EFFICIENT APPROACHES FOR FUNCTIONAL VALIDATION OF SOC DESIGNS USING HIGH-LEVEL SPECIFICATIONS

By

MINGSONG CHEN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2010

To my parents for their love and encouragement

ACKNOWLEDGMENTS

Although four years passed in a twinkling, all the vivid snapshots are deeply engraved in my memory. I always think that I was lucky to be a Gator in UF, not only because I witnessed four National Championships, but also I achieved another milestone in my life here. I need to confess that the journey to get a Ph.D. is challenging. It is impossible to imagine completing it without the precious advice and help from other people.

First of all, I really appreciate what my supervisor Dr. Prabhat Mishra did for me. His expertise and insights helped me to quickly capture the research direction and made this dissertation come true. Throughout my Ph.D. study, he gave me enduring support, guidance and encouragement which helped me to overcome various problems. There is no doubt that his attitude on research has deeply affected me and will be helpful in my future career. Finally I understood why he was always urging me to make progress. His efforts made my CV looks stronger which is beneficial to me.

I would also like to thank my Ph.D. committee members: Prof. Sartaj Sahni, Porf. Jih-Kwon Peir, Prof. Tao Li and Prof. Raymond Issa. Their valuable suggestions at different stages of my research were constructive and thought-provoking. Their criticisms enhanced the quality of my research. Colleagues and friends are an important part in my graduate life. I am very grateful for the friendship of all the members in my research group - Kanad Basu, Hadi Hajimiri, Heon-Mo Koo, Chetan Murthy, Kartik Shrivastava, Xiaoke Qin and Weixun Wang. I really enjoyed the harmonious atmosphere of our lab and the experience of collaborating with them.

Last but not least, I sincerely thank my parents, who unconditionally gave me the love and encouragement. Without their support, I won't reach this far. I dedicate this dissertation to them.

This work was partially supported by grants from Intel Corporation and NSF CAREER award 0746261.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

EFFICIENT APPROACHES FOR FUNCTIONAL VALIDATION OF SOC DESIGNS USING HIGH-LEVEL SPECIFICATIONS

By

Mingsong Chen

August 2010

Chair: Prabhat Mishra
Major: Computer Engineering

Increasing complexity coupled with time-to-market pressure create a critical need to raise the abstraction level for System-on-Chip (SoC) designs. Functional validation is widely acknowledged as a major bottleneck due to lack of automated techniques and limited reuse of validation efforts between abstraction levels. Simulation is the most widely used form of validation using random or constrained-random tests. Directed tests are very promising for simulation since only fewer directed tests are required compared to billions of random tests to achieve a coverage goal. Currently, directed test generation is performed manually which is time-consuming and error-prone. This dissertation presents a novel top-down methodology for automatically generating directed tests from high-level specifications and reuse them across different abstraction levels. The objective is to reduce the overall functional validation effort. My research has four major contributions: i) it proposes a method that can extract formal models from high-level SoC specifications; ii) it presents an approach that can automatically derive properties based on fault models; iii) it proposes efficient clustering, learning and decomposition techniques to reduce the directed test generation time; and iv) it provides validation refinement approaches to enable reuse of the system-level validation efforts for low-level implementation validation as well as to check the consistency between different abstraction layers. Our experimental results using both software and hardware benchmarks demonstrate that the proposed approaches can significantly reduce the overall validation effort.

CHAPTER 1
INTRODUCTION

Functional validation [1] is widely acknowledged as a major bottleneck in System-on-Chip (SoC) design methodology – up to 70% of the overall design time and resources are spent on functional validation. In spite of such extensive efforts, majority of the SoC designs fail at the very first time (silicon failures) primarily due to functional errors [79]. The functional validation complexity is expected to increase further due to the combined effects of increasing design complexity and recent paradigm shift from single processor SoC designs to heterogeneous multiprocessor architectures [90].

Traditional SoC validation adopts a combination of simulation-based approaches and formal methods. Random testing is widely used for SoC simulation. In general, random tests can not guarantee the coverage and it may exercise the same functional scenario for several times because of randomness. Thus directed tests are a better alternative since only a small number of tests are required to achieve a functional coverage goal compared to random or constrained-random tests. However, due to lack of automated tools to generate directed tests, human intervention is necessary during the test generation. All these scenarios can lead to time-consuming and error-prone validation. My research targets to reduce the overall functional validation effort by automating various steps in the the validation flow as well as by developing efficient learning and reuse techniques.

The rest of the chapter is organized as follows. Section 1.1 presents the SoC design flow. Section 1.2 surveys the existing SoC functional validation methods. Finally, Section 1.3 presents the contributions of this dissertation.

[1] The term "validation" generally refers to simulation-based approaches, while "verification" is used for both simulation-based and formal methods. This dissertation focuses on directed test generation for simulation, so it uses the term validation.

1.1 SoC Design Flow

SoC integrates all components of a computer into a single integrated circuit (chip). It consists of both hardware (such as processor, memory and peripherals) and software (such as application programs). SoC may perform a variety of computations including digital, analog and mixed-signal functions. Thus it is widely used in the field of embedded and hybrid systems.

SoC is becoming increasingly complex since new applications require more features. As a result, extensive system-level simulations are required to make the right architectural trade-offs. To efficiently and quickly make the decision on these trade-offs, design architects increasingly leverage system-level specifications instead of implementations to perform such analysis.

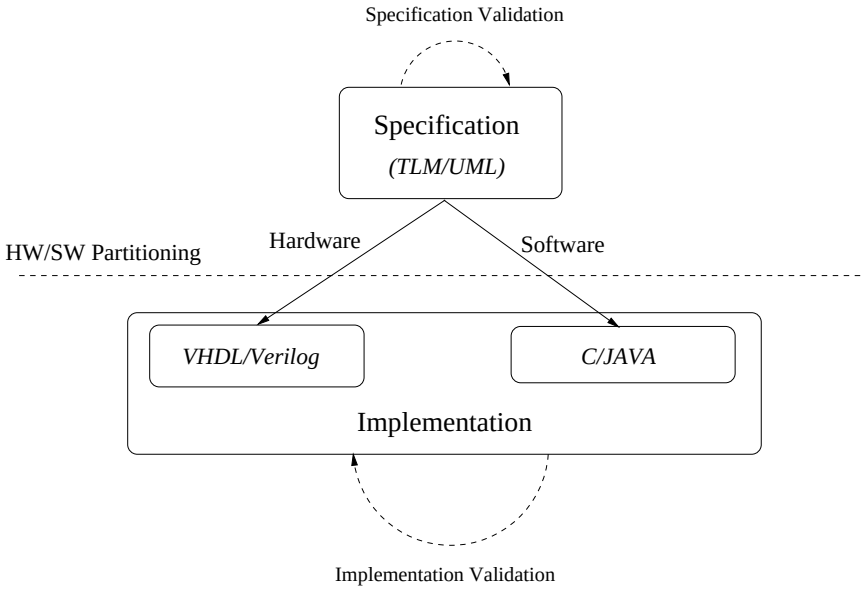


Figure 1-1. SoC design and validation flow

Figure 1-1 presents a SoC design and validation flow. Various hardware and software modeling paradigms are used for SoC specifications. Two of the most widely used specifications are Transaction Level Modeling (TLM) [16, 78] and Unified Modeling Language (UML) [69]. They establish a standard to enable fast simulation speed and easy model interoperability for hardware/software co-design. Generally, TLM is promising for

hardware modeling and UML focuses on software modeling. TLM mainly allows modeling of communication between different hardware components of a system and data processing in each component. UML can capture both structural and behavioral information of a software system. Validated specification can be used as a golden reference model for validation of software and hardware implementations. Although specifications can capture most important functional scenarios (system behaviors), some implementation details can be still missing. For example, TLM provides two kinds of modeling styles: loosely-timed models can be used to model the system behavior with less timing information and approximately-timed models can enable timing analysis of system behavior. Although TLM is promising for system-level modeling and simulation, it is still hard to accurately describe the hardware behavior because it lacks many detailed information such as timing details. So Register Transfer Level (RTL) is needed to model the implementation-level behavior after the system-level simulation. In Figure 1-1, the hardware part will be implemented using a RTL language such as VHDL or Verilog, and the software will be implemented using a programming language such as C or JAVA. Significant amount of validation work is needed to check the specified functional scenarios as well as to check the consistency between the specification and implementation.

1.2 Functional Validation of SoC Designs

Specification validation is extremely important to ensure that the specified design is correct and can be used as a golden reference model for the implementation. According to [79], there are two key contributors to the SoC failures (silicon respin): specification errors and implementation errors. As expected, 82% of the designs with respins resulting from functional flaws had implementation errors. Interestingly, almost 47% of the designs with respins resulting from functional flaws had also incorrect or incomplete specifications [79]. Therefore, it is necessary to validate specifications before validating the implementation.

This section first surveys existing functional validation methods, and then describes several improvement opportunities to reduce the overall functional validation effort.

1.2.1 Overview of Functional Validation Methods

Simulation is the most widely used SoC validation method. Compared to random testing methods which use billions of random and pseudo-random tests in the traditional design flow, directed tests are very promising in reducing the overall validation effort since a significantly smaller number of directed tests can achieve the same coverage goal [61]. However, a major problem in current directed test generation approach is that it is mostly performed by human intervention. Hand-written tests entail laborious and time consuming effort of verification engineers who have deep knowledge of the design under verification. Due to the manual development, it is infeasible to generate all directed tests to achieve a comprehensive coverage goal in a short time. Automatic directed test generation based on a comprehensive functional coverage metric is an alternative to address this problem.

Model checking [21] is one of the most widely used formal methods for automated test generation to validate software/hardware designs [5]. In the context of test generation, a design specification is described using a formal model. The required functional scenarios are described in the form of temporal logic formulas. When checking a false property using a model checker, one counterexample is reported to falsify the property. Because this counterexample is a sequence of variable assignments, it can be used as a directed test to validate the functional scenario of the specification. However, model checking based techniques do not scale well for large designs due to the "state space explosion" [2] .

Simulation based methods are fast but cannot guarantee the convergence of functional coverage. Model checking based methods can automatically generate directed tests but cannot deal with large designs. Currently, most SoC validation approaches use a hybrid method which incorporates both techniques. The hybrid method first performs the

[2] The number of states generated for verifying a property is huge and can not be handled due to the memory capacity of computers.

random simulation to get as much functional coverage as possible. Then the uncovered functional scenarios and corner cases are activated using the directed tests.

1.2.2 Potential Improvement Opportunities

Since system-level specification is treated as the golden reference model in the SoC design flow, a logic error in the system-level specification certainly will cause the malfunction in the implementation. Because implementations are more complex than system-level specifications, finding an error in implementations will be more time-consuming. So it is necessary to guarantee that system-level specification validation can cover as many functional scenarios as possible. In addition, the differences between specification and implementation limit the degree of validation reuse. In the absence of significant reuse of validation efforts between different abstraction levels, the overall functional validation effort will increase since designers have to verify the specification as well as its implementation.

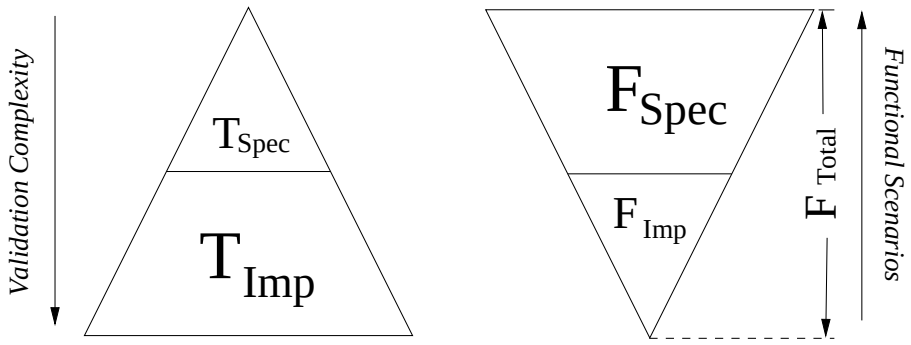


Figure 1-2. Comparison of functional validation between specification and implementation

Figure 1-2 compares specification and implementation levels. Assume that a design D has a total F_{Total} number of functional scenarios that need to be checked. For specifications, there are F_{Spec} number of functional scenarios that need to be checked, and each specification level test generation need an average time of T_{Spec}. In addition to F_{Spec} functional scenarios, there are F_{Imp} functional scenarios need to be checked in implementations, and each implementation test needs an average time of T_{Imp}. Figure 1-2

a) indicates that when checking a functional scenario, implementation validation is more difficult than specification validation. Figure 1-2 b)shows that specifications cover majority of the overall system functional scenarios (e.g., 70%), and implementations inherit all such scenarios with its own new additional functional scenarios (e.g., 30%) due to the introduction of implementation details. In this dissertation, the complexity of validating a functional scenario is equivalent to generating and applying a directed test. So test generation and corresponding simulation time is used to indicate the functional scenario validation effort.

In order to achieve a 100% functional coverage as well as to minimize the overall specification and implementation test generation time, it is necessary to find a method to optimize the Equation (1–1).

$$\begin{aligned} &Minimize: \quad F_{Spec} \times T_{Spec} + \{F_{Spec} + F_{Imp}\} \times T_{Imp} \\ &Subject\ to: \quad \begin{cases} F_{Spec} + F_{Imp} = F_{Total} \\ T_{Spec} << T_{Imp} \\ F_{Spec} > F_{Imp} \end{cases} \end{aligned} \tag{1–1}$$

For directed test generation, there are four feasible options. Table 1-1 compares these approaches.

- **No optimization:** Specification level test generation and implementation level test generation are independent, and in each level there are no optimizations.
- **Specification level optimization:** Specification level test generation and implementation level test generation are independent. The overall specification test generation time can be reduced by certain optimization methods.
- **Reuse between specification and implementation:** No optimization for specification and implementation level test generation, but the specification tests can be reused for implementation level validation.
- **Specification level optimization + reuse between specification and implementation:** Optimizations reduce the overall specification level test generation time, and the specification level tests can be reused for implementation level validation.

Assume that in system validation we can find a specification level test generation optimization that can produce α times ($\alpha > 1$) speedup, and we can obtain another β times ($\beta > 1$) speedup due to validation reuse. According to the comparison shown in Table 1-1, the last option can achieve the best possible performance. The goal of this dissertation is to develop efficient techniques to reduce the overall validation effort using the fourth (last) option.

Table 1-1. A comparison for four optimizations

Optimization	Time
None	$F_{Spec} \times T_{Spec} + F_{Total} \times T_{Imp}$
Specification level	$F_{Spec} \times T_{Spec}/\alpha + F_{Total} \times T_{Imp}$
Reuse	$F_{Spec} \times T_{Spec} + F_{Spec} \times T_{Imp}/\beta + F_{Imp} \times T_{Imp}$
Specification level + Reuse	$F_{Spec} \times T_{Spec}/\alpha + F_{Spec} \times T_{Imp}/\beta + F_{Imp} \times T_{Imp}$

1.2.3 Challenges

Each of the components (such as IP cores, processors and memories) in a SoC design can be verified using existing validation approaches. However, the validation of the overall system is extremely complex due to exponentially large number of possible interactions that are extremely hard to model, analyze and validate. Although the potential improvements proposed in the previous section seems promising, there are four fundamental problems in automated generation of directed tests for SoC architectures. The first challenge is to decide specification models for SoC architectures and how to verify the specification to ensure that it can be used as a golden reference model. The next challenge is to identify a comprehensive functional coverage metric to enable coverage-driven generation of properties and associated high-level tests. The third and most important challenge is how to significantly reduce the test generation complexity to avoid state space explosion problem. Finally, due to significant differences between specifications and implementations, a major challenge is how to efficiently reuse the specification-level properties and tests for validation of SoC implementations.

1.3 Dissertation Contributions

My research employs a top-down validation methodology using a combination of simulation based approaches and formal methods to address the four challenges mentioned in Section 1.2.3. The objective of my research is to develop tools, techniques and methodologies to enable automatic generation of directed functional tests to drastically reduce the overall verification effort as well as to improve the quality of SoC designs.

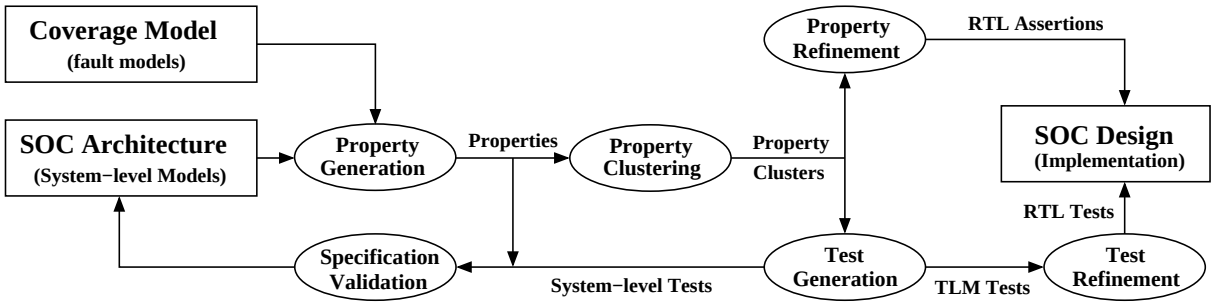


Figure 1-3. Top-down validation of SoC architectures

Figure 1-3 outlines the proposed validation methodology for SoC architectures using system-level specification. It consists of four major contributions as follows:

- **Formal modeling of SoC designs:** Since most existing SoC specifications are not formal enough to enable automated test generation, this dissertation proposes an approach for automatic specification analysis. It can extract formal models from semi-formal hardware and software specifications.

- **Coverage-driven property generation:** Functional coverage plays an important role to determine the adequacy of functional validation. This dissertation defines various fault models for SoC specifications. Based on these fault models, we can automatically derive properties to validate the specified functional scenarios.

- **Efficient directed test generation:** To reduce the overall test generation time for the same design with a large set of properties, this dissertation proposes various clustering methods which can cluster the similar properties together to share the learnings during test generation. The proposed framework investigates two kinds of learnings based on conflict clause forwarding as well as decision ordering. Such learnings can be used to avoid repeated validation efforts between similar properties. For complex properties without learning opportunity, this dissertation proposes two decomposition techniques that can actively achieve the learning to reduce its test generation time.

- **Automated refinement of validation efforts:** This dissertation develops a prototype tool which can automatically convert TLM level tests and properties into

RTL tests and assertions to enable implementation level validation. Based on this validation effort reuse, this dissertation proposes a methodology which can check the assertion-based functional equivalence between specifications and implementations.

The rest of this dissertation is organized as follows. Chapter 2 describes how to extract formal models from system level specifications of SoC designs. Chapter 3 describes how to generate properties based on our proposed fault models. Chapter 4 to 6 discuss how to efficiently generate tests to enable functional validation. Chapter 4 describes how to divide the properties into several groups such that each group contains similar properties that can benefit from each other during test generation. Chapter 5 presents the decision ordering based learning techniques which can drastically reduce the overall test generation time. Chapter 6 proposes various decomposition techniques to actively find the learnings for a complex property. Chapter 7 presents the methodology for automated property and test refinements. It also describes how to utilize the validation refinement for functional equivalence checking. Finally, Chapter 8 concludes the dissertation and outlines several future research directions.

CHAPTER 2
FORMAL MODELING OF SOC SPECIFICATIONS

Modeling plays a central role in design automation of SoC architectures. It is necessary to develop a specification language that can model complex systems at a higher level of abstraction and also enable automatic analysis and generation of efficient reference models. The language should be powerful enough to capture high-level description of a wide variety of SoC architectures as well as should be simple enough to allow correlation of the information between the specification and the architecture/system manual.

As a system level specification, SystemC TLM [78] 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. Although UML is being used as a de facto software modeling tool, UML Profile for SoC [68] is proposed as an extension of UML 2.X to enable SoC hardware modeling. It can be used to capture the system behavior for both SoC software and hardware components [19, 65, 77]. However, both SystemC TLM and UML diagrams are not formal enough for automatic test generation using model checking techniques [5]. Consequently, the ambiguity, incompleteness, and contradiction in specifications can lead to different interpretations. Therefore it is necessary to formalize the semantics of SoC specifications.

This chapter introduces two widely used SoC specifications: SystemC TLMs for hardware modeling, and UML activity diagrams for software modeling. Next, it describes how to automatically extract the formal models from specifications to enable subsequent validation steps. The rest of the chapter is organized as follows. Section 2.1 introduces the formal modeling of SystemC TLMs. Section 2.2 proposes the formal modeling techniques of UML activity diagrams. Section 2.3 presents the case studies using both SystemC TLM designs and UML activity diagrams. Finally, Section 2.4 summarizes the chapter.

2.1 Specification using SystemC TLMs

As a framework built on C++, SystemC [70] deliberately mimics the hardware description languages such as VHDL and Verilog. With an event-driven simulation kernel, SystemC can be used to simulate the behavior of concurrent processes which can communicate with each other using procedure calls or other mechanisms offered by the SystemC library. Generally, SystemC is often associated with Transaction-Level Modeling (TLM) [16, 78], because SystemC TLM provides a wrapper to facilitate the process of communication modeling. Since SystemC TLM provides a rapid prototyping platform for the architecture exploration and hardware/software integration [30], it is widely used to enable early exploration for both hardware and software designs. It can reduce the overall design and validation effort of complex SoC architectures.

To enable automated analysis, various researchers have tried to extract formal representations from SystemC TLM specifications. Abdi et al. [2] introduced *Model Algebra*, a formalism for representing SoC designs at system level. The work by Kroening et al. [48] formalized the semantics of SystemC by means of labeled Kripke structures. Moy et al. [64] provided a compiler front-end that can extract architecture and synchronization information from SystemC TLM designs using HPIOM. Karlsson et al. [41] 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. [34] 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 validation. All these modeling techniques focus on the formal modeling of SystemC specifications. However, none of them investigate the automated test generation for transaction validation. This section discusses how to extract the formal models from SystemC TLM specifications to enable automated test generation.

2.1.1 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 "non-deterministic" 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 the transaction data token and the 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 specification, but also can be used to automatically derive the properties for TLM test generation. Definition 1 gives the formal model of SystemC TLM designs.

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

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

- $T = \{t_1, t_2, \ldots, t_n\}$ *is a set of transitions.*
- $A \subseteq \{P \times T\} \cup \{T \times P\}$ *is a set of arcs between places and transitions.*
- $E = \{e_1, e_2, \ldots, e_k\}$ *is a set of arc expressions. 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.*
- $M : 2^{P \times \Sigma} \times T \rightarrow 2^{P \times \Sigma}$ *is a function that describes the internal operations on input transaction data and output transaction data of a transition.*
- $I \in 2^{P \times \Sigma}$ *specifies the initial state.*
- $F \subseteq 2^{P \times \Sigma}$ *specifies the final states.* ■

In our framework, we use the graph model as an immediate form to capture the execution as well as interconnection of processes.

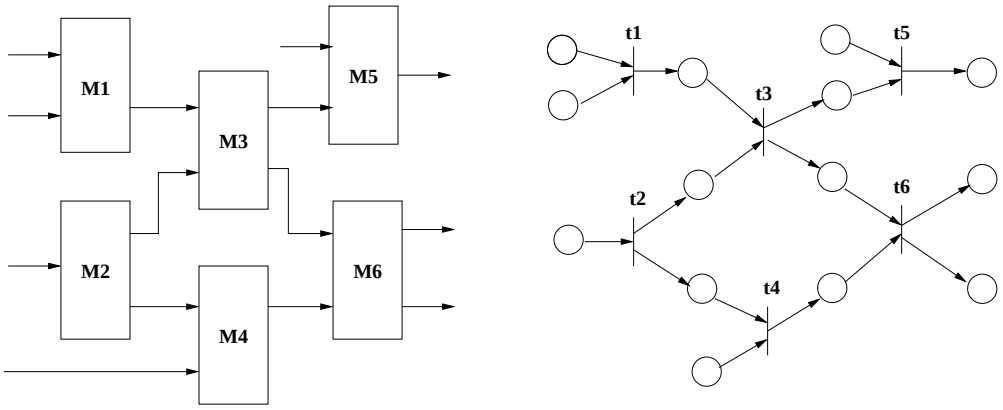


Figure 2-1. Mapping from a SystemC structure to corresponding graph model

Figure 2-1a) shows an interconnection of six modules. Each arrow indicates a port binding between two modules. Figure 2-1b) 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*. A transaction data token flows from the initial places to the target places and token values may change in transitions when necessary. The internal logic of a transition determines the flow of the transaction.

2.1.2 Transformation from SystemC TLM to SMV

Model checking techniques are very promising for directed test generation in hardware and software domains [5, 9, 71]. In our framework, we adopt SMV [56] 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. It is important to note that, 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 [8]) in our framework to enable the timing verification of transactions.

As an intermediate form for TLM to SMV translation, the graph 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 graph 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. We use the example shown in Section 2.3.1 to illustrate how to extract the formal model from a router example.

2.1.2.1 Structure Extraction

In TLM, the content of a transaction data token indicates the transaction flow and the output of each component. So it consists of the key part of TLM tests. Generally a transaction token consists of several attributes with different 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 lengths of user-defined type need to be considered. Figure 2-2 gives an example of the router *packet* in the form of SystemC TLM and SMV respectively. For example, *sc_uint* $< 2 >$ has 2 bits and will be transformed to a range 0..3 in SMV.

```
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
```

Figure 2-2. An example of data type transformation

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;
};
```

Figure 2-3. An example of SystemC TLM module

Figure 2-3 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 2-4 shows the SMV module skeleton corresponding to example in Figure 2-3 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 2-4, 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_out*0 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;
    ......
}
```

Figure 2-4. An example of SMV module

2.1.2.2 Behavior Extraction

TLM behavior describes the run-time information of TLM 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.

```
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);
}
```

Figure 2-5. An example of TLM process

Figure 2-5 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*.

TLM modeling provides some synchronization mechanism for the communications between modules. As shown in Figure 2-5, 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 2-6 is the translated SMV specification of the TLM example presented in Figure 2-5. 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 the value of the *packet_in* instead of the value of *input_* shown in the TLM example in Figure 2-5.

```
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;
    }
}
```

Figure 2-6. An example of SMV process

2.1.3 A Prototype Tool For TLM to SMV Translation

We developed a prototype tool $TLM2SMV$ which can transform SystemC TLM specifications to corresponding SMV models for automated directed test generation. The details of the implementation are described in Section 7.2.3.1.

2.2 Specification using UML Activity Diagrams

Formal verification can be used to verify the correctness of specifications, so it can be used to guarantee the quality of UML models [84]. UML activity diagram adopts Petri-net semantics which is promising to describe the concurrent behavior [18, 51, 88]. There are several approaches that use model checking techniques to verify UML activity diagrams. Eshuis [75] presented a translation procedure from UML activity diagrams to the input language of NUSMV [20]. However, the translation is used to verify the consistency between UML activity diagrams and class diagrams. It focuses on checking the consistency between two different models. Guelfi and Mammar [31] provided a formal definition for timed activity diagrams. They outlined the translation from the semantic specifications into PROMELA - an input language of the SPIN model checker. Das et al. [22] proposed a method to deal with timing verification of UML activity diagrams. All these verification

work primarily focus on checking the consistency or correctness of the model itself instead of generating directed test cases.

In this chapter, we adopt UML 2.1.2 [69] as our specification. To reduce the complexity of the testing work, we restrict our testing target and investigate a subset of activity diagrams. The subset mainly contains action nodes, control nodes, object nodes and control and data flow. Especially for the object node, we assume that it can hold at most one object at a time and it does not support *competition* and *data store*. This section first gives the notations used in UML activity diagrams. Then it presents the formal definitions of the UML activity diagrams. Finally, it describes the translation from UML activity diagrams to SMV formal models.

2.2.1 Notations

UML activity diagram is used to coordinate the execution of actions. An action takes a set of inputs and converts them into corresponding outputs. An activity (behavior) consists of a set of actions and flow edges. The actions are connected by object flow edges to show how object tokens flow through and connected by control flow edges to indicate the execution order.

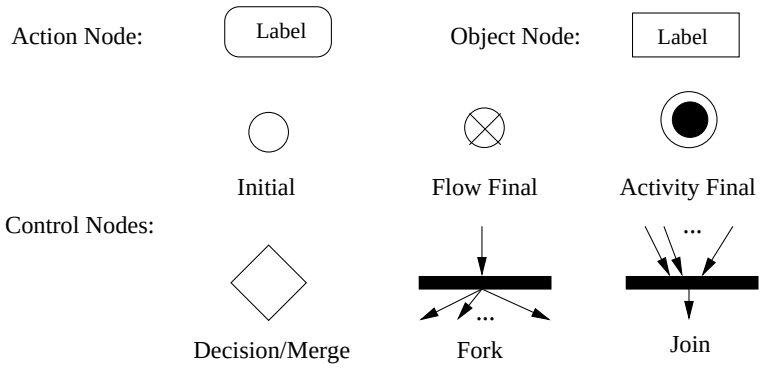


Figure 2-7. UML activity nodes

UML activity diagrams adopt the semantics like Petri-net [72]. It is a type of directed graphical representation. Tokens which indicate control or data values flow along the edges from the source node to the sink nodes driven by the actions and conditions. An activity

diagram has two kinds of modeling elements: activity nodes and activity edges. More specially, there are three kinds of nodes in activity diagrams:

- ***Action Node***: Action nodes consume all input data/control tokens when they are ready, generate new tokens and send them to output activity edges.
- ***Object Node***: Object nodes provide and accept data tokens, and may act as buffers, collecting data tokens as they wait to move downstream.
- ***Control Node***: Control nodes route tokens through the graph. The control nodes include constructs to choose between alternative flows (decision / merge), to split or merge the flow for concurrent processing (fork / join).

Figure 2-7 shows the basic constructs of activity nodes. An action node is denoted by round cornered boxes. It represents an execution of operations on input tokens, and generated new tokens will be delivered to an outgoing edges. An object node denoted using rectangle boxes is used to temporarily hold the data tokens waiting to be processed or delivered. For simplicity, we assume that object nodes do not support *competition* and *data store* for test case generation. A flow in an activity starts from the initial node. When a token arrives at a flow final node, it will be destroyed. The flow final node has no outgoing edges, so there is no downstream effect. When no tokens exist in an activity diagram, the activity will be terminated. The activity final nodes are similar to flow final nodes, except that when a token reaches one activity final node, the entire flow will be terminated. Decision nodes and merge nodes use the same shape of diamond. Decision nodes choose one of the outgoing flows according to the value of Boolean expressions labeled on the outgoing edge. Merge nodes select only one of incoming flows to deliver to the next activity node. Forks or joins are shown by multiple arrows leaving or entering the synchronization bar, respectively, to describe the concurrent behavior of a system. When a token arrives at a fork node, it will be duplicated across the outgoing edges. Join nodes synchronize multiple flows. The tokens must be available on every incoming edge in order to be passed to outgoing edges.

Activity nodes are connected by activity edges along which tokens may flow under some condition. Activity edges include control and data flow edges as follows:

- ***Control Flow Edge***: Control flow edges indicate the execution sequence of actions.
- ***Object Flow Edge***: Object flow edges indicate the relation of data token transmissions. It provides the inputs to actions.

In our method, we simplify the syntax and semantics of UML activity diagrams. We combine the control and data token together as a new kind of token which contains both control and data information. Such token can flow through activity edges. In other words, we do not distinguish control flow edges and object flow edges in our framework.

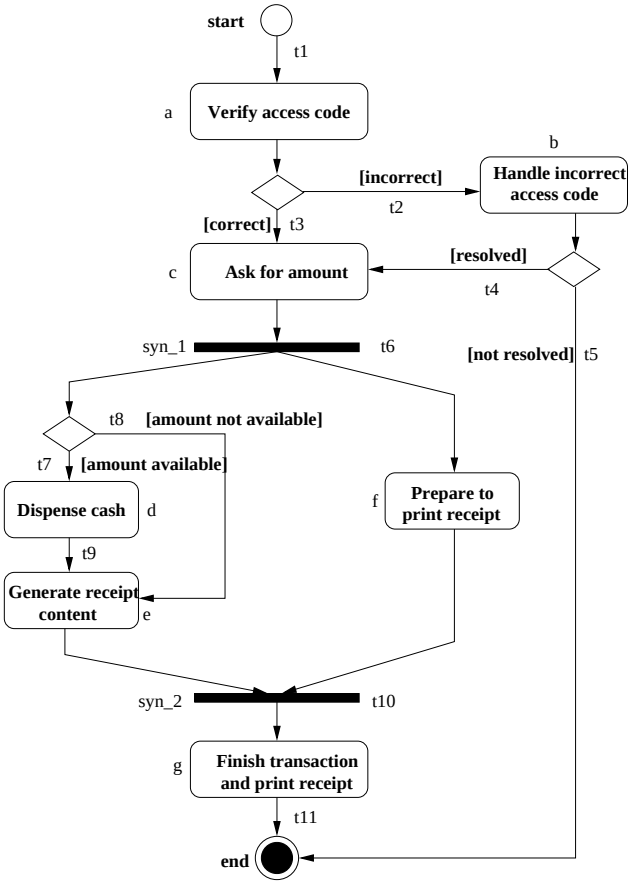


Figure 2-8. The UML activity diagram of an ATM

Figure 2-8 shows an example which uses most of the elements shown in Figure 2-7. It describes the functionality of withdrawing money from an Automated Teller Machine

(ATM) [26]. A user needs to enter the access code first. In case of failure, the user can input the access code again. The operation will abort if access code is wrong in both cases. If the input access code is right, the user can enter the amount of money he wants to withdraw. At the same time, the printer will be ready to print a receipt. Once the ATM decides whether there is enough money the user can withdraw, it provides the cash and generates the information for this transaction. Finally, the printer prints the receipt and the transaction is complete.

The token for this example contains the ATM transaction information such as the input access code and input cash amount, the context information such as the available cash amount and correct access code. In general, a token reflects all the data information required for this activity. Table 2-1 shows the composition of a token of the ATM activity diagram. It consists of 5 variables which will be used to make the decisions illustrated in Table 2-2.

Table 2-1. Break down of a token in Figure 2-8

Variable	Type	Description
access_code	string	user's access code
access_code_input	string	user access code input
access_code_resolve	string	user access code input correction
amount_input	integer	user cash amount input
amount_available	integer	cash amount available

Table 2-2. Condition on the flow edges in Figure 2-8

Activity Edge	Condition	Description
t2	incorrect	$access\_code! = access\_code\_input$
t3	correct	$access\_code = access\_code\_input$
t4	resolved	$access\_code = access\_code\_resolve$
t5	not resolved	$access\_code! = access\_code\_resolve$
t7	amount available	$amount\_input <= amount\_available$
t8	amount not available	$amount\_input > amount\_available$

2.2.2 Formal Modeling of UML Activity Diagrams

Without formalism, it is hard to describe and model the activity diagrams accurately. UML activity diagram itself is a semi-formal specification that cannot be directly mapped

to a model checker input (e.g., SMV models). We use Petri-net as an intermediate formal model between activity diagrams and SMV model, because the Petri-net formalism can capture the major functional scenarios as well as guide the translation.

Definition 2 describes the relation between the activity nodes and flow edges with a Petri-net semantics. It does not model the full features of activity diagrams and formally depicts the static abstracted structure of activity diagrams which can be used to describe the scenarios that need to be tested.

Definition 2. *An activity diagram is a directed graph described using eight-tuple (A, T, F, C, V, A, a_I, a_F) where*

- $A = \{a_1, a_2, \ldots, a_m\}$ *is a set of action nodes.*
- $T = \{t_1, t_2, \ldots, t_n\}$ *is a set of completion transitions.*
- $F \subseteq \{A \times T\} \cup \{T \times A\}$ *is a set of flow edges between activity nodes and completion transitions.*
- $C = \{c_1, c_2, \ldots, c_n\}$ *is a finite set of guard conditions. Here, c_i $(1 \leq i \leq n)$ is a predicate (expression) based on the input variables. There is a mapping from $f_i \in F$ to c_i, referred as $Cond(f_i) = c_i$.*
- *Let V be the set of all possible assignments for input variables $V_1, V_2, \ldots, V_k$ where k is a positive integer.*
- $M : A \times V \rightarrow V$ *is a mapping that describes the value change of the input variables inside an activity node.*
- $a_I \in A$ *is the initial node, and $a_F \in A$ is the final node. There is only one completion transition $t \in T$ and $c \in C$ such that $(a_I, t) \in F$, and for any $t' \in T$, $(t', a_I) \notin F$ and $(a_F, t') \notin F$.* ■

In our formalization, a node can be an action node, an initial node or a final node. We use the *completion transition* and *flow edge* to model the behavior of the control nodes. In the graph, the nodes are connected by flow edges associated with a completion transition. Because activity diagrams allow tokens to exist in the flows concurrently,

the completion transition can be used to synchronize the token flows. If a completion transition has multiple incoming flow edges, it will do the join operation. If there are multiple outgoing flow edges, then it will do the fork operation. For each flow edge, there may be a condition which can guide the token traverse. The graph has one initial node that indicates the start of control and data flows. Activity diagrams have two kinds of final nodes: flow final nodes and activity final nodes. We combine them together and use a join operation to get a new activity final node. So in the definition there is only one final node.

When analyzing dynamic behaviors of an activity diagram, we need to use the *states* (a set of actions executing concurrently) to model the status of a system. Current state (denoted by CS) of an activity diagram indicates the actions which are being activated.

