Giuseppe Di Guglielmo

On the Validation of Embedded Systems through Functional ATPG

Ph.D. Thesis

March 12, 2009

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
prof. Franco Fummi

Series N°: *TD-02-09*

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

$\mathcal{M}$

Acknowledgment.
Writing a thesis is not something you just do by yourself at an unimpeded moment. During the time period of three years, many people have been of great support. Colleagues, friends, and family have helped me in accomplishing this assignment. Therefore, I would like to take this opportunity to thank a number of them in particular.

It is difficult to overstate my gratitude to my Ph.D. Advisor, Prof. Franco Fummi. With his enthusiasm and his inspiration, he helped to make the hard work fun for me. Throughout these months, he provided encouragement, sound advice, and lots of good ideas.

I would like to thank the person who has taught me what it means to be a researcher, Dott. Graziano Pravadelli. Nearly three years ago, a conversation with Graziano started me on the path I traveled. Graziano has been instrumental in ensuring my academic, professional, and human well being ever since. In every sense, none of this work would have been possible without him.

I am indebted to my many colleagues for providing a stimulating and fun environment in which to learn and grow. I am especially grateful to my office-mates, Cristina Marconcini, Nicola Bombieri, and Francesco Stefanni. I would also acknowledge all the ESD Group and EDALab colleagues for their support at everyday work.

The last scientific acknowledgements goes to the students of ESD Laboratory, each one of them gave a little, but fundamental contribution to this work.

I wish to thank my friends, for helping and tolerating me to get through the hard times, and for all the emotional support, entertainment, and caring they provided, Alessandro Balzanello, Giordano Dalla Verde, Matteo Francescato, Oscar Francescato, Paolo Gaburro, Laura Maschi, Anna Piccoli, Cristiana Rossi, Alice Scala, Andrea Soave, Luca Stefanelli, Elisa Stevanella, Silvia Stizzoli, Matteo Tosato, Viola Zerbetto.

I would also like to express my thanks and gratitude to my friend, colleague and brother, Luigi Di Guglielmo for his endless support and encouragement at all times.

Lastly, and most importantly, I wish to thank my parents, Maria Antonietta Scanzano and Michele Di Guglielmo. They bore me, raised me, supported me, taught me, and loved me.

Verona, March 12^{nd}, 2009

Contents

Achronisms

The singular and plural of an acronym are always spelled the same.

ABV	Assertion-Based Verification
ADD	Assignment Decision Diagram
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Processor
ASSP	Application Specific Standard Product
ATPG	Automatic Test Pattern Generation
BDD	Binary Decision Diagram
BGL	Boost Graph Library
BMC	Bounded Model Checking
BTG	Block Transition Graph
CDC	Controllability-Don't-Care Set
CDC^{ext}	External Controllability-Don't-Care Set
CLP	Constraint Logic Programming
CNF	Conjunctive Normal Form
CP	Configurable Platform
CS	Controllability Set
CTL	Computation Tree Logic
DAG	Direct Acyclic Graph
DCG	DUV dependent Component Generator
DPI	Direct Programming Interface
DSP	Digital Signal Processing
DUV	Design Under Validation
EDA	Electronic Design Automation
EEFSM	Extended Event Finite State Machine
EFSM	Extended Finite State Machine
ETD	Easy-to-Detect
ETT	Easy-to-Traverse
FATE	Functional ATPG to Traverse unstabilized EFSM
FF	Flip-Flops
FSM	Finite State Machine

FSMD	Finite State Machine with Data Path
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
GCD	Greatest Common Divisor
HDL	Hardware Description Language
HIF	HDL Intermediate Format
HLDD	High Level Decision Diagram
HTD	Hard-to-Detect
HTT	Hard-to-Traverse
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronic Engineers
IIR	In-memory Intermediate Representation
IP	Intellectual Property
ISS	Instruction Set Simulator
LEFSM	Largest EFSM
LPV	Linear Programming Verification
LTL	Linear Time Logic
MSB	Most Significant Bit
ODC	Observability-Don't-Care Set
PI	Primary Input
PLI	Programming Language Interface
PO	Primary Output
PSL	Property Specification Language
REFSM	Reference EFSM
ROBDD	Reduced Ordered Binary Decision Diagram
RTE	Run Time Engine
RTL	Register Transfer Level
RH	Re-configurable Hardware
SBDD	Shared Binary Decision Diagram
SEFSM	Smallest EFSM
S^2EFSM	Stabilized SEFSM
SoC	System-on-Chip
STG	State Transition Graph
STL	Standard Template Library
TBA	To be Activated
TBP	To be Propagated
TL	Transactional Level
VHDL	VHSIC Hardware Description Language (VHDL)
VHSIC	Very High Speed Integrated Circuit

1 Introduction

In 1980, Gordon Moore stated a law according to which the number of transistors on silicon chips was doubling every eighteen months. Although it is not a real physical law, this prediction revealed to be incredibly exact. Between 1971 and nowadays, the density of transistors actually doubled every 1,96 years. As a consequence, electronic devices became more and more powerful and less and less costly.

Nowadays, embedded systems technology powers many of today's innovations and products. The rapid technologic advancement fuels the increasing chip complexity, which in turn enables the latest round of products. Embedded systems touch many aspects of everyday life, form the pocket-sized cell phone, and digital camera, to the high-end server that searches an online database, verifies credit cards found, and sends the order to the warehouse for immediate delivery. Also expectations for these chips grow at an equal rate, despite the additional complexity. For example for critical applications, like the chips that monitor the safety processes in cars. It is not acceptable that these chips fail during the normal use of the vehicle, nor it is acceptable, for example, to be denied access to an online brokerage accounts because the server is down.

Then, an enormous amount of engineering goes into each of these chips, whether it is a microprocessor, memory device o entire system on a chip. All types of chips have a set of challenges that the engineers must solve for the final product to be successful in the marketplace.

The verification flow has become a bottleneck in the development of today's digital systems. Chip complexity and market competitiveness has increased to the point that design teams require to spend around 70% of their efforts in finding the bugs that lurk in their design. In particular, the two main phases of the verification flow are testing and functional validation. The latter aims to ensure that the design satisfies its specification before manufacturing, by detecting and removing design errors. Testing focuses on the detection of production defects quickly as chips come off the manufacturing line. Even thought testing and functional validation are often grouped together, the two disciplines have little in common. A chip that successfully runs through testing may still add one to one and get three if the design had poor functional validation. Testing only confirms that the manufactured chip is equivalent to the circuit design specified to the manufacturing process. It makes no statement about the logical functionality of the chip itself.

However, the design teams dealing with the verification of a system have to handle three constraints: schedule, costs and quality.

Digital systems success depends heavily on hitting the marketplace at the right time, therefore schedule has become an imperative point. Then, the use of automatic tools reduces both the verification time and the probability of committing errors. A valid solution is represented by dynamic verification, that exploits simulation based techniques and automatic test pattern generators (ATPGs) to generate the required test sequences.

Customers expect that delivered products meet the quality standard. Obviously, as introduced before, for critical application quality is also essential. Furthermore, if the marketplace perceives that a product is of poor quality, it can have a devastating effect on the company.

Another critical constraint is cost that can influence the different verification phases. Costs of undetected bugs grow over time. If a bug is detected early during verification, it costs little to fix it. The designer reworks the high-level design description and the verification time can show that the update fixed the ordinal problem. A bug found in a system test, however, may cost hundreds of thousands of dollars: hardware must be re-fabricated and there is additional time -to-market. Finally, and most costly, a customer discovering bug not only invokes warranty replacement but may tarnish the image of the company or brand of products.

Functional validation is the highest lever that affects all three constraints. A chip can be produced early if the verification team is able to remove design errors efficiently. Then, costs of re-fabricating a chip multiple times can drive the development expenses to an unacceptable level and negatively affect the product schedule. Functional validation reduces the number of re-spins and removes latent problems, avoiding also quality problems of the developed products.

Another advantage of functional validation is that that designers have to work at the higher abstraction level, and the design descriptions are more tractable than gate-level ones. On the other side, efficient logic-level ATPGs are available for digital system and already part of the state of the art, while high-level functional ATPG frameworks are still in a prototyping phase.

1.1 Problem formulation

Most of the publications focusing on the field of Electronic Design Automation (EDA) begin claiming the importance of validation [1] and testing [2] for shipping successful digital systems. While the purpose of testing is to verify that the design was manufactured correctly, validation aims to ensure that the design meets its functional intent before manufacturing. In particular, functional validation of digital systems is the process of ensuring that the logical design of the device satisfies the architectural specification by detecting and removing every possible design error. As digital systems become more complex with each generation, verifying that the behavior is correct has become a very challenging task. Between 60% and 80% of hardware design group effort is now dedicated to verification. The trend is even more crucial for embedded systems, which are composed of a heterogeneous mix of hardware and software modules. The presence of design errors in the early phases of the design flow may lead to a complete failure of time-to-market fulfillment.

Because validation is on the critical path of embedded system design flow, the time it needs must be greatly reduced to satisfy the time-to-market requirements. Two ways are particulary promising for accomplishing the goal: abstraction of design level and automation of verification. A higher level of abstraction allows us to work more efficiently without worrying about low-level details. After all, verification is interested in ensuring functionality of a design, not caring about how it is implemented. However, abstraction reduce visibility and control over low-level details that could be exploited to arrange a more efficient verification strategy. On the other hand, the use of automatic tools reduce both the verification time and the probability of committing errors. Unfortunately, automation is not always possible, especially in process that are not well-defined and continue to require human ingenuity and creativity.

A good validation methodology must choose wisely the abstraction level and the degree of the automation according to what is being verified. Investigating about the correctness of synchronization between components of an embedded system may require a different approach with respect to addressing the correctness of each component alone. From this point of view, functional validation cannot be dependent on a single tool. A mix of different static and dynamic techniques is required, including hardware modeling and simulation, random and focused stimulus generation, coverage analysis, formal checking software to compare model behavior against specification. Tool-independent methodologies yield a more predictable and adaptable process, they have lower adoption cost, and they are not dependent on the continued technical superiority of any one EDA tool vendor. Thus, semiformal techniques, where formal and simulation-based approaches integrate each other, are arising more and more. In this way, respective limitations can be mitigated yielding efficient and effective verification processes for an early detection of design errors.

In this context, many functional validation techniques based on Automatic Test Pattern Generators (ATPGs), have been proposed in the paste. They are applied either to an explicit HDL description of the Design Under Validation (DUV), or to mathematical models, such as for example, a Finite State Machine (FSM). All these methods generate good test sequences for easy-to-detect faults, but hard-to-detect faults may require a very long execution time. This thesis would develop a methodology for addressing also hard-to-detect faults when a high-level ATPG is applied to verify functional descriptions of sequential circuits.

The proposed methodology is based on the Extended Finite State Machine model (EFSM), which allows to improve observability and controllability of faults injected into functional descriptions of sequential circuits. The EFSM is a valuable alternative to FSM model, preserving many characteristics of an FSM and reducing the state explosion problem. In this case, control and datapath are mixed, but the number of states is sensibly lower with respect to a corresponding FSM, since the EFSM does not require an explicit representation of internal registers. The EFSM paradigm can be very effective, from a design point of view, to describe concurrent systems; moreover it has not been widely applied for verification, and particularly for automatic test pattern generation. The reason depends on the difficulty of traversing an EFSM, which is a fundamental requirement to control and to observe faults. In fact, moving from a state of the EFSM to another depends on the value of primary inputs, but also on the value of internal registers.

In particular, this thesis proposes a procedure that generates a particular kind of EFSM that is exactly equivalent to the HDL description of design under validation (DUV). The proposed methodology changes the detectability properties of the DUV by exploiting the different EFSM features, allowing to generate test sequences for hard-to-detect faults. It can be applied to every sequential circuit improving the observability and controllability of hard-to-detect faults. A functional ATPG framework has been defined that is fast, since it relies on simulation, but also very effective on covering corner cases as it can exploit the EFSMs features to uniformly explore the state spaces of the DUV.

1.2 Thesis Overview

This thesis proposes a methodology based on both dynamic verification and semi-formal techniques to perform testing and functional validation of an embedded system. The thesis structure reflects all aspects that have to be addressed to define a deterministic ATPG and to estimate the efficiency of generated test sequences on high-level design errors.

Chapter 2 summarizes the necessary background that has been investigated for three years to plan and implement the methodology described in this thesis. Section 2.1 presents the state of the art for modeling and simulating embedded systems, from most popular Hardware Description Languages (HDLs) to implicit mathematical representations of digital systems functionalities. Section 2.2 is devoted to present an overview of formal and simulation-based methods to verify embedded systems. In particular, it presents the main issues dealing with the dynamic verification, pattern generation and fault modeling. Section 2.3 introduces the Mutation Analysis as a technique for software unit testing. Section 2.4 describes co-simulation concepts and strategies which allow to simulate and verify HW/SW embedded systems before the real platform is available. In this field, there is a large variety of approaches, that rely on different communication mechanisms to implement an efficient interface between the SW and the HW simulators. An overview is provided. Finally, Section 2.5 summarizes two different approaches used to solve the constraint solving problem, based respectively on Constraint Logic Programming and Model Checking.

Chapter 3 describes the motivation and goals driving this thesis work organized into four main aspects related to ATPG-framework design: the definition of an abstract model to represent the DUV, the definition of a functional deterministic ATPG engine, the definition of a fault model to guide the pattern generation and a metric to evaluate the generated patterns, and finally the definition of an efficient simulation engine.

