evaluation. In *Proceedings of International Workshop on Microprocessor Test and Verification (MTV)*. 89–94.

GHENASSIA, F. 2005. *Transaction level modeling with SystemC*. Springer, Dordrecht, Netherlands.

HABIBI, A. AND TAHAR, S. 2004. Partial order reduction for scalable testing of SystemC TLM designs. In *Proceedings of International High-Level Design Validation and Test Workshop (HLDVT)*. 19–22.

HABIBI, A. AND TAHAR, S. 2006. Design and verification of SystemC transaction-level models. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI) 14,* 1, 57–68.

IEEE P1850. *Property Specification Language Homepage*. Available at http://www.eda.org/ieee-1850/.

ITC-IRST AND CMU. *NuSMV*. Available at http://nusmv.fbk.eu/NuSMV/.

JENSEN, K. 1997. A brief introduction to coloured Petri nets. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. 203–208.

JINDAL, R. AND JAIN, K. 2003. Verification of transaction-level SystemC models using RTL testbenches. In *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. 199–203.

KARLSSON, D., ELES, P., AND PENG, Z. 2006. Formal verification of SystemC designs using a Petri-net based representation. In *Proceedings of Design, Automation, and Test in Europe (DATE)*. 1228–1233.

KOO, H. AND MISHRA, P. 2009. Functional test generation using design and property decomposition techniques. *ACM Transactions on Embedded Computing Systems (TECS) 8(4)*.

KUPFERMAN, O. AND VARDI, M. 1999. Vacuity detection in temporal model checking. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*. 82–96.

LARSEN, K., PETTERSSON, P. AND WANG, Y. 1997. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT) 1(1-2),* 134–152.

KROENING, D. AND SHARYGINA, N. 2005. Formal verification of SystemC by automatic hardware/software partitioning. In *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. 101–110.

LAHBIB, Y., KAMDEM, R., BENALYCHERIF, M., AND TOURKI, R. 2005. An automatic ABV methodology enabling PSL assertions across SLD flow for SoCs modeled in SystemC. *Computers & Electrical Engineering 31,* 4, 282–302.

MCPEAK, S. *Elsa*. Available at http://www.eecs.berkeley.edu/~smcpeak.

MISHRA, P. AND DUTT, N. 2008. Specification-driven directed test generation for validation of pipelined processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 13,* 3, 1–36.

MOY, M., MARANINCHI, F., AND MAILLET-CONTOZ, L. 2005. Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *Proceedings of the International Conference on Application of Concurrency to System Design*. 26–35.

ROSE, A., SWAN, S., PIERCE, J., AND J.M., J. F. 2005. Transaction level modeling with SystemC. Springer.

STRICHMAN, O. 2001. Pruning techniques for the sat-based bounded model checking problem. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*. 58–70.

SYNOPSYS. *VCS verification library*. Available at http://www.synopsys.com.

WANG, Z. AND YE, Y. 2005. The improvement for transaction level verification functional coverage. In *Proceedings of International Symposium on Circuits and Systems (ISCAS)*. 5850–5853.