Definition 3. *Let D be an activity diagram. The current state CS of D is a subset of A. For any transition $t \in T$,*

- *${}^\bullet t$ denotes the preset of t, then ${}^\bullet t = \{\ a \mid (a,t) \in\ F\}$.*
- *$t^\bullet$ denotes the postset of t, then $t^\bullet = \{a \mid (t,a) \in F\}$.*
- *enabled(CS) denotes the set of completion transitions that are associated with the outgoing flow edges of CS, then enabled(CS) $= \{\ t \mid {}^\bullet t\ \subseteq\ CS\ \}$.*
- *firable(CS) denotes the set of transitions that can be fired from CS , then firable(CS)=$\{\ t \mid t \in$ enabled(CS) $\bigwedge$ ${}^\bullet t$ are all completed $\bigwedge$ $\exists\ n\ \in\ A.\ \ Cond((t,n))$ is satisfied $\bigwedge$ $(CS - {}^\bullet t) \cap t^\bullet = \emptyset\}$. After some t is fired, the new current state $CS' = fire(CS,t) = (CS - {}^\bullet t) \cup t^\bullet$.* ■

The current state of an activity diagram indicates which activity nodes are holding the tokens. For example, when $\{d, f\}$ is the current state of the activity diagram in Figure 2-8, two tokens are in the activity nodes d and f individually. At this time, only the transition associated with t_9 is firable. If it is fired, then the next state is $\{e, f\}$.

Because of the inherent concurrency, several transitions can be fired at the same time. For an activity diagram, all the firable transitions in a state form a *concurrent transition*.

Definition 4. *Let D be an activity diagram. For a state CS of D, a concurrent transition τ is a set of completion transitions $t_1, t_2, ..., t_n \in firable(CS)$ where*

1. $\forall\ i, j (1 \leq i < j \leq n)$, ${}^\bullet t_i \cap {}^\bullet t_j = \emptyset$;
2. $\forall\ t \in (enabled(CS) - \{t_1, t_2, ..., t_n\})$, *there exists* i $(1 \leq i \leq n)$ *such that* ${}^\bullet t \cap {}^\bullet t_i \neq \emptyset$.

After firing τ from state CS, the current state $CS' = fire(CS, \tau) = \bigcup_{i=1}^{n}(fire(CS, t_i)) = \bigcup_{i=1}^{n}((CS - {}^\bullet t_i) \bigcup t_i^\bullet)$. ■

An instance of dynamic behavior of an activity diagram can be represented by a sequence of states and concurrent transitions. We call it a **path** of the activity diagram. Because a path may have cycles, during the model checking, it is hard to determine the cycle numbers, so we neglect the cycles on a path. We call such a path as **key path**.

Definition 5. *A path ρ of the activity diagram D is a sequence of states and concurrent transitions, let*

$$\rho = s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_{n-1}} s_n$$

where $s_0 = \{a_I\}$, $s_n = \{a_F\}$, *and* $s_{i+1} = fire(s_i, \tau_i)$ *for any* i $(0 \leq i < n)$. *ρ is a* **key path** *if there is no state repetition in ρ, i.e.* $\forall i, j\ (0 < i < j \leq n)$, $s_i \bigcap s_j = \emptyset$. ■

There are five key paths in Figure 2-8:

- ρ_1=$\{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t5\}} \{end\}$
- ρ_2=$\{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$,
- ρ_3=$\{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t8\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$,
- ρ_4=$\{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$,
- ρ_5=$\{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t8\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$.

We insert a *dummy* node here because we assume that outgoing edges of the fork node must connect to an activity rather than a selection node. For a key path, when firing transitions, we need to consider guard conditions. For clarity, in Figure 2-8, we did not label the condition guards for each transition.

Definition 6. *Let D be an activity diagram. An interaction of the activity diagram is a set of activity nodes (actions) that can be activated simultaneously. A "k-interaction" is a set that contains k activity nodes.*

In order to detect whether a concurrent state of an activity diagram is reachable or can be activated, we use the term **interaction** [1] to describe the scenario that a set of actions can be activated simultaneously. For example, in the Figure 2-8, $\{d, f\}$ is an example of "2-interaction" in the ATM.

2.2.3 Transformation from UML Activity Diagrams to SMV

Our technique can extract both the control and data flows by parsing a UML activity diagram. The translation consists of two parts: static information extraction and dynamic information extraction. Static information extraction analyzes the structure of an activity diagram and then generates a skeleton of the SMV input. The dynamic information extraction analyzes the dynamic behavior of the system by focusing on control and data flow analysis (i.e. the state change of activities, data manipulation in activities and the condition of the transitions).

2.2.3.1 Static Information Extraction

This step collects both the input data manipulated by the activities and the predicates used as guard conditions of the transitions. For example in Figure 2-8, there are five input data variables that determine the data and control flows: *access_code*, *access_code_input*, *access_code_resolve*, *amount_input*, and *amount_available*. Because there may be a number of possible values for a variable, during model checking it will cause the state space explosion. In our approach, we adopt the model checker SMV which does not support complex data types (e.g., float, double and etc.). For each variable, it is required

[1] Unlike the interaction in UML Interaction overview diagram, the interaction here means that several actions are actived at the same time.

that the value range should be specified explicitly. To avoid state space explosion, we use the following methods to reduce the complexity of data types:

- *Scaling:* Scaling is used to proportionally reduce the value range of a variable.
- *Reduction:* Reduction is used to reduce the cardinality of possible values for a variable.

Since it is hard to implement the above techniques automatically, before the SMV translation, the variable type information is tuned manually for activity diagrams.

In our translation, we assign each activity with a *state variable* which has three possible state values: *unvisited* (0), *visiting* (1) and *visited* (2). *Unvisited* indicates that no token has passed through this activity node. $Visiting$ indicates currently the activity is holding one or more tokens. *visited* indicates that some token has passed through this activity node and currently there is no token in this activity node. The extraction procedure instantiates the activity state variables and assigns suitable values to them. During initialization, the initial activity node is assigned *visiting* that means there is a token ready at the initial state. Other nodes are initialized to *unvisited*. Also, we assign each flow edge a state variable which has two possible values: $fired$ (1) and $unfired$ (0). $Fired$ means some tokens have flowed from the incoming activity nodes to its outgoing activity nodes. $Unfired$ means no token has passed through this activity edge. Initially we set them with value 0.

Figure 2-9 shows the generated skeleton of Figure 2-8 in SMV format [20, 56]. There are 3 modules in this skeleton. The module *state* defines the token information (described in Table 2-1) as well as the state variable for activity nodes and flow edges. For example, $verify\_access\_code$ is a state variable for an action with three states. Initially it is assigned the state *unvisited* (0). Module ATM gives a static skeleton without dynamic behavior information. In this phase, we just collect variables without any processing. The missing state transition details will be described in Section 2.2.3.2. The module *main* creates the module instances and elaborates them together. For example, *st* is an instance

of state module and *atm* is an instance of ATM module. We bind the *st* and *atm* together, because *atm* will handle the state changes of variables in *st*.

```
MODULE state
 VAR
   access_code: { A1, B1, C1 };
   access_code_input: { A1, B1, C1, D1 };
   start: 0..2;
   syn_1: 0..2;
   verify_access_code: 0..2;
   t2_cond: 0..1;
   t3_cond: 0..1;
   ......
 ASSIGN
   init(start):=1;
   init(syn_1):=0;
   init(verify_access_code):=0;
   ......
MODULE ATM(st)
 ASSIGN
   next(st.start):=
   next(st.t2_cond):=
   ......
   next(st.prepare_print_receipt):=
   ......
   next(st.dispense_cash):=
   next(st.t7_cond):=
   ......
MODULE main() {
   st: state;  atm: ATM(st);
   p_print:  prepare_print(st);
   check:  check_amount(st);
}
```

Figure 2-9. The generated skeleton after structure extraction

2.2.3.2 Dynamic Information Extraction

After static information extraction, we need to extract both data manipulations and transitions of state variables, because they will determine the data and control flows.

In our method, we define a set of rules that specify the state transition for each activity node and the value changes of each data. Figure 2-10 shows the details of the rules. In these rules, we use the preset and postset notations. In these rules, the assignment and constraint to a set means the assignment and constraint to each element

Rule 1: If n is an initial node

$init(n) := 1;$
$next(n) := 2;$

Rule 2: If n is a final node, and there are k incoming transitions $t_1, t_2 ... t_k$.

$init(n) := 0;$
$next(n) := case$
$((^\bullet t_1 = 1 \ \& \ cond(t_1)) \mid (^\bullet t_2 = 1 \ \& \ cond(t_2)) \mid$
$... \mid (^\bullet t_k = 1 \ \& \ cond(t_k))) : 2;$
$1 : n;$
$esac;$

Rule 3: If n is an activity node (not join or fork), and there are k incoming transitions $t_1, t_2 ... t_k$.

$init(n) := 0;$
$next(n) := case$
$n = 1 : 2;$
$(^\bullet t_1 = 1 \ \& \ cond(t_1)) \mid (^\bullet t_2 = 1 \ \& \ cond(t_2)) \mid$
$... \mid (^\bullet t_k = 1 \ \& \ cond(t_k))) : 1;$
$1 : n;$
$esac;$

Rule 4: If n is a fork node, and the corresponding transition is t.

$init(n) := 0;$
$next(n) := case$
$n = 1 \ \& \ t^\bullet > 0 : 2;$
$^\bullet t = 1 : 1;$
$1 : n;$
$esac;$

Rule 5: If n is a join node of transition t, and $a_1, a_2 ... a_k$ are k elements of $^\bullet t$.

$init(n) := 0;$
$next(n) := case$
$n = 1 : 2;$
$n = 0 \ \& \ (a_1 + a_2 + ... + a_k = 2 * k) : 1;$
$n = 2 \ \& \ (a_1 + a_2 + ... + a_k < 2 * k) : 0;$
$1 : n;$
$esac;$

Rule 6: If t is a transition which corresponds to the flow edges.

$init(t) := 0;$
$next(t) := case$
$! \ cond(t) \ \& \ ^\bullet t = 1 : 0;$
$cond(t) \ \& \ ^\bullet t = 1 : 1;$
$1 : t;$
$esac;$Dynamic Information

Rule 7: If v is a variable whose new value is changed by expression exp_i in the activity act_i $(1 \leq i \leq n)$.

$next(v) := case$
$act_1 = 1 : exp_1;$
$act_2 = 1 : exp_2;$
$.....$
$act_n = 1 : exp_n;$
$1 : v;$
$esac;$

Figure 2-10. Translation rules for state and data transitions

in the set. For example, if ${}^\bullet t = \{a_1, a_2, \ldots a_k\}$, then ${}^\bullet t = 1$ means $a_1 = 1 \,\&a_2 = 1 \,\& \ldots \,\& a_k = 1$ and $cond(t)$ means $cond((a_1, t)) \,\& cond((a_2, t)) \,\& \ldots \,\& cond((a_k, t))$.

Rule 1 specifies the translation rule for the initial node. The token will be first put at the initial state and the node is marked as *visited* in the next step. Rule 2 specifies the translation rule for the final node. At first, the state is *unvisited*, when one of the incoming edges is activated, its state will become *visited*. Rule 3 defines the state changes of an activity. Initially, the state of an activity is *unvisited*. If the incoming edge is activated, the state will become *visiting* in the next step. If the current state is *visiting*, the state will change to *visited* in the next step. Rule 4 presents the state transition of the fork nodes. When the incoming edge is activated, the fork node will maintain the *visiting* status until all the outgoing edges are visiting or visited. Rule 5 gives the state transition of join nodes. The join node is used to synchronize the token flows. When all the incoming flows are ready, the transition corresponding to the join node can be fired. In this rule, if we want to fire the transition, we need to wait until all the activity nodes in the preset of the transition are visited. Rule 6 shows how to manipulate the state change of the transition when it is fired. Rule 7 presents the translation for value change of the variables. If an activity performs some operation on the variable, we can modify the value of the variable only when the activity state is *visiting*.

2.2.4 A Prototype Tool For UML to SMV Translation

Based on the framework proposed in Section 2.2.3, we developed a prototype tool which can automate the process of test case generation. The tool takes three inputs: type definition of the data which is used in the activity diagram, the context information which set the parameters for the execution of an activity diagram (e.g. when to trigger the initial node and so on), and UML activity diagrams. The UML activity diagrams are stored in the format of XML Metadata Interchange (XMI) files. The tool can parse the XMI files to get the static and dynamic information for formal model translation. Combined with the

context information and data type information, a formal model can be generated using the proposed mapping rules.

2.3 Case Study

This section presents five representative high-level specifications for SoC designs. First, it describes three TLM specifications: router, MIPS processor and Alpha AXP processor. Next, it presents two UML activity diagrams: a control system and a online stock exchange system (OSES).

2.3.1 Example 1: A Router

Figure 2-11 shows the TLM structure of a router design. The router consists of five modules: one master, one router and three slaves. The SystemC program consists of 4 classes (one class for packet definition, one class for the driver, one class for the router and one class for the slave), 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.

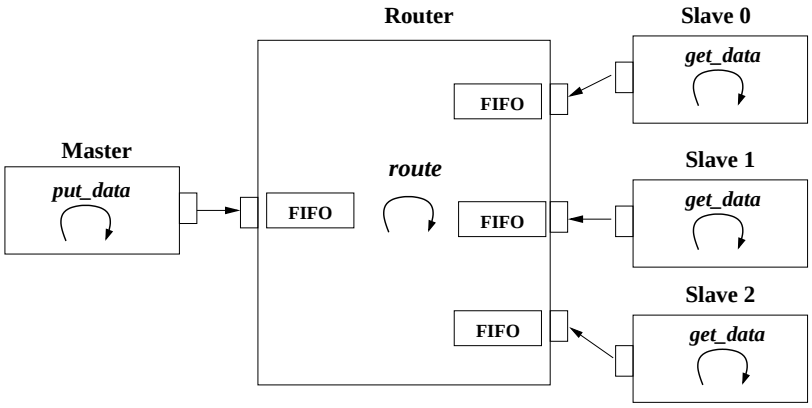


Figure 2-11. The TLM structure of the router

At the beginning of a transaction, the master module creates a packet. 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 header of the packet 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 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 controlled by the address *to_chan* in the packet header. By using our proposed approach in Section 2.1.2, the automatically generated SMV model contains four modules and 145 lines of code.

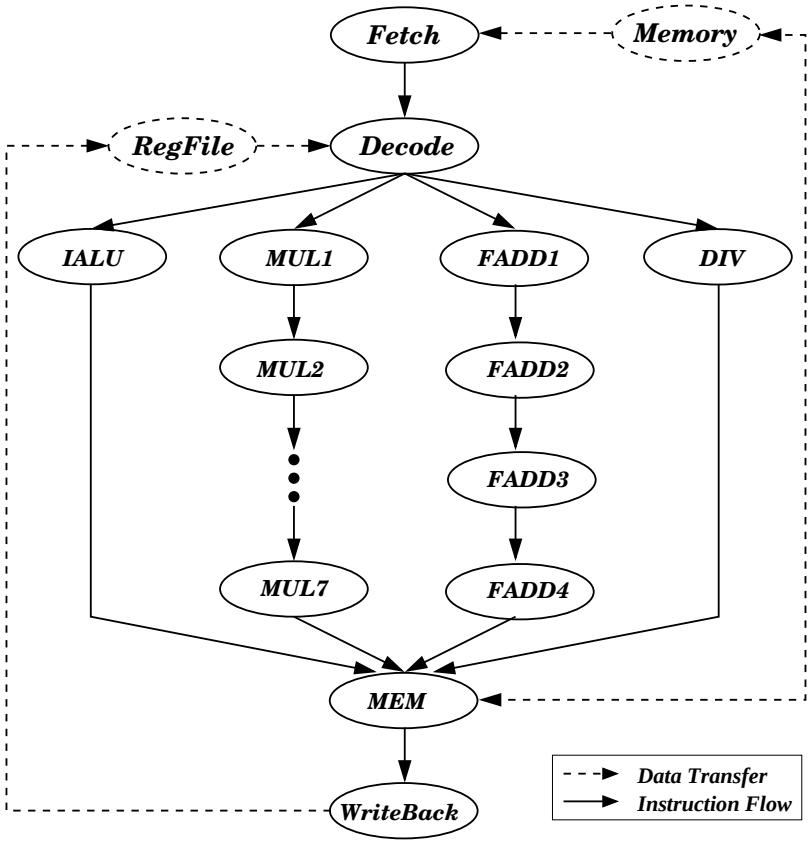


Figure 2-12. Graph model of a VLIW MIPS processor

2.3.2 Example 2: A MIPS Processor

Figure 2-12 shows a simplified version of a single-issue MIPS [35] architecture. It has five pipeline stages: fetch, decode, execute, memory (MEM), and writeback. The *execute* stage has four parallel execution paths: integer ALU, 7 stage multiplier (MUL1 - MUL7), four stage floating-point adder (FADD1 - FADD4), and multi-cycle divider (DIV). The oval boxes represent units and dashed boxes represent storages. The solid lines

represent instruction-transfer paths and dotted lines represent data-transfer paths. After TLM-to-SMV transformation, the SMV model has 1134 lines of code.

2.3.3 Example 3: An Alpha Processor

Figure 2-13 shows a simplified TLM specification structure of the Alpha AXP processor. It consists of five stages: Fetch (IF), Decode (ID), Execute (EX), Memory (MEM) and Writeback (WB). IF module fetches instructions from the instruction memory. ID module decodes instructions and fetches the operand data if necessary. EX module does ALU operations as well as asserts whether the conditional or unconditional branch happens. Memory module reads and writes data from (to) the data memory. Writeback module stores the result in specified registers. The communication between two modules uses the port binding associated with a blocking FIFO channel with only 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 an incoming instruction. 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. After the TLM-to-SMV transformation, the generated SMV model has 6 modules and 821 lines of code.

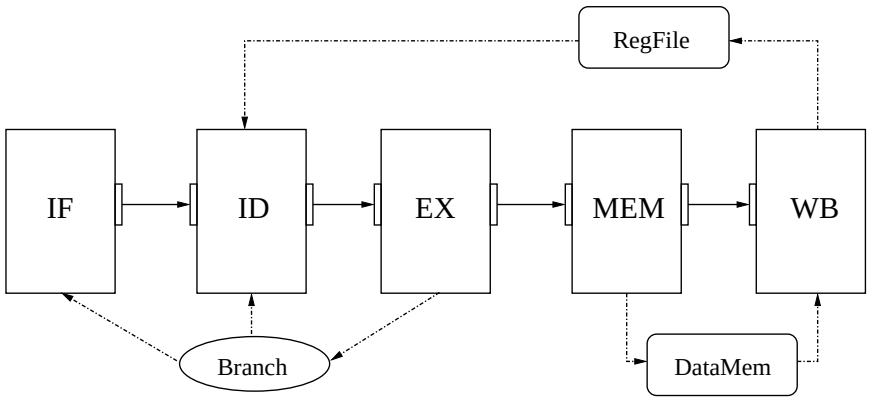


Figure 2-13. TLM of the Alpha AXP processor

2.3.4 Example 4: A Control System

As shown in Figure 2-14, the UML activity diagram representation of the control system consists of 17 activities, 23 transitions and 6 key paths. It has a global integer variable i which determines token flows. The generated SMV files have 365 lines of code.

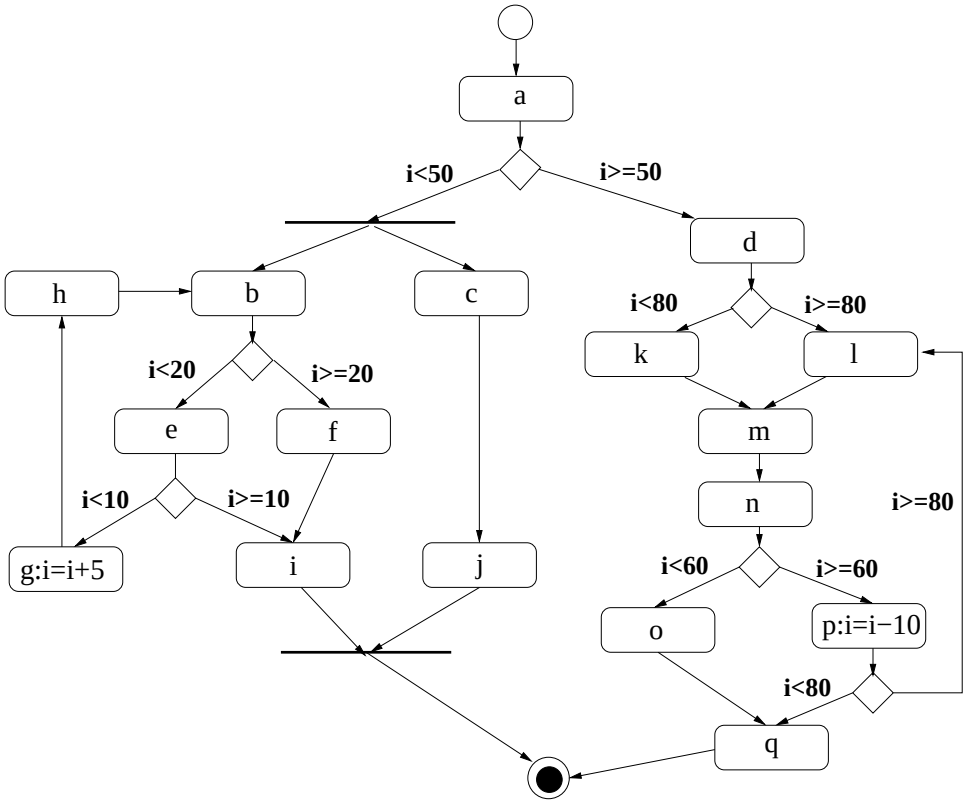


Figure 2-14. The activity diagram for a control system

2.3.5 Example 5: A Stock Exchange System

The purpose of the on-line stock exchange system (OSES) is to process three scenarios: accept, check and execute the customer's orders (market order and limit order). The system uses the UML activity diagram as its behavior specification. Figure 2-15 shows the specification of the stock system. It has 27 activities, 29 transitions and 18 key paths. The generated SMV model has 756 lines of code.

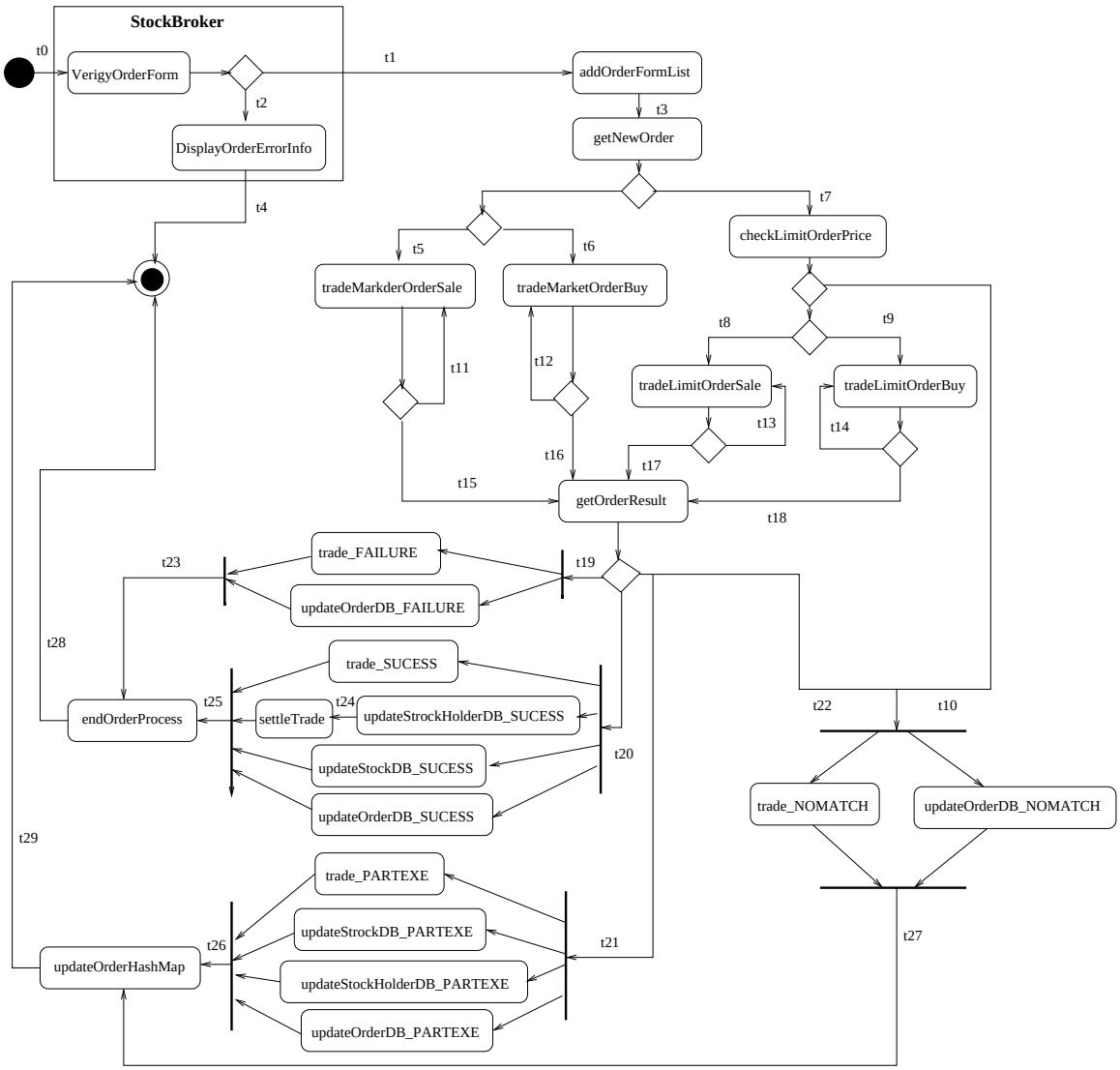


Figure 2-15. The activity diagram for a stock exchange system

2.4 Summary

This chapter introduced two high level design specifications for SoC designs: SystemC TLM to model the hardware bahavior and UML activity diagrams to describe the concurrent software behavior. The main contribution of this chapter is to devise mechanisms to extract both static and dynamic information from specifications and then convert them to formal SMV models to enable automatic analysis and directed test generation.

CHAPTER 3
COVERAGE-DRIVEN AUTOMATIC GENERATION OF DIRECTED TESTS

Figure 3-1 presents our methodology for specification driven test generation using model checking techniques. First, a design is described using a specification language that can capture both structure and behavior of SoC systems. Next, the design specification is translated to a formal model (described in Chapter 2), and the properties in the form of CTL or LTL formulas are generated based on the functional *fault models* (see Section 3.1). Finally, the properties are applied on the formal model using a model checker to generate required tests (counterexamples). The model checker exhaustively searches all reachable states of the model to check if any state violates the property. If it finds a violation, it will produce a counterexample. The counterexample contains a sequence of input assignments from an initial state to a state where the specified property fails. If we assume that the design is correct and the property is a false property, the model checker will always generate a valid counterexample unless it encounters state space explosion problem. The generated tests can be used for validating both specifications and implementations.

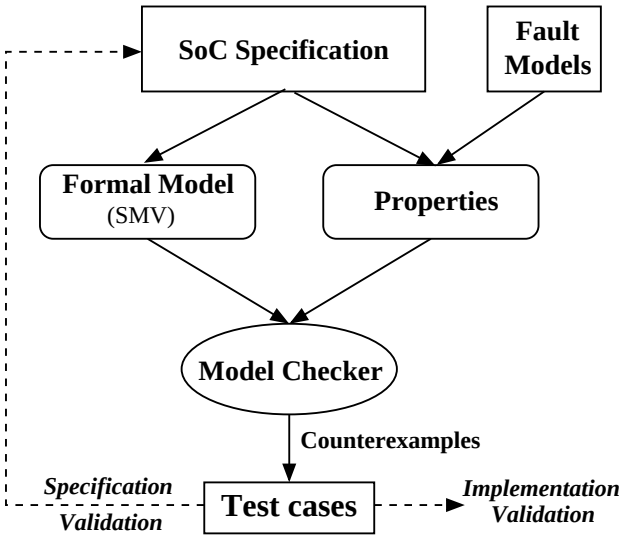


Figure 3-1. Test generation using model checking

There are three major challenges in implementing this test generation methodology in practice: i) automatic extraction of formal models from SoC specifications, ii) development

of efficient functional fault models and associated coverage-driven property generation, and iii) how to address state space explosion problem. We have discussed the formal model generation in Chapter 2. Chapter 4 to 6 will present novel approaches for addressing state space explosion problem. In the following sections, we will focus on the automatic generation of properties and corresponding directed tests. The rest of this chapter is organized as follows. Section 3.1 presents the property generation using various fault models. Section 3.2 describes the test generation methods using both unbounded model checking and bounded model checking. Section 3.3 demonstrates two case studies based on UML activity diagrams. Finally, Section 3.4 summarizes the chapter.

3.1 Coverage-Driven Property 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 $\sim F(p)$ asserts that a specified scenario can not happen (i.e., property p can not 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 used as a test to validate the specified scenario. In our method, the quality of the generated tests is determined by the corresponding properties. During the property generation, it is required to guarantee that the generated properties can sufficiently validate the system.

The coverage metrics [32] 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 approach, 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 a 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 $\sim p$ as a fault. By checking the property $\sim F(p)$, we can derive a test to check

the scenario where property p holds. Since in this dissertation we focus on only safety property generation for above fault models, majority of the properties will be in the form of $\sim$ F (p) or G($\sim$ p). However, other forms of safety properties are also possible and allowed in our framework.

3.1.1 Fault Models

Fault model [28] plays an important role in directed test generation. Each fault model represents a kind of "false functional scenarios". The efficiency of directed tests is directly related to the generated properties which in turn are related to the associated fault model. The following three subsections present the generic fault models for graph model as well as its two variants: fault models for SystemC TLM designs and fault models for UML activity diagrams. It is important to note that these fault models are by no means the "golden" model rather it is a representative model which can be refined or modified for improved verification methodology.

3.1.1.1 Generic Fault Models for Graph Based Models

For a simple graph model, there is only node and edge information. By investigating the status of the nodes and edges we can infer various system behaviors. There are four widely used fault models for graph models as follows.

- **Node Fault:** Each node is faulty. For example, a node cannot be activated.
- **Edge Fault:** Each edge is faulty. For example, the respective nodes cannot be activated in that order.
- **Path Fault:** Each execution path is faulty. For example, the associated nodes and edges are either faulty or their behavior cannot be composed correctly to activate the path.
- **Interaction Fault:** Each interaction is faulty. For example, an interaction involving a set of nodes cannot be activated simultaneously.

We generate one property for each fault in a fault model. So the transformation from the fault model to the properties in the form of temporal logic is a one-to-one mapping.

Because a fault is already a negation of the system required behavior, it can be directly used to derive a property for test generation.

Let's consider Figure 2-12 in Chapter 2 as an example of a graph model. The following example shows four properties (one for each fault type) for the graph model.

```
Property 1: The node Fetch cannot be activated.
  LTL formula: ~ F (fetch_active = 1)
Property 2: The edge between node MUL4 and MUL5 cannot be activated.
  LTL formula: ~ F(mul4_active = 1 -> X(mul5_active = 1))
Property 3: The path of FADD cannot be activated.
  LTL formula: ~F (fetch_active = 1 & decode_active = 1 & fadd1_active = 1
    & fadd2_active = 1 & fadd3_active = 1 & fadd4_active = 1 & mem_active = 1
    & writeback_active = 1)
Property 4: DIV, FADD4 and MUL7 cannot be activated at the same time.
  LTL formula: ~ F(div_active = 1 & fadd4_active = 1 & mul7_active = 1)
```

Depending on the design, the generated properties may lead to redundant tests. Therefore, property compaction can be employed to reduce the number of properties without affecting the coverage goal [45].

3.1.1.2 Fault Models for SystemC TLM Specifications

In TLM, transaction data, transaction flow and events are three most important factors. They reflect both the structure and behavior information of system level hardware designs. In addition to the fault models presented in Section 3.1.1.1, in our framework we have defined another three fault models based on transactions as follows.

- **Transaction data fault model** investigates the content of the variables relevant to the transaction. For each variable, it is assumed that a specific value can/cannot be assigned in some scenario.

- **Transaction flow fault model** investigates the controls along the path where the transaction flows. For each branch condition along the transaction path, it is assumed that it can/cannot be activated in some scenario.

- **Transaction event fault model** investigates the event occurrence within a transaction. For each event, it is assumed that it can/cannot be activated.

Transaction data fault model deals with the possible value assignment for each part of the transaction data. However, during property generation, due to the large size of value space, trying all possible values of a data is time-consuming and impractical. By checking each bit of a variable (data bit fault) separately, the data content coverage can be partially guaranteed. Transaction flow fault model deals with the controls along with the transaction flow. To ensure transaction flow coverage, one can cover branch conditions which exist in *if-then-else* and *switch-case* statements. The goal is to check all possible transaction flows. Transaction event indicates the execution stage of a transaction or the interaction between processes. The activation and the order of transaction events is an important issue. Section 7.2.1.1 gives an example for each type of TLM transaction faults.

3.1.1.3 Fault Models for UML Activity Diagrams

In traditional software testing, the definition of testing adequacy is given in [32] as a measurement function. The case of UML activity diagrams is different because it is in the form of model instead of code. Especially the coverage of activity diagram is more complex because of the concurrency. We create four fault models for UML activity diagrams (AD) which are similar to the generic fault models presented in Section 3.1.1.1 as follows.

- **Activity Fault Model**. For each activity of AD, the model assumes that such activity is not reachable.
- **Transition Fault Model**. For each transition of AD, the model assumes that such transition can not be fired.
- **Key Path Fault Model**. For each key path of AD, there is no corresponding executable *path*.
- **Interaction Fault Model**. For each interaction of AD, the activities associated with the interaction cannot be activated at the same time.

From these four different models, we can generate various properties to validate activity diagrams. The activity fault model can be used to check the reachability of each activity. So it can be used to check whether there exists infinite loops in the system. The

transition fault model can be used to check the execution order of the activities. It can also be used to check whether the condition guard of the transition can be satisfied. We also need to check all the dynamic behaviors of the system, so key path fault model is preferable in this case. The interaction fault model can be used to check whether several activities can be activated simultaneously. In general, if all the interactions have only one activity, the interaction fault model is the same as the activity fault model.

The following example shows four properties (one for each fault type) for the UML activity diagram shown in Figure 2-8.

```
Property 1:The activity dispense_cash is not reachable.
  LTL formula: ~ F (st.dispense_cash = 2)
Property 2:The transition with condition [amount available] can not be fired.
  LTL formula: ~ F( st.t7_cond = 1 )
Property 3:The key path 4 can not be covered.
  LTL formula: ~F ( st.start = 2 & st.verify_access_code=2 & st.handle_access_code = 2
      & st.ask_for_amount = 2 & st.prepare_print_receipt = 2 & st.dispense_cash = 2
      & st.generate_receipt_content=2 & st.finish_transaction_print_receipt = 2
      & st.end = 2 & st.t2_cond=1 & st.t4_cond=1 & st.t7_cond=1 )
Property 4:The activities dispense_cash and   prepare_to_print_receipt can not
           be activated simultaneously.
  LTL formula: ~ F( st.dispense_cash = 1 & st.prepare_to_print_receipt =1)
```

Figure 3-2. Fault model examples

3.1.2 Functional Coverage Based on Fault Models

The *functional coverage* of a system level design is defined based on the overall faults of a fault model and the faults activated by the derived tests.

Definition 7. *For a design D, we are given its fault model F and a test suite T. F is a complete set of same type faults. Each fault indicates the negation of a required functional behavior of D. T is a set of directed tests which is derived from F. By applying T on D, the functional coverage D_F using T can be calculated as:*

$$D_F = \frac{\#\ of\ exercised\ F\ type\ functional\ scenarios}{|F|}$$

3.2 Test Generation using Model Checking Techniques

Model checking [21, 56] is a formal method that can enumerate all the possible state to check whether a finite state system M satisfies a property p in the form of temporal logic (e.g. LTL or CTL [21]), i.e., $M \models p$. When the property fails at some state, it will report a counterexample to falsify the specified property p. Let's consider a test generation example for a pipelined processor. To activate a fault in the stall functionality of a decode unit (i.e., the decode unit can never be stalled), the system will generate the property *"$\sim$ F(dec_stall = 1)"*. Taking the property and the processor model as inputs, the model checker will generate a counterexample to stall the decode unit which can be used as a test to activate the stall functionality of the decode unit. The counterexample contains a sequence of instructions from an initial state to a state where the property fails. In this section, we briefly introduce two kinds of test generation methods based on different model checking techniques.

3.2.1 Test Generation using Unbounded Model Checking

This section introduces the preliminary knowledge of the unbounded model checking and gives a general algorithm for test generation.

3.2.1.1 Unbounded Model Checking

Symbolic Model Verifier (SMV [21]) is a widely used model checker. By taking model of the design and temporal logic properties as inputs, SMV can determine whether the design satisfies the property. During the verification, SMV abstracts the given model into a formal Kripke structure which consists of a set of states, a set of transitions between states, and a function that labels each state with a set of properties that are true in this state. Then SMV does the state space search on this Kripke structure. The model checking algorithm stops because: i) it encounters a false state for the property, then the counterexample which leads to this state will be generated, or ii) all the states have been explored and no error is detected. Generally, the implementation of the state search

adopts the data structure based on BDDs. However, they are not scalable to handle large practical systems in practice.

3.2.1.2 Test Generation Algorithm

Algorithm 1 outlines the general test generation approach using unbounded model checking (UBMC) [47, 59, 60]. The algorithm takes a SMV model M and a set of false properties P (based on coverage) as inputs and generates a test suite extracted from counterexamples. For each property P_i, one test is generated. The algorithm iterates until all the properties are checked. For each iteration, one property is handled and the corresponding test will be generated. In this dissertation, we focus on the generation of safety properties which assert that the specified scenarios cannot happen.