Chapter 4 describes the HDL manipulation infrastructure based on *HIF Suite*. The HIF Suite is a set of tools and application programming interfaces that provide support for modeling and validation of hardware and software systems. The core of the HIF Suite is the HDL Intermediate Format (HIF) language upon which a set of front-end and back-end tools have been developed to allow the conversion of HDL core into HIF code and vice-versa. Thus, HIF Suite allows designer to manipulate and integrate heterogeneous components implemented by using different hardware description languages (HDLs). Moreover, the tools proposed in the following sections, rely on HIF APIs, for manipulating HIF descriptions in order to support model generation, fault injection and post-refinement validation.

Chapter 5 presents the computational models adopted to describe the DUV. The EFSM model is introduced (Section 5.2) and the testability of EFSMs is characterized by classifying hard-to-traverse transitions (Section 5.3). Such transitions are mainly responsible of hard-to-detect faults, since ATPGs spend a lot of computational effort trying to find a path to activate them. In Section 5.4 and then Section 5.5 is showed how such transitions can be replaced by manipulating the EFSM model, thus producing a new EFSM (S^{2}EFSM) with a higher degree of testability. This allows deterministic ATPGs to more uniformly analyze the state space of the resulting EFSM, thus, reducing the number of hard-to-detect faults. In Section 5.6 a particular variant of EFSM has been defined to manage properly both synchronous and asynchronous modules in a uniform way and then theoretical basis has been proposed to perform EFSM composition by bounding state and transition growth (Section 5.7.The aim of composition is to improve functional ATPG whose effectiveness and efficiency may be limited when separate EFSM are used to model the design under test. Finally, the High Level Decision Diagrams (HLDD) are presented as an alternative paradigm in Section 5.8 and in Section 5.9 is summarized how the HLDD models are generated starting from EFSM ones.

Chapter 6 describes the proposed ATPG engine and the techniques defined to improve its determinism. Section 6.1 describes the architecture of the proposed ATPG working on multiprocess DUVs. Section 6.2 presents the ATPG engine, oriented to EFSM traversal and thus fault activation, that relies on learning, backjumping and constraint solving to deterministically generate the test vectors for traversing all EFSM's transitions. Section 6.3 compares the defined scheduling algorithm and the EFSM-composition approach to deal with multiprocess designs. Finally, in Section 6.4 describes an extension of the proposed ATPG which exploits both EFSM and HLDD paradigms. HLDDs and EFSMs are deterministically explored by using propagation, justification, learning and backjumping. The integration of such strategies allows the ATPG to more efficiently analyze the state space of the design under validation.

Chapter 7 describes the methodologies defined to measure the effectiveness of the generated test sequence and to guide pattern generation. Section 7.1 describes the high-level bit-coverage fault model and the injection technique adopted. Section 7.2 describes how to exploit Mutation Analysis concepts, typical of the unit test of software engineering, to define a fault model for hardware description.

Validation via fault injection and fault simulation is a widely adopted technique to evaluate the correctness of a design implementation. However, the complexity of industrial designs and the huge number of faults that must be injected into them require efficient fault simulators, in order to make validation via fault simulation an affordable task. To optimize fault simulation performances, some parallelization techniques have been proposed at gate level. On the contrary, they have not been fully exploited at functional level, where functional fault models, instead of gate-level ones, are considered. Thus, Chapter 8 analyzes the impact of such parallelization techniques on functional faults. In particular, possible issues are presented together with optimizations that can be implemented to speed up the simulation.

Chapter 9 extends the proposed methodology to support co-simulation techniques. In particular, the chapter shows how the approach proposed in this thesis constitutes a fundamental block of the novel design and validation methodology for embedded systems developed within the VERTIGO European Project (FP6-2005-IST-5-033709), which has partially supported this research studies.

Finally, chapter 10 concludes this thesis pointing out its main results and addressing future works.

2

Background

This chapter describes general concepts related to modeling and validation of embedded systems necessary to better understand the ideas presented in the subsequent chapters of this thesis. Section 2.1 is devoted to summarize the design flow of digital systems spending particular effort to present some of the most popular description languages and mathematical formalisms used for system modeling. The intent of functional validation lies in ensuring that models meet their specification. This topic is addressed in Section 2.2. Section 2.3 introduces Mutation Analysis, which is a method of software testing, which involves modifying program's source code in small ways. These, so-called mutations, are based on well-defined mutation operators that either mimic typical programming errors or force the creation of valuable tests. The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution. Modeling complex embedded platforms requires to co-simulate one or more CPUs, connected to some hardware devices, running applications on top of an operating system. Therefore, Section 2.4 introduces co-simulation issues and summarizes some literature and industrial co-simulation frameworks. Finally, Section 2.5 introduces constraint solving tools. Constraint Logic Programming lies at the intersection of logic programming, optimization and artificial intelligence. In the field of logic in computer science, Model Checking refers to the following problem: given a simplified model of a system, test automatically whether this model meets a given specification. Typically, the systems are hardware or software systems, and the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Both CLP and Model Checking have proved successful tools for application in a variety of areas including functional validation.

2.1 Embedded Systems: Modeling

The design of an embedded system is a very challenging task which involves the cooperation of different experts: system architects, SW developers, HW designers, verification engineers, etc. Each of them operates on different views of the system

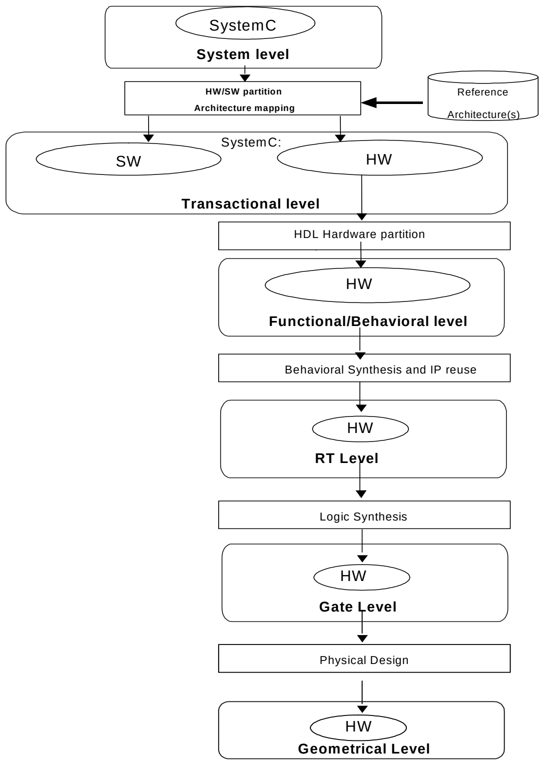


Fig. 2.1. Embedded system design flow.

starting from a very abstract informal specification and refining the model through different abstraction levels.

Generally, designers adopt a top-down methodology starting from a high (behavioral) level and going down to the geometrical level, as described in Figure 2.1 where a general tool-independent design flow is shown. The translations between abstraction levels are called *synthesis steps*, and generally they are performed by using automatic tools, while it is purely manual in the higher levels.

At every level of abstractions, a model of a digital system can be view as a black box, processing the information carried to its input to produce outputs.The I/O mapping realized by the box defines the behavior of the system.

Historically, the highest level of abstraction for digital system is the *behavioral level*, where the focus is centered on the logic function of the design ignoring every implementation detail. Some EDA books make a more accurate classification, and they refer to this level as the *functional level*, while a behavioral level model is intended as a functional representation of the design coupled with a description of the associated timing relations. Each one of these two abstraction levels keeps quite low the complexity of digital system models allowing their rapid simulation. However, the advent of more complex digital designs that integrate a mix of HW and SW components, as embedded systems and System-on-Chip (SoC), induced to start the design flow at a much higher level, denoted as *electronic system level (ESL)*.

ESL design is an embedded system design developed at a level of abstraction above RTL. ESL design enables functional validation and performance analysis of complex architectures and protocols very early during the system development. At this high level of abstraction, designers can run large numbers of test cases to identify and fix corner case problems that otherwise would be discovered only during the system integration, too late to safely operate corrections in short times. With an ESL design methodology, HW/SW partitioning can be paired with up-front performance analysis that identifies bottlenecks long before physical implementation. The need for additional processing capacity and deployment of dedicated hardware accelerators is identified while there is still time available to explore different ways in order to meet requirements.

At system level, the design is considered an interconnection of independent subsystems which communicate via block of words (messages), where a word is a group of logic values. At this level there is no distinction between the HW components and the embedded SW; indeed, every subsystems is represented by a high-level algorithm. SystemC (see Section 2.1.3) is a very suited language for system level modeling, because it joins the flexibility of C++ language and the typical features of the more traditional hardware description languages, as VHDL (see Section 2.1.1), Verilog (see Section 2.1.2), etc. This definitely allows an easier HW/SW partitioning process, whose output is a *transactional level* model that conforms to the characteristics of the reference architecture selected for the platform mapping.

After HW/SW partitioning, the HW part follows a traditional design flow. The HW model is possible partitioned in various behavioral/functional descriptions that better characterize the different HW units. Then, high-level synthesis translates the behavioral/functional model into a *Register Transfer Level (RTL)* model, where the functionalities of the design are divided and represented by a structural connection of combinational and sequential components (generally described as finite state machines).

Finally, logic synthesis is used to translate the RTL model to a gate-level model, where the design is mapped into a structural view of primitive components (AND, OR, flip-flop, etc.) from which the physical mask of the circuit can be easily generated.

After every synthesis step, a verification/testing phase is mandatory to avoid the propagation of errors between the different abstraction levels. Indeed, synthesis is a dangerous process that may introduce further bugs. This can be due to different causes: incorrect use of synthesis tools, incorrect code writing style that may prevent the synthesis tool to adequately infer the required logic, bugs of the synthesis tool, etc.

At each of the previously described abstraction levels, the system is described through the use of a hardware design language which extends the features of the traditional software languages to deal with typical hardware characteristics as concurrency, timing, I/O communication, etc. There exist many Hardware Design Languages (HDL), the features of the most popular ones are summarized in the

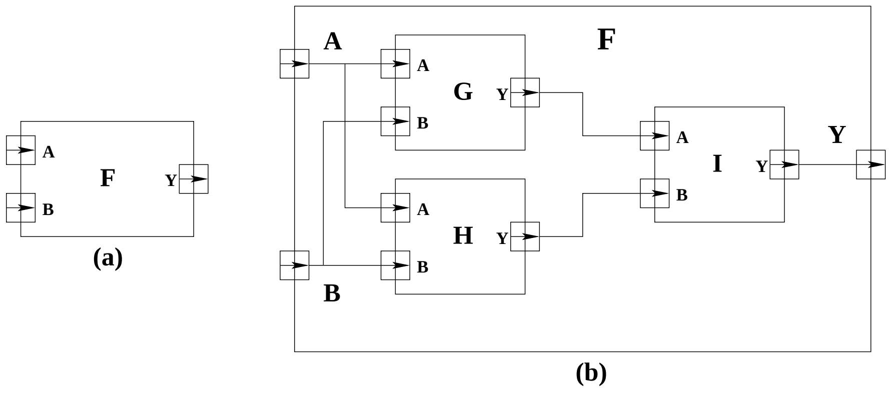


Fig. 2.2. Example of a structural description.

next three sections. On the contrary, Section 2.1.4 describes how digital systems can be represented by means of some implicit mathematical formalisms that have been reveled to be very efficient for the development of many verification techniques.

2.1.1 VHDL

The VHSIC Hardware Description Language (VHDL) [3] is a language for describing digital electronic systems. It arose out of the United States government's Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980. In the course of this program, it became clear that there was a need for a standard language for describing the structure and function of Integrated Circuits (IC). Hence VHDL was developed, and subsequently adopted as a standard by the Institute of Electrical and Electronic Engineers (IEEE).

VHDL was designed to fill a number of needs in the design process. It allows description of the structure of a design, that is, how it is decomposed into sub-designs, and how those sub-designs are interconnected. VHDL allows the specification of design functionalities by using familiar programming language constructs. Moreover, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and cost of hardware prototyping.

A digital electronic system can be described as a module with inputs and/or outputs. The electrical values on the outputs are some function of the values on the inputs. Figure 2.2(a) shows an example of this view of a digital system. The module F has two inputs, A and B, and an output Y. Using VHDL terminology, the module F is called a design *entity*, and the inputs and outputs are called *ports*.

One way of representing the function of a module is to describe how it is composed of sub-modules. Each of the sub-modules is an instance of some entity, and the ports of the instances are connected through *signals*. Figure 2.2(b) shows how the entity F might be composed of instances of the entities G, H and I. This kind of description is called a structural description. Note that each of the entities G, H and I might also have a structural description.

In many cases, it is not appropriate to describe a module structurally. One such case is a module which is at the bottom of the hierarchy of some other structural description. Such a description is called a functional or behavioral description. To illustrate this, suppose that the function of the entity F in Figure 2.2(a) is the exclusive-or function. Then, a behavioral description of F could be the Boolean function $Y = \overline{A}.B + A.\overline{B}$.
More complex behaviors cannot be described purely as a function of inputs. In systems with feedback, the outputs are also a function of time. VHDL solves this problem by allowing description of behavior in the form of an executable program.

Once the structure and behavior of a module have been specified, it is possible to simulate the module by executing its behavioral description. This is done by simulating the passage of time in discrete steps. At some simulation time, a module input may be stimulated by changing the value on an input port. The module reacts by running the code of its behavioral description and scheduling new values to be placed on the signals connected to its output ports at some later simulated time. This is called scheduling a transaction on that signal. If the new value is different from the previous value on the signal, an event occurs and other modules with input ports connected to the signal may be activated.