Algorithm 1: Test Generation using UBMC

Input: i) SMV Model, M, and ii) A set of false properties P

Output: Testsuite

TestSuite = ϕ;

for *each property* P_i *in the set* P **do**
    $test_i$ = ModelChecking(P_i, M);
    TestSuite = TestSuite $\cup$ $test_i$;
end

return TestSuite;

3.2.2 Test Generation using Bounded Model Checking

This section introduces the preliminary knowledge of the SAT-based Bounded Model Checking (BMC). It also describes how to pre-determine the bounds of properties. Finally a BMC based test generation algorithm is presented.

3.2.2.1 SAT-Based Bounded Model Checking

For complex designs and properties, BDDs based methods usually cause the state space explosion problem. As an alternative, Boolean satisfiability (SAT) based approaches have emerged, especially for the bounded model checking (BMC). SAT-based BMC [11] is a promising method which can prove whether there is a counterexample for the

property within a given bound. Given a model M, a safety property p, and a bound k, SAT-based BMC will unfold the model k times and encode it using the the Boolean formula Equation (3–1).

$$BMC(M, p, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i) \tag{3–1}$$

Here, $I(s_0)$ means the initial state of the system, $T(s_i, s_{i+1})$ describes the state transition from state s_i to state s_{i+1}, and $p(s_i)$ tests whether property p holds on state s_i. Then this formula will be transformed to a Conjunctive Normal Form (CNF) and checked by a SAT solver. If there is a satisfiable assignment, the property is false and a satisfiable variable assignment will be reported, i.e., $M \not\models_k p$. Otherwise, it implies that the property is true within the specified time steps. In other words, there is no counterexample with length k for this property, written $M \models_k p$. Test generation using BMC is similar to model checking based approach except that it needs to determine the bound for each property. SAT-based BMC takes model M, negated property p_i, and $bound_i$ as inputs and generates a counterexample (test).

3.2.2.2 Test Generation Algorithm

Algorithm 2 describes the widely used test generation procedure using BMC [46, 62]. This algorithm takes the model M generated from a design model and properties as inputs and generates test suite extracted from the counterexamples. For each property P_i, one test is generated. The algorithm iterates until all the properties are covered. In each iteration, the bound k_i of each property P_i is decided. SAT-based BMC takes model M, negated property P_i, and bound k_i as inputs and generates a counterexample (test).

During the test case generation, bound determination plays an important role. If it can be known a priori, SAT-based BMC can be more effective than BDD based model checking techniques. However, any incorrect bound determination will increase test case generation time as well as memory requirement. Therefore, the techniques of deciding property bounds determine the efficiency of test case generation using SAT-based BMC. In

our method, because the property is derived from models, the bound can be derived from the structure of the models.

Algorithm 2: Test Generation using BMC

Input: i) Design Model, M and ii) A set of false properties P (based on fault models)

Output: Testsuite

TestSuite = ϕ;

for *each property* P_i *in the set* P **do**

  $bound_i$ = DetermineBound(M, P_i);

  $test_i$ = BoundedModelChecking(P_i, M, $bound_i$);

  TestSuite = TestSuite $\cup$ $test_i$;

end

return TestSuite;

3.2.2.3 Determination of Bound

Biere et al. [10] described several ways to determine the bound. If $M \models_k p$ for all k within the bound, then $M \models p$. However there is no deterministic way of computing the bound of the system. In fact, determining the minimal bound for a property is as hard as the model checking itself. So bounded model checking is promising only when the bound can be pre-determined and is shallow.

According to the definition of the diameter in [10], the bound for each node error instance is decided by the temporal distance between the root node and the node under verification. For example, in UML activity diagrams, the bound for the key path error is determined by the activities and transitions along the path. In Figure 2-8, the length of the key path $\rho_4 = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t2\}} \{b\} \xrightarrow{\{t4\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\} \xrightarrow{\{t7\}} \{d, f\} \xrightarrow{\{t9\}} \{e, f\} \xrightarrow{\{t10\}} \{g\} \xrightarrow{\{t11\}} \{end\}$ is 9. The property derived for this key path is shown in the Figure 3-2. In our translation rules, an activity state transition needs one step delay. Fork node needs one step delay, and join node needs two steps delay. One step delay at the start node is also required. The bound size will be $9 + 1 + 2 + 1 = 13$. The bound of the activity error or transition error is determined by the delay of activities and transitions

on a valid shortest path from the *start* node to the activity or transition which need to be verified in the UML activity diagram. For example, when we want to check the activity error model instance "*prepare_to_print_receipt* can not be activated", the system will generate the property $\sim F(st.prepare\_print\_receipt = 2)$. The shortest path from start to such an activity is $\rho = \{start\} \xrightarrow{\{t1\}} \{a\} \xrightarrow{\{t3\}} \{c\} \xrightarrow{\{t6\}} \{dummy, f\}$. In a similar way, the bound for this property is 4+1+1=6. Sometimes in the system, there is a counter that acts like a clock which counts the execution steps. Such variable in a property will affect the bound of the property. For example, because of the introduction of a counter, the property $\sim F(clk = 10 \ \& \ st.prepare\_print\_receipt = 2)$ has a bound of 10 instead of 6.

Different properties based on different fault models have different methods to compute the bounds. Assume that there is no counter variables, the determination of the bound of a graph based model can use the following rules:

- **Node or edge based faults**. Extract all the paths without loops from the initial node to the target node or edge. Calculate the bound for each extracted path and choose the shortest one as the property bound.
- **Path based faults**. Calculate the bounds for the path based on the delay of nodes and edges on the path.
- **Interaction faults**. Calculate the bound for each element (node or edge) in the interaction. Choose the largest bound as the property bound.

If a property contains a counter variable. Then bound of the property is the larger one of the counter value and the bound calculated using the above rules. Therefore, the complexity of bound determination is polynomial to the nodes in the graph-based models. In general, it is more efficient to use BMC for shallow counterexamples when the bound can be pre-determined.

3.3 Case Studies

In this section, we demonstrate two case studies for UML activity diagrams: a control system and a on-line stock exchange system. We do not give the case study for TLM

designs since Section 7.3 will give the details of automated directed test generation for TLM designs. We compared our model checking based approach with the random test based method [51], which is the best known result in the category of test generation for UML activity diagrams. The experimental results indicate that our method can drastically reduce the overall validation effort by producing fewer tests. Furthermore, for UML activity diagrams, the generated high-level test can be directly applied on the low-level implementations (e.g. Java code). Therefore it can be used to check the consistency between UML activity diagrams and its low-level implementations. We used Cadence SMV model checker [56] in our study. All the experiments were conducted using 2.0 GHz Intel Core2 Duo CPU with 1 GB RAM.

3.3.1 A Control System

The first case study is a small control system. This case study is based on the example presented in Section 2.3.4.

Table 3-1. Comparison of two methods

	Coverage (%)			Time
Method	activity	transition	path	(second)
random 30	90	85	50	1.33
random 50	95	93	67	2.35
random 100	100	100	83	5.13
random 150	100	100	100	8.83
Our Approach (UMC)	100	100	100	0.91

Table 3-1 shows the comparison between our approach and the random test based method [51]. For generating tests with highest coverage, the random method requires 8.83 seconds to run 150 random tests, however our approach using unbounded model checking method (UMC) just needs 0.91 seconds. In this case study, UMC approach improves the test generation time by an order of magnitude.

Table 3-2. Implementation level coverage of the control system

Package	Class	Method	Block	Line
100%	100%	90%	88%	93%

We applied the generated tests to the Java implementation of the control system. Table 3-2 shows the coverage of the Java code. The generated tests obtained 100% *package* as well as *class* coverage. However, the *method*, *block* and *line* coverage are around 90%. Our analysis showed that the Java implementation have many "try" and "catch" blocks to handle exceptions whereas the specification does not have any information on the exception scenarios. As a result, the generated tests did not activate any of the exception blocks which resulted in low coverage of methods, blocks as well as lines. Clearly, this is an issue of incomplete specification. Based on this observation, we added exception information at the specification level and generated tests which led to the required coverage in all the categories of the implementation.

3.3.2 A Stock Exchange System (OSES)

The stock exchange system is based on the example presented in Section 2.3.5. It uses the UML activity diagram as its behavior specification. The system is implemented in JAVA and consists of 7 packages, 39 classes, 372 methods and 2510 lines.

Table 3-3. Comparison of three methods

	Coverage (%)			Time
Method	activity	transition	path	(minute)
random 800	96	83	89	19.06
random 1000	96	86	94	24.26
random 1500	100	100	100	30.25
Our Approach (UMC)	100	100	100	3.47
Our Approach (BMC)	100	100	100	0.15

In Table 3-3, the first three rows depict the results by using 800, 1000, 1500 random tests respectively. The result by our method is shown in the last two rows. In the case of *random* 800, two key paths are missing due to the randomness. So the coverage metrics are not 100%. If we increase the number of the random tests to 1000, one key path is still missing. Based on our observation, in the random method, it is hard to determine what is an appropriate upper bound for the number of required random tests. As a result, it is hard to obtain 100% specification coverage using the random tests. The result of the UMC shows that we can get an order of magnitude improvement compared to the random

method. Because the bounds of the properties of OSES system are shallow and can be pre-determined, we applied SAT-based BMC in this situation. The result shows that BMC method can be an order of magnitude faster than UMC method. Clearly, BMC approach reduces the validation effort by two hundred times compared to the best known result [51] in this category.

Table 3-4. Implementation level coverage of OSES

Package	Class	Method	Block	Line
100%	100%	58%	55%	51%

Table 3-4 presents the coverage of the implementation by applying the generated tests. The coverage of method, block and line are not sufficient because the activity diagram does not consider all the scenarios of the system, such as the registration of the customers and so on. In this case, we needed to add the missing details in the specification to obtain the required coverage.

3.4 Summary

In this chapter, we presented a framework to automatically generate directed tests from SoC specifications. Our experimental results demonstrated that the generated tests can produce the required functional coverage and also can make a significant reduction in validation effort for specifications as well as implementations. Model checking based test generation is promising for automated test generation but it can lead to state space explosion in the presence of complex designs and properties. So in the following chapters, we will present various optimization techniques to reduce the overall test generation complexity.

CHAPTER 4
PROPERTY CLUSTERING FOR EFFICIENT TEST GENERATION

Although model checking techniques are promising for automated directed test generation, it is costly for complicated designs due to the state space explosion problem. Especially for a complex design, there will be a large number of properties to be validated. When validating a specific system component, it is common that several properties have a large overlap on sub-functionalities. Validating the properties individually will be a waste of time due to the repeated validation efforts on the same functional scenarios. Potentially these redundancy can be avoided and consequently the overall test generation time can be significantly reduced.

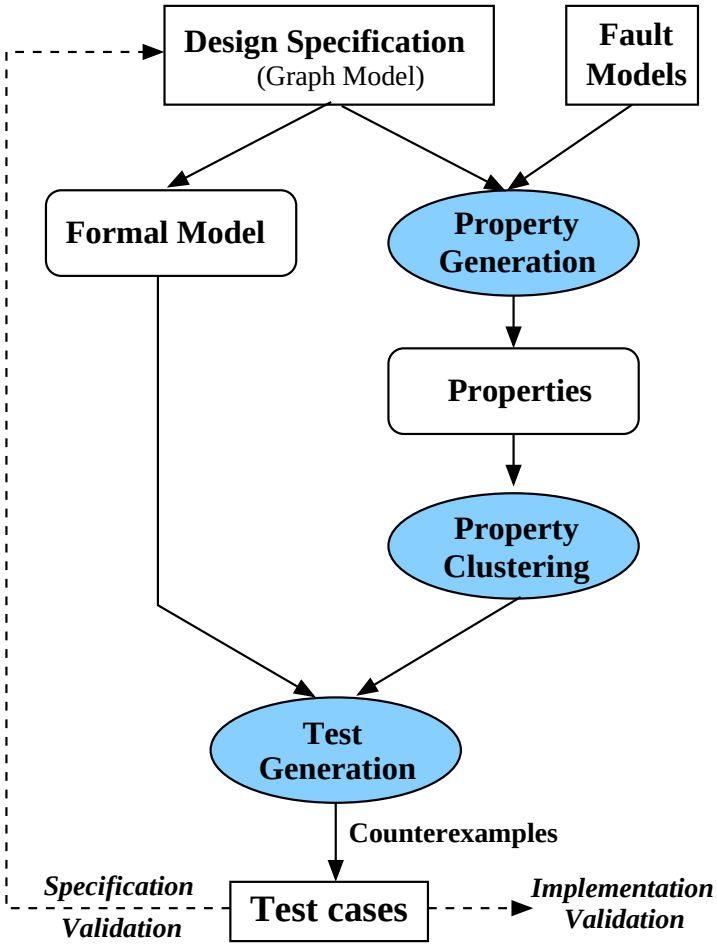


Figure 4-1. Our test generation methodology

The target of property clustering is to reduce the overall test generation time by exploiting the similarities among properties. Figure 4-1 shows the test generation framework using our property clustering approach. The proposed methodology has three important steps: coverage-driven property generation, clustering of similar properties, and test generation using learning techniques. It is important to note that each of these three steps can be independent. For example, our method uses the coverage of our fault models to derive properties. The other two steps will produce beneficial results even if other fault models are used to generate properties. Designers can even add various properties manually to the set of generated properties without affecting the usefulness of our approach.

This chapter makes two primary contributions: i) it proposes novel methods to cluster similar properties; and ii) it utilizes the conflict clause based learning to reduce the overall test generation time for a cluster of similar properties. The rest of this chapter is organized as follows. Section 4.1 presents related work on efficient model checking techniques. Section 4.2 introduces the implementation details of state-of-the-art SAT solvers. Section 4.3 proposes our property clustering approaches. Section 4.4 presents how to efficiently generate tests using property clustering and conflict clause forwarding technques. Section 4.5 demonstrates case studies on both hardware and software designs. Finally, Section 4.6 summarizes the chapter.

4.1 Related Work

Due to the scalability issues of conventional Binary Decision Diagram (BDD) based methods, SAT-based BMC is proposed as a complementary solution for large designs. Many studies in both software and hardware domains [4] show that BMC has better capacity and productivity over unbounded model checking for real designs. Currently, various techniques based on conflict clause forwarding and variable ordering [63] are proposed to further improve the efficiency of BMC based test generation.

As a promising learning based approach, incremental SAT [40, 67, 89, 92] tries to leverage the similarity between the elements of a sequence of SAT instances – most do so by re-utilizing learned knowledge based on conflict clauses. When many closely related instances need to be solved, caching solutions [43] and incremental translation [7] can also be effective. If a SAT instance is obtained from another by augmenting some clauses (as in [38]), all conflict clauses of the first can be forwarded to the second. Therefore, when clauses are only added through a sequence of instances, there is no need to screen conflict clauses to determine which ones can be forwarded. This, on the other hand, is necessary when arbitrary clauses are both added or deleted to create a new instance. A common approach for such a general case is to have incremental SAT solvers keep track of whether a conflict clause depends on some removed clauses. Majority of the existing approaches exploit incremental satisfiability to improve the test generation time involving only one property with different bounds. There are very few approaches such as [17] where both static and dynamic learning are used across test generation instances for path-delay fault model by dynamically excluding the untestable path during test generation. Since the learning is employed across all test scenarios without efficient clustering methods, the improvement in test generation time is small (6% on average) and has a wide variation (-7% to 27%) on different ISCAS circuits.

To the best of our knowledge, our approach is the first attempt to cluster similar test generation instances involving multiple properties and utilize shared knowledge across similar instances in the context of directed test generation.

4.2 Background: SAT Solver Implementation

This section introduces the preliminary knowledge of SAT solver implementation. In the context of directed test generation, we describe how SAT-based BMC can be used to improve test generation time by employing learning techniques.

4.2.1 DPLL Algorithm

Most modern SAT solvers such as GRASP [39] and Chaff [63] adopts the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm [52, 53].

Algorithm 3: DPLL search procedure of zChaff

```
while TRUE do
    run_periodic_functions();
    if decide_next_branch() then
        while deduce() == CONFLICT do
            blevel = analyze_conflicts();
            if blevel < 0 then
                return UNSAT;
            end
        end
    else
        return SAT;
    end
end
```

Algorithm 3 shows the DPLL implementation in zChaff. It contains three parts:

- Periodic function updates the SAT configuration triggered by some specified events, such as updating the scores of literals after a certain number of backtracks.
- Boolean Constraint Propagation (BCP) is implemented in *deduce*. It figures out all possible implications by previous decision assignment.
- Conflict analysis does a proper backtrack when encountering a conflict. It analyzes the reason for the conflict and make it as a conflict clause to avoid the same conflict in future processing.

Studies in [63] show that modern SAT solvers spend approximately 80% of time to carry out BCP. In addition, during the conflict analysis, long distance backtracks will increase the burden of SAT solvers.

4.2.2 Conflict Clause Based Learning

As shown in Algorithm 3, SAT solvers use the conflict analysis technique to trace the reason for a conflict. The conflict analysis contains two part: conflict-driven back-tracking

and conflict-driven learning. Conflict-driven backtracking enables the non-chronological backtracking up to the closest decision which caused the conflict. Conflict-driven learning learns some knowledge and save them in conflict clauses and adds them to the original clauses, in order to avoid the same conflict in the future. Both techniques can drastically boost the performance of the SAT solvers.

The kernel of the conflict analysis technique is the implication graph [39, 91]. The graph keeps the current state and the implication history of the search during the SAT solving by recording the dependence of the variable assignments. The implication graph is a directed acyclic graph where each vertex represents an assignment to a variable and each edge implies that all the in-edges implicate the assignment of the vertex.

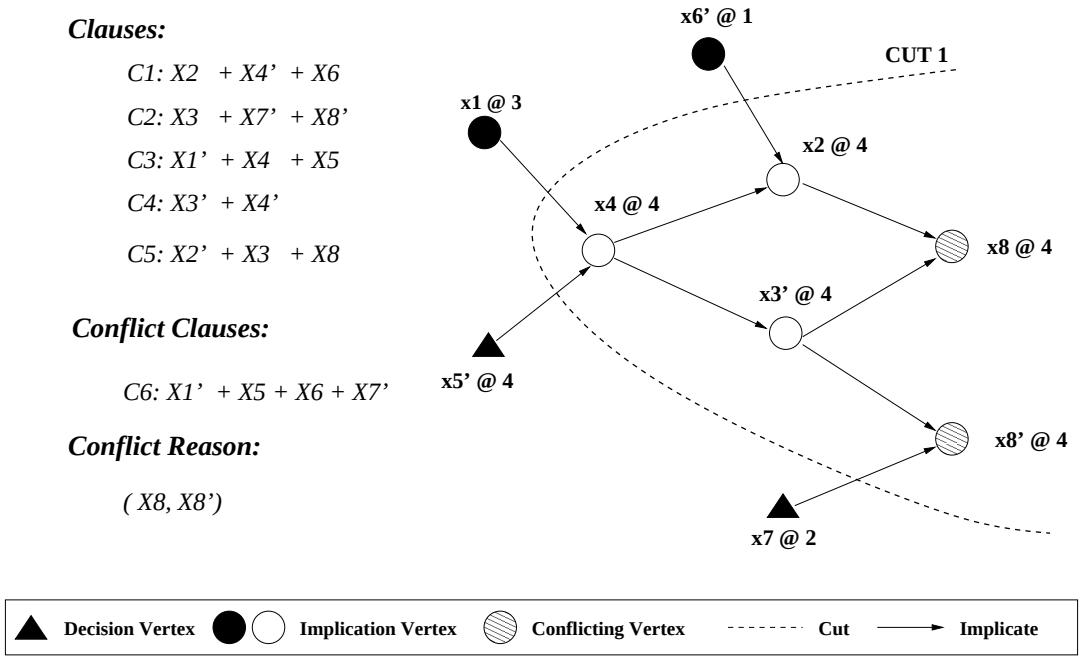


Figure 4-2. Conflict analysis using an implication graph

Figure 4-2 shows a small example of conflict analysis using an implication graph. As shown at the left of the figure, there are five original clauses *C1-C5*. The right part is a scenario of implication graph for *C1-C5*. In this example, $x4@4$ means variable $x4$ is assigned value 1 at decision level 4. The node has a corresponding clause *(x1'+x4+x5)*, we call it the antecedent clause of $x4$, i.e., the assignments $x1 = 1$ and $x5 = 0$ imply

$x4 = 1$. Only the implication vertex (non-decision vertex) has an antecedent clause. A conflict happens when there are two nodes in the implication graph that have different value assignments for the same variable. For example, the implications in the graph lead to the ambiguous assignment to variable $X8$ ($X8 = 0$ and $X8 = 1$). When encountering a conflict, conflict analysis will trace back along the implication relations to find the reason for the conflict and encode the reason using a conflict clause. A conflict clause can be found by a bipartition of the implication graph. The side containing the conflicting vertex called conflict side, and the other side is called reason side which can be used to form the conflict clause. In Figure 4-2, $CUT1$ is a cut that divides the implication graph into two parts. The conflict analysis stops at $CUT1$. The left part of $CUT1$ in the implication graph is the reason side, and the right part is the conflict side. From the reason side, we can get the *conflict cluase C6=(X1 + X5' +X6' +X7)*. That means, the assignment of variables $X1 = 1$, $X5 = 0$, $X6 = 0$ and $X7 = 1$ will always lead to a conflict because of the clauses *C1-C5*. Lemma 1 indicates that the generated conflict clauses during the SAT search can be added to original clause set as an assignment constraint. Therefore we can add the clause $C6$ to the original clause set to avoid the same conflict in the future.

Lemma 1. *Given a set of CNF clauses* $S1$ *and* ψ *is a conflict clause derived during the conflict analysis, then* $S1$ *is satisfiable iff* $S1 \bigwedge \psi$ *is satisfiable.*

Proof. Because $S1 \bigwedge \psi$ is a super set of $S1$, so if $S1 \bigwedge \psi$ is satisfiable then $S1$ is satisfiable. According to the definition of the conflict clause, the assignments that make the clause ψ false will make the clause set $S1$ false. If $S1$ is satisfiable, then there exists a variable assignment that makes $S1$ true. This assignment should make ψ true. So the assignments will make $S1 \bigwedge \psi$ true. □

For two SAT instances, if one instance is a subset of the other SAT instance, according to Theorem 1, the conflict clauses generated from the smaller SAT instance can be forwarded to the larger SAT instance. In other words, the local learning can be

forwarded as a knowledge for global searching. Usually the average cost of locally learned conflict clauses is much cheaper than the globally learned conflict clauses.

Theorem 1. *Given two CNF clause sets $S1$ and $S2$, where $S1 \subseteq S2$, and ψ is a conflict clause derived from the clauses in $S1$, written $S1 \vdash \psi$, then $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable.*

Proof. Since $S2 \bigwedge \psi$ is a super set of $S2$, if $S2 \bigwedge \psi$ is satisfiable then $S2$ is satisfiable. Because $S1 \vdash \psi$ and $S1 \subseteq S2$, then ψ is also a conflict clause of $S2$. According to Lemma 1, $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable. □

According to the Equation (3–1), similar properties share a large part of the CNF clauses. Regardless of the cone of influence, the equation shares the system part (transition relation $T(s_i, s_{i+1})$) and the part of property testing (i.e., $p(s_i)$). Sharing a large part of CNF clauses indicates that when checking the first property, the learned knowledge (conflict clauses) can be forwarded to the second property without affecting the truth assignment of the CNF clauses of the second property.

Theorem 2. *Assume that we have two sets of CNF clauses $S1$ and $S2$, and let $\omega = S1 \bigcap S2$ be the common clauses shared by both $S1$ and $S2$. ψ is a conflict clause derived only by the clauses in ω, written $\omega \vdash \psi$. Then $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable.*

Proof. Because $S2 \bigwedge \psi$ is a super set of $S2$, so $S2 \bigwedge \psi$ is satisfiable then $S2$ is satisfiable. Because $\omega \vdash \psi$ and $\omega \subseteq S2$, then $S2 \vdash \psi$. According to Lemma 1, $S2$ is satisfiable iff $S2 \bigwedge \psi$ is satisfiable. □

4.3 Property Clustering

Given a set of properties, a clustering method determines how to divide the properties into several groups such that each group contains similar properties that can benefit from each other during test generation. The similarity can be structural or behavioral but the assumption is that there is a significant overlap between the counterexample generation traces involving a set of similar properties.

Algorithm 4: Property Clustering

Input: i) A set of properties, P

ii) Similarity strategy CS, and threshold W_{th}

Output: Clusters consisting of similar properties

$PropertyClusters = \phi$;

1. Construct a graph, G where each node is a property;

for *each pair of nodes* (n_i, n_j) *in* G **do**

Weight w_i^j = ComputeSimilarity $CS(n_i, n_j)$;

if($w_i \geq W_{th}$) Create an edge between n_i and n_j with weight w_i^j;

end

2. $k = 1$; /* first cluster */ **while** G *is not empty* **do**

$Base_k$ = Node with highest overall edge weight;

$Cluster_k$ = all the nodes connected to $Base_k$;

$G = G$ - $Cluster_k$;

$PropertyClusters = PropertyClusters \cup Cluster_k$;

$k = k + 1$;

end

return PropertyClusters;

Algorithm 4 outlines the major steps in property clustering. The first step constructs a property graph[1] where the properties are nodes and edges represent similarity. An edge is added between two properties (nodes) when they are similar. Each edge e_j includes weight information (w_j, $0 \leq w \leq 1$) to quantify the similarity. An edge with weight 0 or 1 is not possible since an weight of 0 means no similarity, and an weight of 1 implies same (identical) property. To compute the weight information for each edge we propose four methods – structural, textual, influence and CNF intersection based similarity. Each method will use a similarity threshold for clustering. In other words, there will be no edge

[1] In this chapter, we use three different types of graphs for three different purposes. The graph model of the design (or *design graph* in short) is used to model the design. The *implication graph* is used to store the dependence of variable assignments that is used for conflict analysis. The *property graph* models the similarity between properties and used for clustering.

between two properties when the weight value is below certain threshold. The second step determines the clusters based on the base property. The base property is the property (node) with highest weight (summation of weights of all edges connected to that node). The cluster is formed by adding all the adjacent nodes with the base property. All the nodes selected for a cluster are deleted from the property graph for the next iteration. The remainder of this section describes four different ways of computing similarity between properties.

4.3.1 Similarity based on Structural Overlap

A simple and natural way to cluster properties is to exploit the structural information of the design model and its properties. The intuition is that two similar properties will share similar variable assignments (global and local variables[2]) in the counterexamples. In fact, a conflict clause is a constraint on the assignment of the variables. Therefore, properties with similar structural information will share a lot of conflict clauses.

As mentioned earlier, in the context of directed test generation, properties are generated based on functional coverage of the design. These properties try to cover different parts of the design (e.g., all computation nodes, various interactions, etc.). Therefore, we can cluster the properties that try to cover a specific functionality or interactions. For example, in an SoC environment, the properties can be clustered based on whether they are related to verifying the processor, coprocessor, FPGA, memory, bus synchronization, or controllers. Each cluster can be further refined based on structural details of each component. For example, the processor related properties can be further divided based on which execution path they activate such as ALU pipeline, load-store pipeline etc.

[2] In a graph model, a local variable is defined locally in a node whereas the scope of a global variable is valid across nodes.

In the pipelined processor example in Figure 2-12, there are four execution pipelines: $IALU$, MUL, $FADD$ and DIV. The corresponding paths are as follows.

- $\rho_1 = FET \rightarrow DEC \rightarrow IALU \rightarrow MEM \rightarrow WB$
- $\rho_2 = FET \rightarrow DEC \rightarrow MUL1 \cdots \rightarrow MUL7 \rightarrow MEM \rightarrow WB$
- $\rho_3 = FET \rightarrow DEC \rightarrow FADD1 \cdots \rightarrow FADD4 \rightarrow MEM \rightarrow WB$
- $\rho_4 = FET \rightarrow DEC \rightarrow DIV \rightarrow MEM \rightarrow WB$

Consider two properties $p1 =\sim F(fadd3\_active = 1)$ and property $p2 =\sim F(fadd4\_active = 1)$. They share the same path ρ_3, and the bound of $p1$ is just one smaller than $p2$. So we can cluster them together. Also for the interaction property $p3 =\sim F(fadd4\_active = 1 \ \& \ mul3\_active = 1)$ and $p4 =\sim F(fadd3\_active = 1 \ \& \ mul4\_active = 1)$, the two interactions are related to the same set of paths ρ_2 and ρ_3 and have similar bounds. Therefore, clustering them together is a good choice.

4.3.2 Similarity based on Textual Overlap

Another simple way to quantify similarity is to measure the textual differences between two properties. For example, the similarity between $\sim F(a \ \& \ b \ \& \ c)$ and $\sim F(b \ \& \ c \ \& \ d)$ is 67% since they share a common sub-expression consisting of two variables b and c.

In this section, we focus on bounded model checking of invariants (safety properties) such as the property in the form $\sim$ F(p). Informally, $BMC(M, p, k)$ is true means from cycle 0 to cycle k, the property will be false. So the invariant cannot always be true and one counter example will be reported. Because the part $I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ comes from the design, so for different properties this part is same. The part $\bigvee_{i=0}^{k} \neg p(s_i)$ usually determines the difference among the properties.

The negative format of each literal in the conflict clause is a false assignment for the logic formula $BMC(M, p, k)$. In fact, the conflict clause can be regarded as a constraint for the variable assignment. Let P and Q be two properties of the model, the properties P, $P \wedge Q$ and $P \vee Q$ can be expanded as follows:

- $BMC_1(M, P, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(s_i)$
- $BMC_2(M, P \wedge Q, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg(P \wedge Q)(s_i)$
  $= I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} (\neg P(s_i) \vee \neg Q(s_i))$
- $BMC_3(M, P \vee Q, k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} (\neg P(s_i) \wedge \neg Q(s_i))$

In the expanded CNFs above, we assume that the same variable in respective expansion has the same meaning. Let A be a partial assignment of the CNF variables that can make the whole CNF false, then $A \nvDash BMC_1$ implies $A \nvDash BMC_3$, $A \nvDash BMC_2$ implies $A \nvDash BMC_1$, and $A \nvDash BMC_2$ implies $A \nvDash BMC_3$. In other words, the conflict clauses of BMC_1 can be forwarded to BMC_3, and conflict clause of BMC_2 can be forwarded to both BMC_1 and BMC_3.

In most existing BMC tools, the variables in the generated CNF file do not have specific meaning. The conflict clauses of the stronger property cannot be directly forwarded to some weaker properties. For example, some conflict clauses of property $P \wedge Q$ cannot be forwarded to check property $P \vee Q$. However, when properties have the relation of implication, and their textual similarity is high, clustering them together will have a positive effect. If two properties are in the same format and have a significant (more than 50%) textual overlap, the two properties can benefit from each other.

Textual clustering is very fast but it may not be very accurate. For example, the properties $\sim F(a)$ and $\sim F(c)$ have no overlap, however, it is possible that both variables are very closely related in the design model (such as activates the same path), and therefore they are good candidate for clustering. Unfortunately, in the absence of such structural information, pure textual clustering may not generate significant savings in test generation time. Textual clustering is beneficial when information regarding the design or original fault model are not available and/or when there are too many properties.

4.3.3 Similarity based on Influence

An assignment to a global variable determines the state transition of various components in the design (graph) model. For example, in the MIPS model, when the

instruction buffer contains only division instruction, only the components in DIV path will be activated. However, it is time consuming to analyze all the global and local variables of the model since it need to consider the state transition of each component.

Based on the graph model structure, we can determine various cause-effect relations. For example, the state change of $MUL6$ will be one clock cycle later than $MUL5$. That means the execution of $MUL5$ has an influence on the execution of $MUL6$. The influence nodes indirectly reflect the assignment of the global variables, since the assignment of global variables is relevant to the variable assignment in the counterexample.

Prior to clustering, we need to figure out the influence node set for each node in the graph model. We can compute the influence node set for each node using Depth First Search (DFS) algorithm. If there is a path starting from the start node to the current node, then all the nodes on this path are influence nodes for the current node. DFS can explore all the paths (except the paths with loops) from the start node to the current node. For example the influence node sets for $MUL2$, $FADD3$ and WB are as follows:

- $Influence(MUL2) = \{FET,\ DEC,\ MUL1,\ MUL2\}$
- $Influence(FADD3) = \{FET,\ DEC,\ FADD1,\ FADD2,\ FADD3\}$
- $Influence(WB) = \{n|$ n is a node in the MIPS graph model.$\}$

A property corresponds to several nodes (modules) in the graph model. So the influence node set of a property is the union of the influence of all relevant nodes. When comparing the similarity of two properties, we need to compute the intersection of influence sets. For example, the influence set of property $\sim F(mul2\_active = 1\ \&\ fadd3\_active = 1)$ is S_1= $\{FET, DEC, MUL1, MUL2, FADD1, FADD2, FADD3\}$ and the influence set for $\sim F(mul3\_active = 1\ \&\ fadd3\_active = 1)$ is S_2= $\{FET, DEC, MUL1, MUL2, MUL3, FADD1, FADD2, FADD3\}$. The two sets share a large intersection. For set S_1, the similarity with S_2 is $7/7 = 100\%$. For set S_2 the similarity with S_1 is $7/8 = 87.5\%$. Based on our experience, when the overlap of influence sets are

larger than 70%, forwarding conflict clauses is beneficial. In this example, S_1 and S_2 can be clustered together.

4.3.4 Similarity based on CNF Intersection

One obvious, but costly, way to determine property similarity for clustering is to compute intersections of CNF clauses between properties. We can cluster properties that have a relatively large number of clauses in the intersection. Based on our experience, a threshold of 0.9 is beneficial. In other words, when two properties share at least 90% common clauses, it is beneficial to forward conflict clauses between two instances.

This method is very time consuming because it requires $O(n^2)$ intersections for n properties. When n is large, this method is not feasible, because the calculation of intersection of irrelevant properties may waste more time than actual SAT solution time. Moreover, in certain scenarios, forwarding conflict clauses may not improve the overall test generation time for a cluster, since it may change variable ordering and searching heuristics. CNF based clustering is a good choice when the number of properties is small or when other methods fail to find beneficial clusters.

4.3.5 Determination of Base Property

Determination of base property in a cluster is crucial for test generation using learning techniques. The base property is solved first and its conflict clauses are shared by the remaining properties in the cluster. Although, any property in the cluster can be used as the base property for that cluster, our studies have shown that certain properties serve better as base property and thereby generate better overall savings for the cluster. We need to consider two important factors while choosing a base property for a cluster. First, the base property should be able to generate a large number of conflict clauses. In other words, a weak base property may find the satisfiable assignment quickly without making mistakes (generating conflict clauses). In this scenario, the remaining properties have nothing to learn from the base property. Moreover, the SAT checking time for the base property should be relatively small. This will ensure that the overall gain is maximized

by reducing the solution time of the properties which takes longer time to solve. None of these requirements can be determined without actually solving them. Based on our experience, we have observed that the following heuristics works well most of the time.

- Choose a property that has significant variable and/or sub-expression overlap with other properties in the cluster.
- If bound for each property is known, choose the property whose bound is closest to the remaining properties.
- Compute intersection of every pair of properties in the cluster, and choose the one that shares the most with the remaining properties.

4.4 Efficient Test Generation using Learning Techniques

Incremental SAT-based BMC [54] is very promising to reduce the test generation complexity. However, existing approaches are restricted for a test generation scenario consisting of one design and only one property (with varying bounds). Many properties generated from the design specification share a lot of similar information. If the shared information can be exploited and re-utilized across the similar properties, many repeated verification efforts can be avoided. In other words, the knowledge learned during the execution of one property can benefit other similar properties. Therefore, knowledge sharing can reduce complexity and improve the overall verification effort. Although each test generation instance requires a different property, several properties related to testing specific functionalities are similar or have a significant overlap. Reuse of learned knowledge (e.g., constraints) derived from such overlap can avoid the repeated state space search. In this section, we discuss two kinds of learning techniques which can drastically reduce the overall test generation time in a cluster of similar properties.

4.4.1 Conflict Clause Forwarding Techniques

The basic idea is to learn from solving one property and share learning (through conflict clauses) for solving the similar properties in the cluster. While solving the first property (base property), the SAT solver may have taken many wrong decisions (lead to conflicts) and therefore needs long time to find a counterexample. Forwarding

conflict clauses ensures that these wrong decisions are avoided while solving the similar properties. An important question is whether all the wrong decisions of the first property are relevant to all the other properties in the clusters? Since the properties are similar but not the same, some of the decisions are not relevant. In our approach, we determine the common CNF clauses by computing the intersection of clauses and use this intersection information to exactly identify the conflict clauses that are relevant to solving the respective properties.

Algorithm 5: Test Generation using Learning Techniques

Input: i) Design model D and ii) Clusters of similar properties

Output: Tests

for *each cluster, i, of properties* **do**

  Generate CNF for the base property P_1^i, CNF_1^i;

  for *j is from 2 to the* $size_i$ *of cluster i* **do**

    /* P_j^i is the j^{th} property in the i^{th} cluster */;

    1. Generate CNF, $CNF_j^i = BMC(D, P_j^i, bound_j^i)$;

    2. Perform name substitution on CNF_j^i;

    3. INT_j^i = ComputeIntersection(CNF_1^i, CNF_j^i);

    4. Mark the clauses of CNF_1^i using INT_j^i;

  end

  /* Generate a counterexample and record conflict clauses */;

  5. ($ConflictClauses_i$, $test_1^i$) = SAT(CNF_1^i);

  $Tests = \{test_1^i\}$;

  for *j is from 2 to the* $size_i$ *of cluster i* **do**

    /* Find relevant ones for P_j^i from conflict clauses */;

    6. CC_j^i = Filter ($ConflictClauses_i$, j);

  end

  for *j is from 2 to the* $size_i$ *of cluster i* **do**

    7. $test_j^i$ = SAT($CNF_j^i \bigcup CC_j^i$);

    $Tests = Tests \cup test_j^i$;

  end

end

return *Tests*;

Algorithm 5 describes our test generation methodology. It accepts a list of clusters where each cluster consists of a set of similar properties. Since one property is used to generate a test, the number of input properties is exactly the same as the number of output tests. The first step generates the CNF clauses for all the properties in each cluster using the design and respective bounds. The second step performs name substitution to maximize knowledge sharing. The third step computes the intersection of CNF clauses between the base property and all the remaining properties in the cluster. The first three steps can be omitted, if CNF intersection based clustering is employed. The fourth step marks the clauses in the base property to indicate whether a particular clause is also in the clause set of another property in the cluster. The next step uses a SAT solver to generate the conflict clauses and the counterexample for the base property. Based on the intersection information with the base property, the set of conflict clauses is filtered to identify the relevant ones for solving the remaining properties in step 6. The final step uses the relevant conflict clauses to solve the remaining properties using our approach. The algorithm reports all the generated counterexamples.

We use a simple example to illustrate how Algorithm 5 works. Let us assume that we are generating tests using n properties for a design. The input is a list of m ($m \leq n$) clusters based on property similarities. Each cluster can have different number of properties. In the worst case, each cluster can have only one property which will be verified normally. However, this scenario is rare in practice since a typical design uses thousands of properties for directed test generation and majority of them share significant parts of the design functionality. For ease of illustration, let us assume that there is a cluster with three similar properties, $\{P_1, P_2, P_3\}$. Let us further assume that the second step selects P_1 as the base property. The fourth step computes intersection of CNF clauses of P_1 with P_2, and P_1 with P_3. This information is used to filter conflict clauses (generated while solving P_1) relevant for P_2 and P_3 in step 6. The last step adds the relevant conflict clauses while solving the respective properties to reduce the test generation time.

The following subsections describe two important techniques in our approach, name substitution for computation of intersections, and identification of relevant conflict clauses.

4.4.2 Name Substitution for Computation of Intersections

Name substitution is an important preprocessing step in Algorithm 4. Currently, few BMC tools support the name mapping from the variables of the CNF clauses and the names in the model of the unrolled design. As a result, the variables of the CNF clauses of two different properties may not have any name correspondence. In other words, the same variable in two properties may have different name in their respective CNF clauses. Therefore, without name substitution (mapping), it will miss the overlap information. As a result, the computed intersection will be small and will adversely affect the sharing of learned conflict clauses. We observed that the improvement in test generation time without using name substitution is negligibly small due to very small number of clauses being forwarded as a result of small number of clauses in the intersection. Since the properties are similar and the design is exactly the same, the size of the intersection is very large when our name substitution method is employed.

Our framework uses zChaff SAT solver [74] which accepts the input in the DIMACS format. The generated DIMACS file for each property provides the name mapping from the CNF variable to the unrolled design. For example, "c 8 = V1_var[6]" shows that the variable 8 is used in the CNF file to refer to the 7^{th} bit of variable var in the design specification at time step 1. This can also be written as, $8 => var[6]\_1$.

Given two DIMACS files $f1$ and $f2$ for two properties P_1 and P_2 respectively, the name substitution is a procedure that changes the names of clause variables of $f2$ using the name mapping defined in $f1$. Figure 4-3 shows an example for name substitution. Before the name substitution, the intersection $(f1 \cap f2)$ is empty. However, after name substitution, there are two common clauses in the intersection $(f1 \cap f2')$. The complexity of both name substitution and computation of intersection is linear (using hash table) to the size of the DIMACS file of the properties. Therefore, the time required for name

substitution and intersection computation is negligible compared to the SAT solving time for complex properties.

It is important to note that the same variable at different time steps can be assigned a different number. Therefore, the name mapping (substitution) method needs to consider the same variable at different time steps in the CNF clauses of the same property as well as in the CNF clauses for the different properties in the same cluster. Moreover, the name mapping routine needs to remap some of the variables in the CNF clauses. For example in Figure 4-3, when the variable 4 in file $f2$ is replaced with the variable 1 (in $f2'$), the name mapping routine needs to remap the original variable 1 in file $f2'$ to a different variable.

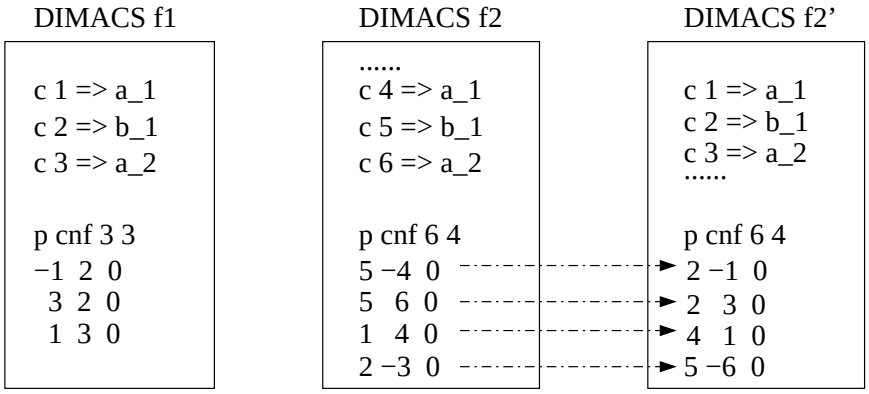


Figure 4-3. An example of name substitution

4.4.3 Identification and Reuse of Common Conflict Clauses

Our implementation of relevant conflict clause determination is motivated by the work of [67] which proved that for two sets of CNF clauses C_1 and C_2, and their intersection φ, use of conflict clauses generated from φ when checking C_1 will not affect the satisfiability of the CNF clauses $C_2 \bigcup \varphi$. Therefore, the conflict clauses generated from the intersection when checking the base property can be shared by other properties in the cluster.

Strichman [67] suggested an isolation procedure that can isolate the conflict clauses which are deduced solely from the intersection of two CNF clause sets. We have modified the isolation procedure to improve the efficiency of test generation for a cluster of properties. We have modified zChaff [74] SAT solver and used it in our framework. The zChaff provides utilities for implementing incremental satisfiability. For each clause,

it uses 32 bits to store a group id to identify the group where this clause belongs. Use of group id allows us to generate the conflict clauses for different properties when checking the base property. If the i^{th} bit of the clause's group id is 1, it implies that the clause is shared by the CNF clauses of property P_i. If the clause of the base property is not shared by any property, the field will be 0.

Assume that there are $k+1$ properties in a cluster with C_i as the set of CNF clauses for the property P_i. Moreover, assume that P_0 is the base property. In other words, there are $k+1$ sets of clauses with C_0 as the base set, and $C_1, C_2, ..., C_k$ are k similar sets with C_0. We use the following steps to calculate the conflict clauses for $C_1, C_2, ..., C_k$ when solving C_0.

- During preprocessing, for each clause cl in C_1, if this clause also exists in C_i ($2 \leq i \leq k$), then mark the i^{th} bit of cl's group id as 1.
- When one conflict clause is encountered during the checking of the base property, collect all the group ids of the clauses in the conflict side. The group id of the conflict clause is logical "AND" of all these group ids.
- For each conflict clause, if the i^{th} bit of the group id is 1, then this conflict clause can be shared by C_i.

In our approach, each conflict side clause has a group id which is marked during the preprocessing step or marked during the conflict analysis if it is a conflict clause. The procedure of group id determination of a conflict clause is described in Algorithm 6.

This algorithm traces back from the conflicting assignment to a cut such as $first$ *Unique Implication Point* (UIP) [91] in zChaff. The conflict side will contain all the implications of the variable assignments of the reason side. For UIP, they are implication variable assignments in the same decision level as the conflicting variable assignment which led to the conflict. The group id of the conflict clause is the logical "AND" value of all the group ids of the conflict side clauses. This algorithm can guarantee that if the i^{th} bit of the group id of the conflict clause is 1, then this conflict clause can be forwarded to the i^{th} CNF clause set.

Algorithm 6: Determination of conflict clause and its group ID

Input: i) Conflicting node N

Output: Conflict clause with its group id

$Visited = \{ N\}$;

$ConflictAssign = \{\}$;

$groupId$ = group id of $N's$ antecedent clause;

while *the set* $Visited$ *is not empty* **do**

1. v = RemoveOneElement($Visited$);

2. *clause* = AntecedentOf(v);

$groupID = groupID$ *"AND"* group id of *clause*;

if v *is on the conflict side* **then**

3. Put all the nodes of *clause* in implication graph except v to the set $Visited$;

else

4. $ConflictAssign = ConflictAssign \cup \{v\}$;

end

end

5. $ConflictClause$ = Logical disjunction of negated assignments of all elements in $ConflictAssign$;

return $ConflictClause$ and $groupId$;

Figure 4-4 illustrates how this computation is done. The implication graph belongs to a base property of a cluster. Each clause in this graph is marked with the group id information. Here we use four bits to express the group id. For example, the group id of clause $(x3' + x4')$ is "1010". It means that this clause exists both in CNF clause set 2 and CNF clause set 4. The group id of the conflict clause is the logical "AND" of all conflict side clauses, and the result is 0010. That means, this conflict clause can be forwarded to clause set C_2. Therefore, the use of this conflict clause in solving P_2 will reduce the SAT solving (test generation) time.

4.5 Case Studies

We have applied our test generation methodology for validation of various software and hardware designs. In this section, we present two case studies: the VLIW implementation of the MIPS architecture, and a stock exchange system. Both experiments were performed

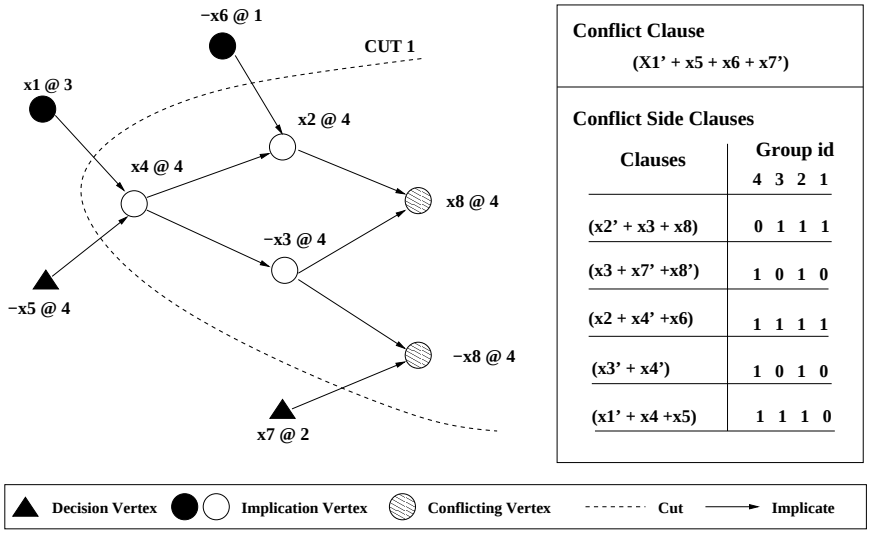


Figure 4-4. An example of conflict clause reuse

on a Linux PC using 2.0GHz Core 2 Duo CPU with 1 GB RAM. In our experiments, we used the NuSMV [27] as our BMC tool to generate the CNF clauses (in the DIMACS format) for the design and properties. We developed the tool *PropertyCluster* which accepts the graph model, the coverage criteria and the clustering strategies as inputs. This tool generates the required properties (using different coverage criteria presented in Section 3.1) and clusters them using the clustering strategies proposed in Section 4.3. We also modified zChaff [74] to integrate our techniques including name substitution, clause intersection, and constraint sharing based test generation described in Section 4.4.1.

4.5.1 A VLIW MIPS Processor

We applied our methodology on the single-issue MIPS presented in Section 2.3.2. The *PropertyCluster* generated 171 properties using the node coverage, 2-interaction coverage, and the path coverage criteria. In this section we first present results for each clustering technique, and then present a summary to compare the clustering techniques.

4.5.1.1 Structure-based Clustering

The graph model of MIPS processor has four parallel pipeline paths. Each of them shares four units (fetch, decode, memory and writeback), and differs only in the execution units. The structural similarity is established based on the path that a set of properties

activates. For example, the following 7 properties is grouped in a cluster because all of them refer to the division path.

- $p13 =\sim F(fet\_active! = 1 \& div\_active! = 1)$
- $p28 =\sim F(dec\_active! = 1 \& div\_active! = 1)$
- $p133 =\sim F(div\_active! = 1 \& mem\_active! = 1)$
- $p134 =\sim F(div\_active! = 1 \& wb\_active! = 1)$
- $p150 =\sim F(div\_active! = 1))$
- $p165 =\sim F(fet\_active! = 1 \& dec\_active! = 1 \& div\_active! = 1)$
- $p170 =\sim F(fet\_active! = 1 \& dec\_active! = 1 \& div\_active! = 1 \& mem\_active! = 1 \& wb\_active! = 1)$

Table 4-1 presents the verification details for the above cluster. This cluster has 7 properties where $p13$ is the base property. The second column shows the property type (node coverage, edge coverage, interaction coverage etc.). The third column indicates the bound for that property. The fourth column shows the number of CNF clauses (size) for that property. The fifth column presents the number of conflict clauses forwarded from the base property. The next column presents the test generation time (original, in seconds) using unmodified zChaff. The seventh column presents the test geneation time using our approach. The new time is larger for the base property since it includes the intersection calculation time with other properties in the cluster. The speedup is computed using the formula (Original Time / New Time). The overall speedup for this cluster is 4.18x.

Table 4-1. Verification results for a structure-based cluster

Prop.	Type	Bound	Size	Forward	Orig.(s)	New(s)	Speedup
$p13$	Inter.	8	461122	-	15.61	21.99	0.71
$p28$	Edge	7	395566	32576	8.31	0.16	51.94
$p133$	Edge	7	395564	32576	11.99	0.18	66.60
$p134$	Inter.	7	395564	32576	9.07	0.19	47.74
$p150$	Node	6	330002	21748	4.70	0.16	29.38
$p165$	Path	8	461132	35121	22.87	0.27	84.70
$p170$	Path	8	461142	35121	24.45	0.26	94.04
Avg.	-	7.29	414299	-	13.86	3.32	4.18

Table 4-2. Structure-based clustering results for MIPS processor

Cluster Index	Size (# Prop)	Base Time (s)	Original Time (s)	Improved Time		Speedup
				Verify(s)	Overhd(s)	
1	10	1.21	68.26	32.01	5.91	1.78
2	8	1.84	83.26	37.43	6.02	1.88
3	17	15.90	193.21	2.49	15.44	6.18
4	17	18.31	173.20	3.81	14.47	5.23
5	7	15.61	81.40	1.22	6.38	4.18
6	7	2.03	120.38	40.05	5.71	2.56
7	4	2.15	15.94	5.79	2.62	1.71
8	1	8.56	8.56	8.56	0.00	1.00
9	17	30.92	582.80	59.44	17.57	5.69
10	17	2.30	149.75	50.74	12.83	2.31
11	7	10.54	140.31	30.77	6.78	3.14
12	17	9.40	669.83	164.34	17.39	3.55
13	11	21.21	365.79	44.1	12.26	4.99
14	4	10.62	46.58	3.84	3.54	3.18
15	14	15.84	142.78	4.00	11.47	5.07
16	13	2.65	263.93	149.19	11.92	1.63
Avg.	10.69	10.57	194.12	39.86	9.39	3.42

Table 4-2 provides the overall verification details of the clusters generated using the structural similarity. The total 171 properties are grouped into 16 clusters shown in the first column. The example presented in Table 4-1 is the expansion of the fifth cluster in Table 4-2 (row 5). The second column presents the size of that cluster in terms of number of properties. The base time is the execution time of the base property. The original time is the running time of the remaining properties (except the base property) without using any knowledge sharing techniques. Since intersection calculation is necessary before executing the base property, we show the improved time (our approach) in two parts: new verification time, and overhead (intersection calculation time). The last column shows the speedup using the formula (Base time + Original time) / (Base time + Improved time). In this table, we can see that the overhead has a linear relation with the number of properties in the cluster. Using structural clustering, we can achieve a speedup of 3.42x [3] .

[3] Clustering time using structural similarity is negligible and not shown in the table.

4.5.1.2 Clustering based on Textual Similarity

Since the properties are generated based on fault models, they use similar format and therefore helpful for clustering based on textual similarity. In this case, we assume that 50% is a reasonable threshold for textual similarity. For example, the following properties are textually similar. In this case, $p49$ is the base property, and other 6 properties has 50% similarity with it. So they can be clustered together.

- $p49 =\sim F(m1\_active! = 1 \& m6\_active! = 1)$
- $p50 =\sim F(m1\_active! = 1 \& m7\_active! = 1)$
- $p61 =\sim F(m2\_active! = 1 \& m6\_active! = 1)$
- $p72 =\sim F(m3\_active! = 1 \& m6\_active! = 1)$
- $p82 =\sim F(m4\_active! = 1 \& m6\_active! = 1)$
- $p91 =\sim F(m5\_active! = 1 \& m6\_active! = 1)$
- $p100 =\sim F(m6\_active! = 1 \& m7\_active! = 1)$

Table 4-3. Verification results for a textual cluster

Prop.	Type	Bound	Size	Forward	Orig.(s)	New(s)	Speedup
$p49$	Inter.	10	592239	-	59.54	68.81	0.87
$p50$	Inter.	11	657806	78826	81.09	5.88	51.94
$p61$	Inter.	10	592239	78826	60.72	0.31	195.87
$p72$	Inter.	10	592239	78826	62.37	0.31	201.19
$p82$	Inter.	10	592239	78826	61.91	0.31	199.71
$p91$	Edge	10	592239	78826	67.96	0.31	219.23
$p100$	Edge	11	657806	78826	84.17	6.08	13.84
Avg.	-	10.29	610972	-	68.25	11.72	5.82

Table 4-3 shows the verification details for a cluster consisting of above 7 properties. The numbers in the table are in the same format as Table 4-1. Due to knowledge sharing, the speedup for this cluster is 5.82x.

Table 4-4 shows the test generation details for all 32 clusters using textual similarity. Table 4-3 is the expansion of the 22^{nd} cluster of Table 4-4 (row 22). In this case, our approach is able to obtain the overall speedup of 3.72.

Table 4-4. Textual clustering results for MIPS processor

Cluster Index	Size (# Prop)	Base Time (s)	Original Time (s)	Improved Time		Speedup
				Verify(s)	Overhd(s)	
1	1	0.11	0.11	0.11	0	1.00
2	1	0.12	0.12	0.12	0	1.00
3	1	0.35	0.35	0.35	0	1.00
4	1	0.35	0.35	0.35	0	1.00
5	3	1.28	4.62	2.57	1.53	1.10
6	5	2.75	15.63	6.02	3.34	1.52
7	8	5.56	72.61	15.23	6.55	2.86
8	11	11.30	183.44	26.31	10.57	4.04
9	11	17.72	249.19	40.57	12.03	3.80
10	10	30.58	456.97	48.44	12.38	5.33
11	1	0.30	0.30	0.30	0 .00	1.00
12	3	1.28	4.65	2.00	1.57	1.22
13	5	2.69	17.78	7.82	3.40	1.47
14	8	5.00	77.04	21.91	6.62	2.45
15	11	4.7	100.19	34.17	9.16	2.18
16	3	1.55	4.77	1.22	1.62	1.44
17	5	2.73	18.17	4.28	3.42	2.00
18	2	1.21	1.84	1.42	0.97	0.85
19	17	15.67	269.53	6.18	16.45	7.39
20	13	7.74	127.90	4.49	11.24	5.78
21	4	2.04	7.78	1.13	2.38	1.77
22	7	59.54	418.22	13.22	9.27	5.82
23	7	10.34	69.91	9.16	5.82	3.17
24	3	29.07	61.34	0.32	3.39	2.76
25	4	95.77	288.45	0.61	5.66	3.77
26	6	21.63	104.19	0.85	5.98	4.42
27	4	4.02	29.97	4.24	3.05	3.00
28	2	10.46	10.50	0.15	1.72	1.70
29	5	18.64	81.71	0.83	5.08	4.09
30	5	21.07	78.80	6.61	5.22	3.04
31	3	22.25	44.91	0.46	3.05	2.61
32	1	28.78	28.78	28.78	0	1.00
Avg.	5.34	13.64	88.44	9.07	4.74	3.72

4.5.1.3 Influence-based Clustering

The following 7 properties are grouped using influence-based clustering with $p111$ as the base property. We set the threshold of the similarity as 70%. For instance, the influence nodes of $p111$ are $\{FET, DEC, MUL1, MUL2, MUL3, MUL4, MUL5, MUL6, MUL7, FADD1, FADD2, FADD3, FADD4\}$, and the influence of $p108$ is $\{FET, DEC, MUL1, MUL2, MUL3, MUL4, MUL5, MUL6, MUL7, FADD1\}$. The similarity between $p108$ and $p111$ is $10/13 = 77\%$.

- $p111 =\sim F(m7\_active! = 1 \& f4\_active! = 1)$

- $p104 =\sim F(m6\_active! = 1 \& f4\_active! = 1)$
- $p110 =\sim F(m7\_active! = 1 \& f3\_active! = 1)$
- $p103 =\sim F(m6\_active! = 1 \& f3\_active! = 1)$
- $p109 =\sim F(m7\_active! = 1 \& f2\_active! = 1)$
- $p102 =\sim F(m6\_active! = 1 \& f2\_active! = 1)$
- $p108 =\sim F(m7\_active! = 1 \& f1\_active! = 1)$

Table 4-5 shows the verification results for an influence-based cluster consisting of the above 7 properties. In this case, the overall speedup using our approach is 4.52x.

Table 4-5. Verification results for an influence-based cluster

Prop.	Type	Bound	Size	Forward	Orig.(s)	New(s)	Speedup
$p111$	Inter.	10	592239	-	54.80	63.40	0.87
$p104$	Inter.	9	526687	66773	25.98	0.22	118.09
$p110$	Inter.	10	592239	70975	54.26	0.25	217.04
$p103$	Inter.	9	526687	66773	25.83	0.22	117.41
$p109$	Inter.	10	592239	70975	49.16	0.25	196.64
$p102$	Inter.	9	526687	66773	33.27	0.22	151.23
$p108$	Inter.	10	592239	70975	49.74	0.26	191.31
Avg.	-	9.57	564145	-	41.86	9.26	4.52

Table 4-6 shows the verification results using influence-based clustering for all 27 clusters. The details of the first cluster (row 1) is shown in Table 4-5. The overall speedup using our approach is 4.30x.

4.5.1.4 Intersection-based Clustering

Intersection-based clustering is intuitive and easier to implement since it does not require any prior knowledge about the structure of the graph model or the format of the properties. It only uses the mapping of the variables for name substitution and the intersection between the CNFs. Due to use of data structure *hashmap*, the intersection time is linear to the size of the CNF file. The following properties are grouped as a cluster using a threshold for the intersection as 90%.

- $p50 =\sim F(m1\_active! = 1 \& m7\_active! = 1)$
- $p62 =\sim F(m2\_active! = 1 \& m7\_active! = 1)$

Table 4-6. Influence-based clustering results for MIPS processor

Cluster Index	Size (# Prop)	Base Time (s)	Original Time (s)	Improved Time		Speedup
				Verify(s)	Overhd(s)	
1	7	54.80	238.24	1.42	8.60	4.52
2	15	55.31	874.07	38.38	19.18	8.23
3	6	0.07	72.30	83.01	5.18	0.82
4	11	21.22	173.93	4.81	10.44	5.35
5	17	25.94	570.77	48.36	19.22	6.38
6	7	10.49	62.39	4.89	5.92	3.42
7	14	8.98	188.18	22.39	12.64	4.48
8	6	9.41	19.76	0.86	4.45	1.98
9	17	11.76	192.75	20.44	14.62	4.37
10	7	4.06	44.33	10.76	5.29	2.41
11	8	4.39	49.22	7.26	5.91	3.05
12	4	24.29	49.00	0.90	3.92	2.52
13	6	15.54	73.46	0.72	5.74	4.05
14	5	2.19	8.99	2.25	2.86	1.53
15	6	2.18	12.60	1.42	3.44	2.10
16	7	12.98	84.54	8.65	6.45	3.47
17	6	19.49	63.14	1.01	5.59	3.17
18	2	4.58	1.83	0.11	1.27	1.08
19	1	2.31	2.31	2.31	0.00	1.00
20	9	10.57	107.50	16.85	8.14	3.32
21	2	1.54	0.35	0.08	0.74	0.80
22	3	18.24	26.83	0.43	2.90	2.09
23	1	0.35	0.35	0.35	0.00	1.00
24	1	0.30	0.30	0.30	0.00	1.00
25	1	1.21	1.21	1.21	0.00	1.00
26	1	0.12	0.12	0.12	0.00	1.00
27	1	0.12	0.12	0.12	0.00	1.00
Avg.	6.33	11.94	108.1	10.35	5.65	4.30

- $p73 =\sim F(m3\_active! = 1 \& m7\_active! = 1)$
- $p83 =\sim F(m4\_active! = 1 \& m7\_active! = 1)$
- $p92 =\sim F(m5\_active! = 1 \& m7\_active! = 1)$
- $p100 =\sim F(m6\_active! = 1 \& m7\_active! = 1)$

Table 4-7 presents the verification details for the above cluster using $p50$ as the base property. The speedup for this cluster is 5.96x.

Table 4-8 presents the intersection clustering verification for all the 171 properties. The details of the 9^{th} cluster are shown in Table 4-7. The overall speedup using our approach is 5.90x.

Table 4-7. Verification results for an intersection-based cluster

Prop.	Type	Bound	Size	Forward	Orig.(s)	New(s)	Speedup
$p50$	Inter.	11	657806	-	80.91	89.41	0.90
$p62$	Inter.	11	657806	91548	95.87	0.58	165.29
$p73$	Inter.	11	657806	91548	95.75	0.46	208.15
$p83$	Inter.	11	657806	91548	96.29	0.59	163.20
$p92$	Inter.	11	657806	91548	96.83	0.59	164.12
$p100$	Inter.	11	657806	91548	83.99	0.59	142.36
Avg.	-	11	657806	-	91.61	15.37	5.96

Table 4-8. Intersection-based clustering results for MIPS processor

Cluster Index	Size (# Prop)	Base Time (s)	Original Time (s)	Improved Time		Speedup
				Verify(s)	Overhd(s)	
1	4	1.22	4.08	0.27	1.75	1.64
2	13	1.82	28.44	1.31	7.48	2.85
3	17	15.68	266.61	2.76	16.99	7.97
4	17	7.72	147.75	1.80	14.51	6.47
5	17	3.65	66.50	2.00	11.96	3.98
6	14	26.19	383.10	2.28	15.91	9.22
7	13	60.61	691.41	2.68	16.58	9.42
8	17	8.51	172.23	3.10	14.20	7.00
9	6	80.91	468.73	2.81	8.50	5.96
10	17	20.57	323.98	2.73	16.71	8.61
11	12	13.01	120.28	2.17	10.26	5.25
12	4	4.74	15.29	0.41	2.88	2.49
13	2	0.11	0.11	0.04	0.30	0.49
14	3	0.35	0.65	0.16	0.89	0.71
15	13	18.91	249.84	2.40	13.29	7.77
16	1	30.63	30.63	30.63	0	1
17	1	29.54	29.54	29.54	0	1
Avg.	10	19.07	176.42	5.12	8.95	5.90

4.5.1.5 Comparison of Clustering Technqiues

Table 4-9 compares the four clustering technqiues. The first row shows our proposed clustering methods. The second row indicates the number of clusters using the respective clustering methods, and the third row shows the corresponding clustering time (in seconds). The fourth row presents the test generation time for the base property. Similar to the previous tables, the original time refers to traditional (no clustering) verification time for all the properties excluding the base property. The sixth row presents the verification time for all the properties except the base property using the respective clustering method. The speedup is computed using the formula (Base time + Original time) / (Clustering time + Base time + Improved time). For the first three clustering

methods, the clustering is very fast and the associated cost (time) is negligible. However, for the intersection-based clustering, the intersection time is longer compared to other three methods and is not negligible. Therefore, for intersection-based clustering, we provide speedup values for both scenarios – without considering clustering time (the first number) as well as with clustering time (the number in parenthesis).

Table 4-9. Property clustering and verification for MIPS processor

Methods	*Structure*	*Textual*	*Influence*	*Intersection*
Cluster No.	16	32	27	17
Clust. Time	0.24	0.06	0.22	187.90
Base Time	169.09	436.60	322.44	324.18
Orig. Time	3105.98	2830.13	2918.56	2999.16
Impr. Time	788.09	442.53	431.92	239.28
Speedup	3.42	3.72	4.33	5.90 (4.42)

It is important to note that intersection-based clustering is most beneficial for reducing overall test generation time. However, the clustering overhead is much more than other strategies. When a large number of complex properties are involved, the intersection overhead may become prohibitively large. In such cases, influence-based clustering is most beneficial. Interestingly, textual clustering consumes least amount of clustering time but generates better results than structure based clustering. When detailed information about the design is not available, textual clustering is most beneficial.

4.5.2 A Stock Exchange System

This section presents the test generation results of the on-line stock exchange system (OSES) (described in Section 2.3.5). The specification is used to generate 51 properties based on the fault model. We applied the clustering methods discussed in Section 4.3 on all the properties to generate the tests.

Table 4-10 presents the test generation results using structure-based clustering for all the 51 properties. The overall speedup using our approach is 2.26x.

Table 4-11 presents the test generation results using textual clustering for all the 51 properties. The overall speedup using our approach is 2.33x.

Table 4-10. Structure-based clustering results for OSES

Cluster Index	Size (# Prop)	Base Time (s)	Original Time (s)	Improved time		Speedup
				Verify(s)	Overhd(s)	
1	2	4.48	3.72	0.63	0.97	1.35
2	4	6.14	45.5	13.13	1.92	2.44
3	2	1.76	2.03	0.60	0.97	1.14
4	4	59.56	160.99	15.16	1.90	2.88
5	2	9.34	11.09	19.58	0.98	0.68
6	4	10.74	123.79	5.97	1.95	7.21
7	2	0.40	0.32	0.25	0.97	0.44
8	4	96.44	150.45	31.11	1.91	1.91
9	2	6.62	7.40	0.71	1.13	1.66
10	4	10.08	82.61	48.02	2.26	1.54
11	2	3.36	4.69	1.22	1.13	1.41
12	4	101.16	154.62	38.48	2.22	1.80
13	2	29.55	36.5	2.90	1.14	1.97
14	4	106.51	168.30	2.24	2.24	1.95
15	2	0.21	0.20	19.34	1.14	0.02
16	4	95.91	588.49	120.00	2.26	3.14
17	2	18.91	15.53	1.16	0.82	1.65
18	1	0.88	0.88	0.88	0.00	1.00
Avg.	2.83	31.23	86.51	19.51	1.44	2.26

Table 4-11. Textual clustering results for OSES

Cluster Index	Size (# Prop)	Base Time (s)	Original Time (s)	Improved time		Speedup
				Verify(s)	Overhd(s)	
1	1	0.68	0.68	0.68	0.00	1.00
2	2	15.55	18.86	7.73	0.81	1.43
3	9	4.33	196.59	60.88	4.26	2.89
4	8	60.25	135.37	36.83	3.80	1.94
5	1	33.57	33.57	33.57	0.00	1.00
6	6	11.62	246.23	2.05	2.86	15.60
7	9	6.44	469.61	130.68	5.01	3.35
8	8	10.61	155.82	95.90	4.50	1.50
9	7	0.21	760.38	390.69	3.91	1.93
Avg.	5.67	15.87	224.12	84.33	2.79	2.33

Table 4-12 presents the test generation results using influence-based clustering for all the 51 properties. The overall speedup using our approach is 2.44x.

Table 4-13 presents the test generation results using intersection-based clustering for all the 51 properties. The overall speedup using our approach is 2.84x without considering clustering overhead. If clustering overhead is considered the overall speedup is 2.69x.

Table 4-14 summarizes the results using four clustering methods where 2–3 times improvement is achieved. It is important to note that the results for OSES are consistent with the results for MIPS in Table 4-9. As Table 4-14 shows, intersection-based clustering

Table 4-12. Influence-based clustering results for OSES

Cluster Index	Size (# Prop)	Base Time (s)	Original Time (s)	Improved time		Speedup
				Verify(s)	Overhd(s)	
1	5	22.97	147.84	50.48	2.75	2.24
2	8	10.10	369.97	120.27	4.40	2.82
3	3	36.62	59.65	38.78	1.69	1.26
4	5	10.66	135.98	11.37	2.37	6.01
5	4	0.32	4.00	3.28	1.90	0.78
6	1	93.48	93.48	93.48	0	1.00
7	7	28.89	629.39	132.41	3.89	3.98
8	2	12.87	9.85	0.37	0.98	1.58
9	6	14.23	302.63	115.31	2.83	2.40
10	7	34.66	261.80	69.81	3.34	2.75
11	2	15.87	18.98	7.63	0.81	1.43
12	1	0.75	0.75	0.75	0	1.00
Avg.	4.25	23.12	169.50	53.65	2.08	2.44

Table 4-13. Intersection-based clustering results for OSES

Cluster Index	Size (# Prop)	Base Time (s)	Original Time (s)	Improved time		Speedup
				Verify(s)	Overhd(s)	
1	7	4.84	53.91	16.64	3.31	2.37
2	3	10.93	94.79	6.2	1.46	5.69
3	2	7.13	56.72	5.81	0.98	4.59
4	2	35.32	68.96	24.97	0.98	1.70
5	3	5.06	20.60	22.56	1.45	0.88
6	7	84.18	243.60	22.78	3.30	2.97
7	8	6.54	393.75	147.45	4.53	2.53
8	6	3.37	98.46	42.39	3.32	2.07
9	3	29.45	68.71	19.07	1.74	1.95
10	3	107.27	457.52	39.59	1.69	3.80
11	4	0.20	247.46	62.83	2.24	3.79
12	2	18.74	15.35	1.17	0.82	1.64
13	1	0.7	0.7	0.7	0	1.00
Avg.	3.92	24.13	140.04	31.70	1.99	2.84

is most beneficial for reducing overall test generation time. However, when clustering overhead is prohibitively large, influence-based clustering is beneficial. Similarly, when detailed information about the design is not available, textual clustering is the best choice.

Table 4-14. Property clustering and verification for OSES

Methods	*Structure*	*Textual*	*Influence*	*Intersection*
Cluster No.	18	9	12	13
Clust. Time	0.05	0.01	0.05	42.77
Base Time	562.05	142.81	277.42	313.73
Orig. Time	1557.11	2017.11	2034.05	1820.53
Impr. Time	377.15	784.16	668.72	437.98
Speedup	2.26	2.33	2.44	2.84 (2.69)

On two case studies (MIPS and OSES) our approach demonstrated 3–5 times improvement in overall test generation time using efficient integration of property clustering and conflict clause forwarding based learning techniques.

4.6 Summary

Directed test vectors can reduce overall validation effort since fewer tests can obtain the same coverage goal compared to the random tests. The applicability of the existing approaches for directed test generation is limited due to capacity restrictions of the automated tools. This chapter addressed the test generation complexity by clustering similar properties and exploiting the commonalities between them. To enable knowledge sharing across multiple properties, we have developed a number of conceptually simple, but extremely effective, techniques including name substitution and selective forwarding of learned conflict clauses. Our experimental results using both hardware and software designs demonstrated an average of four times speedup in directed test generation time.

CHAPTER 5
DECISION ORDERING BASED INTRA- AND INTER-PROPERTY LEARNING

The primary goal of efficient test generation is how to quickly get satisfiable assignments for SAT instances. Various heuristic methods and tools [39, 63] are proposed to improve the SAT searching time. *Decision ordering* [55] plays an important role during the search because different decision ordering implies different decision tree as well as different search path which strongly affect the search time. Existing decision ordering methods focus on exploiting the useful information of general SAT problem with a single SAT instance. Most of them are based on the statistics of SAT instances without considering any other learning information. For test generation, a design may have various properties and generally model checking techniques will check each of them individually. For a given design, similar properties describe correlated functional scenarios. Therefore the respective counterexamples are expected to have a significant overlap which can be used for sharing learning. Furthermore, even for a single SAT instance, the result of the local search can also benefit the global search. The method proposed in this chapter exploits the learning from decision ordering in the context of test generation involving one or more properties of a design. This chapter makes three contributions: i) investigates the decision ordering based learning for a single SAT instance; ii) applies the decision ordering based learning between similar SAT instances; and iii) exploits the relation between the decision ordering and conflict clause forwarding based methods.

The rest of the chapter is organized as follows. Section 5.1 presents related work on decision ordering based heuristics. Section 5.2 describes our learning techniques based on decision ordering. Section 5.3 proposes the test generation methodology using efficient decision ordering techniques. Section 5.4 presents the experimental results. Finally, Section 5.5 summarizes the chapter.

5.1 Related Work

Different variable ordering will lead to different search trees, therefore branching heuristics can improve the SAT searching performance significantly [55]. As a popular SAT solver, zChaff uses the Variable State Independent Decaying Sum (VSIDS) heuristic [63]. This heuristic contains two parts: i) the static part collects the statistics of the Conjunctive Normal Form (CNF) literals prior to SAT solving and sets the initial decision ordering, and ii) during the SAT solving, the dynamic part periodically updates the priority based on conflict clauses. Although the above general-purpose heuristics are promising for propositional formulas, they neglect some unique information of BMC. In [81], Strichman exploited the characteristics of the BMC formulas for a variety of optimizations including decision ordering. When the bound is unknown, SAT-based BMC needs to increase the unrolling depth one-by-one until finding a counterexample. Wang et al. [87] analyzed the correlation among different SAT instances of a property. They used the *unsatisfiable core* of previously checked SAT instances to guide the variable ordering for the current SAT instance.