The simulation starts with an initialization phase and then proceeds by repeating a two-stage simulation cycle. In the initialization phase, all signals are given initial values, the simulation time is set to zero, and each module's behavior program is executed. This usually results in transactions being scheduled on output signals for some later time.

In the first stage of a simulation cycle, the simulated time is advanced to the earliest time at which a transaction has been scheduled. All transactions scheduled for that time are executed and this may cause events to occur on some signals.

In the second stage, all modules which react to events occurring in the first stage have their behavior program executed. These programs will usually schedule further transactions on their output signals. When all of the behavior programs have finished the execution, the simulation cycle repeats. If there are no more scheduled transactions, the whole simulation is completed.

The purpose of the simulation is to gather information about the changes in system state over time. This can be done by running the simulation under the control of a simulation monitor. The monitor allows signals and other state information to be viewed or stored in a trace file for later analysis. It may also allow interactive stepping of the simulation process, much like an interactive program debugger.

Figures 2.3 and 2.4 show small examples of a VHDL description of a two-bit counter to give you a feel for the language and how it is used. Figure 2.3 shows the description of an entity by specifying its external interface, which includes a description of its ports. This specifies that the entity `count2` has one input and two outputs, all of which are bit values; that is, they can assume the values '0' or '1'. It also defines a *generic constant* called `prop_delay`, which can be used to control the operation of the entity (in this case its propagation is delayed). If no value is explicitly given for this value when the entity is used in a design, the default

```
ENTITY count2 IS
  GENERIC (prop_delay : Time := 10 ns);
    PORT (
           clock  : IN BIT;
           q0     : OUT BIT;
           q1     : OUT BIT);
END count2;
```

Fig. 2.3. Example of VHDL entity.

```
ARCHITECTURE behavior OF count2 IS
BEGIN
    count_up: PROCESS (clock)
       VARIABLE count_value : NATURAL := 0;
   BEGIN
       IF clock = '1' THEN
           count_value := (count_value + 1) mod 4;
           q0 <= bit'val(count_value mod 2) AFTER prop_delay;
           q1 <= bit'val(count_value / 2) AFTER prop_delay;
       END IF;
   END PROCESS count_up;
END behavior;
```

Fig. 2.4. Example of VHDL architecture.

value of 10ns will be used. An implementation of the entity is described in an *architecture* body. There may be more than one architecture body corresponding to a single entity specification, each of which describes a different view of the entity. For example, a behavioral description of the counter could be written as shown in Figure 2.4. In this description of the counter, the behavior is implemented by a process called `count_up`, which is sensitive to the input `clock`. A *process* is a piece of code which is executed whenever any of the signals is sensitive to changes value. This process has a variable called `count_value` to store the current state of the counter. The variable is initialized to zero at the start of simulation and it retains its value between activation of the process. When the `clock` input goes from '0' to '1', the state variable is incremented, and transactions are scheduled on the two output ports based on the new value. The assignments use the generic constant `prop_delay` to determine how long after the `clock` change the transaction should be scheduled. When control reaches the end of the process body, the process is suspended until another change occurs on the clock.

VHDL language is very similar to the Ada programming language. Comments start with two adjacent hyphens (`'--'`) and extend to the end of the line. Identifiers in VHDL are used as reserved words and as programmer defined names. Note that the case of letters is not considered significant, so the identifiers `cat` and `Cat` are the same. VHDL provides a convenient way of specifying literal values for arrays of type bit. VHDL provides a number of basic, or *scalar*, types and a means of forming *composite* types. The scalar types include numbers, physical quantities and enumerations (including enumerations of characters), and there are a number of standard predefined basic types. The composite types provided are arrays and records. VHDL also provides *access* types (pointers) and *files*.

Moreover, VHDL supports a physical type that is a numeric type for representing some physical quantity, such as mass, length, time or voltage. The declaration of a physical type includes the specification of a base unit, and possibly a number of secondary units, being multiples of the base unit. An array in VHDL is an indexed collection of elements all of the same type. Arrays may be one-dimensional (with one index) or multidimensional (with a number of indices). In addition, an array type may be constrained, in which case the bounds for an index are established when the type is defined, or unconstrained, in which the bounds are established subsequently. VHDL provides basic facilities for records, which are collections of named elements of possibly different types. The use of a subtype allows the values taken on by an object to be restricted or constrained subset of some basic type.

Expressions in VHDL are much like expressions in other programming languages. An expression is a formula combining primaries with operators. Primaries include names of objects, literals, function calls and parenthesized expressions.

The logical operators `and`, `or`, `nand`, `nor`, `xor` and `not` operate on values of type bit or boolean, as well as on one-dimensional arrays of these types. For array operands, the operation is applied between corresponding elements of each array, yielding an array of the same length as the result. For bit and boolean operands, `and`, `or`, `nand`, and `nor` are 'short-circuit' operators; that is, they only evaluate their right operand if the left operand does not determine the result. So, `and`, and `nand` only evaluate the right operand if the left operand is true or '1'; and `or`, and `nor` only evaluate the right operand if the left operand is false or '0'.

The relational operators `=`, `/=`,`<`, `<=`, `>` and `>=` must have both operands of the same type, and yield boolean results. The equality operators (`=` and `/=`) can have operands of any type. For composite types, two values are equal if all of their corresponding elements are equal. The remaining operators must have operands which are scalar types or one-dimensional arrays of discrete types.

The sign operators ($+$ and $-$) and the addition ($+$) and subtraction ($-$) operators have their usual meaning on numeric operands. The concatenation operator (&) operates on one-dimensional arrays to form a new array with the contents of the right operand following the contents of the left operand. It can also concatenate a single new element to an array, or two individual elements to form an array. The concatenation operator is most commonly used with strings.

The multiplication ($*$) and division (/) operators work on integer, floating point and physical types. The modulus (`mod`) and remainder (`rem`) operators only work on integer types. The absolute value (`abs`) operator works on any numeric type. Finally, the exponentiation ($**$) operator can have an integer or floating point left operand, but it must have an integer right operand. A negative right operand is only allowed if the left operand is a floating point number.

Like other programming languages, VHDL provides subprogram facilities in the form of *procedures* and *functions*. VHDL also provides a package facility for collecting declarations and objects into modular units. *Packages* also provide a measure of data abstraction and information hiding.

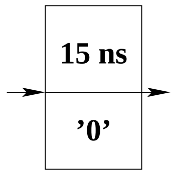


Fig. 2.5. Example of timing

A subprogram declaration in this form simply names the subprogram and specifies the parameters required. The body of statements defining the behavior of the subprogram is deferred. For function subprograms, the declaration also specifies the type of the result returned when the function is called. This form of subprogram declaration is typically used in package specifications, where the subprogram body is given in the package body, or to define mutually recursive procedures.

VHDL allows two subprograms to have the same name, provided the number or base types of parameters differ. The subprogram name is then said to be overloaded. When a subprogram call is made using an overloaded name, the number of actual parameters, their order, their base types and the corresponding formal parameter names (if named association is used) are used to determine which subprogram is meant. If the call is a function call, the result type is also used.

A digital system is usually designed as a hierarchical collection of modules. Each module has a set of ports which constitute its interface to the outside world. In VHDL, an entity is such a module, which may be used as a component in a design, or which may be the top level module of the design. The entity declarative part may be used to declare items which are to be used in the implementation of the entity. Usually such declarations will not be included in the implementation itself, so they are only mentioned here for completeness. Also, the optional statements in the entity declaration may be used to define some special behavior for monitoring the operation of the entity.

The entity header is the most important part of the entity declaration. It may include specification of *generic constants*, which can be used to control the structure and behavior of the entity, and *ports*, which channel information into and out of the entity.

A signal assignment schedules one or more transactions to a signal (or port). The target must represent a signal, or be an aggregate of signals. If the time expression for the delay is omitted, it defaults to 0fs. This means that the transaction will be scheduled for the same time as the assignment is executed, but during the next simulation cycle. Each signal has a projected output waveform associated with it, which is a list of transactions giving future values for the signal. A signal assignment adds transactions to this waveform. For example, the signal assignment `s <= '0' after 10 ns;` will cause the signal `s` to assume the value `'0'` 10ns after the assignment is executed. We can represent the projected output waveform graphically by showing the transactions along a time axis. So if the above assignment were executed at time 5ns, the projected waveform would be as shown in

```
wait_statement ::=
  wait[ sensitivity_clause ][ condition_clause ][ timeout_clause ];
sensitivity_clause ::= on sensitivity_list
sensitivity_list ::= signal_name { , signal_name }
condition_clause ::= until condition
timeout_clause ::= for time_expression
```

Fig. 2.6. Syntax of a VHDL wait statement.

Figure 2.5. When simulation time reaches 15ns, this transaction will be processed and the signal updated.

The primary unit of behavioral description in VHDL is the process. A process is a sequential body of code which can be activated in response to changes in state. When more than one process is activated at the same time, they execute concurrently. A process statement is a concurrent statement which can be used in an architecture body or block. The declarations define items which can be used locally within the process. Note that variables may be defined here and used to store state in a model. A process may contain a number of signal assignment statements for a given signal, which together form a driver for the signal. Normally there may only be one driver for a signal, so the code that determines a signal value is confined to one process. A process is activated initially during the initialization phase of simulation. It executes all of the sequential statements and then repeats, starting again with the first statement. A process may suspend itself by executing a wait statement (Figure 2.6).

The *sensitivity list* of the wait statement specifies a set of signals to which the process is sensitive while it is suspended. When an event occurs on any of these signals (that is, whenever the value of the signal changes), the process resumes and evaluates the condition. If it is true or if the condition is omitted, execution proceeds with the next statement, otherwise the process re-suspends. If the sensitivity clause is omitted, then the process is sensitive to all of the signals mentioned in the condition expression. The timeout expression must evaluate to a positive duration and indicates the maximum time for which the process will wait. If it is omitted, the process may wait indefinitely. If a sensitivity list is included in the header of a process statement, then the process is assumed to have an implicit wait statement at the end of its statement part. The sensitivity list of this implicit wait statement is the same as that in the process header. In this case, the process may not contain any explicit wait statements.

Often, a process describing a driver for a signal contains only one signal assignment statement. VHDL provides a convenient short-hand notation called a *concurrent signal assignment* statement for expressing such processes.

When VHDL descriptions are written, they are done so in a design file then a compiler is invoked to analyze them and insert them into a design library. A number of VHDL constructs may be analyzed separately for inclusion in a design library. These constructs are called *library units*. Primary library units are entity declarations, package declarations and configuration declarations. Secondary library units are architecture bodies and package bodies. These library units depend on the specification of their interface in a corresponding primary library unit, so the primary unit must be analyzed before any corresponding secondary unit. A design file may contain a number of library units. Libraries are referred to using identifiers called logical names. These names must be translated by the host operating system into an implementation dependent storage name.

2.1.2 Verilog

Verilog [4] was introduced in 1985 by Gateway Design System Corporation, now a part of Cadence Design Systems, Inc.'s Systems Division. Until May 1990, with the formation of Open Verilog International (OVI), Verilog HDL was a proprietary language of Cadence. Cadence was motivated to open the language to the public domain with the expectation that the market for Verilog HDL-related software products would grow more rapidly with broader acceptance of the language.

Verilog allows a hardware designer to describe designs at a high-level of abstraction, such as at the architectural or behavioral level, as well as at the lower implementation levels (i.e., gate and switch levels) leading to integrated circuits layouts and chip fabrication. A primary use of HDL is the simulation of designs before the designer must commit to fabrication. This handout does not cover all of Verilog but focuses on the use of Verilog at the architectural or behavioral levels. The handout emphasizes design at the register transfer level.

The Verilog language provides the digital designer with a means of describing a digital system at a wide range of abstraction levels, while, at the same time, providing access to computer-aided design tools to aid in the design process at these levels.

Verilog allows hardware designers to express their design with behavioral statement and to put off the details of implementation to a later stage of design. An abstract representation helps the designer to explore architectural alternatives through simulations and to detect design bottlenecks before detailed design begins.

From the simulation point of view, Verilog is a discrete event time simulator; that is, events are scheduled for discrete times and placed in an ordered-by-time wait queue. The earliest events are at the front of the wait queue and the later events are behind them. The simulator removes all the events for the current simulation time and processes them. During the processing, more events may be created and placed in their proper place in the queue for later processing. When all the events of the current time have been processed, the simulator advances time and processes the next events at the front of the queue.

Figure 2.7 shows a simple Verilog program. Some instructions are C-like, such as assignment statements and comments have a C++ flavor; for example, they are shown by "`//`" to the end of the line. The Verilog language describes a digital system as a set of *modules*; in this example, there is only a single module called `simple`.

In the module `simple`, `A` and `B` are declared as 8-bit registers and `C` is a 1-bit register or flip-flop. Inside the module, the one *always* and two *initial* constructs describe three threads of control; that is, they run concurrently. Within the initial construct, statements are executed sequentially, such as in C or other traditional imperative programming languages. The always construct is identical to the initial construct, except that it loops forever as long as the simulation runs.

The notation `#1` constrains the execution of the statement after one unit of simulated time. Therefore, the thread of control caused by the first initial construct will delay for 20 time units before calling the system task `$stop` to end the simulation.