To the best of our knowledge, all the existing approaches exploit variable ordering to improve the SAT solving time involving only one property (one SAT instance or several correlated SAT instances with different bounds). Our approach is the first attempt to use both decision ordering and conflict clauses to reduce the BMC based test generation time for a single SAT instance as well as for a cluster of similar SAT instances. The comparison between various learning techniques is provided in Section 5.4.

5.2 Decision Ordering Based Learnings

Decision ordering plays an important role during the SAT search. It indicates which variable will be selected first and which value (true or false) will be first assigned to this variable. Similar to BDD based methods [15], variable ordering determines the performance of the SAT solving time. In the VSDIS heuristics implementation of zChaff, each literal l is associated with a *zchaff_score(l)* which is used for decision ordering at

decide_next_branch() (see Algorithm 3 in Chapter 4). Initially the score is equal to the literal count in corresponding CNF file. During the SAT solving, the score will be updated in periodic function after a certain numbers of backtracks. The calculation of the new literal score is as follows:

$$chaff\_score(l) = chaff\_score(l)/2 + lits\_in\_new\_confs(l) \tag{5–1}$$

where *lits_in_new_confs(l)* is the number of newly added conflict clauses which contain literal l since last update.

Similar properties usually have similar counterexamples which indicates that they may have similar Boolean constraints during the test generation. Consequently the generated SAT instances should have a large overlap in CNF clauses and can be clustered to share the learning. This section presents our decision ordering heuristic which will be incorporated in the test generation approaches in Section 5.3.

5.2.1 Overview

As discussed in Section 4.2.1, the most time consuming parts are BCP and long distance backtracking. They are indicated by implication number and conflict clause number which represent the successful decision ratio and backtrack number respectively. Ideally, a search method can get a satisfiable assignment by making the assignment for each variable only once. However, generally it is impossible to achieve such scenario. For a cluster of similar properties and pre-determined bounds, the objective of our method is to reduce the number of implications and conflict clauses of unchecked properties by incorporating the learned decision ordering knowledge from previously checked properties.

Assuming that we have two similar properties, both properties will have a large overlap on CNF clauses and counterexample assignments. Figure 5-1 shows the partial views of search trees and search paths of the two properties. The search paths are formed according to the decision ordering (shown on top of the search trees). For each variable v in the ordering, there are two literals (v means *v=1* and *v'* means *v=0*). As shown in

Figure 5-1a, there are 6 conflicts encountered. The search stops after finding a satisfiable assignment $a = 1$, $b = 0$, $c = 0$, $d = 1$ in this scenario. In Figure 5-1b, the search will be successful only when $a = 0$, $b = 0$, $c = 0$, $d = 1$ after encountering 14 conflicts. Therefore the search of the second example will be more time-consuming because of more backtracks.

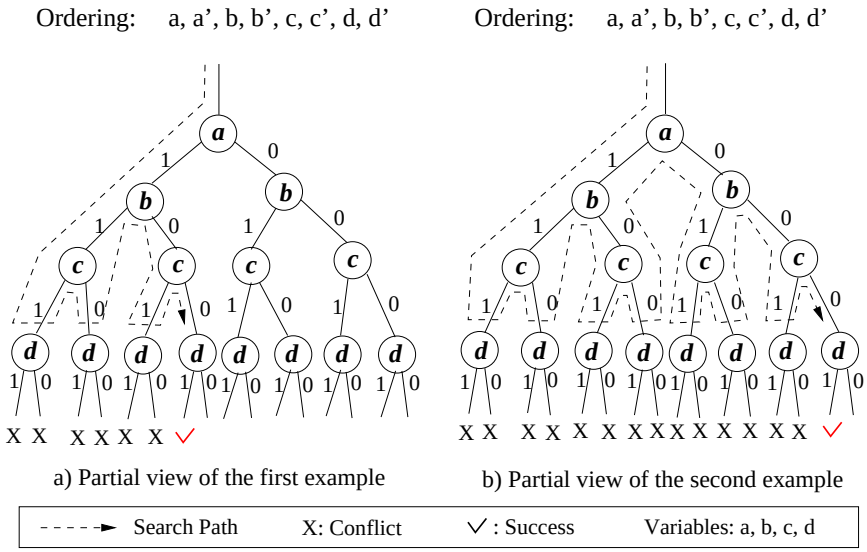


Figure 5-1. Two examples of SAT search

Because of the large overlap in the assignment of counterexamples, the result of previously checked properties can be used as a learning for unchecked properties. For example, in Figure 5-1, the result of first example strongly indicates the assignment of the second example because of the satisfiable assignment intersection $b = 0$, $c = 0$, $d = 1$. If the second example uses the decision ordering based on the variable assignments in the first example, the searching time of the second example can be drastically reduced as shown in Figure 5-4.

5.2.2 Bit Value Ordering

Similar properties generally have a large intersection on both corresponding CNF clauses and counterexample assignments. This indicates that the satisfiable assignment of checked SAT instances contain rich decision ordering knowledge for unchecked satisfiable SAT instance. In SAT search, incorrect value selection for each variable will cause conflicts which will result in backtracks to remove the reason of the conflicts. A good decision

ordering can mostly avoid such faulty assignments. Unlike pruning the search tree using conflict clause forwarding [58], bit value ordering changes the *search path*. By setting the *bit priority* (choose 0 or 1 first) for each variable using the knowledge of previous property checking, the length of the search path can be reduced.

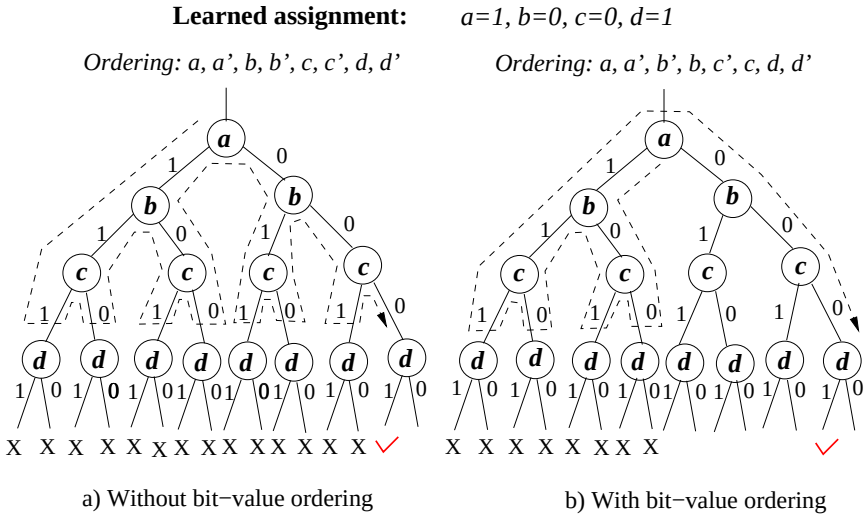


Figure 5-2. A scenario where bit-value ordering works

Figure 5-2 shows an example where bit-value ordering works. As shown in Figure 5-1a, we can get a satisfiable assignment $a = 1$, $b = 0$, $c = 0$ and $d = 1$. This assignment can be used to change the bit-value ordering of the second example. That means, when node b is encountered, the search chooses $b = 0$ first in its search path. The same rule also applies on other nodes. Applying such heuristic in Figure 5-2b, there are only 8 conflicts encountered compared to 14 conflicts in Figure 5-2a. In addition, the search path is also shortened. Therefore, the searching time is reduced.

It is important to note that the bit-value ordering itself is not always helpful for the SAT searching. For example in Figure 5-3, $a = 1$, $b = 1$, $c = 0$, $d = 1$ is the only satisfiable assignment in the given scenario. The searching in Figure 5-3a without bit value ordering is faster than the searching in Figure 5-3b because of less conflicts. If the learning assignment in Figure 5-3 was a=0, b=1, c=0 and d=1, the searching performance will be much worse than the search in Figure 5-3b. Clearly, in the search tree, the high level

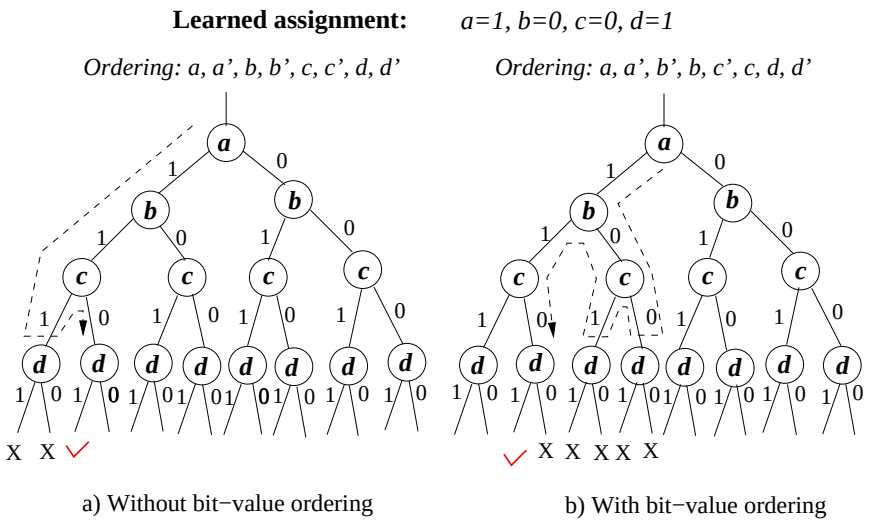


Figure 5-3. A scenario where bit value ordering fails

variables (e.g., node a) strongly affect the performance of the searching if they are not consistent with learned bit-value ordering.

5.2.3 Variable Ordering

Although bit-value ordering is promising in general, there are still a lot of conflicts encountered during the search. According to the example shown in Figure 5-3, if high level nodes (e.g., node a) make the wrong decision, the search path will be lengthened due to the long distance backtrack. To reduce the searching time, it is necessary to restrict the conflict detection and reasoning in a small area.

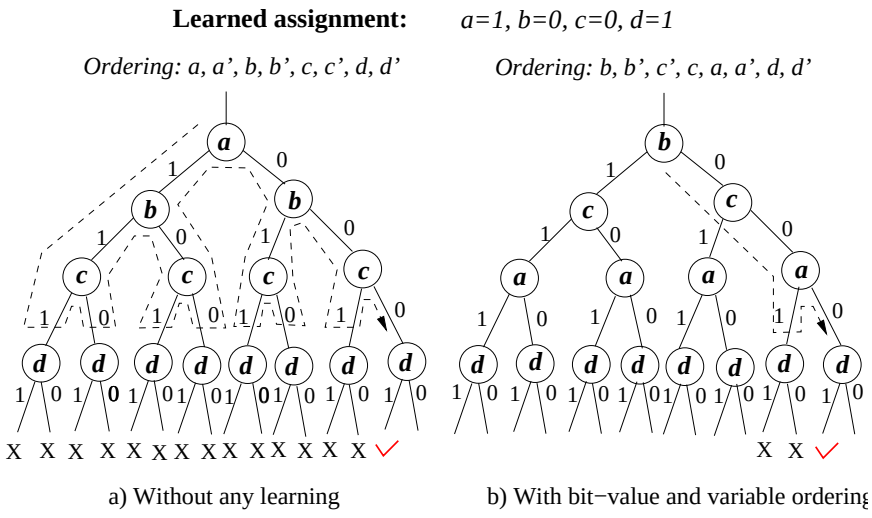


Figure 5-4. An example of bit-value and variable ordering

Efficient combination of variable ordering and bit-value ordering is very promising. As shown in Figure 5-4b, the search time is better than that in Figure 5-4a due to a shorter search path and less conflicts. The reason of this improvement is that we enhance the priority of variables b and c. Since a is the variable with different values between the two satisfiable assignments shown in Figure 5-1, lowering down the priority of such variables (ones with different values between two CNFs) can efficiently avoid the long distance backtrack. Generally, before SAT solving, it is hard to figure out the difference between two satisfiable CNF variable assignments. However, based on the value assignment statistics of the checked properties, the variable ordering can be constructed. For a variable with the lower assignment value variation, which indicates high chance of same value, we will enhance its priority by increasing the score of its two literals.

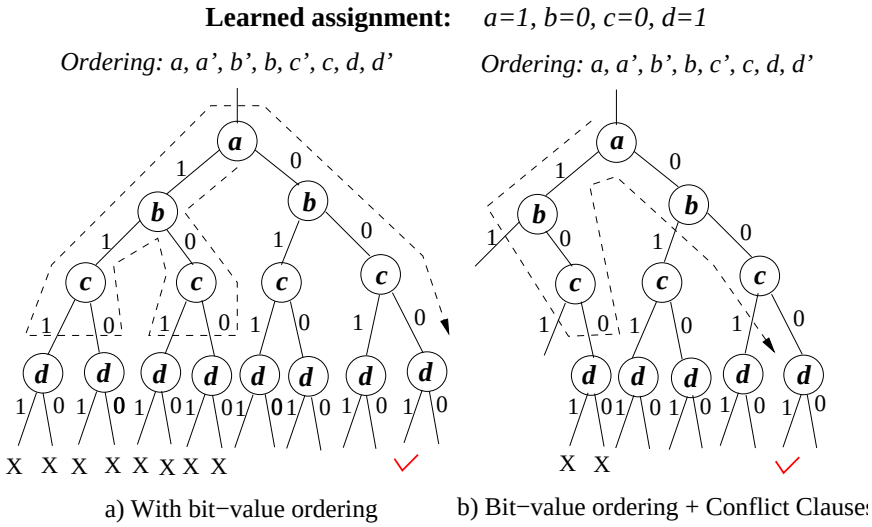


Figure 5-5. An example of conflict clauses based variable ordering

5.2.4 Conflict Clause based Decision Ordering (Hybrid)

Conflict clause is promising to avoid repeated conflicts during the SAT searching. Therefore it can be used as a learning during the test generation (described in Chapter 4). In essence, conflict clause forwarding can be used to prune the decision tree and can be utilized as a complementary approach for the decision ordering techniques proposed in Section 5.2.2 and Section 5.2.3. For two similar SAT instances, if the conflict clauses of the

checked SAT instance can be forwarded to the unchecked one, it will reduce the conflicts, thus further shorten the search path.

Figure 5-5a shows application of bit-value ordering on the example shown in Figure 5-1b. There are 8 conflicts during the SAT search in this case. Let's assume the conflict clauses generated from Figure 5-1a can be forwarded to the CNF clauses of Figure 5-1b. The generated 6 conflict clauses are as follows:

$$\left.\begin{array}{l}(a' \vee b' \vee c' \vee d') \\ (a' \vee b' \vee c' \vee d) \\ (a' \vee b\prime \vee c \vee d') \\ (a' \vee b' \vee c \vee d)\end{array}\right\} \Rightarrow (a' \vee b') \qquad (5\text{–}2)$$

$$\left.\begin{array}{l}(a' \vee b \vee c' \vee d') \\ (a' \vee b \vee c' \vee d)\end{array}\right\} \Rightarrow (a' \vee b \vee c')$$

Equation 5–2 shows the resolution of the forwarded conflict clauses. Based on the result, we can prune the search tree as shown in Figure 5-5b. It indicates that there are only 2 conflicts by applying the bit value ordering on the pruned search tree. Therefore the test generation time can be significantly reduced. For the example shown in Figure 5-4b, the conflict clause forwarding is not beneficial since the search does not traverse the pruned part of the decision tree. Generally, the conflict clause forwarding can further improve the performance of the decision ordering based methods.

5.3 Test Generation using Decision Ordering

For model checking based test generation, each property is a negation of a desired system behavior. Consequently each property can produce a counterexample. Since our method adopts SAT-based BMC, we assume that the bound can be pre-determined and the generated SAT instances are satisfiable. The goal of the test generation for the property with a known bound is to figure out a satisfiable assignment for this SAT instance.

To reduce the overall test generation effort, this section utilizes the heuristics proposed in Section 5.2 as a learning. Section 5.3.1 applies the learning based on the decision ordering for test generation of a single property. In Section 5.3.2, we present an algorithm which shares learning from the decision ordering among a cluster of similar properties.

5.3.1 Test Generation for a Single Property

When checking a base property using property clustering techniques, or when checking only a single property, current methods solve the SAT instance alone since there is no source of learning. Therefore it is time-consuming and it can be a major bottleneck of the clustering based test generation.

During test generation, if the bound of a property is increased by one, the complexity will be drastically increased. Based on the observation of [81], the reason of time-consuming search is due to the long distance backtracking. Since large set of clauses that belong to different distant cycles are being satisfied independently (locally), [81] found that there are three typical scenarios which can cause the conflicts:

- Distant cycles are being satisfied independently until they collide each other with assignment conflict.
- Some cycle assignment collides with the constraints imposed by the initial state.
- Some cycle assignment collides with the constraints imposed by the negation of the specified property.

The resolution of such conflicts needs to cancel large number of variable assignment between the conflicting cycles. Especially for the SAT instance with large bound, the cost of non-chronological backtracking is still huge since large bound indicates huge number of clauses and variables.

To alleviate long distance backtrackings during test generation, learning is required to guide the SAT search. Conflict clause is a promising learning that can prune the decision tree. However, in a SAT instance with large bound, the cost of deriving a conflict clause is

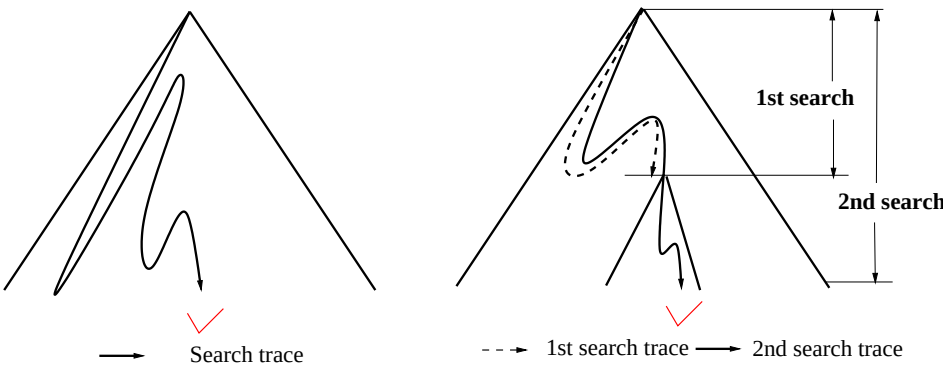
costly due to large interleaving of irrelevant variables during the SAT search. Furthermore, large set of CNF clauses is likely to generate a large number of conflict clauses which can affect the search performance. Therefore if we can get conflict clauses from a smaller SAT instance, then the average cost of conflict clause generation will be reduced. As an alternative, decision ordering can be used as learning. Since the SAT instance is assumed to be satisfiable, each segment[1] of the CNF clauses should be satisfiable. The searching time for a segment is much shorter than the original SAT instance. Although a segment can not reflect the global view of the system, if the satisfiable assignment of the segment is consistent to the partial variable assignment of the original SAT instance, it will be helpful to reduce the overall test generation time of the original SAT instance.

5.3.1.1 Heuristic Implementation

The basic idea of our heuristic for test generation involving a single property is to use the learnings from a small part of the SAT instance to guide the search of the whole SAT instance. By dividing the SAT instance into two segments, we can get the first segment which contains the initial state constraints and the second segment which contains property constraints. After checking any one of them, we can get the partial variable assignments which can be used as decision ordering learning, and we can get the conflict clauses which can be forwarded to the original property according to Theorem 2 in Chapter 4.

Figure 5-6 demonstrates an example of using such learnings. In Figure 5-6b, we first check one part of the SAT instance and get the corresponding learnings. Then during the checking of whole SAT instance, under the guidance of the learned knowledge, the overall search path is shortened compared to Figure 5-6a.

[1] A CNF SAT instance can be viewed as a union of a set of segments where each segment consists of a set of CNF clauses.



a) A search without any learnings b) A search with two kinds of learnings

Figure 5-6. Learning techniques for a single property

Our decision ordering heuristics implementation uses an array $var[sz]$ (sz is the largest variable number for CNFs) to indicate the satisfiable assignment result of the first search. Each element of the array $var[i]$ ($0 < i \leq sz$) has three values: 1 means that the i^{th} variable is assigned with 1; 0 means that the i^{th} variable is assigned with 0; and -1 implies that the variable is not assigned during the first search. So during the second search, the literal score is calculated using the following formula where $max(v_i) = MAX(chaff\_score(v_i), chaff\_score(v_i'))+1$.

$$score(l_i) = \begin{cases} max(v_i) & (var[i] == 1 \ \& \ l_i = v_i) \\ & or(var[i] == 0 \ \& \ l_i = v_i') \\ chaff\_score(v_i) & otherwise \end{cases} \quad (5\text{–}3)$$

5.3.1.2 Test Generation

Algorithm 7 describes our test generation procedure for a single property using learnings from some part of the SAT instance corresponding to the original property. Step 1 initializes all the elements of var with -1. Step 2 generates the CNF clauses for the property p. After dividing the CNF into two parts in step 3, step 4 solves the clauses in any one part and derives the learning in the form of decision ordering and conflict clauses. Step 5 updates the var. Finally, step 6 uses the learning to guide the test generation of the original property.

Algorithm 7: Test Generation for a Single Property

Input: i) Formal model of the design, D

ii) Property p with bound b

Output: A test t for p with generated conflict clauses

1. Initialize var;

2. $CNF = BMC(D,\ p,\ b)$;

3. Divide CNF into CNF_1 and CNF_2;

4. ($assign$, $conf\_clauses1$)=SAT(CNF_1 or CNF_2, var, NULL);

5. Update var using $assign$;

6. (t, $conf\_clauses2$) = SAT(CNF, var, $conf\_clauses1$);

return (t, conf_clauses1 + conf_clauses2);

It is important to note that our heuristic for a single property is based on the assumption that the decision ordering knowledge learned from the first search has a large overlap with a satisfiable assignment of the second search. Although the forwarded conflict clauses can prune the decision space, it is still possible that the first search may mislead the second search which will aggravate the overall searching time. Since we halve the SAT instance and each part can be checked individually, for test generation, we use the following three strategies in parallel:

- Directly solve the original SAT instance.
- Solve the first part and use the learnings to solve the original instance.
- Solve the second part and use the learnings to solve the original instance.

Once one of the above methods finds a satisfiable assignment, the remaining two processes will be terminated. Therefore, we can guarantee the worst case of the test generation time is the same as directly solving the original SAT instance.

5.3.2 Test Generation for a Cluster of Similar Properties

For similar properties, there exists a large overlap between corresponding counterexamples. Therefore the satisfiable assignments of checked properties can be used as a learning for

other properties in the cluster. Some of the derived conflict clauses can also be forwarded as learning. This sub-section will discuss how to extract the bit-value ordering and variable ordering based learnings from the checked properties in details. Also we will describe an algorithm to utilize the learning based on decision ordering for test generation of a cluster of similar properties.

5.3.2.1 Heuristic Implementation

In our heuristic implementation, we predict the decision ordering based on the statistics collected from the checked properties. Let $varStat[sz][2]$ (sz is the largest variable number for CNFs) be a 2-dimensional array to keep the count of variable assignments. Initially, $varStat[i][0] = varStat[i][1] = 0$ $(0 < i \leq sz)$. $varStat$ will be updated after checking each property. Assuming we are now checking property p_j, if the value of variable v_i in the assignment of the p_j is 0, then $varStat[i][0]$ will be increased by one; otherwise, $varStat[i][1]$ will be increased by one. This updated information of $varStat$ will be utilized when checking property p_{j+1}.

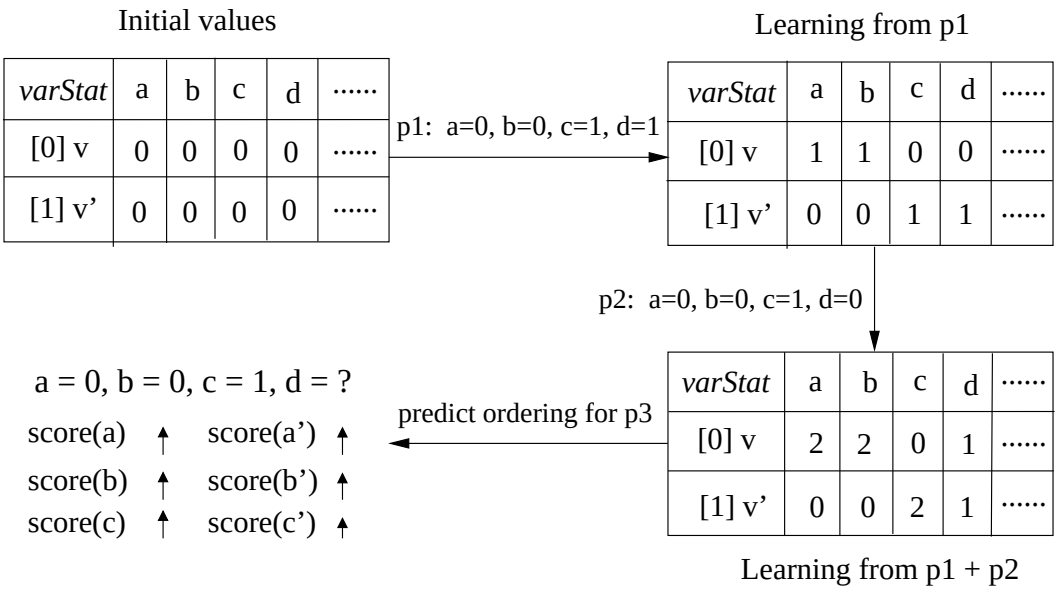


Figure 5-7. Statistics for two properties

For example, if we have three properties p_1, p_2 and p_3, the statistics after checking p_1 and p_2 are shown in Figure 5-7. When checking p_3, we can predict its decision ordering based on the collected information saved in $varStat$. The content of $varStat$ indicates

that variables a and b are more likely to be 0, c is more likely to be 1 and d can be assigned any value. Furthermore, *varStat* implies that the assignments for variable a, b and c are more consistent than the assignment for variable d. Thus the score of variable a, b and c will be increased. In other words, they will be searched first as described in Section 5.2.3.

Assuming l_i is a literal of v_i, we use the following equation to predict the bit value assignment of v_i when checking p_{j+1}.

$$potential(l_i) = \begin{cases} 1 & (varStat[i][1] > varStat[i][0] \& l_i = v_i) \\ & or(varStat[i][1] < varStat[i][0] \& l_i = v_i') \\ 0 & otherwise \end{cases} \tag{5–4}$$

Here, $potential(l_i) = 0$ means that value of l_i is more likely to be 0 in the satisfiable assignment of p_{j+1}. For example, in Figure 5-7, $potential(a) = 0$ which means that a is more likely to be assigned with 0. Let

$$ratio(i) = \frac{max(varStat[i][0], varStat[i][1]) + 1}{min(varStat[i][0], varStat[i][1]) + 1} \tag{5–5}$$

indicates the assignment variance of variable v_i. The larger $ratio_i$ means that the value assignments for variable v_i are more consistent. So it can be used for variable ordering. e

Our decision ordering heuristic is based on VSIDS. The only difference is that our method incorporates the statistics of previously checked properties. For each literal l_i, we use $score(l_i)$ to describe its priority. Initially, $score(l_i)$ is equal to the literal count of l_i. At the beginning of search as well as periodically decaying time, the literal score will be recalculated using the following equation where *max(v_i)=MAX(score(v_i), score(v_i'))+1.*

$$score(l_i) = \begin{cases} max(v_i) * ratio(i) & pontential(l_i) = 1 \\ score(l_i) * ratio(i) & otherwise \end{cases} \tag{5–6}$$

5.3.2.2 Test Generation

Algorithm 8: Test Generation for A Property Cluster

Input: i) Formal model of the design, D

ii) Property cluster, P, with satisfiable bounds

Output: *Test-suite*

1. Initialize $varStat$;

2. Select the base property p_1 and generate CNF, CNF_1;

for *i is from 2 to the size of cluster P* **do**

3. Generate CNF, $CNF_i = BMC(D,\ p_i,\ bound_i)$;

4. $INT_i = ComputerIntersection(CNF_1, CNF_i)$;

5. Mark the clause of CNF_1 using INT_i;

end

6. ($test_1$, $conf\_clause$) = Algorithm7(D, p_1, $bound_1$);

Test-suite = $\{test_1\}$;

for *i is from 2 to the size of cluster P* **do**

7. Update $varStat$ using $test_{i-1}$;

/*Figure out the learned conflict clauses from p_1*/;

8. $CC_i = Filter(conf\_clause, i)$;

9. ($test_i$,) = SAT(CNF_i, $vatStat$, CC_i);

Test-suite = *Test-suite* $\cup$ $test_i$;

end

return *Test-suite*;

Algorithm 8 describes our test generation methodology. The inputs of the algorithm are a formal model of the design and a cluster of similar properties. The first step initializes $varStat$ which is used to keep statistics of the variable assignments. Step 2 generates the CNF clauses for the base property p_1. Step 3 generates the CNF clauses for other properties. After figuring out the intersection between the base property with other properties in step 4, step 5 marks the clauses of base property (the marking is used for conflict clause identification in step 8). Step 6 solves the base property using Algorithm 7,

and generates a test as well as the $conf\_clause$ which can be used as learnings for the test generation of the remaining properties in the cluster using steps 7-9. After solving each property, we need to update the $varStat$ in step 7. Step 8 finds the proper conflict clauses which can be forwarded to the current property. Step 9 solves the current property using the learnings based on conflict clauses and decision ordering. Finally, the algorithm reports all the generated counterexamples (tests). It is important to note that this algorithm combines both the conflict clause and decision ordering based learnings. If only decision ordering learning is used, steps 4, 5, 8 should be omitted. Similarly, if only conflict clause forwarding is applied, then step 7 should be omitted.

5.4 Case Study

This section presents case studies for efficient test generation using our decision ordering as well as conflict clause based heuristics. Section 5.4.1 presents the case studies using intra-property learnings for checking individual SAT instances. The benchmarks collected are all pre-generated satisfiable SAT instances. By using inter-property learning for a cluster of similar SAT instances, Section 5.4.2 presents two case studies: a VLIW implementation of the MIPS architecture (described in Section 2.3.2) and the stock exchange system (described in Section 2.3.5). We used NuSMV [27] to generate the CNF clauses (in DIMACS format). We modified the SAT solver zChaff [74] to incorporate our proposed decision ordering heuristic on top of VSDIS. The experimental results are obtained on a Linux PC using 2.4GHz Core 2 Duo CPU with 2 GB RAM.

5.4.1 Intra-Property Learning

The benchmarks are collected from [83] and [85]. In [83], there are 13 SAT instances given in the benchmark set which are all taken from real industrial hardware designs (contribution of IBM research and Galileo). We chose four complex instances from them, because most SAT instances provided in [83] take short time during falsification. Apart from these four benchmarks, we also chose the benchmarks of two complex designs from

[85] as follows. Since we are focusing on test generation, the collected SAT instances are all satisfiable.

- **VLIW-SAT-4.0**, buggy VLIW processors with instruction queues and 9-stage pipelines; the processors support advanced loads, predicated execution, branch prediction, and exceptions.
- **PIPE-SAT-1.1**, buggy variants of the pipe benchmarks as presented in [86].

For the intra-property learning, we divide each SAT instance into two segments with the same number of clauses. Table 5-1 shows the test generation details using various intra-property learning techniques. The first column shows the names of the SAT instances. The second and third columns indicate the CNF size information including the variable number and clause number. The fourth column indicates the checking time by directly using zChaff without any other learning information. The fifth column shows the checking time using intra-property learning based on conflict clause forwarding, and the sixth column shows the test generation time using our decision ordering based heuristics. The seventh column presents the result which incorporates both conflict clause forwarding and decision ordering techniques as described in Section 5.2.4. Since we run different methods on different computers with the same settings, when one machine gets the satisfiable assignment, all the remaining SAT searches on the other machines will be terminated. Therefore the SAT searching time is the minimum searching time among these techniques. Based on such minimum time, the last column indicates the maximum speedup using the following formula:

$$speedup = \frac{MIN(zChaff,\ Conflict\ Clause,\ Decision\ Ordering,\ Hybrid)}{zChaff} \qquad (5\text{–}7)$$

where $zChaff$, $Conflict\ Clause$, $Decision\ Ordering$ and $Hybrid$ indicate the results of columns 4-7 in Table 5-1, respectively.

It is important to note that the execution time in columns 5-7 which adopt the intra-property learning techniques includes the learning time from divided/segmented

Table 5-1. Test generation results using intra learnings

SAT	CNF Size		zChaff [74]	Conflit Clause	Decision Ordering	Hybrid	Max
Instance	#Variable	#Clause	Time(s)	Time(s)	Time(s)	Time(s)	Speedup
bmc-galileo-8	58074	294821	0.99	0.43	0.74	**0.41**	2.30
bmc-galileo-9	63624	326999	1.74	0.99	0.94	**0.56**	3.11
Bmc-ibm-10	59056	323700	7.98	**3.96**	8.23	7.86	2.02
Bmc-ibm-11	32109	150027	6.98	4.58	**1.8**	6.97	3.88
VLIW-1	521188	13378461	1366.78	1070.4	2074.19	**489.15**	2.79
VLIW-2	521158	13378532	198.12	**77.45**	221.17	298.16	2.56
VLIW-3	521046	13376161	145.46	151.85	55.66	**52.93**	2.75
VLIW-4	520721	13348117	1126.13	295.15	599.22	**94.4**	11.93
VLIW-5	520770	13380350	879.24	757.09	703.1	**167.78**	5.24
VLIW-6	521192	13378781	211.50	**51.49**	544.26	317.26	4.11
VLIW-7	521147	13378010	**87.61**	189.41	357.63	400.74	1.00
VLIW-8	521179	13378617	1227.75	952.13	443.38	**377.74**	3.25
VLIW-9	521187	13378624	962.82	**107.99**	1523.44	1590.54	8.92
VLIW-10	521182	13378625	1769.14	**915.73**	930.2	1595.05	1.93
PIPE-1	138917	4678756	1327.92	752.52	279.82	**278.17**	4.77
PIPE-2	138918	4678718	1710.66	1703.37	403.97	**403.97**	4.23
PIPE-3	138917	4678757	825.78	394.07	**365.04**	969.33	2.26
PIPE-4	138563	4675040	1080.10	32.57	408.13	**14.06**	76.82
PIPE-5	138918	4678760	626.9	566.75	603.45	**114.57**	5.47
PIPE-6	138795	4671352	**0.43**	0.65	117.44	117.23	1.00
PIPE-7	138918	4678760	1734.26	987.88	1359.35	**534.72**	3.24
PIPE-8	138711	4688614	113.07	2.06	0.65	**0.65**	173.95
PIPE-9	138916	4676007	6062.27	6065.7	**355.56**	355.62	17.05
PIPE-10	138918	4678760	1430.29	1074.18	**277.98**	978.93	5.15

CNFs. This table shows that our method can drastically reduce the test generation time (up to 174 times). We can observe that in majority of the cases, the conflict clauses forwarding based intra-property learning can improve the test generation time compared to zChaff. However, decision ordering method and hybrid method are not always helpful. This is because the decision ordering based method may lead the search in a wrong way with more conflicts.

Figure 5-8 and 5-9 show the statistics of conflicts and implications for the collected benchmarks using various intra-property learning methods. We normalized the generated conflict clauses for each learning method using the total conflict clauses and implications generated by the four different methods shown in Table 5-1. The vertical axis of the stacked graphs shows the normalized percentage of conflict clauses and implications respectively. We can find that the result of the percentage of conflict clauses and implications is consistent. In other words, less conflicts will result in less implications. Furthermore, these figures also are consistent to the test generation performance shown in Table 5-1. It indicates that, by using the proposed intra-property learning methods in parallel, we can drastically reduce the conflicts as well as implications during the SAT searching. Consequently we can save the test generation time.

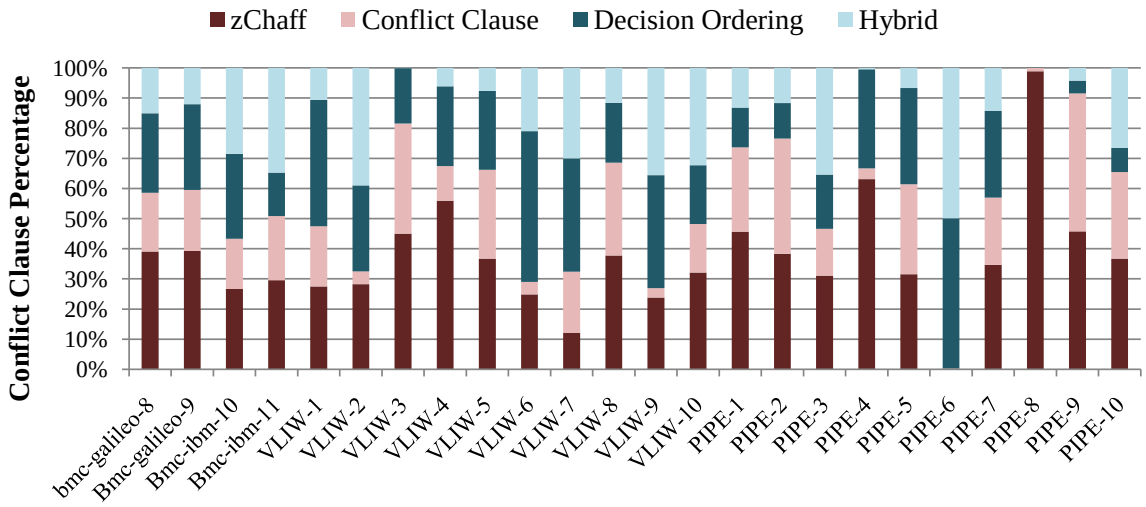


Figure 5-8. Conflict statistics using various intra-property learnings

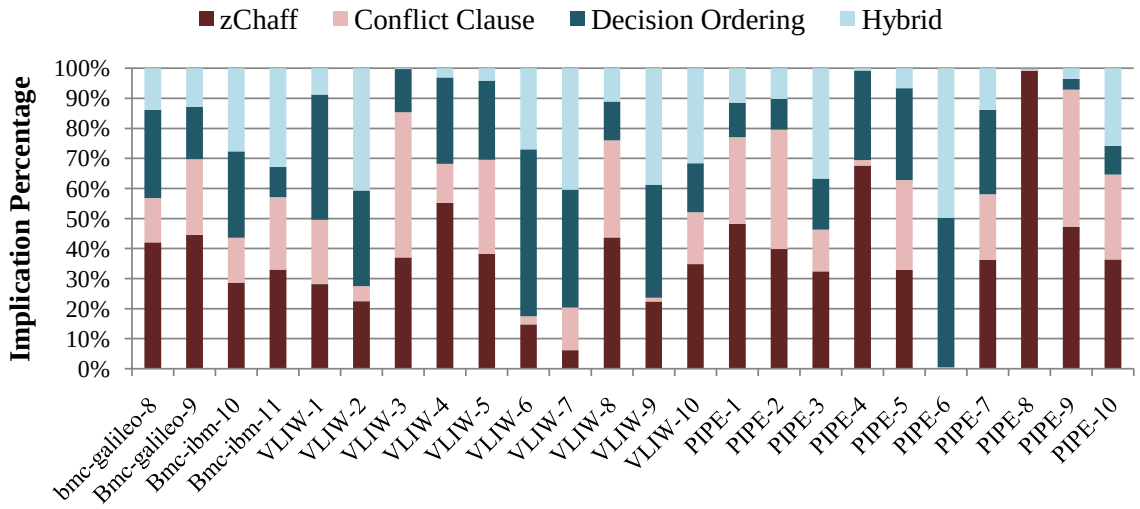


Figure 5-9. Implication statistics using various intra-learnings

5.4.2 Inter-Property Learning

5.4.2.1 A MIPS Processor

The MIPS processor design is based on the example described in Section 2.3.2. We applied our methodology to generate the required directed tests for four pipeline paths in the execute stage (ALU, FADD, MUL and DIV).

Due to the similarity, we cluster the properties of each path together to share the learning. There are 16 properties divided into 4 clusters. Each cluster has a base property. Table 5-2 shows the results. The first column indicates the component under test. The second column shows the properties used for test generation. The third column gives the test generation time using zChaff directly. The fourth column shows the result by forwarding conflict clauses among properties. It has three sub-columns. Since the conflict clauses forwarding based method needs to explore the common clauses, we need to figure out the intersection between SAT instances. Therefore the first sub-column gives the intersection time. The second sub-column gives the checking time under the learning of conflict clauses. The third sub-column gives the speedup over zChaff (speedup = $\frac{zChaff\ Time}{Intersection\ Time\ +\ Checking\ Time}$). The fifth column gives the test generation result using decision ordering based learnings. It has two sub-columns: i) test generation time, and ii)

speedup over zChaff. The last column shows the result which uses both conflict clauses and decision ordering based learnings.

For the base property of each cluster, we adopt the intra-property learning techniques. Since the base property is a major bottleneck of the clustering based methods described in Chapter 4, the test generation time reduction for the base property can drastically increase the overall performance. In Table 5-2, we also give the summary for each property cluster. We found that the hybrid method needs less time during the test generation. However, since the conflict clause forwarding needs to consider the SAT instance intersection, the overall performance of hybrid method is worse than the decision ordering based method. In general, the decision ordering based method can achieve the best performance. In this case study of four clusters with four property in each cluster, we can achieve 4-6 times improvement.

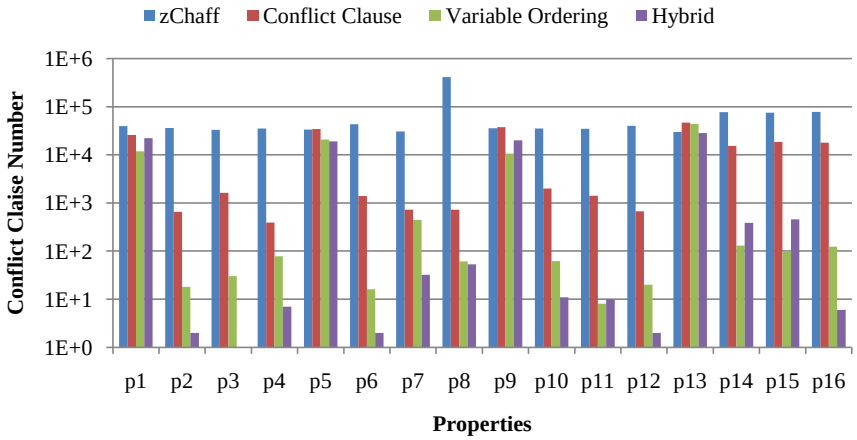


Figure 5-10. Conflict statistics for MIPS processor

During the SAT searching, the number of conflict clauses and number of implications strongly indicate the searching time. Figure 5-10 illustrates the conflict clause generation for each property during the search using different methods. Figure 5-11 shows the corresponding implication numbers. It can be seen that, by using our method, the number of conflict clauses and implications can be reduced drastically by several orders-of-magnitude, which results in significant improvement in test generation time.

Table 5-2. Test generation result for MIPS processor

MIPS Component	Prop. (Tests)	zChaff [74] Time (s)	Conflict Clause			Decision Ordering		Hybrid		
			Inter. (s)	Time (s)	Speedup	Time (s)	Speedup	Inter. (s)	Time (s)	Speedup
ALU Unit	p_1 [1]	19.78	0	11.9	1.66	13.08	1.51	0	9.36	2.11
	p_2	16.55	2.49	0.87	4.93	0.13	127.31	2.49	0.11	7.84
	p_3	15.41	2.08	1.82	3.95	0.15	102.73	2.08	0.11	6.45
	p_4	16.21	2.66	0.54	5.07	0.18	90.06	2.66	0.12	5.69
Summary	all	67.95	22.36		3.04	13.54	**5.02**	16.71		4.07
DIV Unit	p_5 [1]	15.21	0	16.14	0.94	16.09	0.95	0	8.34	1.82
	p_6	19.83	2.77	1.84	4.30	0.12	165.25	2.77	0.11	9.40
	p_7	13.74	2.79	0.98	3.64	0.49	28.04	2.79	0.15	5.56
	p_8	13.24	2.84	0.91	3.53	0.14	94.57	2.84	0.18	4.66
Summary	all	62.02	28.27		2.19	16.84	3.68	15.76		**3.94**
FADD Unit	p_9 [1]	16.01	0	18.00	0.89	11.59	1.38	0	9.33	1.72
	p_{10}	15.38	2.61	2.60	2.95	0.16	96.13	2.61	0.12	5.01
	p_{11}	15.63	2.08	1.80	4.03	0.12	130.25	2.08	0.12	6.65
	p_{12}	18.37	2.88	0.92	4.83	0.12	153.08	2.88	0.12	7.09
Summary	all	65.39	30.89		2.12	11.99	**5.45**	17.34		3.77
MUL Unit	p_{13} [1]	50.90	0	38.9	1.31	31.88	1.60	0	26.18	1.94
	p_{14}	51.27	3.35	13.14	3.11	0.29	176.79	3.35	0.66	15.40
	p_{15}	47.85	3.14	15.06	2.63	0.22	217.50	3.14	0.61	12.24
	p_{16}	53.44	2.89	14.59	3.06	0.25	213.76	2.89	0.15	16.75
Summary	all	203.46	91.07		2.23	32.64	**6.23**	36.61		5.56

[1] Base property

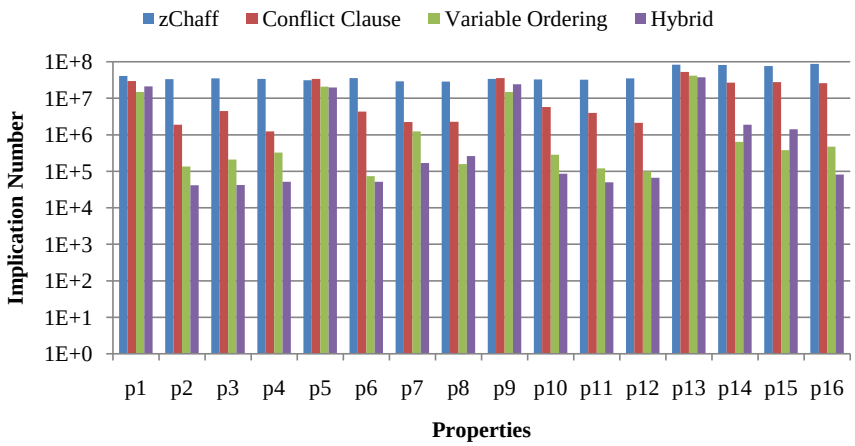


Figure 5-11. Implication statistics for MIPS processor

It is important to note that the hybrid method can achieve least number of conflicts and implications, which justifies our discussion in Section 5.2.4.

5.4.2.2 A Stock Exchange System

The formal NuSMV description of the on-line stock exchange system (OSES) is derived from its UML activity diagram specification described in Section 2.3.5 in Chapter 2. A path in the UML activity diagram indicates a stock transaction flow. There are a total of 49 properties generated based on path coverage criteria. According to their similarity, we group them into nine clusters.

Table 5-3. Test generation result for stock exchange system

Cluster	Size	zChaff [74] (s)	Conflict Clause (s)	Decision Ordering (s)	Hybrid (s)	Max speedup
C_1	3	13.63	11.93	8.55	10.71	1.59
C_2	4	26.35	35.37	3.99	7.75	6.60
C_3	8	463.54	183.06	41.24	50.43	11.24
C_4	4	3.36	5.01	1.49	5.56	2.26
C_5	4	66.59	40.47	6.38	11.30	10.44
C_6	8	343.88	270.48	12.28	23.33	28.00
C_7	2	17.81	6.73	7.03	6.20	2.87
C_8	8	666.61	343.94	51.80	71.19	12.87
C_9	8	208.50	101.91	34.99	34.83	5.99
Average	-	201.14	110.99	18.64	24.59	10.79

Table 5-3 shows the test generation results involving all the 9 clusters. The first column indicates the clusters. The second column indicates the size of each cluster

(number of properties). The third column presents the test generation time (including base property) using zChaff. The fourth column gives the result using conflict clause based inter- and intra- property learnings. The fifth column presents the result using decision ordering based inter- and intra- property learnings. The sixth column indicates the test generation time using both learnings (i.e., hybrid method). The last column indicates the maximum speedup using our heuristic methods. In this case study, our approach can produce an average of 10.79 times overall improvement in test generation time compared to zChaff. It is important to note that the decision ordering method can achieve the best performance, which is consistent with the result obtained in Section 5.4.2.1.

5.5 Summary

To address the complexity of test generation using SAT-based BMC, this chapter presented a novel methodology which explores the intra-property learnings within a SAT instance and inter-property learnings between similar SAT instances. All these learnings are based on decision ordering heuristics as well as conflict clause forwarding techniques. To the best of our knowledge, our work is the first attempt to share the decision ordering learnings on different parts of a SAT instance as well as across multiple properties. By exploiting the commonalities during the search of satisfiable assignments, the test generation time of a single property as well as a set of similar properties can be reduced. The experimental results using both hardware and software designs demonstrated the effectiveness of our method. Our studies show that hybrid learning is more profitable for solving one SAT instance, whereas decision-ordering based learning is more beneficial for solving a set of similar SAT instances.

CHAPTER 6
EFFICIENT PROPERTY DECOMPOSITION TECHNIQUES

Checking the first (base) property is a major bottleneck during the test generation using clustering and learning techniques, since the base property can not actively obtain learnings from others to improve its test generation time. Especially when checking a large design with complex properties (i.e., properties with large cone of influence or deep bounds), BMC based methods are very costly since large SAT instances indicate long SAT search time.

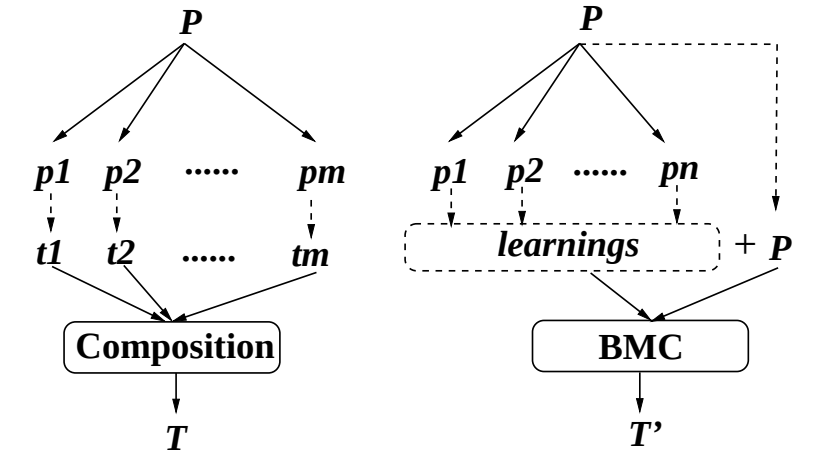


Figure 6-1. Two property decomposition techniques

To address this problem, Koo et al. proposed a property decomposition technique [47] as shown in Figure 6-1a. The basic idea is to decompose a complex property into several simple sub-properties, and then compose the tests corresponding to sub-properties to derive a test for the original property. Since the test generation time of sub-properties is typically several orders of magnitude smaller than the original property, the state space explosion problem can be avoided in many scenarios. However, the composition of tests of sub-properties is a major bottleneck in this method since it is hard to automate. The inevitable human intervention and expert knowledge is required during the test composition. In many cases, it may not be possible to obtain the required counterexample by composing partial (local) counterexamples. As an alternative, in this chapter, we

propose a learning-oriented decomposition technique shown in Figure 6-1b which can be fully automated. Unlike the test-oriented method in [47], our approach is based on the learned knowledge (i.e., decision ordering) during the test generation of decomposed profitable sub-properties. Such learnings can be used to drastically accelerate the original property falsification. Therefore the overall test generation effort can be significantly reduced. Our method makes three important contributions: i) it proposes a method that can spatially or temporally decompose a complex property into several simple but profitable sub-properties; ii) it proposes an approach that can derive learnings from the decomposed sub-properties; and iii) it proposes a method that can guide the complex property checking using derived learnings.

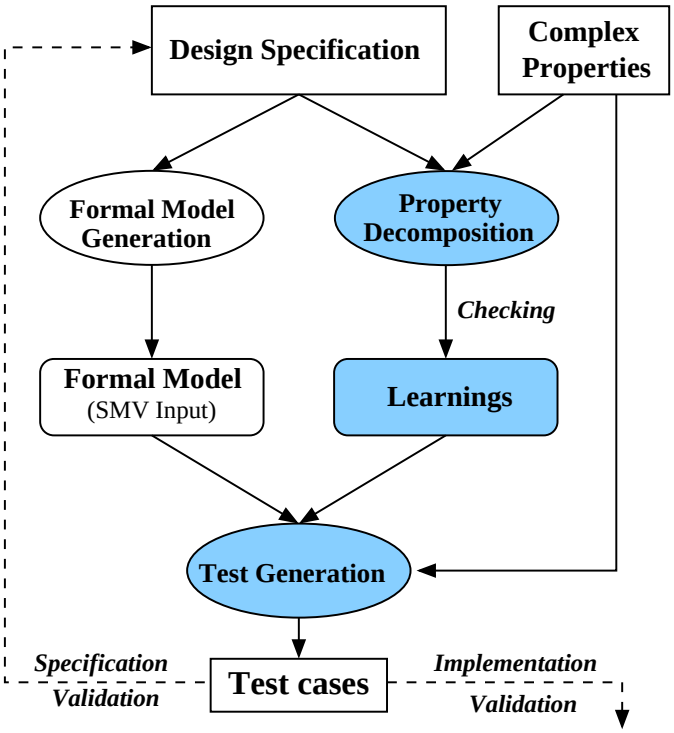


Figure 6-2. Our test generation framework

Figure 6-2 shows our test generation framework. The inputs to this framework are the design specification and required properties. To reduce this complexity, there are three important steps (three shaded boxes in the figure). First, we propose two novel property decomposition techniques which can significantly reduce the complexity during property

falsification. Next, by checking the selected profitable sub-properties, we can collect useful learnings for the original property checking. Finally, the learned knowledge can be utilized as a decision ordering heuristic to avoid the unnecessary conflicts during the test generation. Therefore, the test generation time can be drastically reduced.

The rest of the chapter is organized as follows. Section 6.1 proposes two novel property decomposition methods based on learning techniques. Section 6.2 presents the decision ordering based learning techniques for original property checking. Section 6.3 describes how to use the learned knowledge from the decomposed properties for test generation. Section 6.4 shows an example using our decomposition techniques. Section 6.5 presents case studies using both hardware and software designs. Finally, Section 6.6 summarizes the chapter.

6.1 Learning-Oriented Property Decomposition

This section first discusses the potential learnings of the properties for test generation. Next, we propose our spatial and temporal decomposition techniques.

6.1.1 Potential Learnings for Complex Properties

During test generation using BMC based methods, there are two kinds of complex properties which are often encountered: i) properties which describe complex scenarios involving multiple components of the design; and ii) properties which indicate events with long delay. Both cases will result in large SAT instances because of the corresponding large Cone of Influence (COI) and large bounds. Therefore it is necessary to explore learnings to reduce the complexity during the test generation.

For a complex system level property which describes interactions between different components, it can be partitioned into multiple component level sub-formulas. As an example shown in Figure 6-3, a system level property P can be broken into 3 component level sub-properties P_1, P_2 and P_3 with different COI. When checking a sub-property such as P_1 with a small COI, it usually needs much less time and space than that of checking the complex property P. The knowledge learned during checking P_1 can be used

for test generation of the property P. In Section 6.1.2, we propose a spatial property decomposition method to explore such learnings.

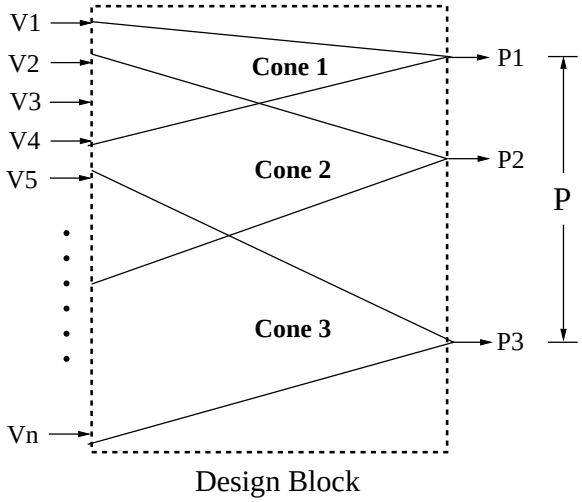


Figure 6-3. The COI of a design block

Transactions are widely used to describe SoC system level behaviors. A transaction is a sequence of strongly relevant *events*. As an example shown in Figure 6-4, there are 3 transactions, and each transaction has two events. We classify the relation between these events in two categories. The *cause effect* relation (marked by $\Rightarrow$) defines the relation of intra-transaction events. For example, in transaction $T1$, if $e1$ happens, then $e2$ should happen in future. The *happen before* relation (marked by $\prec$) specifies the relation of inter-transaction events. It indicates which events happen before other events. For example, $e4 \prec e5$ means $e4$ happens before $e5$.

During the test generation for transactions, we specify a negated safety property to indicate the occurrence of event e in the form of $\sim F(e)$. Generally, if an event happens with a long delay, BMC will unroll the design many times which will drastically increase the checking complexity. According to the definition, the "$\Rightarrow$" relation can be used to derive helpful learnings. For example in Figure 6-4, let property $P_1 =\sim F(e_1)$ and property $P_2 =\sim F(e_2)$. Since $e_1 \Rightarrow e_2$ implies $F(e_1) \rightarrow F(e_2)$, i.e., $\sim P_1 \rightarrow\sim P_2$, it shows that the $P1$'s counterexample will be helpful for deriving $P2$'s counterexample. Such information can be used as a learning. The "$\prec$" relation also can be used to indicate

the learning information. Assuming $e_4 \prec e_5$, the counterexample of $\sim F(e_4)$ is shorter than the counterexample of $\sim F(e_5)$. However, by our observation, counterexample of $\sim F(e_4)$ may have a large overlap of variable assignments with the counterexample of $\sim F(e_5)$. Therefore the learning from $\sim F(e_4)$ can benefit the test generation of $\sim F(e_5)$. In Section 6.1.3, we propose a temporal property decomposition method to explore such learnings.

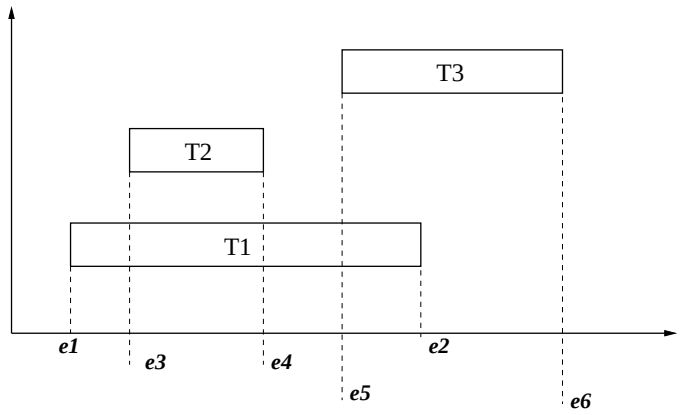


Figure 6-4. A functional scenario with three transactions

6.1.2 Spatial Property Decomposition

A complex false safety property can be decomposed into a set of sub-properties with equivalent semantics. If the partial counterexamples generated by the sub-properties can be refined to guide the complex property falsification, the original property is *spatially decomposable*.

Definition 8. *Let P be a false safety property. P is spatially decomposable in the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ or in the form $p_1 \vee p_2 \vee \ldots \vee p_n$ if all the following conditions are satisfied.*

- *If the decomposed properties are in the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n$, then at least one property p_i ($1 \leq i \leq n$) has a counterexample. In this case, the bound of P is the minimum bound of p_i which has a counterexample.*

- *If the decomposed properties are in the form $p_1 \vee p_2 \vee \ldots \vee p_n$, then each property p_i ($1 \leq i \leq n$) has a counterexample. In this case, the bound of P is the maximum bound of all decomposed properties.*

- *The counterexamples generated from properties p_i ($1 \leq i \leq n$) can guide the test generation for property P.* ■

According to Definition 13, the following rules can be used for complex property decomposition.

$$\begin{aligned} &\sim X(p \vee q) \equiv \sim X(p) \wedge \sim X(q) \\ &\sim X(p \wedge q) \equiv \sim X(p) \vee \sim X(q) \\ &\sim F(p \vee q) \equiv \sim F(p) \wedge \sim F(q) \end{aligned} \tag{6–1}$$

The false property in the form of $\sim F(p \wedge q)$ and $\sim F(p \rightarrow q)$ cannot be directly decomposed into conjunctive or disjunctive form. However, by introducing a synchronization clock clk, they can be spatially decomposed. It is important to note that the value of the clk indicates the bound of the false property. The Equation (6–2) shows that the counterexample of $\sim F(p \wedge q \wedge clk = k)$ can be refined by the counterexamples of $\sim F(p \wedge clk = k)$ and $\sim F(q \wedge clk = k)$.

$$\begin{aligned} &\sim F(p \wedge q \wedge clk = k) \equiv \sim F(p \wedge clk = k) \vee \sim F(q \wedge clk = k) \\ &\text{where } \sim F(p \wedge q \wedge clk = k) \text{ is false.} \end{aligned} \tag{6–2}$$

For a false property in the form $F(p \rightarrow q)$, p describes the pre-condition, and q indicates the post-condition. When the property $G(\sim p)$ holds, $F(p \rightarrow q)$ will be vacuously true, and the checking of $\sim F(p \rightarrow q)$ will report a counterexample without satisfying the precondition p. This counterexample may not match the original intention. In Equation (6–3), we only consider the case where the pre-condition p is satisfied.

$$\begin{aligned} &\sim F(p \rightarrow q \wedge clk = k) \equiv \sim F(p \wedge clk = k) \vee \sim F(q \wedge clk = k) \\ &\text{where } \sim F(p \rightarrow q \wedge clk = k) \text{ and } \sim F(p \wedge clk = k) \text{ are false.} \end{aligned} \tag{6–3}$$

Based on the rules presented in Equation (6–1)-Equation (6–3), a system level property can be decomposed in the form of $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ or $p_1 \vee p_2 \vee \ldots \vee p_n$. In fact, if the complex property can be decomposed in the form $p_1 \wedge p_2 \wedge \ldots \wedge p_n$, it is not necessary to

use the learning information. We just need to sort the property p_i $(1 < i \leq n)$ according to their increasing bounds, and check the sub-properties from the small bounds to large bounds. The counterexample of first falsified property can be used as a counterexample for the complex property.

When checking a complex property in the form of $p_1 \vee p_2 \vee \ldots \vee p_n$, it is not necessary to check all its sub-properties. Because the bounds of sub-properties are the same as the complex property, if the COI of a sub-property is similar to the complex property, the test generation complexity of such sub-property will be similar to the complex property. In this case, it is not economical to use learning. Therefore we need to figure out sub-properties with small COI from the complex property.

$$p_i \vee p_j \vee p_k = (p_i \vee p_k) \vee p_j \tag{6–4}$$

According to the commutative law, for a complex property, we can classify its atomic sub-properties into several clusters. For example, in Equation (6–4), p_i and p_k are clustered together, and p_j belongs to another cluster. For each cluster, we generate a *refined property* which represent all the atomic sub-properties in the cluster to derive the learning. Based on our experience, the following clustering rules work well for most of the time.

Structural similarity: In each cluster, all the variables in the sub-formulas should come from the same component (e.g., fetch module in a processor design).

Functional similarity: In each cluster, all the sub-formulas should describe the related functional scenarios (e.g., fetching instructions and/or data).

Algorithm 9 presents our spatial decomposition method which can derive a set of refined sub-properties with small COI for learning. The inputs of the algorithm are a design model D and a complex property P in disjunctive form. Step 1 initializes the *SD_props* with an empty set. Step 2 tunes sub-properties' order according to the commutative law and clusters sub-properties using the similarity rules. Step 3 selects the

i_{th} cluster. If the COI of such cluster is smaller than $\frac{k}{n}$ of P's COI, step 4 will generate a new refined property $newP$ for the i_{th} cluster. Step 5 adds $newP$ to SD_props. The refined property $newP$ for learning represents a cluster of sub-properties as shown in step 3. Finally this algorithm will return a set of refined sub-properties for deriving learnings (described in Section 6.2). Since the COI of a refined property in SD_props is small, its test generation time will be much smaller than that of the original complex property. It is important to note that this algorithm may return an empty set which means the property cannot be spatially decomposed.

Algorithm 9: Spatial Decomposition

Input: i) The design model, D

ii) A property P in the form $p_1 \vee p_2 \vee \ldots \vee p_n$

Output: A set of refined sub-properties for learning, SD_props

1. $SD\_props = \{\}$;

2. $(cluster_1, \ldots, cluster_m) = clustering(P, modular/functional)$;

for *i is from 1 to m* **do**

  3. $cluster\_i = \{prop_1, \ldots, prop_k\}$;

  if $COI(cluster_i) \leq \frac{k}{n} COI(P)$ **then**

    4. generate a refined property $newP$ for the $cluster_i$;

    5. $SD\_props = SD\_props \bigcup newP$;

  end

end

return SD_props;

6.1.3 Temporal Property Decomposition

Temporal property decomposition tries to eclipse the bound effect. The basic idea of temporal decomposition is to deduce a long bound property from a sequence of short bound properties. For example, $P1$, $P2$, $P3$ and $P4$ ($P4 = P$) are properties indicating four different stages of property P. The bound of them are $K1$, $K2$, $K3$ and $K4$, respectively, and $K1 < K2 < K3 < K4$. Because $P1$'s counterexample is similar to the prefix of the $P2$'s counterexample, $P1$'s counterexample contains rich knowledge

that can be used when checking $P2$. Similarly, during the property checking, $P3$ can benefit from $P2$ and $P4$ can benefit from $P3$. Therefore the knowledge learned from lower bound properties can be reused by the larger bound property. Such learning can avoid some unnecessary random SAT searching and can quickly obtain the counterexample for property P.

Definition 9. *Let P be a false safety property, and P is temporally decomposable if all the following conditions are satisfied.*

- *P can be divided into false properties p_1, p_2, ... and p_n ($P = p_n$) with increasing bounds.*
- *$\sim p_i \rightarrow \sim p_{i+1}$ $(1 \leq i \leq k_n - 1)$, which indicates the counterexample generated from properties p_i can guide the test generation for property p_{i+1}.* ■

If the counterexamples of lower bound property can be used to reason about P, the property P is *temporally decomposable*. In temporal decomposition, finding the implication relation ("$\rightarrow$") between properties is a key process. In our framework, we construct such implication relation by exploring the order between events, i.e. "$\Rightarrow$" or "$\prec$".

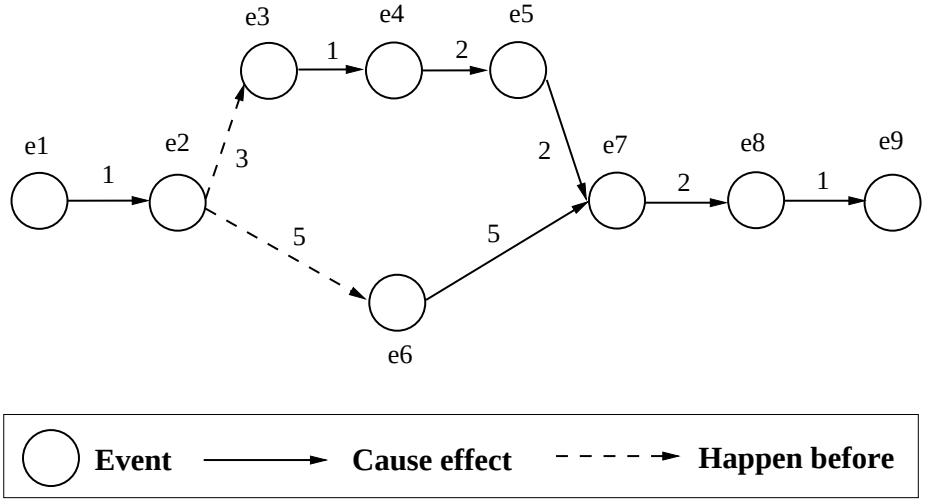


Figure 6-5. A DAG of event relation

When checking a large bound property for a transaction, there may be many events along the path to the target events. Checking all these events to obtain learnings is time-consuming. For example, assuming that we want to check the property $\sim F(e_9)$,

the relation between events is described using a directed acyclic graph (DAG) shown in Figure 6-5. Each node indicates an event, and each directed edge indicates the relation of "$\Rightarrow$" or "$\prec$", and each edge is associated with the delay between events. In this DAG, there are 8 events that happen before $e9$. However, it is not necessary to check all of them. Since the branch nodes of a DAG contain the critical variable assignment information, in our decomposition method, we only consider the events which determine the branches along the path from initial state $e1$ to the target state $e9$.

Algorithm 10 describes how to obtain a sequence of properties based on temporal decomposition. It accepts an event DAG with the initial and target events as inputs. Step 1 uses Dijkstra's algorithm [24] to find a shortest *path*. Step 2 initializes the sequence *TD_props* with a property for the initial event. Step 3 and 4 select the *branch events* and append their correspond properties to the *TD_props*. Finally the algorithm reports the property sequence for deriving learnings. By using this algorithm, ($\sim F(e1)$, $\sim F(e3)$, $\sim F(e7)$) is a property sequence from the temporal decomposition in Figure 6-5.

Algorithm 10: Temporal Decomposition

Input: i) An event DAG, D

ii) Initial event *src*, target event *dest*

Output: A property sequence *TD_props*

1. *path* = Dijkstra(D, *src*, *dest*) to find the shortest delay path;

2. *TD_props* = (property for *src*);

for *i is from 2 to len (number of events in path)* **do**

  3. $(e_{i-1}, e_i) = (i-1)_{th}$ edge of *path*;

  if *out_degree*(e_{i-1}) + *in_degree*$(e_i) > 2$ **then**

    4. Append the property for e_i to *TD_events*;

  end

end

return *TD_props*;

6.2 Decision Ordering Based Learning Techniques

SAT based model checking encodes a property checking problem into a SAT instance (a Boolean formula). A counterexample of the property is a satisfiable variable assignment for this formula. Although the variable assignment of counterexamples derived from the decomposed sub-properties may not satisfy the SAT instance of the complex property, it has a large overlap with the complex property on the variable assignment. Such information can be used as a learning to bias the decision ordering when checking the complex property.

During the SAT search, decision ordering plays an important role to quickly find a satisfiable assignment. The learning approach in this chapter is motivated by the work proposed in Chapter 5. It is based on Variable State Independent Decaying Sum (VSIDS) method [63]. A major difference is that our method incorporates the statistics of decomposed properties. Since different sub-properties have different bounds, we consider such information in our heuristics.

Let *bounds* be an array which stores the bound of k sub-properties. Because in spatial method the decomposed sub-properties may be independent, the learning between sub-properties is not significant. So we set $bound[i] = 1 (1 \leq i \leq k)$. However for temporal decomposition, the *vstat* information of lower bound properties can further benefit the larger bound property checking. Moreover the larger bound sub-property is closer to the final properties than smaller bound sub-properties. Therefore, for temporal decomposition based method, the sub-properties is sorted according to the increasing *bound* and $bound[i]$ indicates the bound of i_{th} property. Let $vstat[sz][2]$ (sz is the variable number of the complex property) be a 2-dimensional array to record the statistics of variable assignments. Initially, $vstat[i][0] = vstat[i][1] = 0$ $(0 < i \leq sz)$. *vstat* will be updated after checking each sub-property. When checking the sub-property p_j, if variable v_i is evaluated and its value in the counterexample is 0 (false), $vstat[i][0]$ will be increased by $bounds[j]$; otherwise if $v_i = 1$ (true), $vstat[i][1]$ will be increased by $bounds[j]$.

Assuming l_i is a literal of v_i (v_i has two literals, v_i and v_i'), we use $score(l_i)$ to indicate its decision ordering. Initially, $score(l_i)$ is equal to the literal count of l_i. However, at the beginning of SAT searching and periodic score decaying, the literal score will be recalculated. Let

$$bias = \frac{MAX(vstat(v_i), vstat(v_i')) + 1}{MIN(vstat(v_i), vstat(v_i')) + 1}$$

indicate the variable assignment variance.

$$score(l_i) = \begin{cases} max(v_i) * bias & (vstat[i][1] > vstat[i][0] \& l_i = v_i) \\ & or(vstat[i][1] < vstat[i][0] \& l_i = v_i') \\ score(l_i) & otherwise \end{cases}$$

The new literal score will be updated using the above formula where $max(v_i) = MAX(score(v_i), score(v_i')) + 1$.

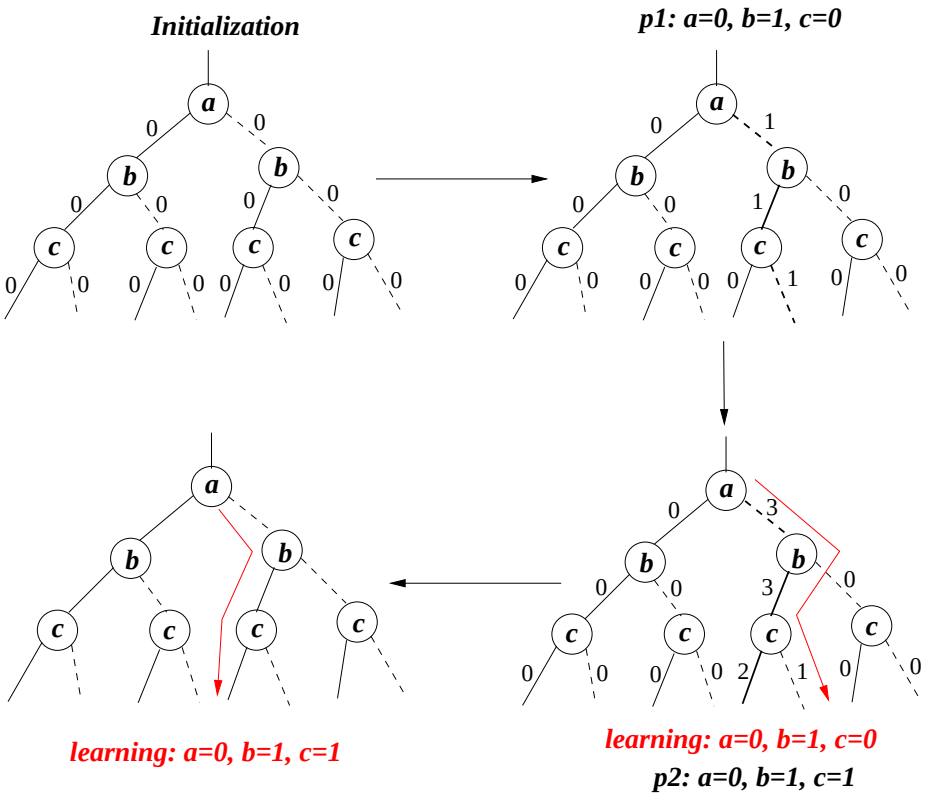


Figure 6-6. Learning statistics applied on decision trees

Figure 6-6 shows an example of temporal decomposition using our heuristic. The complex property P is decomposed into three properties p_1, p_2 and $p_3(= P)$ with bound 1, 2 and 3 respectively and we assume that we always check the variables in the order of a, b, c. Initially, when checking p_1, there is no learning information. However, after checking p_1, we can predict the decision ordering for p_2 based on the collected $vstat$ information from p_1. Also we can predict the decision ordering of $p_3(= P)$ from the $vstat$ of p_1 and p_2. When checking P, the content of $vstat$ indicates that variables a is more likely to be 0, b and c are more likely to be 1.