The `$display` system task allows the designer to print a message, such as `printf` does in the C language. In every time unit when one of the values of the listed variables changes, the `$monitor` system task prints a message. The system function `$time` returns the current value of simulated time.

```
module simple;
// The register A is incremented by one. Then first four bits
// of B is set to "not" of the last four bits of A. C is the
// "and" reduction of the last two bits of A.
//declare registers and flip-flops
reg [0:7] A, B;
reg       C;
// The two "initial"s and "always" will run concurrently
initial begin: stop_at
   // Will stop the execution after 20 simulation units.
   #20; $stop;
end
// These statements done at simulation time 0 (since no #k)
initial begin: Init
   // Initialize the register A. The other registers have
   // values of "x"
   A = 0;
   // Display a header
   $display("Time A B C");
   // Prints the values anytime a value of A, B or C changes
   $monitor("%0d %b %b %b", $time, A, B, C);
end
//main_process will loop until simulation is over
always begin: main_process

   // #1 means do after one unit of simulation time
   #1 A = A + 1;
   #1 B[0:3] = ~A[4:7]; // ~ is bitwise "not" operator
   #1 C = &A[6:7];
   // bitwise "and" reduction of last two bits of A
end
endmodule
```

Fig. 2.7. An example of a digital model in Verilog.

The structure of the module `simple` is typical of Verilog programs; there is an initial construct to specify the length of the simulation, another initial construct to initialize registers and specify which registers must be monitored, and an always construct for the digital system one is modeling. Notice that all the statements in the second `initial` are done at time = 0, since there are no delay statements, i.e., `#<integer>`.

The Verilog language describes a digital system as a set of modules. Each of these modules has an interface to other modules to describe how they are interconnected. Usually a Verilog file includes one module but this is not a constraint. The modules may run concurrently, but usually there is one top level module which specifies a closed system containing both test data and hardware models. The top level module executes instances of other modules.

As for VHDL entities, Verilog modules can represent pieces of hardware ranging from simple gates to complete systems, like a microprocessor for example. Modules can either be specified behaviorally or structurally (or a combination of the two). A behavioral specification defines the behavior of a digital system (module) using traditional programming language constructs, e.g., `if`, assignment statements, etc. A structural specification expresses the behavior of a digital system (module) as a hierarchical interconnection of sub modules. At the bottom of the hierarchy, the components must be primitives or specified behaviorally. Verilog primitives include gates (e.g., nand) as well as pass transistors (switches). The structure of a module is shown in Figure 2.8.

```
module <module name> (<port list>);
  <declares>
  <module items>
endmodule
```

Fig. 2.8. Module structures.

The <*module name*> is an identifier that uniquely names the module. The <*port list*> is a list of `input`, `inout` and `output` ports which are used to connect to other modules. The <*declares*> section specifies data objects as registers, memories and wires, as well as procedural constructs such as *functions* and *tasks*.

The <*module items*> may be `initial` constructs, `always` constructs, continuous assignments or instances of modules.

The semantics of the module construct in Verilog is very different from subroutines, procedures and functions in other languages. A module is never called. It is instantiated at the start of the program and stays around for the life of the program. A Verilog module instantiation is used to model a hardware circuit where we assume no one unsolders or changes the wiring. Each time a module is instantiated, we give its instantiation a name.

Verilog makes an important distinction between *procedural assignment* and the *continuous assignment* `assign`. Procedural assignment changes the state of a register, i. e., sequential logic, whereas the continuous statement is used to model combinational logic. Continuous assignments drive wire variables and are evaluated and updated whenever an input operand changes value. Continuous assignments use the keyword `assign`, whereas procedural assignments have the form <*reg variable*> `=` < *expression*>, where the <*reg variable*> must be a register or memory. Procedural assignments may only appear in initial and always constructs.

The Verilog language has two forms of the procedural assignment statement: blocking and non-blocking. The two are distinguished by the `=` and `<=` assignment operators. The blocking assignment statement (`=` operator) acts much like that in traditional programming languages. The whole statement is done before control passes on to the next statement. The non-blocking (`<=` operator) evaluates all the right-hand sides for the current time unit and assigns the left-hand sides at the end of the time unit.

The primary data type of Verilog to model digital hardware is for modeling registers (`reg`) and wires (`wire`). The `reg` variables store the last value that was procedurally assigned to them, whereas the `wire` variables represent physical connections between structural entities such as gates. A `wire` does not store a value. A `wire` variable is really only a label on a wire.

In addition to modeling hardware, there are other uses for variables. Verilog has several data types that do not have a corresponding hardware realization. These data types include `integer`, `real` and `time`. The data types `integer` and `real` behave pretty much as in other languages (e.g., C). Be warned that a `reg` variable is unsigned and that an `integer` variable is a signed 32-bit integer. This has important consequences when you subtract.

Verilog has a rich collection of control statements which can be used in the procedural sections of code, i.e., within an `initial` or `always` block. Most of them will be familiar to the programmers of traditional programming languages like C or Pascal. The main difference is that instead of C { } brackets, Verilog uses `begin` and `end`. In Verilog, the { } brackets are used for concatenation of bit strings.

Finally, the Verilog language provides two types of explicit timing control over when simulation time procedural statements are to occur. The first type is a delay control in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The second type of timing control is the event expression, which allows statement execution.

2.1.3 SystemC

SystemC [5] is a new modeling language based on C++ that is intended to enable system level design and Intellectual Property(IP) exchange. The emergence of the system-on-chip era is creating many new challenges at all stages of the design process. At the systems level, engineers are reconsidering how designs are specified, partitioned and verified. Today, with systems and software engineers programming in C/C++ and their hardware counterparts working in hardware description languages such as VHDL and Verilog, problems arising from the use of different design languages, incompatible tools and fragmented tool flows are becoming common. The SystemC standard is controlled by a steering group composed of thirteen major companies in the EDA and electronics industries. For the past year the technical members of this organization have been focusing on developing system level modeling extensions for SystemC. SystemC was developed as a standardized modeling language intended to enable system level design and IP exchange at multiple abstraction levels for systems containing both hardware and software components. The source code for the SystemC reference simulator can be freely downloaded from web site (`http://www.systemc.org`) under an Open Community Licensing agreement.

SystemC is a C++ class library and methodology that can be used to effectively create a cycle-accurate model of software algorithms, hardware architecture and interfaces of your SoC and system-level designs. It is possible to use SystemC and standard C++ development tools to create a system-level model, quickly simulate to validate and optimize the design, explore various algorithms and provide the hardware and software development team with an executable specification of the system. An executable specification is essentially a C++ program that exhibits the same behavior as the system when executed. C or C++ is the language choice for software algorithm and interface specifications because they provide the control and data abstractions necessary to develop compact and efficient system descriptions. Most designers are familiar with these languages and the large number of development tools associated with them.

The SystemC class library provides the necessary constructs to model system architecture including hardware timing, concurrency and reactive behavior that are missing in standard C++. Adding these constructs to C would require proprietary extensions to the language, which is not an acceptable solution for the industry. The C++ object-oriented programming language provides the ability to extend the language through classes, without adding new syntactic constructs. SystemC provides these necessary classes and allows designers to continue to use the familiar C++ language and development tools.

There are many benefits to creating an accurate executable specification of your complex system at the beginning of your design flow. These benefits are expressed as follows:

- An executable specification avoids inconsistency and errors and helps ensure completeness of the specification. This is because in creating an executable specification, you are essentially creating a program that behaves the same way as the system. The process of creating the program unearths inconsistencies and errors, and the process of testing the program helps ensure completeness of the specification.
- An executable specification ensures unambiguous interpretation of the specification. Whenever implementers are in doubt about the design, they can run the executable specification to determine what the system is supposed to be doing.
- An executable specification helps validate system functionality before implementation begins.
- An executable specification helps create early performance models of the system and validates system performance.
- The testbench used to test the executable specification can be refined or used as is to test the implementation of the specification. This can provide tremendous benefits to implementers and drastically reduce the time for implementation verification.

SystemC supports hardware-software co-design and the description of the architecture of complex systems consisting of both hardware and software components in a C++ environment. The following features of SystemC version 2.2 allow it to be used as a co-design language:

- *Modules*: SystemC has a notion of a container class called a module. This is a hierarchical entity that can have other modules or processes contained in it.
- *Processes*: Processes are used to describe functionality. Processes are contained inside modules. SystemC provides three different process abstractions to be used by hardware and software designers.
- *Ports*: Modules have ports through which they connect to other modules. SystemC supports single-direction and bidirectional ports.
- *Signals*: SystemC supports resolved and unresolved signals. Resolved signals can have more than one driver (a bus), while unresolved signals can only have a single driver.
- *Rich set of port and signal types*: To support modeling at different levels of abstraction, from the functional to the RTL, SystemC supports a rich set of port and signal types. This is different from languages like Verilog that only support bits and bit-vectors as port and signal types. SystemC supports both two-valued and four-valued signal types.
- *Rich set of data types*: SystemC has a rich set of data types to support multiple design domains and abstraction levels. The fixed precision data types allow for fast simulation, the arbitrary precision types can be used for computations with large numbers, and the fixed-point data types can be used for DSP applications. SystemC supports both two-valued and four-valued data types. There are no size limitations for arbitrary precision SystemC types.
- *Clocks*: SystemC has the notion of clocks (as special signals). Clocks are the timekeepers of the system during simulation, and the system supports multiple clocks, with arbitrary phase relationship.
- *Cycle-based simulation*: SystemC includes an ultra light-weight cycle-based simulation kernel that allows high-speed simulation.
- *Multiple abstraction levels*: SystemC supports untimed models at different levels for abstraction, ranging from high-level functional models to detailed clock cycle accurate RTL models. It supports iterative refinement of high-level models into lower levels of abstraction.
- *Communication protocols*: SystemC provides multi-level communication semantics that enable you to describe SoC and system I/O protocols with different levels for abstraction.
- *Debugging support*: SystemC classes have run-time error checking that can be turned on with a compilation flag.
- *Waveform tracing*: SystemC supports tracing of waveforms in VCD, WIF and ISDB formats.

The SystemC design approach offers many advantages over the traditional approach for system level design. The SystemC design methodology for hardware is shown in Figure 2.9. This technique has a number of advantages over the current design methodology, including the following:

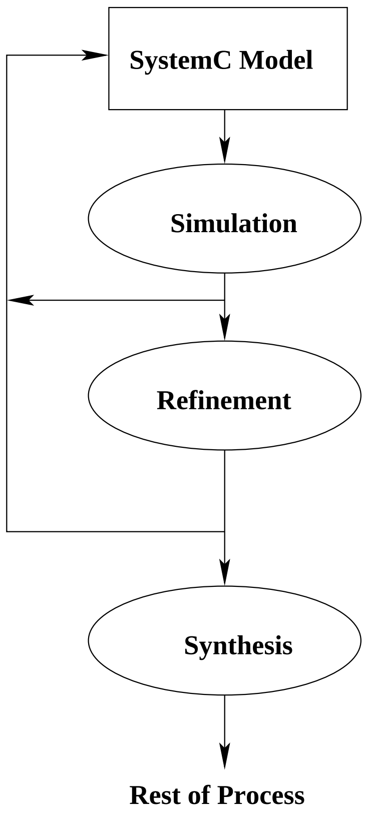


Fig. 2.9. SystemC modeling technique.

- *Refinement methodology.* With the SystemC approach, the design is not converted from a C level description to an HDL in one large effort. The design is slowly refined in small sections to add the necessary hardware and timing constructs to produce a good design. Using this refinement methodology, the designer can more easily implement design changes and detect bugs during refinement.
- *Written in a single language.* Using the SystemC approach, the designer does not have to be an expert in multiple languages. SystemC allows modeling from the system level to RTL, if necessary.

The SystemC approach provides higher productivity because the designer can model at a higher level. Writing at a higher level can result in smaller code, that is easier to write and simulates faster than traditional modeling environments.

Testbenches can be reused from the system level model to the RTL model, which saves conversion time. Using the same testbench also gives the designer higher confidence that the system level and RTL model implement the same functionality.

Modules are the basic building block within SystemC to partition a design. They also allow designers to break complex systems into smaller more manageable pieces. Modules help split complex designs among a number of different designers in a design group. Modules allow designers to hide internal data representation and algorithms from other modules. This forces designers to use public interfaces to other modules, and the entire system becomes easier to change and easier to maintain. For example, a designer can decide to completely change the internal data representation and implementation of a particular module. However, if the external interface and internal function remain the same, the users of the module do not know that the internals were changed. This allows designers to optimize the design locally.

Modules are declared with the SystemC keyword `SC_MODULE` and they can contain a number of other elements such as ports, local signals, local data, other modules, processes and constructors. These elements implement the required functionality of the module. Module *ports* pass data to and from the processes of a module. Port mode are declared as `in`, `out`, or `inout`. The data type of the port is also declared as any C++ data type, SystemC data type or user defined type.

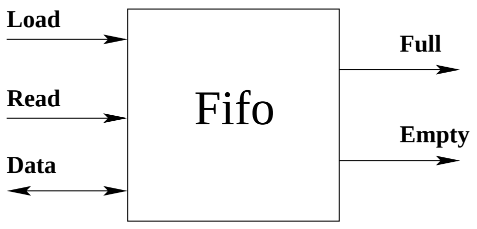


Fig. 2.10. Example of a module.

Figure 2.10 shows a FIFO module with a number of ports. The ports on the left are input ports or inout ports, while the ports on the right are output ports. Each port has an identifying name. Graphic symbols like the one shown above typically do not contain port types, so it is not clear from the symbol which port types are present. The SystemC description of this module is shown in Figure 2.11. Each port on the block diagram has a matching port statement in the SystemC description. Port modes `sc_in`, `sc_out`, and `sc_inout` are predefined by the SystemC class library.

Signals can be local to a module and are used to connect ports of lower level modules. These signals represent the design's physical wires that interconnect devices on the physical implementation. Signals carry data, while ports determine the direction of data between module. Signals are not declared with a mode such as in, out, or inout. The direction of the data transfer is dependent on the port modes of the connecting components. The local signals are declared using the SystemC template class `sc_signal`. The type of signal being passed is entered between the angle brackets (`<>`). In the example the type of signal is a SystemC data type `sc_uint`. Notice that there is an extra space inserted between the `32>` and the `>`

```
SC_MODULE(fifo) {
  sc_in<bool>        load;
  sc_in<bool>        read;
  sc_inout<int>      data;
  sc_out<bool>       full;
  sc_out<bool>       empty;
  ...
  sc_signal<sc_uint<32> > q, s;
  ...
  SC_CTOR(fifo) {
    ...
  }
  ...
}
```

Fig. 2.11. A SystemC description for the module of Figure 2.10.

in the declaration. This is required to allow the description to compile. The three modules in this design are instantiated in the constructor `SC_CTOR`.

The real work of the modules is performed in processes. Processes are functions that are identified to the SystemC kernel and called whenever signals that these processes are "sensitive to" change value. A process contains a number of statements that implement the functionality of the process. These statements are executed sequentially until the end of the process occurs, or until the process is suspended by one of the wait function calls.

Processes look very much like normal C++ methods and functions with slight exceptions. Processes are methods that are registered with the SystemC kernel. There are a number of different types of processes including method processes, thread processes and clocked thread processes. The process type determines how the process is called and executed. Processes can contain calls to a function named `wait()` that will halt their execution at different points. Signal value changes cause the process to receive events and execute statements in a process.

The module constructor `SC_CTOR` creates and initializes an instance of a module. The constructor creates the internal data structures that are used for the module and initializes these data structures to known values. The module constructors in SystemC are implemented such that the instance name of the module is passed to the constructor at instantiation (creation) time. This helps identify the module when errors occur or when reporting information from the module.

SystemC provides the designer with the ability to use any and all C++ data types as well as unique SystemC data types to model systems. The SystemC data types include the following:

- `sc_bit` 2 value single bit type
- `sc_logic` 4 value single bit type
- `sc_int` 1 to 64 bit signed integer type
- `sc_uint` 1 to 64 bit unsigned integer type
- `sc_bigint` arbitrary sized signed integer type
- `sc_biguint` arbitrary sized unsigned integer type

- `sc_bv` arbitrary sized 2 value vector type
- `sc_lv` arbitrary sized 4 value vector type
- `sc_fixed` templated signed fixed point type
- `sc_ufixed` templated unsigned fixed point type
- `sc_fix` untemplated signed fixed point type
- `sc_ufix` untemplated unsigned fixed point type

Many operations and conversions are also implemented between these and standard data types.

SystemC simulation is cycle-based: processes are executed and signals are updated at clock transitions. The SystemC library includes a cycle-based scheduler that handles all events on signals, and it schedules processes when the appropriate events occur at their inputs. SystemC simulation follows the evaluate-update paradigm where all processes that are ready to be executed are executed, and only then their output signals are updated.

The scheduler in SystemC executes the following steps during simulation:

1. All clock signals that change their value at the current time are assigned their new values.
2. All `SC_METHOD/SC_THREAD` processes with inputs that have changed are executed. The entire body of `SC_METHOD` function processes are executed, while `SC_THREAD` processes are executed until the next `wait()` statement suspends execution of the process. `SC_METHOD/SC_THREAD` processes are not executed in a fixed order.
3. All `SC_CTHREAD` processes that are triggered have their outputs updated and are saved in a queue to be executed later in step 5. All outputs of `SC_METHOD/SC_THREAD` processes that were executed in step 1 are also updated.
4. Steps 2 and 3 are repeated until no signal changes its value.
5. All `SC_CTHREAD` processes that were triggered and queued in step 3 are executed. There is no fixed execution order of these processes. Their outputs are updated at the next active edge (when step 3 is executed), and therefore they are saved internally.
6. Simulation time is advanced to the next clock edge and the scheduler goes back to step 1.

If processes communicate using signals, the process execution order should not affect the simulation results. However, if global variables and pointers are used, process execution order affects the simulation results. Note that these simulation semantics are similar to Verilog simulation semantics with deferred signal assignments and VHDL simulation semantics.

Testbenches are used to provide stimulus to a design under test and the check design results. The testbench can be implemented in a number of ways: The stimulus can be generated by one process and results checked by another; or the stimulus can be embedded in the main program and results checked in another process.

2.1.4 Implicit Modeling

Generally digital systems are described explicitly by using hardware description languages, as the ones presented in the previous sections of this chapter. This is principally due to the convenience of having an executable or simulatable description of the design that allows a rapid prototyping of its functionalities. A not less relevant aspect for the adoption of HDL is the availability of many synthesis tools that allow to efficiently convert a HDL-based high-level representation of the system in one at a lower abstraction level, up to the gate-level. However, for verification purposes, implicit representations of the system can be more suited than HDL descriptions. Next paragraphs summarize some of the most popular formalism used to implicit represent digital systems.

Conjunctive Normal Form

The functionalities of a circuit can be represented as a boolean function. Boolean variables and functions assume only two possible values: 0 stands for *false*, 1 stands for *true*. A *literal* is either a boolean variable (for instance, v_1) or its complement ($\bar{v}_2$). A *clause* C_i is a set of literals $\{l_{k_i}\}$ (for instance, $C_1 = \{v_1, \bar{v}_2\}$). A Conjunctive Normal Form (CNF) is a boolean formula used to represent a boolean function f on a set of boolean variables $\{v_j\}$ and it is defined as a conjunction (logical *AND*, $\cdot$) of the clauses $\{C_i\}$ each of which is interpreted as the disjunction (logical *OR*, $+$) of its literals. For example the boolean function $f(v_1, v_2, v_3, v_4)$ represented by the formula $v_1v_2v_3 + \bar{v}_1v_4 + \bar{v}_2v_4$ is expressed in CNF, where the set of clauses is $\{C_1, C_2, C_3\}$ and $C_1 = \{v_1, v_2, v_3\}$, $C_2 = \{\bar{v}_1, v_4\}$, $C_3 = \{\bar{v}_2, v_4\}$.

Conjunctive normal form is the classical formalism accepted by SAT-solvers.

Binary Decision Diagrams

Binary Decision Diagrams (BDD) [6] are another formalism frequently used to implicitly represent digital systems. A binary decision diagram is a Directed Acyclic Graph (DAG); the root node of the DAG identifies the function, f, represented by the BDD, the internal nodes are labeled with the variables belonging to the true support of f (i.e. the set of variables on which f actually depends), and the terminal nodes are labeled with the values 0 and 1. As an example, the BDD for the function $f(v_1, v_2, v_3, v_4) = v_1v_2v_3 + \bar{v}_1v_4 + \bar{v}_2v_4$ is given in Figure 2.12.

A particular kind of BDD are the Reduced Ordered Binary Decision Diagrams (ROBDD). They do not contain duplicated and redundant nodes, and in addition, they are ordered, that is, all the variables appear in the same order along all paths from the root to the terminal nodes. Given an ordering, the reduced BDD for a function is unique. Hence, BDD are canonical representations, that is, two functions f and g are equivalent (i.e. $f = g$) if and only if they have the same BDD.

Assignment Decision Diagram

An Assignment Decision Diagram (ADD) [7] can be considered as an extension of the classical graph that represents the data path of an RTL description. In this structure it is possible to identify the read and the write nodes, the operator nodes, and the assignment decision nodes. An example of ADD is shown in Figure 2.13.

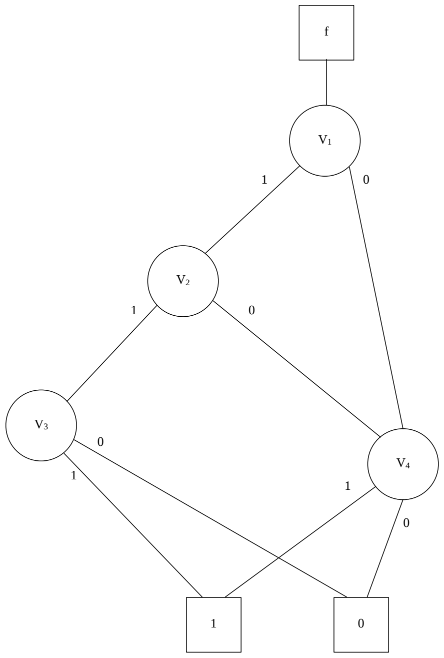


Fig. 2.12. A BDD for function $f(v_1, v_2, v_3, v_4) = v_1v_2v_3 + \bar{v}_1v_4 + \bar{v}_2v_4$.

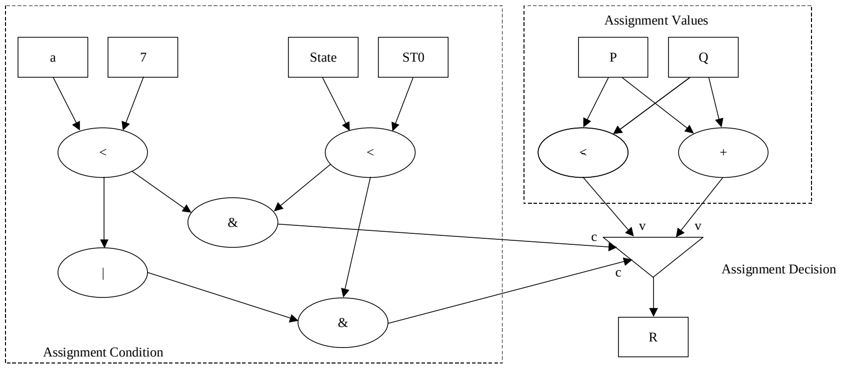


Fig. 2.13. An Assignment Decision Diagram.

The critical point is the assignment decision node: it can be identified an assignment value part, which is constituted by the set of read nodes and operation nodes that represent the computation of values that are to be assigned to a storage unit or output port; and an assignment condition part which consists of read nodes and operation nodes that produce the boolean value which is the guarding condition for the assignment value.

The assignment decision part consists of an assignment decision node that selects a value from the set of values that are provided at its value inputs. If one of the conditions to the assignment decision node evaluates to true, then the corresponding input value is selected.

The assignment target is represented by the write node. The write node is associated with the selected value from the corresponding assignment decision node. A value is assigned to the write node only if one of the condition inputs to an assignment decision node evaluates to true. Since only one value can be assigned to a target at a time, all assignments conditions for a given target are mutually exclusive. The interesting observation is in the fact that an ADD tries to model the influence of input ports and variables over the assignment operation, considering in a distinct way the effects over output and conditional operations.

Finite State Machine

A Mealy-type Finite State Machine (FSM) M is a model of computation defined as the 6-tuple: $M = \left\{I, O, S, s^0, \delta, \lambda\right\}$ where:

- I is the input alphabet;
- O is the output alphabet;
- S is a set of states;
- $s^0 \in S$ is the unique reset state;
- $\delta : I \times S \to S$ is the next-state function;
- $\lambda : I \times S \to O$ is the output function.

In the Moore machine, the output value depends only on the current state and it does not depend on the current input values. In this case the λ function is defined as: $\lambda : S \to O$.

A FSM, M, can also be represented by a State Transition Graph (STG). Every vertex of such a graph corresponds to a state of M, and it is labeled with an element of S, while every edge corresponds to a transition, and it is labeled with an element of $I \times O$.

A structural representation of the FSM model is the Huffman model (Figure 2.16), that separates the next state and the output functions from the memory elements that store the state of the FSM.

The description through the FSM formalism of some classes of sequential circuits requires a number of states that become intractable very fast and therefore different representations are adopted, such as the FSM + Data Path model [8] or the Extended FSM model [9].

A *Finite State Machine with Data-path* (FSMD) is a finite state machine with an elaboration unit (data-path).
Given:

- the set $V = \{v_1, v_2, \ldots, v_v\}$ of variables;
- the set Exp =$\{f(v_1, v_2, \ldots, v_v) | v_1, v_2, \ldots, v_v \in V\}$ of functions;
- the set $Asg = \{(v_1, v_2, \ldots, v_v) \leftarrow e | (v_1, v_2, \ldots, v_v) \in V, e \in Exp\}$ of elements that can be associated with V;

- the set $Stat = \sigma(a, b)|a, b \in Exp$ of state variables defined as a logic relation between two functions of the Exp set;

a finite state machine with a computation unit, FSMD, is defined by a 6-tuple $FSMD = \{S, I \cup B, O \cup A, \delta, \lambda, s_0\}$ where:

- S is a set of states;
- I is the input alphabet;
- B is a subset of $Stat$;
- O is the output alphabet;
- A is a subset of Asg;
- $\delta : I \times S \rightarrow S$ is the transition function;
- $\lambda : I \times S \rightarrow O$ is the output function;
- $s^0 \in S$ is the initial state.

The *Extended Finite State Machine* (EFSM) is a generalization of the classical FSM model that provides a compact representation of local data variables and preserves many nice properties of the traditional state machine model. The EFSM can be defined as a 5-tuple $EFSM = \{S, I, O, D, T\}$ where:

- S is a set of states;
- I is a set of inputs;
- O is a set of outputs;
- D is an n-dimensional linear space $D_1 \times D_2 ... \times D_n$;
- T is a transition relation, $T : S \times D \times I \rightarrow S \times D \times O$.

The model differs from the classical FSM model, since each transition from a state to another one does not present only the label with the required input and the associated produced output, in the classical form $(i)/(o)$, but it is more complex.