6.3 Test Generation using Our Methods

In this chapter, we assume that the bound of complex property and decomposed properties can be pre-determined. Determination of bound is hard in general. However, for directed test generation, the bound can be determined by exploiting the structure of the design. An example of bound determination is presented in Section 6.4.

Algorithm 11: Test Generation based on Property Decomposition

Input: i) Formal model of the design, D

ii) Decomposed properties *props* and satisfiable *bounds*

iii) The complex property P, with the satisfiable bound $bound_p$

Output: A test $test_P$ for P

1. $CNFs = BMC(D, props, bounds)$;

2. $(CNF_1, \ldots, CNF_n)$ = sort $CNFs$ using increasing file size;

3. Initialize $vstat$;

for *i is from 1 to the n* **do**

  4. $test_i$ = SAT(CNF_i, $vstat$);

  5. $Update(vstat, test_i, bounds[i])$;

end

6. Generate $CNF = BMC(D,\ P,\ bound_P)$;

7. $test_P$ = SAT(CNF, $vstat$);

return $test_P$;

Algorithm 11 describes our test generation methodology. The inputs of the algorithm are a formal model of the design, a set of decomposed properties *props* and their satisfiable bounds *bounds*, and the complex property P with its satisfiable bound $bound_p$. Step 1 generates CNF files in the DIMACS format [74] for each decomposed property in *props*. Step 2 sorts the CNFs by their DIMACS file size. Step 3 initializes *vstat* which is used to keep statistics of the variable assignments for decomposed sub-properties. Then for each decomposed sub-property, we collect its counterexample assignments from step 4 to step 5. For each iteration, we need to update *vstat* statistics. In step 6 and step 7, the complex property P is checked using the decision ordering derived from the decomposed sub-properties. Finally, the algorithm reports a test for the complex property P.

6.4 An Illustrative Example

This section presents an example of how to use decomposition methods on a design illustrated in Section 2.3.2. Assume that we want to check a complex scenario that the units $MUL5$ and $FADD4$ will be active at the same time. We generate the property P which is a negation of the desired behavior as follows. The remainder of this section will solve it using spatial and temporal decomposition methods.

```
/* Original complex property P */

P: ~ F(mul5_active=1 & fadd3_active=1)
```

6.4.1 Spatial Decomposition

In the MIPS design, each functional unit has a delay of one clock cycle. To trigger the functional unit $MUL5$, we need at least 7 clock cycles (there are 7 units along the path $Fectch \rightarrow Decode \rightarrow \ldots \rightarrow$ MUL5). Similarly, to trigger the functional unit $FADD3$, we need at least 5 clock cycles. Plus one clock cycle for initialization, we need 8 clock cycles for triggering this interaction. Thus the bound of this property is 8. According to Equation (6–2) and Algorithm 9, property P can be spatially decomposed into two sub-properties as follows, assuming the COI of $P1$ and $P2$ are both smaller than half of COI of P.

```
/* Modified original complex property P' */
P': ~ F(mul5_active=1 & fadd3_active=1 & clk=8)
/* Spatially decomposed properties */
P1: ~ F(mul5_active=1 & clk=8)
P2: ~ F(fadd3_active=1 & clk=8)
```

When checking $P1$ and $P2$ individually, we can get the following two counterexamples.

Counterexamples for P1 and P2

Cycles	P1's Instructions	P2's Instructions
1	NOP	NOP
2	MUL R2, R2, R0	NOP
3	NOP	NOP
4	NOP	FADD R1, R1, R0
5	NOP	NOP
6	NOP	NOP
7	NOP	NOP
8	NOP	NOP

However, according to Algorithm 11, the test generation for $P2$ is under the guidance of $P1$'s result. Thus, the counterexample of $P2$ guided by $P1$ contains $P1$'s partial behavior (see clock cycle 2 below). So the score of literals which have repetitive occurrences is enhanced.

Counterexample for P2 guided by P1

Cycles	P2's Instructions
1	NOP
2	MUL R2, R2, R0
3	NOP
4	FADD R1, R1, R0
5	NOP
6	NOP
7	NOP
8	NOP

The statistics saved in *vstat* indicates an assignment which has a large overlap of the assignments with the real counterexample that can activate property P. Thus it can be used as the decision ordering learning to guide the property checking of P.

6.4.2 Temporal Decomposition

For temporal decomposition, we need to figure out the event implication relation first. Because we want to check the property $F(mul5\_active = 1 \ \& \ fadd3\_active = 1)$, the target event is $mul5\_active = 1 \ \& \ fadd3\_active = 1$. Figure 6-7 shows the implication for this event. There are 7 events in this graph, and e_7 is the target event.

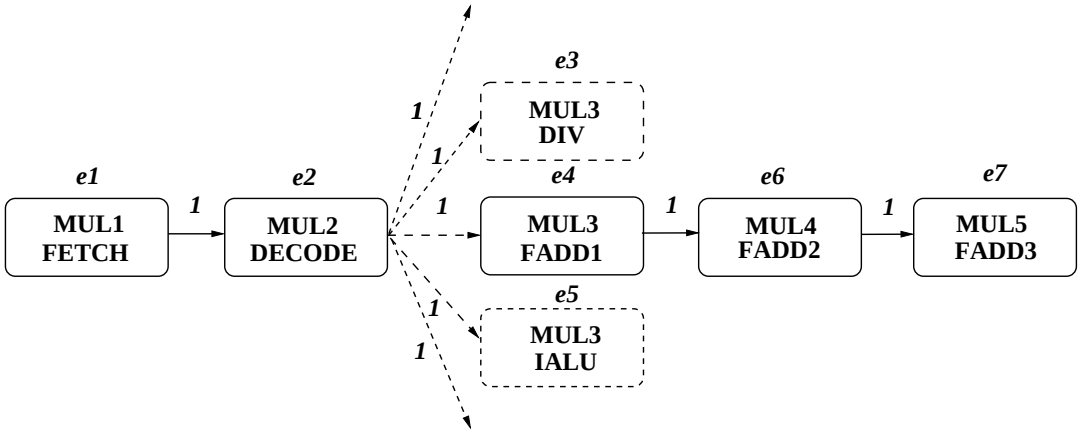


Figure 6-7. Event implication graph for property P

Assuming $e1$ is the initial event, from $e1$ to $e7$, there is only one path $e_1 \rightarrow e_2 \rightarrow e_4 \rightarrow e_6 \rightarrow e_7$. Along this path there is a branch node e_2. According to the Algorithm 2, we need to check two events e_1 and e_4 using following properties. By using our learning technique, during the test generation, *P_e4* can benefit from *P_e1*, and P can benefit from *P_e4*.

```
/* Spatially decomposed properties*/

P_e1: ~ F(fetch_active=1 & mul1_active=1)

P_e4: ~ F(mul3_active=1 & fadd1_active=1)
```

6.5 Experiments

This section presets two case studies: the VLIW implementation of the MIPS architecture (described in Section 4.5.1) and the stock exchange system (described in

Section 4.5.2). In our framework, we used NuSMV [27] to generate the CNF clauses (in DIMACS format) and integrated our proposed methods in the zChaff [74] SAT solver. The experimental results are obtained on a Linux PC using 2.0GHz Core 2 Duo CPU with 1 GB RAM.

6.5.1 A VLIW MIPS Processor

This section presents the experimental result using a five-stage pipelined MIPS processor design. The details of the design are illustrated in Section 2.3.2 and Section 6.4. Since the generated properties are in various complex formats, it is difficult to figure out the implication between events. Therefore in this case study, we only investigate the spatial decomposition based learnings.

Table 6-1. Test generation result for MIPS processor

Property (Tests)	zChaff [74] (sec)	Cluster #	Refinement #	Spatial (sec)	Speedup zChaff vs Spa.
Property format: $\sim F(p \vee q)$					
p_1	119.96	3	3	0.03	3999
p_2	56.22	2	2	0.03	1874
p_3	2.32	2	2	0.01	232
Property format: $\sim F(p \wedge q)$					
p_4	43.96	3	2	18.88	2.33
p_5	15.24	2	1	6.57	2.32
p_6	9.28	2	1	4.42	2.10
Property format: $\sim F(p \rightarrow q)$					
p_7	13.59	2	1	4.16	3.27
p_8	68.33	2	1	13.16	5.19
p_9	160.51	3	2	30.31	5.30

We select nine complex properties from the MIPS design. Table 6-1 shows the test generation results using our spatial decomposition method. The first column indicates the selected properties. The second column gives the test generation time using zChaff. The third and fourth columns present the number of sub-property clusters and the number of refined sub-properties for deriving learnings. The last two columns show test generation time using learnings and the improvement of our spatial decomposition based method over the method using zChaff. We cluster 9 properties into 3 groups (3 properties in each

group), and each group has a specific property format. For example, the first group can be decomposed as $p_1 \wedge p_2 \cdots \wedge p_n$. Thus the test generation can be done when finding a counterexample from a lower bound sub-property without any learnings. For the second and third groups, the properties can be decomposed in the form of $p_1 \vee p_2 \cdots \vee p_n$. Each sub-property are of the same bound. Therefore we need to cluster the sub-property according to the similarity rules presented in Section 6.1.2. Compared to the method without any learnings (column 2), our spatial decomposition based learning method can drastically reduce the test generation time.

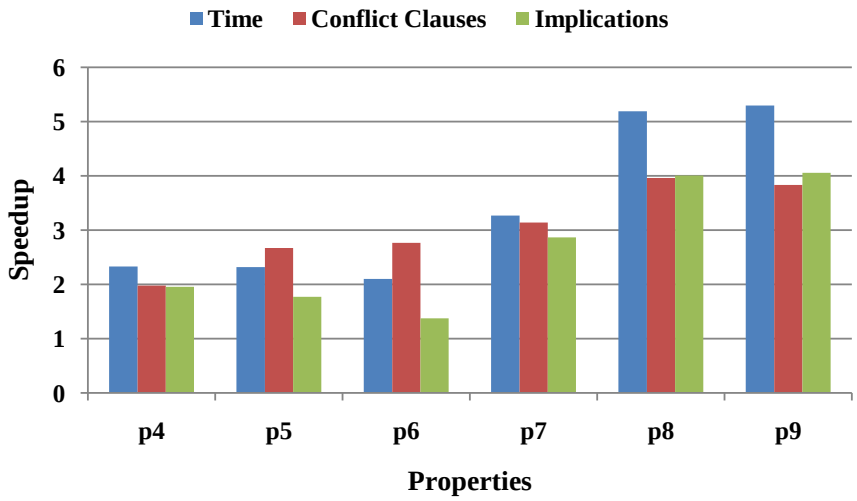


Figure 6-8. Property checking result for MIPS processor

During the SAT-based BMC falsification, conflict clause number and implication number are key factors which determine the test generation performance. Decision ordering learned from decomposed properties can efficiently avoid the conflicts when checking the complex property. Figure 6-8 shows the result of properties $p_4 - p_9$ presented in Table 6-1. It illustrates the performance improvement (spatial method over zChaff) using time, implication number and conflict clause number. It can be seen that, by using spatial method, the number of conflict clauses and implications can be reduced drastically by 2-4 times, which consistently results in significant improvement in test generation time (2-5 times).

6.5.2 A Stock Exchange System

The on-line stock exchange system (OSES) is a software (described in Section 2.3.5) which mainly deals with stock order transactions. We generate 18 complex properties to check the stock transactions. In the UML activity diagram, each transaction is indicated by a path which is a sequence of activities (events). The test generation for a transaction using only one complex property is time consuming. So we temporally decomposed the transaction into several stages which specify the branch activities along the path, and for each stage we create a sub-property.

Among the 18 complex properties, ten of them are time-consuming (more than 10 seconds without using our method). Table 6-2 shows the test generation results for these ten properties using temporal decomposition. The first column indicates the property. The second column indicates the test generation time using zChaff without any decomposition and learning techniques. The third column presents the bound of the complex property. The fourth column indicates the number of temporal sub-properties decomposed along the stock transaction flow. The last two columns indicate the test generation time (using temporal decomposition) and its speedup over zChaff. In this case study, our approach can produce around 3-60 times improvement compared to the method using zChaff.

Table 6-2. Test generation result for OSES

Property (Tests)	zChaff [74] (sec)	Bound	Decomposed #	Temporal (sec)	Speedup zChaff vs Temp.
p_1	25.99	8	3	0.78	33.32
p_2	48.99	10	4	2.69	18.21
p_3	39.67	11	5	3.45	11.50
p_4	247.26	11	5	22.46	11.01
p_5	160.73	11	5	15.68	10.25
p_6	97.54	11	4	1.56	62.53
p_7	31.39	10	4	12.31	2.55
p_8	161.74	11	4	12.62	12.82
p_9	142.91	10	4	17.57	8.13
p_{10}	33.77	10	4	1.76	19.19

6.6 Summary

To address the test generation complexity of a single complex property using SAT-based BMC, this chapter presented a novel method which combines the property decomposition and learning techniques. By decomposing a complex property spatially and temporally, we can get a set of sub-properties whose counterexamples can be used to predict the decision ordering for the complex property. Because of the learning from the simple sub-properties to the complex property, the overall test generation effort can be reduced. The case studies demonstrated the effectiveness of our method using both hardware and software designs that generated significant savings (2-60 times) in test generation time.

CHAPTER 7
REUSE OF VALIDATION EFFORT FOR ASSERTION-BASED EQUIVALENCE

For software designs, the difference between specification level tests and implementation level tests is small. Generally, the specification level tests can be automatically reused and applied on software implementations. Consequently, the consistency between different software designs can be checked. However, due to the significant difference in timing and other details, maintaining the functional equivalence between different hardware abstraction layers is a major challenge during the SoC design. In this chapter, we are focusing on checking the functional consistency between different hardware abstractions. In this chapter, we focus on reusing validation effort between TLM and RTL models.

Since there is no mature automatic TLM to RTL refinement tool and manual conversion is error-prone, various approaches are proposed to guide the TLM to RTL conversion. Simulation is a widely used method for functional validation. By using a transactor [6] between TLM and RTL designs for communication, the previously generated TLM tests can be exercised on the refined RTL implementations to check the functional correctness. However, due to substantial differences between TLM and RTL models, traditional simulation methods can not guarantee the functional equivalence. For black-box methods [66], simulation can not guarantee the bug propagation to the outputs. Similarly, for white-box simulation methods [66], the code coverage [32] and toggle coverage can not fully indicate the required functional coverage. This is due to the lack of functional observation mechanisms for traditional simulation based methods.

Assertion based validation (ABV) [23, 29] has been successfully applied in SoC validation to ensure the functional correctness. It not only increases the design observability based on simulation using ad-hoc tests, but also takes advantage of more emerging formal methods for improving the overall verification quality and results. As a functional observation point, an assertion can be instrumented into TLM or RTL designs to monitor the specified functional scenario. Currently, there are two most popular assertion

languages: Property Specification Language (PSL) [3] and System-Verilog Assertion (SVA) [37]. PSL is platform-independent and can be used in multi-layer designs. SVA is similar to PSL, nevertheless it is only customized for System-Verilog designs. In the context of ABV, a *property* is defined as a logic constraint description built on Boolean expressions, sequences and temporal operators, while *assertion* is defined as a directive to prove the correctness of the property. For simplicity, in this chapter we use the term assertion to indicate both assertion and property.

In this chapter, we propose a methodology to guarantee the functional equivalence between TLM and RTL models based on the observability of assertions. The basic idea is that in the TLM specification, if a test can exercise a specified functional scenario monitored by some assertion, then in the RTL implementation, the counterpart of the TLM test can also activate the counterpart of the TLM assertion. During the TLM-to-RTL functional equivalence checking, we need to addresses the following four issues:

- **How to determine a set of TLM assertions for observing all the functional scenarios?** We proposed several fault models which require that all the specified faults should be covered by the generated assertions.
- **How to activate a given TLM assertion?** We adopted the model checking falsification technique to derive tests for activating TLM assertions. For each assertion, we generate one test to activate it.
- **How to reuse TLM validation effort?** We developed the validation refinement rules which can convert TLM assertions and tests to their RTL counterparts.
- **How to use the correlation between TLM and RTL assertions for equivalence checking?** We proposed a method to verify the TLM-to-RTL equivalence based on the criteria of assertion coverage and assertion ordering.

Our proposed approach addresses the above challenges and makes two major contributions: i) develops a prototype tool for automatic TLM-to-RTL test and assertion refinement, and ii) proposes a method that uses the assertion observability for checking the functional equivalence between TLM and RTL models. Because our work is based on

the reuse of TLM validation effort, there is no extra cost (excludes defining the refinement rules) since it needs to be validated anyway. Furthermore, our method is fully automated and can be easily scaled for large designs.

The rest of this chapter is organized as follows. Section 7.1 presents related work on validation reuse and equivalence checking between TLM and RTL models. Section 7.2 proposes our equivalence checking framework based on validation reuse. Section 7.3 presents the experimental results. Finally, Section 7.4 summarizes the chapter.

7.1 Related Work

TLM is promising to enable early design space exploration and hardware/software co-simulation. Hsiung et al. [36] adopted SystemC TLM models to enable rapid exploration of different reconfigurable design alternatives. In [44], Kogel et al. presented a SystemC based methodology which provides sufficient performance, flexibility and cost efficiency as required by demanding applications. Shin et al. [80] proposed a method to automatically generate TLM models from virtual architecture models which can achieve significant productivity gains.

As a hybrid method based on both simulation and formal verification, ABV is acknowledged as a promising approach for functional validation in RTL level [1]. However, ABV is still a challenging domain in system level design. To address the issues when incorporating PSL within SystemC environments, Lahbib et al. [49] proposed an automated solution which can embed PSL assertions in a SystemC design. Based on static code analysis and genetic algorithms, Habibi et al. [33] presented an efficient method to optimize test generation in order to increase the assertion coverage. Ecker et al. [25] proposed a transaction level assertion framework using a new specialized language. In [73], Pierre described an efficient and tractable solution for verifying the PSL based properties of TLM design during the simulation. However, most researches are focused on implementing PSL assertions in SystemC framework, and none of them use assertions for checking the TLM-to-RTL functional equivalence.

Reusing the validation effort between abstraction levels can reduce overall validation effort. Assertions can be treated as constraints of system specifications. Therefore the assertion reuse can partially guarantee the consistency between different abstraction levels. In [42], Kasuya and Tesfaye presented a mechanism to construct and reuse temporal assertions in various TLM abstraction levels. As an alternative, test reuse can not only reduce the test generation and simulation time, but also enable the co-simulation between different abstraction levels. In [12], Bombieri et al. proposed a transactor-based dynamic verification method. By using transactors, the TLM testbenches can be reused during the TLM-RTL co-simulation. In [14], Bombieri et al. presented a formal definition of functional equivalence based on events order without timing information. However, they did not provide any implementation details for checking the proposed functional equivalence. Similar to our work, the research of incremental ABV methodology described in [13] uses various kinds of assertions to check the correctness of TLM-to-RTL refinement. However, since their work is based on transactors, it is required that RTL implementations should be ready before the co-simulation. Also, their work does not provide any methods about how to activate all the instrumented assertions. Therefore it is difficult to guarantee that the simulation can achieve the required assertion coverage quickly. Furthermore, their method applies assertions on TLM specifications only. It just monitors primary input and output signals without investigating RTL implementation details.

To the best of our knowledge, when checking refinement consistency and correctness, existing approaches focus on test/assertion coverage without considering more details such as the correlation between TLM and RTL assertions. Our approach is the first attempt to reuse the validation effort to enable assertion-based equivalence checking between TLM and RTL models.

Figure 7-1 shows the framework of our methodology. First by analyzing TLM specifications, the TLM assertions and tests can be automatically derived according to specified fault models. Next the refinement process translates the TLM assertions and

tests for RTL validation using our proposed mapping rules. The refined assertions will be instrumented in RTL implementations. The refined tests will be applied on the RTL implementations and the output of the tests and the activated assertions will be monitored by a RTL assertion checker. Finally, by comparing simulation traces recorded by TLM and RTL assertion checkers, the equivalence checker reports the results.

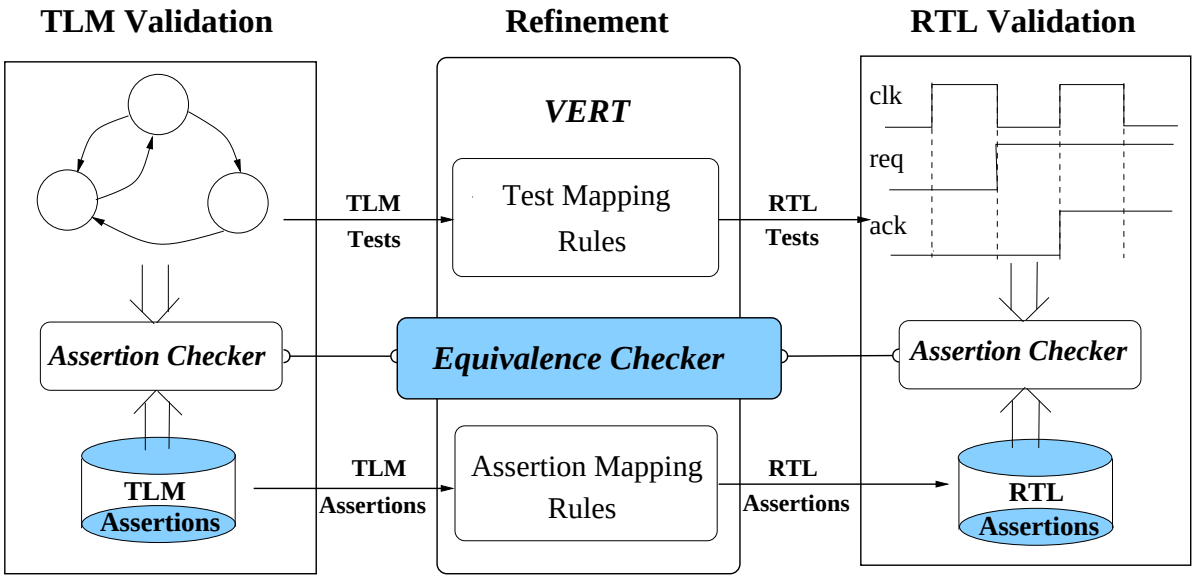


Figure 7-1. Our equivalence checking framework

7.2 A Framework for Checking TLM-to-RTL Functional Equivalence

Our methodology has three important steps: i) automatic validation of TLM specifications (i.e., TLM assertion/test generation), ii) validation effort refinement, and iii) assertion based equivalence checking. The following subsections discuss each of these steps in detail.

7.2.1 Automatic Transaction Level Validation

SystemC TLM emphasizes the functionality of the data transfers instead of actual implementation. Essentially a SystemC TLM design interconnects a set of processes using transactions (i.e., C++ function calls) for communication. Each process does the following tasks: receiving data, processing data and sending data. Due to various complex constructs in C++, extracting all such behavior to enable automated analysis and

validation is difficult. Furthermore, the underlying complex SystemC scheduler aggravates the modeling complexity. In fact, investigating the general features of SystemC TLM is not necessary for functional validation of TLM models. For TLM, the most important factors are the transaction data, the transaction flow and the transaction event order. So during our assertion/test generation process, these factors need to be considered. Other elements can be selectively abstracted.

7.2.1.1 Generation of TLM Assertions

Assertions are used to specify the required functional behaviors of a system. To investigate the equivalence between TLM and RTL models, we need to explore as many assertions as possible. In our method, we define a set of *fault models* to achieve a complete set of assertions. Each fault indicates a required "design behavior" which may be violated during the system design. For example, when validating a desired scenario described by a *sequence* p (*sequence* is a PSL term which indicates a sequential expression), we use the following PSL statement pairs to detect whether the sequence p will happen finally. The $Prop1\_1$ asserts that the sequence p must "*eventually*!" hold strongly during the simulation, and $Prop1\_2$ is used to record the assertion coverage during the simulation by using verification directive "cover".

```
Prop1_1: assert eventually! p;

Prop1_2: cover (p);
```

We consider the three TLM fault models which are described in Section 3.1.1.2. Transaction data fault model deals with the possible value assignment for each part of the transaction data. However, for property generation, due to the large size of value space, trying all possible values of a data is infeasible. By checking each bit of a variable (data bit fault) separately, the data content coverage can be partially guaranteed. The following is an example of a data fault.

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

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

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

Transaction event indicates the execution stage of a transaction or the interaction between processes. Therefore, during the equivalence checking, the order of events should be investigated. In our method, we consider various events in two categories: 1) events of procedure calls, such as *read* and *write*, and *put* and *get* operations; and 2) synchronization events, such as *wait* and *notify* operations. The following is an example of a read procedure call.

```
//The event a=A.read() can be activated.
assert eventually! {A==a};
cover  {A==a};
```

It is important to note that an assertion that is generated from the above three fault models activate a specific functional scenario. In our method it just acts like a functional check point to monitor the occurrence of a specific event instead of describing a complex scenario. The order of the assertion activations plays an important role and will be handled when verifying functional equivalence described in Section 7.2.4.

7.2.1.2 Generation of TLM Tests

Our equivalence checking approach is based on simulation, so we need to generate tests to cover all the assertions derived using the method proposed in Section 7.2.1.1.

Conventional methods use millions of random/constrained-random tests, however, it is difficult to exercise all the assertions in a reasonable time. As an alternative, directed tests are promising since they exploit the structural information and can converge to 100% assertion coverage quickly. However, most directed test generation methods need human intervention which is error-prone and costly. In our framework, we developed a tool which can enable automatic directed test generation. It is important to note that for random test based methods, we may require a large set of tests for each assertion. However, when using directed methods, we just need to derive one test for each assertion. Chapter 3 gives the details for TLM test generation.

7.2.2 Refinement of TLM Assertions and Tests

When TLM assertions and tests are ready, we need to refine them to RTL counterparts for reuse. A major challenge in the translation is how to bridge abstraction gap between TLM and RTL models. As we know, TLM design is significantly different from its RTL implementation in input/output port definition, internal structure and timing information. Thus for TLM-to-RTL validation refinement, it is necessary to provide such missing information which is also needed during the manual or automatic TLM-to-RTL synthesis.

In our framework shown in Figure 7-1, the *Validation Effort Reuse Tool* (VERT) is a major component which enables TLM-to-RTL refinement by specifying rules. The inputs of VERT are TLM assertions/tests as well as a Validation Refinement Specification (VRS) which contains the rules to guide the validation refinement. Generally a VRS contains three parts as follows.

- **Symbol Mapping** specifies the name and type mapping between TLM variables and RTL signals.
- **Assertion Refinement Rules** specify patterns and timing information for RTL assertions.
- **Test Refinement Rules** specify the interface protocols and timing information for RTL input stimulus.

The following subsections describe each part in details.

7.2.2.1 Symbol Mapping

In our prototype tool, we use SystemC for transaction level modeling and Verilog for RTL modeling. Due to the naming convention inconsistency between TLM specifications and RTL implementations, during the validation refinement, it is necessary to have a symbol table which specifies the name mappings. Each item in the symbol table defines the correspondence between TLM variables and RTL variables. Generally it provides the following information: i) name mapping, ii) date type mapping, and iii) bit mapping. The following is an example of symbol mapping.

```
SYMBOL_MAPPING
 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_SYMBOL_MAPPING
```

For each symbol mapping item, the left hand side is the RTL data declaration, and the right hand side is the bit mapping details from TLM data to RTL data. The VRS 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. For example, *parity* is a RTL data with 8 bits. It refers to the TLM variable *packet.parity*. The header is a RTL data whose most significant six bits corresponds 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 $(i+1)^{th}$ element *payload*[i] corresponds to the $(i+1)^{th}$ element of the TLM data *packet.payload*[i].

7.2.2.2 Assertion Refinement Rules

According to the definition in [3], a PSL or SVA assertion consists of four layers:

- **Boolean Layer** defines the Boolean expressions of signals which are evaluated in a single evaluation cycle.

- **Temporal Layer** describes assertions involving complex temporal relations between Boolean expressions. Temporal assertions are evaluated over a series of evaluation cycles.
- **Verification layer** specifies the directives to verification tools to handle the temporal assertions.
- **Modeling layer** is used to model the behavior of design inputs.

Our TLM-to-RTL assertion refinement only considers the first three layers since the fourth layer is not relevant in our framework. As presented in Section 7.2.1.1, the generated TLM assertions are in the simple syntax like "assert eventually! p". Most of them are temporal assertions involving transaction data only without any clock and control signal information. However, RTL assertions generally have such lower level details. Therefore, during the assertion refinement, we need to consider clock expression and control signals. If all such information is provided, the assertion refinement can be done by inserting the timing (i.e., clock expression) and control information as well as by substituting symbols.

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


ASSERTION_SPEC
 ‘set_clock (posedge clock);
 ......
 ‘control
   tmp_packet.to_chan
   @ $rose(write_enb[%tmp_packet.to_chan]);
 ......
END_ASSERTION_SPEC
```

In the above assertion refinement rules, *tmp_packet.to_chan* is a TLM variable that denotes the target slave address of the packet. From the symbol mapping, we can figure

out the corresponding RTL internal signal is *data_o_fsm* which is a 2-bit register. In the *ASSERTION_SPEC* block, the directive `\set_clock` sets the clock expression for the refined assertions. Because in RTL different value of control signals may specify different meaning to input data signals, we use the directive `\control` to set the RTL control signals during the TLM data refinement. The first parameter of `\control` is a TLM variable that appears in the TLM assertion. The second parameter is the corresponding RTL control signal expression for the TLM variable. In this example, only when the RTL signal *write_enb[%tmp_packet.to_chan]* asserts, the RTL signal *data_o_fsm* can indicate the target slave address. Here *%tmp_packet.to_chan* denotes the value of *tmp_packet.to_chan*.

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

The above example shows the usage of the assertion refinement rules. The TLM assertion wants to check whether the packet can be delivered to the slave 1. We can find that the RTL assertion includes the clock expression. The VERT substitutes the TLM variable *tmp_packet.to_chan* for its RTL signal $data\_o\_fsm$ accompanied by its control signal $\$rose(write\_enb[1])$.

7.2.2.3 Test Refinement Rules

From the symbol mappings, we can get the size information of each RTL signal as well as the bit correspondence between TLM data and RTL data. However, the RTL test stimulus is a timed sequence of data signal inputs controlled by control signals. Therefore it is required that the test refinement rules need to be programmable. Similar to Verilog testbench, VRS supports basic programming constructs like *if-then-else* and *for-loop* statements, sub-functions and so on. In essence, the test refinement rules consists of a sequence of statements. These statements are well organized to describe the timing

sequence of RTL data inputs. Based on the symbol mappings and the compiled test refinement rules, the *VERT* will produce one RTL test for one TLM test.

```
TEST_SPEC router(packet)
 ......
 main:
  begin
   initialize();
   reset();
   #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 ;
   slave_read(packet.to_chan, 1);
  end
END_TEST_SPEC
```

The above code is an example of the test refinement rules. It describes a testing scenario of the packet delivery for a router as follows: i) a master sends a packet to the router, ii) the router parses the packet and notifies the corresponding slave to fetch the packet, and iii) the slave receives the packet. Figure 7-6 shows an example of a TLM-to-RTL test translation using the given rules.

7.2.3 A Prototype Tool for TLM-to-RTL Validation Refinement

We developed a prototype tool which incorporates the proposed methods in Section 7.2.2. Figure 7-2 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 and assertion generation.

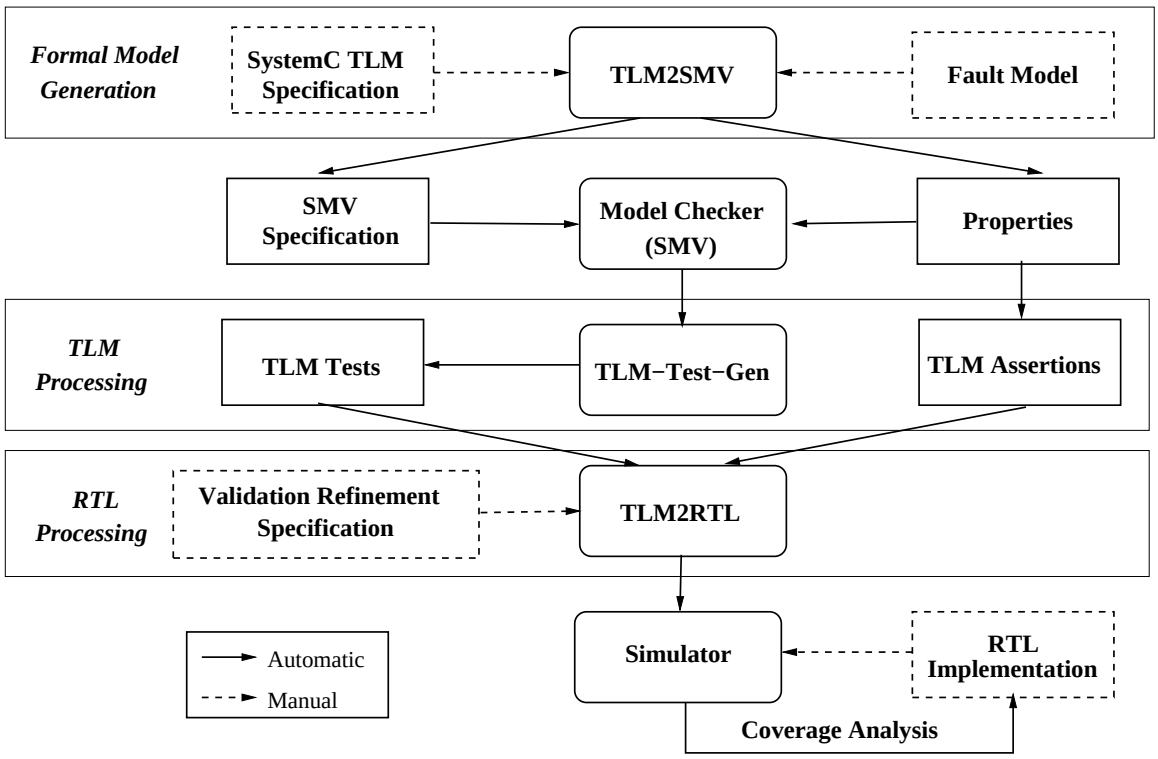


Figure 7-2. The structure of our prototype tool

7.2.3.1 TLM2SMV

Implemented based on the C++ parser Elsa [57], *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.

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, iii) convert the data type if necessary. Then *TLM2SMV* will transform the TLM specification. As described in Section 2.1.2, *TLM2SMV* will extract both static and dynamic information. At the mean time, it also collects the

information such as transaction relevant data, and 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 [56], we can get a set of counterexamples. The TLM tests are extracted from these counterexamples.

7.2.3.2 TLM Test Generation

When a specified safety property is false, SMV model checker will generate a counterexample to falsify it. The 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.

7.2.3.3 TLM2RTL

Because SystemC TLM focuses on the system level modeling, the generated TLM tests/ assertions lack the implementation level knowledge. So the generated TLM tests/assertions are different from RTL tests/assertions and can not 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 details and it needs the detailed timing information for each signal. In our framework, the user should provide a VRS which provides the mapping rules for the TLM to RTL test/assertion translation. With the generated TLM tests/assertions and the VRS as inputs, the *TLM2RTL* can translate the TLM tests/assertions to RTL tests/assertions. Finally, the coverage of the TLM implementation will be reported when simulating the generated RTL tests/assertions on RTL designs.

7.2.4 Assertion-Based Functional Equivalence

After the assertion and test refinement, we need to perform the simulation on both TLM and RTL designs to check the assertion-based functional equivalence. As shown in Figure 7-1, there is a equivalence checker which monitors both the TLM and RTL simulation result and reports the equivalence result based on its comparison.

TLM and RTL indicate different abstraction levels of the system. The traditional simulation based method can only guarantee the correctness by enumerating input tests and comparing the primary output results. However, the significant difference of internal structure of TLM and RTL designs is often eclipsed. Therefore no relation on the internal structure can be assumed during the simulation. As a functional constraint, the assertion can be used as a checkpoint during the simulation. Based on the inherent observability, the exercise of such checkpoints enables revealing the internal functional behaviors.

7.2.4.1 Assertion-Based Functional Coverage

During simulation, an assertion is covered means that the specific functional scenario is activated. Therefore the coverage of the assertions indicates the adequacy of the functional validation. Let T be a TLM design and R be a RTL design of T. We generate a set of TLM assertions $T_{assertion}$ according to the specified fault models of T, and we obtain a set of TLM tests T_{test} to activate such assertions. By using VRS, we can refine the T_{test} to a RTL test set R_{test}, and refine $T_{asserion}$ as a subset of the RTL assertion set $R_{asserion}$. When running the T_{test} and R_{test} on T and R individually, we can get the assertion coverage defined as follows.

Definition 10. *Given a TLM specification T and its RTL implementation R, by applying T_{test} on T and R_{test} on R, the assertion coverage can be calculated as:*

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

In our framework, there are two kinds of assertions: i) TLM assertions which are automatically generated from the TLM specifications, and ii) RTL assertions which are refined from the TLM assertions. The RTL programmers can also provide additional assertions based on other fault models and corner case scenarios. Therefore 100% TLM assertion coverage may not indicate 100% RTL assertion coverage in case additional RTL assertions are introduced.

7.2.4.2 Assertion Ordering

The assertion ordering plays an important role in TLM-to-RTL equivalence checking. For a TLM or RTL design which is instrumented with a large number of assertions, during the simulation, a test may exercise a sequence of assertions. An assertion indicates a functional checkpoint. Such simulation result of a test leads to an *assertion trace* which reveals the temporal order of checked functions in a system behavior. For a TLM test and its refined RTL version, when applying them on the TLM and RTL designs individually, it is required that the TLM functions and RTL functions happen consistently. In other words, the TLM assertions and corresponding RTL assertions should happen in their traces in the same order.

It is difficult to determine the order of the assertions during the simulation of a test. Since the assertions belong to different parallel processes in the TLM specifications and RTL implementations, even in the same assertion trace the assertions may not be activated linearly. That means several assertions may be exercised simultaneously. In addition, due to the existence of loop structure in a design, an assertion may be exercised several times in a design. This will further increase the difficulty in assertion matching between TLM traces and RTL traces. Inspired by the algorithm proposed by Lamport [50], in our framework, each assertion activation in a trace is associated with a "timestamp" to indicate the *happens before* (marked by $\prec$) relation. We use the timed assertion in the form of (a, t) to denote that the assertion a happens at clock cycle t.

Definition 11. *Given two timed assertions* $(a, t1)$ *and* $(b, t2)$ *in an assertion trace. The relations between them are as follows.*

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

Definition 11 describes the relation between the timed assertions. The key issue in determining the order is to figure out the timestamp for an assertion. For RTL design, because the assertions are translated into the VHDL/Verilog code, we can monitor the simulation at each clock cycle. Therefore, we can define the timestamp using the clock cycle number. However, figuring out the assertion order for TLM designs is not trivial due to the multiple classification of TLM abstraction levels. According to the definitions in [16], there are three TLM abstraction layers as follows.

- **Programmer's View (PV)**: Pure transaction based without timing information.
- **Programmer's View with Time (PVT)**: Transaction based with approximate timing information.
- **Cycle Accurate (CA)**: Cycle based with accurate timing information.

In the CA abstraction level, the model is cycle accurate. The CA TLM model is quite similar to the corresponding RTL model with respect to the notion of time. For the PVT abstraction level, the model simulates in non-zero simulation time. In spite of the time inaccuracy, we can still judge the assertion order according to the simulation time. During the simulation, if an TLM assertion is exercised, we can use the SystemC function *sc_time_stamp()* to record the current simulation time. Such *sc_time* information can be used as the timestamp to order the assertions. For the PV abstraction level, both the communication and computation part of the system are untimed. Therefore, it is not suitable to use the system time to order assertions. To determine the order of the interaction between communicating processes, SystemC provides the *delta cycle* concept which adopts the *evaluate-update* paradigm to interpret *zero-delay* semantics. The evaluate phase first executes all the processes that are *ready-to-run*. During the update phase the

scheduler calls the *update*() function to handle the pending processes registered by using the *request_update*() function. Each tiny delta cycle consists of these two steps without advancing the simulation time. Therefore the delta cycle can be utilized for ordering the assertions. For assertion ordering, we need to use a global variable as a counter of delta cycles. This counter can be used as the timestamp for assertions. If two assertions happen in the same delta cycle, then they are concurrent. Otherwise there is a "happen before" relation between them. Let's take the following simple program as an example.