A transition $(s_1 \rightarrow s_2)$ with label $(f, i)/(u, o)$ can be interpreted as: if the machine E is in configuration $\langle s_1, x\rangle$, where $f(x) = 1$ (the transformation u can be applied to the value x, that represents an internal variable), and the input i is received, then E moves to the configuration $\langle s_2, u(x)\rangle$ while generating output o. This transition can be realized only if the vector x satisfies some particular properties about the applicability of the transformation $u(x)$, and this the function f is introduced. Each EFSM can be represented using a graph that is called extended state transition graph.

Kripke Structure

Formal models of concurrent systems can be represented by Kripke structures. Given a formula of first order logic, that represents a concurrent system, it is straightforward to extract the Kripke structure that models the system [10]. Let AP a set of atomic propositions, a *Kripke structure* K over AP is a 4-tuple $K = \{S, S_0, R, L\}$ where:

- S is a finite set of states;
- $S_0 \subseteq S$ is the set of initial states;

- $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s_1 \in S$ such that $R(s, s_1)$;
- $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

2.2 Embedded Systems: Validation

Verification of embedded system can be performed by using two different strategies: *static verification*, presented in Section 2.2.1, and *dynamic verification*, described in Section 2.2.2. While static verification consists of using formal methods to ensure the correctness of a design, dynamic verification relies on simulation. Both techniques present different advantages, but neither formal verification nor dynamic verification can assure the guarantee of shipping bugs-free systems. Recently, to join advantages of both techniques, great interest comes from assertion-based verification which unifies the mathematical rigor of formal techniques and the intuitive and fast approach of simulation and coverage metrics.

2.2.1 Static Verification

Static verification, also noted as formal verification, applies the formalism of mathematical proofs to state about the correctness of the Design Under Validation (DUV). It is static, in contrast to dynamic verification, because it verify the presence of errors without the need of explicitly simulate the behavior of the model.

A formal verification framework has three basic elements: a mathematical model of the system to be verified, a formal language to framing the correctness problem, and a methodology for proving the statement of correctness. Depending on these three factors we can distinguish between two main formal verification mechanism, that apply with different intent in the EDA field: *model checking* and *equivalence checking*. For model checking the correctness problem consists of showing if the model satisfies the specification represented as logic formulas. Its main application is to find design errors in the early stage of the design flow. On the contrary, equivalence checking address the problem of verifying if two models implements the same functionalities. Its main application is to find discrepancies between descriptions of the same system at different abstraction levels.

Model Checking

Model checking was introduced by Clarke and Emerson [11] and independently by Quielle and Sifakis [12] in 1981. It is an automatic technique for verifying finite state concurrent systems that instead of proving the validity of a logical formula for all models, it determines the truth value of the formula in a specific finite model.

Applying model checking to a design consists of three tasks:

- ***Modeling.*** The DUV has to be converted into a formalism accepted by a model checking tool. From a formal point of view, the designs are represented by using Kripke structures, but practically speaking the designs can be described by using traditional HDL or other specific model checker dependent languages.

- ***Specification.*** Before verification, it is necessary to define the properties that the design must satisfy. This specification is usually given by using temporal logics. However, nowadays the advent of assertion-based verification allow one to specify properties by using more HDL-like languages (e.g., PSL, OVA, PEC, cf.
- ***Verification.*** An automatic tool (model checker) is used to show if the model satisfies the specification. Given sufficient resources in terms of time and memory space, the automatic procedure always terminates answering "yes" or "no".

Formally, the model checking problem can be described as follows: given a Kripke structure $K = \{S, S_0, R, L\}$ that represents a finite state concurrent system and a temporal logic formula φ expressing some desired specification, find the set S' of all states in S that satisfy φ: $S' = \{s \in S \mid K, s \vDash \varphi\}$. Normally, some states of the concurrent system, S_0, are designated as initial states. The system satisfies the specification provided that all of the initial states are in the set S'.

For branching time logic, as CTL, the model checking problem is computationally tractable. The algorithm developed by Clarke and Emerson for CTL model checking is polynomial in both the size of the model and in the length of its temporal logic specification. The method first builds a complete state transition graph of the system, then the truth value of a property is determined by propagating formulas in this graph until a fixed point is reached.

For LTL, the model checking problem is PSPACE-complete [13]. LTL model checking is performed by translating the formula to be verified into an automaton by means of a *tableau* construction [11]. A tableau is a graph derived from the formula from which a model for the formula can be extracted if and only if the formula is satisfiable. Each state in a tableau is associated with a set of formulas which are true in that state. Since the number of states in the tableau is exponential in the size of the formula, the method is not really practical. However, Lichtenstein and Pnueli [14] realized the complexity of checking LTL formulas and discovered that although the complexity appears exponential in the length of the formula, it is linear in the size of the state graph. Based on this observation, they argued that the high complexity of linear time model checking can be still acceptable for short formulas.

Also CTL* model checking is a PSPACE-complete problem [15]. The basic idea for CTL* model checking is to combine the state labeling technique from CTL model checking with LTL model checking.

The explicit representation of the model by means of a state transition graph is the main problem that prevents to apply model checking to systems with many concurrent parts where the number of states in the graph becomes too large to handle. To avoid this limitation, McMillan [16] realized that, by using an implicit symbolic representation of the state transition graph, much larger systems can be verified.This symbolic representation is based on BDD, and it allows model checker to operate on sets of states and transitions rather than on individual states and transitions. Each state is encoded by an assignment of boolean values to the set of state variable associated to the system. Thus, the transition relation can be expressed as a boolean formula in terms of two sets of variables, one set encoding the old state and the other encoding the new state. This formula is then represented by a BDD. Given a Kripke structure $K = \{S, S_0, R, L\}$, and the lattice under the set inclusion ordering of all subsets of S [11], the *symbolic model checking* algorithm is based on computing fixpoints of particular functions, called predicate transformers, which map $\mathcal{P}(S)$ to $\mathcal{P}(S)$.

Because the symbolic representation captures some of the regularity in the state space determined by the DUV, it is possible to verify systems with a number of states whose magnitude is many orders larger than explicit state algorithms can do. By using the original CTL model checking algorithm of Clarke and Emerson, it is possible to verify examples with up to 10^5 states. On the contrary, symbolic model checking allows one to handle designs with up to 10^{120} states. In conclusion, the main advantages of model checking with respect to other approaches can be summarized as follows:

- ***Easy to learn and use.*** Model checking is fully automatic, and its application requires no user supervision or expertise in mathematical disciplines such as logic and theorem proving;
- ***Easy to catch bugs.*** When the design fails to satisfy a property, model checking always produces a counterexample that demonstrates a behavior which falsifies the property. This trace allows one to "easily" understand and fix the problem. In such a sense, model checking is a very effective technique to detect design errors.

On the contrary, the main drawbacks of model checking are:

- ***State explosion problem***. Even if the advent of model checkers based on BDD symbolic representation sensibly increases the applicability of model checking to medium-large systems, state explosion is still a problem for complex digital systems where a great number of parallel transitions can occur. To overcame this problem, some techniques, as decomposition, abstraction, symmetry, compositional minimization, etc. must be adopted. However, these techniques risk to make the adoption of model checking less easy and intuitive for users lacking a deep formal background.
- ***Accuracy of verification results***. As already mentioned in the introduction of this thesis, model checking can prove the presence of bugs, but it cannot prove it absence. In fact, properties proven by model checking can be inadequate to assure an effective correctness of the DUV. The problem of accuracy of the model checking process is the main topic of this dissertation.

Equivalence Checking

Equivalence checking is a formal verification process that mathematically proves the equivalence of two different models of the design under validation.

Generally, equivalence checking is used at a lower abstraction level with respect to model checking, and it is more suited to detect errors that depend on circuit transformations rather than design errors.

Commonly, equivalence checking compares two models of the same digital system described at two different abstraction levels. For example, a popular use of equivalence checking is to verify that a netlist obtained by synthesis correctly implements the original RTL code. In fact, synthesis tools are large software systems history shows to be prone to error.

A different use of equivalence checking consists of comparing two netlists to ensure that some netlist post-processing, such as scan-chain insertion, clock-tree synthesis or even manual modifications, did not change the functionality of the circuit.

The most successful equivalence checkers [17, 18, 19] try to make full use of structural similarity of circuits to be compared. In particular, they try to establish some strong relationships (like equivalence or implication) between internal points of the circuits traversing their FSM representations. These relationships are deduced in topological order proceeding from inputs to outputs until the equivalence of corresponding primary outputs of the two circuits is deduced. Circuits are considered to be structurally similar and so easy for equivalence checking if they have many internal points that are related by these strong relationships.

As for model checking, BDD provide a very efficient formalism also to perform equivalence checking.

Equivalence of two combinational circuits [1] may be verified by following the following basic algorithm: “for each circuit, build the BDD for the outputs in terms of the primary inputs; since BDD are a canonical representation, the two combinational circuits implement the same function if and only if they have the same BDD”.

Equivalence of sequential circuits is performed reasoning about sequential circuits as finite state machines, rather than as just a bunch of gates. The problem of comparing two state machines can be converted into the problem of finding all of the reachable states of FSM. Given two FSM to compare, tie the input lines together, send the outputs to a comparator, and clock the two machines together in lockstep. This combination is just another, bigger FSM. The original two FSM have identical behavior if and only if the new machine indicates the outputs are equal for all reachable states.Computing the set of reachable states using BDD requires three basic ideas:

- representing sets of states using BDD;
- computing images, where the image of that BDD is a new BDD that represents the set of all possible states that the FSM could be in exactly one clock tick later;
- iterating by using images to compute all reachable states.

As with combinational verification, this approach for sequential equivalence is limited by the size of the generated BDD, which is highly sensitive to the function being verified and the variable order used. Thus, many valuable papers on equivalence checking aim to define efficient algorithms for the traversal of FSM [20, 21, 22, 23].

2.2.2 Dynamic Verification

A widely adopted alternative to formal verification, is represented by dynamic verification [24] (or simulation-based verification) that faces the correctness of a design by means of simulation-based techniques. In dynamic verification the model

[1] A combinational circuit is a digital circuit without state-holding elements or feedback loops, so the output is a function of the current input. A circuit with state-holding elements is called a sequential circuit.

functionalities are essentially verified by generating a high number of input stimuli (test set) that are simulated to observe the behavior of the DUV at primary outputs. The test set, generated at a specific abstraction level, can be re-used (and possibly incremented) at the lower levels after each synthesis step up to manufacturing test.

What we need to perform dynamic verification is: a HDL description of the design, a simulator for the selected HDL, a testbench to apply stimuli (i.e., vectors for combinational circuits, and sequence of vectors for sequential circuits) to the primary input of the design, and a “method to establish the correctness” of the design with respect to the results of the simulation. While the first three ingredients are almost straightforward, the last one is the crucial aspect of dynamic verification. If state explosion is the big problem of formal verification, the big problem of dynamic simulation is represented by the lack of exhaustiveness of the verification process based on simulation. Thus, as it happens for formal verification, dynamic verification is very good in finding bugs, but it cannot ensure their absence.

Simulation can hypothetically provide an exhaustive answer to the problem of design correctness only if the set of all possible input stimuli applied to the design results, after design simulation, in a set of values for the primary outputs consistent with the set of expected values. Unfortunately, that is almost impossible for two reasons: the set of expected values cannot be always available, and the set of input stimuli for sequential circuits is infinite. Thus, the quality of dynamic verification, and in particular the quality of the generated set of stimuli, is measured by means of *code coverage* or *fault coverage*. Depending on what of these two strategies is used, we can distinguish between two kind of dynamic verification: *logic simulation* and *fault simulation*.

Logic Simulation

In logic simulation the quality of the set of stimuli is measured by using code coverage. This is a class of metrics that has been used in software engineering [25] for quite some time to analyze if testbenches forgot to verify some functionalities.

Code coverage tools work in the following manner. First the source code must be instrumented to add checkpoints at strategic locations to records whether a particular construct has been exercised. The instrumentation method varies from tool to tool. Some may use file I/O features available in the language, others may use special feature built into the simulator. The instrumented code is then simulated using all available, not instrumented, testbenches, the cumulative traces from all simulations are collected, and reports are generated to determine various coverage metrics.

The most popular metrics adopted in logic simulation are:

- ***Statement Coverage.*** It measures how much of the total lines of code are executed by the test set. To bring the statement coverage metric up to 100%, a desirable goal, it is necessary to understand what conditions are required to cause the execution of the uncovered statements. Then, it is necessary to understand why they never occurred. It is because the test set does not contain a stimulus able to activate the condition or it is because the condition can never occur? In the first case, a larger number of (or higher quality) test cases must be generated. On the contrary, if the condition can never occur, the code in question is effectively unreachable. Thus, either the code (and the condition) could be removed, or the design requires some most general refinement to allow the activation of the condition.

- ***Condition Coverage.*** It measures the various ways paths through the code are executed. Consider for example an `if` statement whose condition is `((a < 10) or (a > 20))`. The `then` part of such a statement can be executed in two ways: when the value of `a` is less than 10 (first term of the condition) and when the value of `a` is greater than 20 (second term of the condition). Thus, it is evident that the statement coverage of a code can be 100%, while the condition coverage is lower. To increase condition coverage, it is necessary to identify the possible terms of conditions that are not executed, and if these terms can never be excited or they cannot be activated by the current test set.
- ***Path Coverage.*** It measures all possible ways you can execute a sequence of statements. Again it is important to determine the possible conditions that cause the uncovered path to be executed, and if these conditions can never occur or they cannot be activated by the current test set. Full 100% path coverage is very difficult to achieve, since the number of path in a sequence of statements grow exponentially with the number of control-flow statements.

What does 100% code coverage means? Not much, it indicates how thoroughly the generated test set exercises the source code, but it does not provide precise indications about the correctness of the DUV. However, code coverage can help to identify possibly corner cases that are not exercise by the testbench, and that can be symptoms of design errors.

Fault Simulation

Fault simulation consists of simulating a design in presence of logical faults (faults in the following), which try to emulate the effect of physical faults on the behavior of a system description. Faults can be modeled by means of different fault models, targeting various kinds of errors that may affect a design, depending on the considered abstraction level.

Comparing the fault simulation results with those of the fault-free simulation of the same design, simulated by using the same test set, we can determine the *fault coverage* as the ratio between the number of fault detected by the test set and the number of simulated faults. In that sense, the fault coverage is an alternative measure to evaluate the quality of dynamic verification. Indeed, as for code coverage, we cannot completely ensure the correctness of the design by relying on the fault coverage. In fact, even a test set which achieve 100% fault coverage may still fail to detect faults modeled by a different fault model. Furthermore a not full fault coverage highlight the presence of undetected fault that may be either hard-to-test, requiring to improve the test set, or undetectable, requiring a more accurate analysis to avoid the presence of design errors.

The following paragraphs overview the basic concepts of test generation algorithms and addresses the related state of the art, describing some of the most relevant existing approaches to test pattern generation. Some of the more interesting fault models proposed in the literature to simulate possible erroneous behaviors of digital systems and the most common techniques used to activate faults during fault simulation and test pattern generation are described. Section 7.1 is entirely devoted to the bit coverage fault model which is adopted in the methodology presented in this thesis.

2.2.3 Test Generation

A defect is an error introduced into a device during the manufacturing process. A fault model is a mathematical description of how a defect alters design behavior. A fault is said to be detected by a test pattern if, when applying the pattern to the design, any logic value observed at one or more of the circuit's primary outputs differs between the original design and the design with the fault.

The test generation process for a targeted fault consists of two phases: *fault activation* and *fault propagation*.

Fault activation establishes a signal value at the fault model site that is opposite of the value produced by the fault model.

Fault propagation moves the resulting signal value, or fault effect, forward by sensitizing a path from the fault site to a primary output.

The test generation process can fail to find a test for a particular fault in at least two cases. First, the fault may be intrinsically undetectable, such that no patterns exist that can detect that particular fault. The classic example of this is a redundant circuit, designed so that no single fault causes the output to change. In such a circuit, any single fault will be inherently undetectable.

To create the test set, fault simulation tools commonly exploit stimuli generator engines known as *Automatic Test Pattern Generator* (*ATPG*). The aim of test pattern generation algorithms is to find a possibly compact set of input sequences that detects all modeled faults. To accomplish this goal an ATPG works as follows:

1. it selects a target fault;
2. it generates an input sequence;
3. it simulates the fault-free design and the faulty design by applying such sequence;
4. it marks the target fault as detected if and only if, at some instance during simulation, there is a difference between outputs of the fault-free design and the corresponding outputs of the faulty design;
5. for efficiency aspects, it fault simulates the same sequence for all of the other not yet detected faults;
6. it repeats steps from 1 to 5 up to one of the following conditions is reached:
   a) the desired fault coverage threshold is achieved;
   b) the desired fault coverage threshold is not achieved, but remaining faults are shown to be undetectable;
   c) available resources run out.

An ideal test generation algorithm should produce an effective test sequence with a low cost, in terms of required time and memory resources. The effectiveness of the test sequence is measured by the achieved fault coverage for the given fault model, and by the number of generated test cases. Finally, its cost is affected by the fault model used, the type of circuit under test, the level of abstraction of the circuit description, and the required test quality.

Justification, Implication and Backtracking

Several ATPG algorithms are based on the concepts of *justification*, *implication* and *backtracking* defined for gate-level descriptions.

The *justification* is the process of assigning values to gate inputs when the logic value of the gate output is known.

The values assigned during justification process can uniquely determine the values on some other signals, and this process is called *implication*; finally, due to the presence of re-convergent paths fanouts in the circuit, the assignment of new signal values during implication can conflict with values of these signals assigned in the earlier steps of the test generation.

If this happens, the effects of the last decision have to be reversed and a new decision has to be made to allow the ATPG process to continue. This process is called backtracking. The decision making and backtracking strategies have a strong impact on the efficiency of the ATPG process. However, the proposed pattern that is based on the phases of justification, implication and backtracking is only a general scheme, and there are a number of different ways to implement an algorithm that perform these operations both at gate or at higher levels of abstraction.

Automatic Test Pattern Generator (ATPG)

The most relevant contributions to functional test generation algorithms can classified in two macro-classes: *deterministic ATPG* and *(pseudo-)random ATPG*.

The first class can be divided in three further categories: ATPG based on finite state machines, ATPG based on controllability, observability and structural description of data paths, and ATPG based on assignment decision diagrams. These techniques make a selective exploration of the input sequences space and generate a limited number of high-quality test sequences. This generation is performed by explicitly exploiting information related to the design under validation. Even if a deterministic generation may require a great effort in terms of time, these approaches are particularly suited to investigate corner cases and random-resistant faults.

On the contrary, random based ATPG found their effectiveness on the great number of generated input sequences rather than on their quality. A pure random-based ATPG does not require any information about the DUV and it represents a simple and fast solution to cover the most part of easy-to-detect faults. However, this is not enough to guarantee the absence of design errors, thus random based ATPG are enhanced by using different kinds of heuristic and statistical techniques which guide the sequences generation by exploiting run-time information. In this case we talk of pseudo-random ATPG where genetic algorithm-based ATPG can be considered the most promising approach.

An extensive description of each of these different categories is in the following.

ATPG based on Finite State Machines

One of the most investigated classes of functional test generation algorithms, corresponds to those ATPG that rely on the identification of the control structure that drives the behavior of the system. In fact, given a high-level description of the circuit, no matter if it is realized in a classic HDL or in a C-based description, these methods extract the Finite State Machine associated with the description. Sometime the FSM is defined implicitly in the specification of the component under test, and in this case these algorithms try to automatically transform the high-level description into an appropriate data structure that can be represented as a FSM.

There are several examples of ATPG that use this approach [26,27,28,29], but one of the most important contributions on the application of such a technique is described in [9]. The algorithm presented in this work can be considered fundamental, because it outlines principles typically used in the world of algorithms based on FSM descriptions.

The different steps of the algorithm proposed in [9] are:

1. an EFSM model is automatically extracted from a VHDL or C description;
2. a stabilized EFSM model is derived from the initial EFSM model generated during the first step;
3. a functional test sequence, that guarantees that every statement in the VHDL or C model is exercised at least once, is derived from the stabilized model.

The work presents and accurately explains the algorithm used during the first two phases to obtain an automatic identification of the EFSM and for the stabilization of the obtained model, but the interesting part is the one devoted to the test vectors generation starting from the optimized EFSM. In order to exercise at least once all the statements in the hardware description, all the transitions have to be traversed in the EFSM model. After the stabilization process is completed, each transition in the final stabilized block transition graph originates from a transition in the initial EFSM model. Therefore, traversing all transitions in the stabilized block transition graph guarantees covering all transitions in the initial EFSM model and thus in the original HDL. In practice, the problem of finding a suitable test vector is reduced to the problem of finding the shortest path that covers all edges in a directed graph at least once, and this is a well known problem in the world of Operational Research. Applying a breadth first strategy, the complexity of this procedure is $O(e^2)$ where e is the number of transitions in the stabilized block transition graph.

A more recent and very efficient approach is proposed in [30]. The authors develop an ATPG that relies only on exploring embedded characteristics within the sequential circuit, in order to achieve very high fault coverage. Furthermore, these characteristics should be obtained with only logic-simulation to reduce the computation costs. To do this, they address the following issues:

1. state partitioning by extracting the spectral information of the states that the current test sequence has traversed;
2. state and transition coverage within each partitioned state set.

State partitioning and grouping are an inherent part of this work. It is simply dividing the flip-flops in a circuit into smaller groups. Essentially, state partitioning breaks the global state into groups of flip-flops. However, the way the flip-flops should be divided is a difficult problem. Since the quality of the partitioned space is important, methods in computing the partitions and groups are critical. The spectral information embedded in the states traversed from a given test sequence can help partition the state variables. In particular, the flip-flops whose state-transition behavior contain dominant frequencies are grouped together. In the frequency domain, these flip-flops are observed to have correlation with one another. Furthermore, disjoint partitioning may not be the optimal way to divide the flip-flops. In other words, state groups are allowed to share some common flip-flops. These overlapped state grouping can help the ATPG by observing how some specific flip-flops can contribute to different partitioned state spaces. Next, the STG for each partial state space is dynamically constructed during the ATPG search. By traversing the states and arcs inside each partial STG (similar to the method in finite state machine testing), the vectors generated help to exercise the circuits state subspaces. In terms of FSM testing, the idea is to map out the finite state machine embedded in the design and exercise the states and transitions within it. However, FSM testing remains to be a difficult problem when dealing with large sequential circuits with many flip-flops. In this work, since the number of flip-flops in each partitioned state set is limited to k, the largest number of states in any set is thus 2k, the author avoid the state explosion keeping the size of k to be less than 16. In addition, if the flip-flops that belong to the controller of the circuit can be identified and grouped, the extracted STG can give a view from the control-unit perspective, thereby providing additional guidance as vectors are generated. Finally, since only logic simulation is needed, the proposed technique can also contribute to generating effective simulation vectors for design validation. Experimental results show that higher fault coverages can be achieved with very low computational needs, when compared to ATPG that are based on logic simulation.

ATPG based on Controllability, Observability and Structural Description of Data Paths

The class of ATPG that are based on the importance of the concept of observability has been continuously growing during the last years, so that this approach to the problem is today the most interesting one for researchers [31, 32, 33]. The basic idea of all the ATPG based on observability is very simple and intuitive: check to see whether effects of possible faults activated by program stimuli can be observed at the circuit outputs.

Assume that model A and model B are both exercised thoroughly using a functional vector set (see Figure 2.14). Controllability metrics will report 100% statement coverage for both models. However, it may be that statements in model A are only exercised with vectors for which $c = 0$, implying that the variables assigned in these statements never affect the observed output F for any simulated vector. The previous example shows the importance of observability for the test vectors generation. There are different papers in the literature that propose ATPG based on this concept.

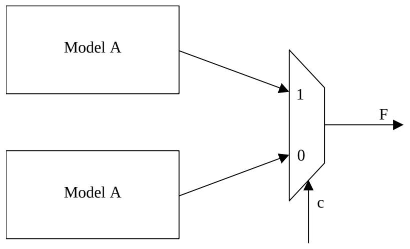


Fig. 2.14. Example of not observable behavior

One of the most relevant works of this area is the OCCOM approach described in [34] and subsequently enhanced in [35,36,37]. All these approaches are based on the use of tags that represent the possibility that incorrect values are computed in some locations. Each location corresponds to a variable assignment within the HDL model of the design under validation. The goal of the methodology is to determine if a tag injected in a particular location is propagated up to the outputs of the circuit, to understand if incorrectly computed values may follow the same propagation path. The problem is that the propagation of the tag toward the output depends on the values applied at other locations in the circuit, in the sense that other data values may block the tags from reaching any circuit output. Therefore, the quality of a set of vectors is determined by calculating how many injected tags are propagated to the outputs, and this percentage is called tag coverage. To compute this coverage the tag-calculus is introduced, that is an algebra that computes the effects of operations over operators that contain tags. The considered description language is not the entire VHDL/Verilog, but only a subset of it. Moreover, some preprocessing transformations of the specification are needed to apply the OCCOM approach to the given description. Any operation that can be described as an assignment plus a series of other logical or arithmetic or control operations, is split in a chain of different assignments, in order to allow the evaluation of the test vectors over that model using the proposed method. Then, the test generation algorithm tries to identify test patterns that increase as much as possible the tag-coverage of the description under analysis. The algorithm solves the tags activation and propagation problem by solving a Hybrid Satisfiability (HSAT) problem, whose goal is to find an input assignment that produces the required output values, or prove that none exists. The algorithm proposed in [37] solves the HSAT problem by using SAT and linear programming techniques. It operates according to the following steps:

1. A tag list is set up. This is analogous to the fault list in stuck-at fault test generation. As the algorithm proceeds, tags are removed from the list every time a vector is found to cover them. Ideally, the tag list should be empty when the algorithm completes.
2. An upper bound on the number of time frames, t_{max}, that will be used for vector generation is selected.
3. If there is no uncovered tag the algorithms stops. Otherwise, it selects an uncovered tag to generate a vector for. The position of the variable V_f and of

the operator V_{op} corresponding to this tag is identified in the graph $G(V, E)$ encapsulating the dependencies between operators.

4. t denotes the number of time frames that the design will be expanded to in the current attempt. t is set to one.
5. The graph $G(V, E)$ is unrolled t times. V_f and all variables in its transitive fanout are marked.
6. HSAT constraints for both tagged and untagged versions of the circuit are generated. For the tagged circuit, it has been ignored constraints with no marked variables in them, and replace the untagged version of the variable with the tagged one in all the involved operators. For the untagged circuit, constraints are only generated for the portion of the circuit in the fanin of the marked output.
7. Constraints are added by expressing the requirement that the tag be detected on at least one of the marked variables which is also an output of the circuit.
8. The HSAT problem is solved.
9. If there is no solution to the HSAT problem and $t < t_{max}$, t is incremented by one and the algorithm reverts to step 5.
10. If there is no solution to the HSAT problem and $t = t_{max}$, the algorithm returns reporting that the tag cannot be covered within t_{max} time frames.
11. For the vector generated, tag simulation is performed using an algorithm that detects all the tags that can be covered by this vector. The algorithm updated the tag list and reverts to step 3.