```
//process1                  //process2

while(true){                while(true){

  a=FIFO.read();              FIFO.write(random());

}                           }
```

Assuming the size of the $FIFO$ channel is 1. The real action sequence of the above code can be " $(write, 1)$, $(read, 1)$, $(write, 2)$, $(read, 2)$, ...". For each delta cycle, only one write and read pair can happen. So we can find the order $(write, 1) \prec (write, 2)$ and $(read, 1) \prec (write, 2)$.

7.2.4.3 Assertion Based Functional Equivalence

In our framework, we define functional equivalence based on the assumption that if a TLM test can trigger a TLM assertion, then its RTL counterpart will also trigger the corresponding RTL assertion. It is important to note that in this chapter we do not intent to introduce new meaning of the classical *equivalence checking*. Our method still relies on the same concept - if two designs are equivalent, when giving the same input tests, they will produce the same outputs. Our goal is to increase the confidence of TLM-to-RTL functional equivalence checking under the monitoring of assertions.

The refinement process is described by two functions - AR for assertion refinement and TR for test refinement as follows.

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

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

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

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

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

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

Definition 12. *Given a TLM specification T and its RTL implementation R, T and R are assertion equivalent iff T_{test} can achieve 100% TLM assertion coverage and*

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

$$where \quad M_{TLM}(t) = \{a_1, a_2, \ldots, a_n\}. \quad \blacksquare$$

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

Definition 13. *Given a TLM specification T and its RTL implementation R, T and R are strongly assertion equivalent iff*

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

Figure 7-3 illustrates an example of assertion equivalence. Assuming the TLM specification and the RTL implementation are assertion equivalent and t is a TLM test and $t' = TR(t)$, we can get $M_{TLM}(t) = \{a1,a2,a3\}$ and $AR(M_{TLM}(t)) = \{b1,b2,b3\}$ which is a subset of $M_{RTL}(t')$. However, the assertion activiation order is not consistent ($a2$ happens before $a1$, but $b1$ happens before $b2$). Therefore, in this case, the TLM design and RTL design are assertion equivalent but not strongly assertion equivalent.

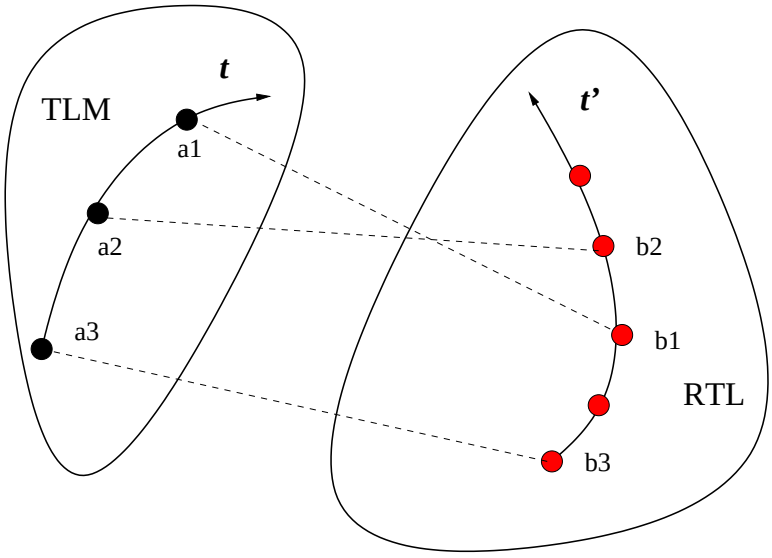


Figure 7-3. An example of assertion equivalence

7.3 Case Study

This section presents two case studies: a router system and a simplified version of the pipelined Alpha AXP processor [76]. We use the prototype tool (described in Section 7.2.3) to automatically generate the TLM assertions and tests as well as refined RTL assertions and tests. The experimental results are obtained on an 3 GHz AMD Opteron server with 16G RAM using Linux operation system.

7.3.1 A Router Example

Figure 2-11 shows the structure of the TLM specification of the router example. The main function of the router is to parse the incoming packets and distribute them to target slaves. The TLM and RTL packet formats are shown in Figure 7-4. 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 (i.e., target slave address). 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. In TLM design, the master module creates a packet first. Then, the master 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*. Triggered by the incoming packets, the route process first collects a packet from the channel connected to the master, next decodes the header of the packet to determine the target slave address, and then sends the packet to the channel connected to the target slave. Finally, the slave modules 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 transaction flow is controlled by the variable *to_chan* in the packet header.

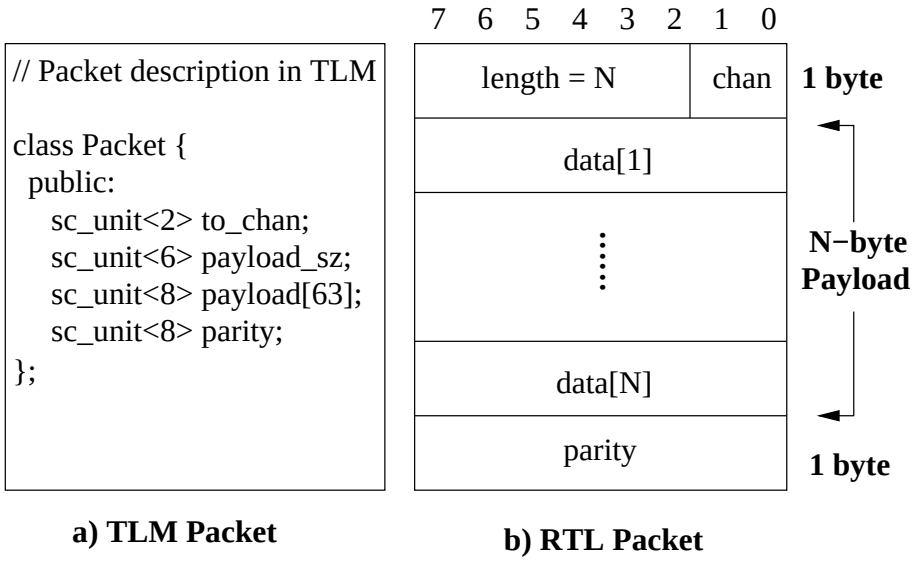


Figure 7-4. The packet format of the router in TLM and RTL

In the TLM specification, the I/O port of the router will deliver one whole packet at a time. However, in RTL implementation, during each clock cycle only a byte can be transferred through the I/O ports. Figure 7-5 shows the RTL I/O interface of the router example. During the validation refinement, we need to specify such mapping rules between the TLM and RTL designs using a VRS. Section 7.2.2.3 shows the partial VRS details of the router example. For instance, in the symbol mapping part, *packet.to_chan* in TLM corresponds to the RTL data $header[0:1]$ and *packet.payload_sz* corresponds to $header[2:7]$. The array of TLM data *packet.payload* will be mapped to RTL data

payload, and the TLM variable *packet.parity* corresponds to RTL variable *parity*. All such RTL packet data will be applied to input signal $DATA[7:0]$

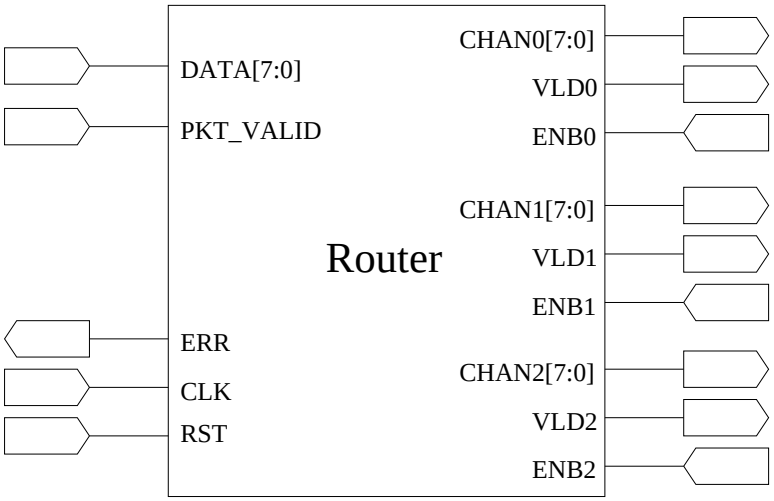


Figure 7-5. The I/O interface of the router example

In Section 7.2.2.3, we have shown the partial VRS to specify test and assertion refinement rules. By using our tool VERT, 95 TLM assertions were generated according to the proposed fault models. For each assertion, we derived a property and used it as an input of a model checker. The model checker generated one counterexample (test) to exercise each assertion. So we obtained 95 TLM assertions and 95 TLM tests from the TLM design. Table 7-1 gives the details. The first row defines the fault types. The second row shows the number of TLM assertions with different fault type. The third row indicates the number of generated TLM tests. The last row gives the test generation time (in minutes) using the SMV model checker.

Table 7-1. Assertion refinement for the router example

Fault Type	Data Faults	Flow Faults	Event Faults	Total
Numbers of TLM Assertions	88	4	3	95
Numbers of TLM Tests	88	4	3	95
Test Generation Time (min.)	73.70	2.60	31.50	107.8

During the TLM specification parsing, we did not consider the FIFO channel information because it is defined in the standard SystemC library. Therefore there is

no assertion for the FIFO channels in the router example. However, to improve the RTL code coverage, we manually created 4 TLM tests (2 tests for FIFO overflow, 1 test for reset check and 1 test for asynchronous read). Finally we got 99 TLM tests and 99 RTL tests for validation purposes. It is important to note that the generation of TLM assertions/tests and refinement are independent. In other words, TLM assertions and tests can come from multiple sources. We use the VRS to describe the both assertion and test refinement rules for the router example. Under the guidance of the VRS, our tool VERT can translate the TLM tests and TLM PSL assertions to the corresponding RTL tests as well as RTL assertions in the form of SVA.

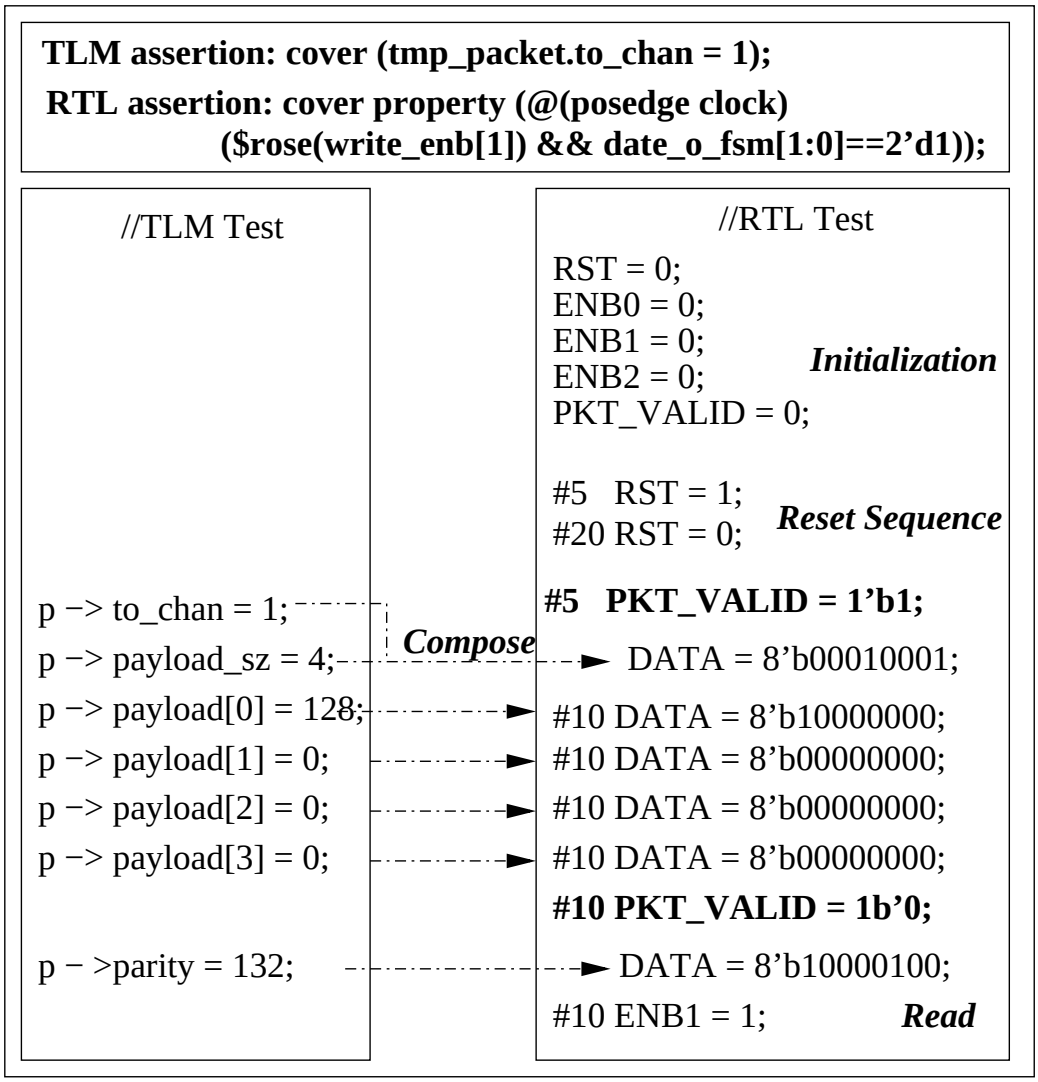


Figure 7-6. An example of TLM-to-RTL refinement

Figure 7-6 shows an example of TLM-to-RTL refinement. The goal of this example is to exercise the scenario that the packet can be sent to slave 1. By using the TLM test, we can activate the TLM assertion. Similarly, the RTL test can activate the RTL assertion.

We applied the TLM and RTL tests on the TLM and RTL designs independently. For the TLM design, we can get 100% coverage on both code and assertions. For the RTL design, we measured various coverage metrics [1] using Synopsys VCS cmView [82]. Table 7-2 shows the coverage obtained using the generated tests. Due to some unreachable code and missing "else" statements in RTL implementation, it is not possible to obtain 100% coverage in all the categories. It is important to note that the directed tests can only give 94.7% assertion coverage on the refined assertions. We investigated the assertions which are not covered. The reason is that the generated assertions and tests try to activate the scenario $to\_chan = 3$ which is used as an error state in TLM. Since RTL implementation did not consider this case, i.e., sending a packet to slave 3, we modified the RTL implementation and finally we can get 100% assertion coverage.

Table 7-2. RTL coverage for the router example

Source	Condition	FSM	Toggle	Path	Assertion
99.5%	76.6%	100%	76.6%	73.6%	94.7%

When applying a test during the validation, several TLM or RTL assertions may be exercised. To check the equivalence between TLM and RTL, our prototype tool recorded the simulation order for assertion activation. Such information is used to check the equivalence between the TLM and RTL design. Our result shows that the TLM and RTL designs of the router example are assertion equivalent, For strongly equivalence checking, we only used the assertions derived from the transaction flow and event faults. By matching the timed assertions on the assertion trace of each test, it shows that the TLM and RTL designs of the router example is also strongly assertion equivalent.

[1] The assertion coverage can not be obtained by VCS cmView.

7.3.2 A Pipelined Processor Example

In Figure 2-13 of Section 2.3.3, we give the TLM specification structure of the Alpha AXP processor. As shown in Table 7-3, we generated 212 TLM assertions using various fault models for the processor model. By using SMV, we generated 212 TLM tests (117 tests for data faults, 86 tests for flow faults and 9 tests for event faults) to exercise all such assertions.

Table 7-3. Assertions refinement for the Alpha AXP processor

Fault Type	Data Faults	Flow Faults	Event Faults	Total
Numbers of TLM Assertions	117	86	9	212
Numbers of TLM Tests	117	86	9	212
Test Generation Time (min.)	369.00	10.83	0.03	379.86

We applied all the generated tests under the observation of our tool VERT. According to the results provided by VCS cmView, we obtained the RTL implementation coverage report shown in Table 7-4. We found that the source and condition coverage can not be improved further because all the uncovered code are due to unreachable *MISSING ELSE* and default *CASEITEM* statements that do not exist in the RTL implementation. We used all the assertion for assertion equivalence checking, and the result shows that assertion equivalence can be achieved. For strongly equivalence checking, we did not include the assertions derived from the transaction data fault model. By comparing the assertion activation sequence, the equivalence checker shows that we can achieve a strong assertion equivalence by using the generated directed tests.

Table 7-4. RTL coverage for the Alpha AXP processor

Source	Condition	FSM	Toggle	Path	Assertion
98.9%	97.0%	NA	70.2%	86.3%	100%

7.4 Summary

Raising the abstraction level in SoC design flow can significantly reduce the overall design effort but introduces two challenges: i) how to guarantee functional equivalence between system level designs and low level implementations, and ii) how to reuse validation effort between different abstraction levels. To address both problems, this chapter proposed a methodology which reuses TLM validation effort to enable RTL validation as well as assertion-based functional equivalence checking between TLM and RTL models. By extracting formal models from TLM specifications, we can generate a set of assertions and corresponding tests to validate all the specified TLM "faults". Then the assertions and tests can be translated to their RTL counterparts using our proposed VRS. During the simulation, the TLM-to-RTL functional equivalence can be verified based on the assertion coverage and assertion ordering. The experimental results using several industrial designs demonstrated the effectiveness and benefits of our approach.

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

SystemC TLMs and UML activity diagrams are widely used to enable early exploration for both hardware and software designs. It can reduce the overall design and validation effort of complex SoC architectures. SoC validation is a major bottleneck due to lack of efficient automated techniques coupled with limited reuse of validation efforts between abstraction levels. This dissertation presented a novel top-down methodology for automatically generating tests from system-level specifications for functional validation at different abstraction levels. This chapter concludes the dissertation and outlines future research directions.

8.1 Conclusions

Existing SoC validation techniques 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 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. Simulation using directed tests is promising for functional validation, since running time can be significantly reduced with fewer tests while the coverage requirement can still be achieved.

A major challenge to enable directed test generation is to automatically extract a formal representation from system level specifications and develop an efficient coverage metric that allows coverage-driven directed test generation. Chapter 2 and 3 described a model checking based framework for directed test generation. This approach can automatically extract formal models from the high level specifications (including SystemC TLMs and UML activity diagrams as described in Chapter 2) as well as can generate properties (assertions) to cover all the errors for the given fault models (described in Chapter 3).

Most automatic directed test generation methods, especially for model checking based techniques, are impeded by the capacity restrictions of corresponding tools. To address the complexity of test generation using SAT-based BMC, this dissertation presented three efficient techniques to reduce the overall test generation time:

- **Property clustering** exploited various similarities between properties in a cluster (described in Chapter 4) to share learnings.
- **Efficient decision ordering** enables beneficial knowledge sharing (described in Chapter 5) between properties to avoid repeated validation effort.
- **Decomposition techniques** tried to scale down the property checking problem into several sub-problems (described in Chapter 6). The learning from the decomposed sub-problems is beneficial to the test generation of the original complex property.

By exploiting the commonalities between properties, the test generation time of a set of similar properties can be significantly reduced.

Furthermore, this dissertation presented a promising methodology that can check the TLM-to-RTL functional equivalence by reusing the TLM level validation effort. The refined assertions as well as tests can not only check the consistency between different abstraction levels, but also can be used for validating the system behavior of RTL designs. Since our method can be automated, complete reuse of TLM tests will lead to a drastic reduction in RTL validation.

In conclusion, this dissertation presented an efficient framework that can automatically generate tests from high-level SoC specifications and enable checking design errors in different stages of the SoC design. Due to drastic reduction in overall validation effort, this research will lead to cost-effective and high-quality systems.

8.2 Future Research Directions

Automated coverage-driven test generation and refinement for validation of SoC is a challenging problem. The work presented in this dissertation can be extended in the following directions:

- The coverage-driven property generation will generate a large set of properties, and many of them may activate the same scenarios. Consequently, there exists a lot of redundancy in the derived tests. Therefore, property compaction can be employed before the automated test generation to reduce the required number of properties. To further reduce the number of directed tests, existing test compaction techniques can be used.

- Find a way to efficiently generate tests for different designs but using the same property set. For example, spiral model is widely used as a software development process. The design is often slightly modified according to new requirements. Thus we need to re-generate the new tests for the properties of the previous design. Because most of the functionality remains the same, proper learning techniques can be used to generate the new tests.

- Currently most assertion based validation methods are based on simulation for both TLM and RTL designs. Generally for a large design, there will be thousands of assertions that need to be checked at the same time. Checking them independently will strongly affect the simulation performance. In the worst case, activating one assertion needs one test. Therefore it is necessary to design a methodology that can investigate the dependence between assertions and generate a small set of tests for the simulation but still can achieve the same assertion coverage.

- It is necessary to develop a framework that can debug the RTL level functional errors using its TLM specification. This can help designers to quickly find the error and fix it.

- Post-silicon debugging is an important stage during SoC design. However, in the post-silicon stage, all the debugging tasks are focused at signal level. It is very difficult to detect and check high level functional scenarios. Therefore it is necessary to refine the high level validation effort into gate-level implementation.

- This dissertation demonstrated the learning techniques (i.e., conflict clause forwarding and decision ordering) are promising for system level test generation. It can be extended to other domains, such as circuit-level validation. By incorporating our learning techniques, we believe that the performance of current SAT-based automatic test pattern generation (ATPG) approaches can be drastically improved.

REFERENCES

[1] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. FoCs - Automatic Generation of Simulation Checkers from Formal Specifications. In *Proceedings of Computer Aided Verification (CAV)*, pages 414–427, 2000.

[2] S. Abdi and D. Gajski. A formalism for functionality preserving system level transformations. In *Proceedings of Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 139–144, 2005.

[3] Accellera. Property Specification Language. [updated May 2008; cited February 2010]. *Available at http://www.eda.org/ieee-1850/.*

[4] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In *Proceedings of Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 254–268. Springer, 2005.

[5] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of International Conference on Formal Engineering Methods (ICFEM)*, pages 46–54, 1998.

[6] F. Balarin and R. Passerone. Functional Verification Methodology based on Formal Interface Specification and Transactor Generation. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 1013–1018, 2006.

[7] M. Benedetti and S. Bernardini. Incremental compilation-to-SAT procedures. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2004.

[8] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and Y. Wang. Uppaal - a tool suite for automatic verification of real-time systems. In *Proceedings of Hybrid Systems (HSCC)*, pages 232–243, 1995.

[9] D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th IEEE International Conference on Software Engineering (ICSE)*, pages 326–335, Los Alamitos, CA USA, 2004.

[10] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of International Conference on Tools and Algorithms for The Construction And Analysis of Systems*, pages 193–207, 1999.

[11] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[12] N. Bombieri, F. Fummi, and G. Pravadelli. 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)*, pages 1–6, 2006.

[13] N. Bombieri, F. Fummi, and G. Pravadelli. Incremental abv for functional validation of tl-to-rtl design refinement. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 882–887, 2007.

[14] N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva. Towards Equivalence Checking Between TLM and RTL Models. In *Proceedings of International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 113–122, 2007.

[15] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, C-35(8):677–691, August 1986.

[16] L. Cai and D. Gajski. Transaction Level Modeling: An Overview. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 19–24, 2003.

[17] K. Chandrasekar and M. S. Hsiao. Integration of learning techniques into incremental satisfiability for efficient path-delay fault test generation. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 1002–1007, 2005.

[18] M. Chen, X. Qiu, and X. Li. Automatic test case generation for uml activity diagrams. In *Proceedings of International Workshop on Automation on Software Test*, pages 2–8, 2006.

[19] A. Chureau, Y. Savaria, and E. M. Aboulhamid. The role of model-level transactors and uml in functional prototyping of systems-on-chip: A software-radio application. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 698–703, 2005.

[20] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model verifier. In *Proc. of Intl. Conference on Computer Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 495–499. Springer, 1999.

[21] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT press, 2000.

[22] D. Das, R. Kumar, and P. P. Chakrabarti. Timing verification of uml activity diagram based code block level models for real time multiprocessor system-on-chip applications. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, pages 199–208, 2006.

[23] K. Datta and P. P. Das. Assertion Based Verification Using HDVL. In *Proceedings of the International Conference on VLSI Design (VLSID)*, page 319, 2004.

[24] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[25] W. Ecker, V. Esen, T. Teininger, M. Velten, and M. Hull. Interactive Presentation: Implementation of A Transaction Level Assertion Framework in SystemC. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 894–899, 2007.

[26] M. Ericsson. Activity diagrams: what they are and how to use them. *The Rational Edge*, 2004.

[27] FBK-irst and CMU. NUSMV. [updated August 2006; cited August 2008]. *Available at http://nusmv.irst.itc.it/.*

[28] F. Ferrandi, F. Fummi, L. Gerli, and D. Sciuto. Symbolic functional vector generation for VHDL specifications. In *Design, Automation and Test in Europe (DATE)*, pages 442–446, 1999.

[29] H. D. Foster, A. C. Krolnik, and D. Lacey. *Assertion-Based Design, 2nd Edition.* Kluwer Academic Publishers, Boston, MA, 2004.

[30] F. Ghenassia. *Transaction Level Modeling with SystemC.* Springer, 2005.

[31] N. Guelfi and A. Mammar. A formal semantics of timed activity diagrams and its promela translation. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, pages 283–290, 2005.

[32] H. Zhu and P. Hall and J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

[33] A. Habibi and S. Tahar. Towards An Efficient Assertion Based Verification of SystemC Designs. In *Proceedings of International High Level Design Validation and Test Workshop (HLDVT)*, pages 19–22, 2004.

[34] A. Habibi and S. Tahar. Design and Verification of SystemC Transaction-Level Models. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 14(1):57–68, 2006.

[35] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Sanfrancisco, CA, 2003.

[36] P. Hsiung, C. Lin, and C. Liao. Perfecto: A SystemC-based Design-Space Exploration Framework for Dynamically Reconfigurable Architectures. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(3), 2008.

[37] IEEE P1800 Working Group. SystemVerilog Assertion. [updated September 2008; cited March 2010]. *Available at http://www.eda.org/sv-ac/.*

[38] J. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15(12):177–186, 1993.

[39] J. Marques-Silva and K. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[40] H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. In *BMC*, pages 51–65, 2004.

[41] D. Karlsson, P. Eles, and Z. Peng. Formal verification of systemc designs using a petri-net based representation. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 1228–1233, 2006.

[42] A. Kasuya and T. Tesfaye. Verification Methodologies in a TLM-to-RTL Design Flow. In *Proceedings of Design Automation Conference (DAC)*, pages 199–204, 2007.

[43] J. Kim, J. Whittemore, J. Marques-Silva, and K. Sakallah. On solving stack-based incremental satisfiability problems. In *Proceedings of International Conference on Computer Design (ICCD)*, pages 379–382, 2000.

[44] T. Kogel, M. Doerper, T. Kempf, A. Wieferink, R. Leupers, and H. Meyr. Virtual Architecture Mapping: A SystemC based Methodology for Architectural Exploration of System-on-Chips. *International Journal of Embedded Systems (IJES)*, 3(3):150–159, 2008.

[45] H. Koo and P. Mishra. Specification-based compaction of directed tests for functional validation of pipelined processors. In *International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 137–142, 2008.

[46] H.-M. Koo and P. Mishra. Test generation using (SAT)-based bounded model checking for validation of pipelined processors. In *Proc. of ACM Great Lakes Symposium on VLSI (GSLVLSI)*, pages 362–365, 2006.

[47] H.-M. Koo and P. Mishra. Functional test generation using design and property decomposition techniques. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(4), 2009.

[48] D. Kroening and N. Sharygina. Formal verification of systemc by automatic hardware/software partitioning. In *Proceedings of International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 101–110, 2005.

[49] M. Lahbib, R. Kamdem, M. Benalycherif, and R. Tourki. An Automatic ABV Methodology Enabling PSL Assertions across SLD Flow for SOCs Modeled in SystemC. *Computers and Electrical Engineering*, 31(4):282–302, 2005.

[50] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communication of ACM*, 21(7):558–565, 1978.

[51] M. Chen and X. Qiu and W. Xu and L. Wang and J. Zhao and X. Li. UML Activity Diagram Based Automatic Test Case Generation for Java Programs. *The Computer Journal*, 52(5):545–556, 2009.

[52] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of ACM*, 7(3):201–215, 1960.

[53] M. Davis, G. Logemann and D. Loveland. A machine program for theorem-proving. *Communication of ACM*, 5(7):394–397, 1962.

[54] M. Prasad and A. Biere and A. Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):156–173, 2005.

[55] J. P. Marques-Silva and K. A. Sakallah. The impact of branching heuristics in propositional satisfiability. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, pages 62–74, 1999.

[56] K. L. McMillan. SMV Model Checker, Cadence Berkeley Laboratory. [updated June 2006; cited August 2008]. *Available at http://www.kenmcmil.com/.*

[57] S. McPeak. Elsa. [updated August 2005; cited August 2008]. *Available at http://www.eecs.berkeley.edu/~smcpeak.*

[58] P. Mishra and M. Chen. Efficient techniques for directed test generation using incremental satisfiability. In *Proceedings of International Conference of VLSI Design*, pages 65–70, 2009.

[59] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proc. of Design Automation and Test in Europe (DATE)*, pages 182–187, 2004.

[60] P. Mishra and N. Dutt. Functional coverage driven test generation for validation of pipelined processors. In *Proc. of Design Automation and Test in Europe (DATE)*, pages 678–683, 2005.

[61] P. Mishra and N. Dutt. *Functional Verification of Programmable Embedded Architectures: A Top-Down Approach*. Springer, 2005.

[62] P. Mishra, H.-M. Koo, and Z. Huang. Language-driven validation of pipelined processors using satisfiability solvers. In *IEEE International Workshop on Microprocessor Test and Verification (MTV))*, pages 119–126, 2005.

[63] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC)*, pages 530–535, 2001.

[64] M. Moy, F. Maraninchi, and L. Maillet-Contoz. 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*, pages 26–35, 2005.

[65] W. Mueller, A. Rosti, S. Bocchio, E. Riccobene, P. Scandurra, W. Dehaene, and Y. Vanderperren. Uml for esl design: basic principles, tools, and applications. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 73–80, 2006.

[66] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing, 2nd Edition*. John Wiley & Sons, Hoboken, New Jersey, 2004.

[67] O. Strichman. Pruning techniques for the SAT-based bounded model checking problem. *Proceedings of Correct Hardware Design and Verification Methods (CHARME), ser. LNCS, T. Margaria and T. Melham, Ed. Springer-Verlag*, 2144:58–70, 2001.

[68] Object Management Group. UML Profile for System on a Chip (SoC), v 1.0.1. [updated August 2006; cited August 2008]. *Available at http://www.omg.org/technology/documents/formal/profile_soc.htm.*

[69] Object Management Group. UML Superstructure V2.1.2. [updated November 2007; cited August 2008]. *http://www.omg.org/docs/formal/07-11-02.pdf.*

[70] Open SystemC Initiative (OSCI). Systemc. [updated august 2006; cited august 2008]. *Available at http://www.systemc.org.*

[71] P. Mishra and N. Dutt. Specification-driven Directed Test Generation for Validation of Pipelined Processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(3):1–36, 2008.

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

[73] L. Pierre and L. Ferro. A Tractable and Fast Method for Monitoring SystemC TLM Specifications. *IEEE Transactions on Computers*, 57(10):1346–1356, 2008.

[74] Princeton Univeristy. zChaff. [updated November 2004; cited August 2007]. *Available at http://www.princeton.edu/~chaff/zchaff.html.*

[75] R. Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering and Methodology*, 15(1):1–38, 2006.

[76] R. L. Sites. Alpha AXP Architecture. *Digital Technical Journal*, 4(4), 1992.

[77] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A uml 2.0 profile for systemc: toward high-level soc design. In *Proceedings of the ACM International conference on Embedded software*, pages 138–141, 2005.

[78] A. Rose, S. Swan, J. Pierce, and J. Fernandez. *Transaction Level Modeling in SystemC*. OSCI TLM Working Group, 2005.

[79] R. Schutten and T. Fitzpatrick. Design for verification methodology allows silicon success. *EETIMES*, (16500856), 2003.

[80] D. Shin, A. Gerstlauer, J. Peng, R. Dömer, and D. Gajski. Automatic Generation of Transaction Level Models for Rapid Design Space Exploration. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 64–69, 2006.

[81] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *Proceedings of the The International Conference on Computer Aided Verification (CAV)*, pages 480–494, 2000.

[82] SYNOPSYS. VCS Verification Library. [updated August 2007; cited August 2007]. *Available at http://www.synopsys.com.*

[83] The Satisfiability Library. SAT Benchmark Problems. [updated September 2003; cited March 2010]. *http://www.satlib.org/Benchmarks/SAT/BMC/description.html.*

[84] B. Unhelkar. *Verification and Validation for Quality of UML 2.0 Models*. John Wiley & Sons, 2005.

[85] M. Velev. Boolean Satisfiability (SAT) benchmarks. [updated November 2006; cited March 2010]. *http://www.miroslav-velev.com/sat_benchmarks.html.*

[86] M. Velev. Automatic abstraction of equations in a logic of equality. In *Proceedings of Proceedings of Analytic Tableaux and Related Methods (TABLEAUX)*, pages 196–213, 2003.

[87] C. Wang, H. Jin, G. D. Hachtel, and F. Somenzi. Refining the SAT decision ordering for bounded model checking. In *Proceedings of Design Automation Conference (DAC)*, pages 535–538, 2004.

[88] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng. Generating test cases from uml activity diagram based on gray-box method. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, pages 284–291, 2004.

[89] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of Design Automation Conference (DAC)*, pages 542–545, 2001.

[90] W. Wolf, A. A. Jerraya, and G. Martin. PDF Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on In Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(10):1701–1713, 2008.

[91] L. Zhang, C. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 279–285, 2001.

[92] L. Zhang, M. Prasad, and M. Hsiao. Incremental deductive & inductive reasoning for SAT-based bounded model checking. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 502–509, 2004.

BIOGRAPHICAL SKETCH

Mingsong Chen received his B.S. and M.E. degrees from the Department of Computer Science and Technology of Nanjing University in China in 2003 and 2006 respectively. His research focuses on design automation of embedded systems, functional verification of System-on-Chip architectures, model checking techniques and software engineering.

In 2002, Mr. Chen joined the Software Engineering Group of Nanjing Unverisity as a research assitant. His research was focused on model checking of real-time systems and automatic test generation for UML activity diagrams. Under the supervision of Prof. Xuandong Li and Jianhua Zhao, he received his master's degree with thesis titled "Dynamic Optimization Techniques for State Space in Timed Automata during Reachability Analysis". Since 2006, he pursued his Ph.D. degree in CISE department of University of Florida. He joined the CISE Embedded Systems Group in 2007 under the supervision of Prof. Prabhat Mihsra. He participated the research project titled "SOC Validation using SystemC Transaction Level Models"which was funded by Intel Corporation and U.S. National Science Foundation. During his Ph.D. study, one of his papers presented in *International Conference on VLSI Design 2009* was nominated for best paper award. He was also a recipient of *DAC Young Student Support Program Award* in 2008.

Mr. Chen currently serves as a reviewer of several ACM and IEEE conferences including DAC, DATE, CODES+ISSS, ASP-DAC, GLSVLSI, VLSI Design, and ISVLSI. He is a student member of IEEE.