The algorithm is very effective for all those circuits with a low sequential depth. However, the fact that the algorithm is unable to find a vector that highlights the observability of a variable within several time frames can be considered a symptom that the corresponding tag is difficult to test.

ATPG based on Assignment Decision Diagrams

The use of assignment decision diagrams can be considered as one of the most recent and innovative proposals in the world of ATPG. The ADD is a data structure able to track the dependency between input and output, also in the case that these dependencies are affected by the value of other variables, not directly involved into the assignment, as for example the variables involved in control instructions. The use of ADD is also based on the availability of a design specification, where the number of primitive elements in the circuit is usually much lower than the elements of gate-level descriptions, thus reducing the problem size.

In [7], the authors illustrate how is it possible to use this data structure to obtain test vectors that are very effective, in particular from the point of view of observability. It is important to underline that the proposed algorithm works under a certain number of constraints:

1. the RTL design in VHDL or Verilog should have a single clock line;
2. the circuit should not present any complicated asynchronous behavior other than the set/reset signal;
3. if the presence in the circuit of a black box or intellectual property is assumed, then each input of the block should be propagated to an output or a combination of outputs of the block in a fixed number of cycles and each output of the

black box should be justified from an input or combination of inputs of that block in a fixed number of cycles;
4. each FSM description in the RTL circuit should have a reset state or a single input line that takes the FSM to a fixed state.

The algorithm proposed by Ghosh and Fujita first converts the HDL description into a graph-like structure, i.e., the ADD. Then the testing algorithm identifies arithmetical operation modules, logic arrays, registers, latches, memories, multiplexers, interconnected and random logic block from the ADD. Each of these elements is then tested by justifying test vectors to its inputs from the primary inputs and propagating test responses from its outputs to the primary outputs. The process of justification and propagation is done symbolically on the ADD representation with the help of a nine valued algebra and a branch and bound search procedure that requires backtracking similar to the sequential ATPG. After this operation, the result is a set of justifications and propagation paths from primary inputs to primary outputs that exercises the elements of the RTL circuit. The resulting path that may span across many clock cycles is termed a test environment for the element under test. Then, a test translation procedure uses test vectors from a well known test set for the RTL element or a pre-computed test set from a test set library and plugs them into the test environment to obtain a system-level test set for the element. Finally, the system-level test sets for various RTL elements are concatenated together to obtain the complete test set for the circuit.

There are at least two interesting points in this approach: the first is the ADD data structure, the second is the use of pre-compiled libraries to fill the test environment according to the particular type of system under test. The justification and propagation technique adopted by the authors is quite similar to the classical approaches adopted in any ATPG, and it is based on a nine value algebra for the identification of observability and controllability requirements. Sometime it is impossible to reach the desired properties with a single input vector, therefore the algorithm introduces the concept of time frame, and the ADD is replicated t times, where t is the number of required time frames. The second interesting point is the fact that the result of the justification and propagation process is not a test vector, but a symbolic string that will be filled in a second phase by using a series of pre-computed solutions stored in a library. The content of the test library is specialized according to the particular kind of component under test: this means that if we want to test a logic array, for example, it could be adopted a specific strategy that is particularly suited for detecting faults in that specific kind of component, and this strategy will be different with respect to the strategy adopted for a different component, such as a storage element.

ATPG based on Genetic Algorithms

Genetic Algorithms (GA) are sound and incomplete search procedures. They are very suitable for exploring spaces with either low or not at all search exploitable information. In fact, they can extract implicit information during the genetic evolution. The search process is performed evolving an initial random population by applying genetic operators (i.e., mutation and crossover). The evolution process is terminated as soon as either the optimal solution is found or the last evolution has been performed. This second terminal condition is the reason why GA are an incomplete search method that cannot generally guarantee to find the optimal result. A general GA is completely defined by the following parameters, which are recalled here to relate them to functional testing.

- G*ene definition.* The optimal solution is obtained by evolving a population of potential solutions. The first step in the definition of a GA is defining what a gene represents.
- G*ene representation.* Each gene will be manipulated by the genetic operator in order to obtain a solution close to the optimal one. An effective representation may have a deep impact on the GA performance.
- P*opulation.* It represents the set of genes evolved during generations. The population may contain either a constant or variable number of individuals. The population size could be increased after some generations without any improvement.
- F*itness function.* This function measures the quality of each individual and drives the GA closer and closer to the solution. Definition of a good fitness function can imply the success/unsuccess of the genetic experiments, thus its definition requires a large effort.
- S*election strategy.* This step of the GA defines how to select a set of genes from which deriving the genes of the next population.
- M*utation operators.* The goal of these operators is to introduce small variants into the gene population to avoid the premature convergence to a solution too far from the optimal solution. Generally, they are applied with a probability, which decreases during the genetic evolution. For more accurate GA, a set o mutation operators may be defined. The applied operator may be selected either randomly or in relation to the current state of the population evolution.
- C*rossover operators.* These operators derive from a set of genes the next generation of genes. They try to mix the positive factors of genes included into the current generation. They are applied with an operator probability. A set of different crossover operators can be defined to be able to apply the most appropriate of them with respect to the genetic evolution.

The characterization of the previous parameters to the functional test generation problem is as follows.

- G*ene definition.* For a genetic ATPG, a gene represents either a test sequence [38,39], if the design under test is sequential, or a test vector for combinatorial circuits [40].
- G*ene representation.* The binary representation is commonly adopted to represent genes [41,42,43,44]. By including other symbols, such as 'X' and '-', it is possible to define more efficient and accurate genetic operators.
- P*opulation.* It contains a set of genes. Thus, it is composed of a set of vectors or sequences [45].
- F*itness function.* An efficient GA for functional validation can be structured in different phases. Each one of them can be characterized by different fitness functions. For example, the first part of the genetic experiment may be focused on the coverage optimization [46,47,48,42,43]. While the second phase may

try to compact the generated test set [49,50] in order to obtain a compact test set with a high fault coverage.

- S*election strategy.* Roulette and tournament selection strategies are commonly adopted to select the set of test sequences to be modified [51,45,49,52].
- M*utation operators.* These operators can flip bits of test patterns, in order to discover alternative test sequences, which represent small variants of the previously generated test sequences [45,48]. A feasibility check is performed on the applied mutations and only feasible test sequences are inserted into the next generation. The feasibility check may be performed by exploiting design information. The effectiveness of these operators can be increased by defining a set of mutation operators, which are selected according to the genetic evolution.
- C*rossover operators.* These operators are liable to increase the variety of the generated test sequences [53,46], and to widely explore the search space. A feasibleness check could be performed for generated sequences according to formal specifications. The operator effectiveness can be increased by applying linkage learning techniques [54] in order to reduce the disrupting power of this operator. By defining a set of crossover operators, it is possible to select the most appropriate one accordingly to the current phase of the genetic evolution [50,43].

The goal of GA is to generate functional test sequences for VLSI designs. The structure of a general GA for functional test generation is presented in Figure 2.15. The initial randomly generated population is composed of input sequences. The

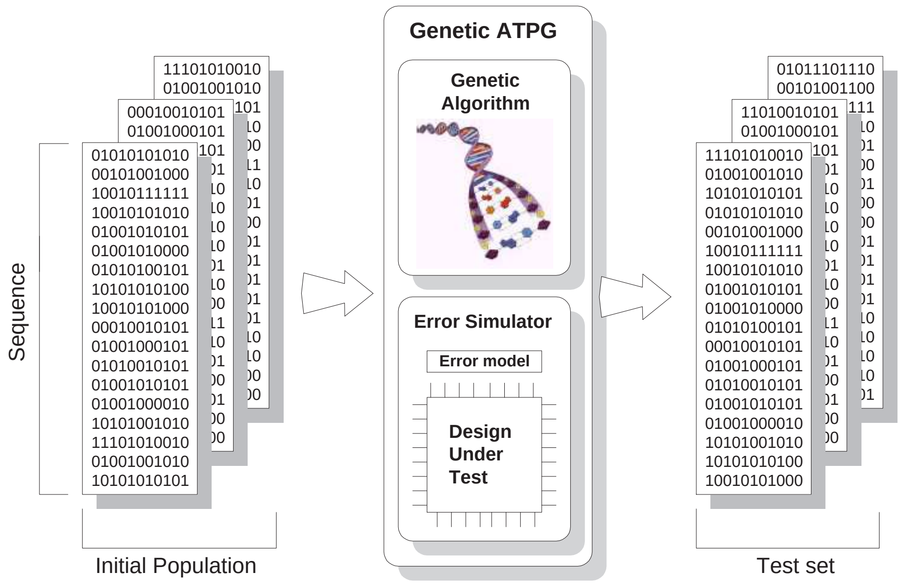


Fig. 2.15. A general architecture of a GA-based functional TPG

fitness of each sequence will be evaluated by the fault simulator. The result of the defined GA based TPG is a set of test sequences for the current design under test.

Sequential ATPG: Time frame expansion

Figure 2.16 illustrates the Huffman model of a sequential circuit. Symbols I, O, S and N are used to represent primary inputs, primary outputs, present state line and next state line, respectively.

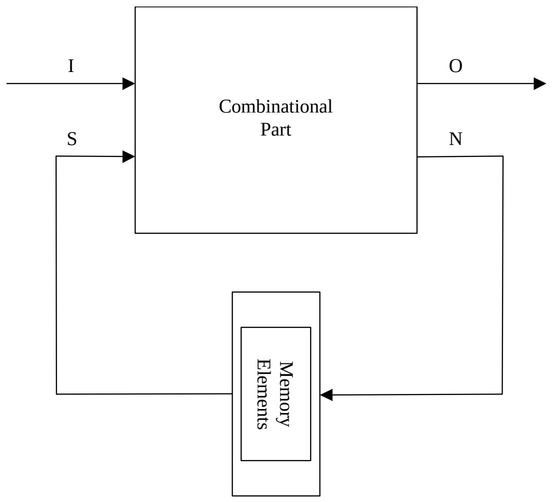


Fig. 2.16. Huffman model of a sequential circuit.

In any clock cycle c_i, arbitrary test values can only be assigned to the primary inputs of the circuit while the values on the next state lines depend on the values on the present state lines at the end of the clock cycle c_{i-1}.

Therefore, the functionality of a sequential circuit can be represented using an iterative array of combinational logic, as shown in Figure 2.17.

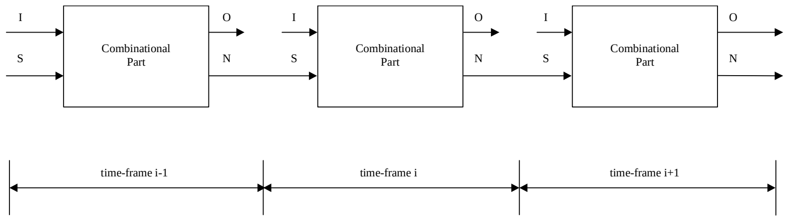


Fig. 2.17. Unrolled sequential circuit.

Each copy of a combinational logic is called *time-frame*. The present state values in time-frame k correspond to the next state values in the time-frame $k-1$.

Due to the time-frame unfolding, a single fault in the original sequential circuit is represented by a multiple fault in the iterative logic array model.

The fault initialization and fault propagation may require several clock cycles, i.e., the circuit may have to go through several time-frames. If activating the faults requires particular assignment on present state lines, the fault initialization has to be carried out such that these present state lines are justified by assigning signal values in the previous time frame. This process continues until no signal assignment on the present state lines is required. Similarly, after the fault has been activated, it may not be possible to propagate the fault effect to the primary outputs in the same time-frame, i.e., it may take several time-frames for the fault effect to become observable on the primary outputs. Obviously, it is harder to discover faults that need several time-frames to be propagated to the primary outputs. There are several issues affecting the complexity of sequential ATPG. Handling highly sequential circuits might not be possible due to the large number of time-frames required. Consider a 20-bit counter. Detecting the stuck-at-0 fault on the most significant bit (MSB) would require a sequence of 2^{19} vectors, that are necessary to set the MSB to 1, and thus this many time-frames would be needed by the ATPG. This is a perfect example of a small but highly sequential circuit that cannot be easily handled by any existing logic-level ATPG tool.

However, sequential circuits combined with lower-cost design-for-testability techniques such as partial scan design offers an attractive, alternative test solution.

2.2.4 Fault models

Fault models represent one of the main aspects for the development of tests for digital circuits. The aim of a fault model is to simulate the effect of the presence of physical faults into the design.

The functional test generation considers mainly two classes of fault models:

Functional fault models. They are based on the input/output relationship of higher level primitives which may represent a large number of gates.

Behavioral fault models. They are described on top of the procedural description of the design functionality.

The main contributions on fault modeling have been derived from the research activities of four prominent professors: Jacob Abraham, John Hayes, James Armstrong, and Sumit Ghosh. Abraham, in [55,56,57,58,59], has described alternative fault models for each abstraction level. Hayes introduced in [H85] a new generic fault model, called induced faults that effectively represents functional faults. His other contributions have been presented in [60,61,62,63]. Armstrong focused his research on functional and behavioral fault models [64,65,66,67,68,69,70]. The failure modes of the language constructs of a generic hardware description language are the basis of fault models introduced by Ghosh in [71,72,73].

Functional faults

An extensive review of research efforts aimed at deriving realistic models at higher levels which can accurately represent the faults at lower levels have been presented in [56]. The considered models can accurately represent the faults at lower levels. The most interesting functional models concerns: general fault models for functional blocks, models for small functional modules, and fault models for microprocessors. A general fault model assumes that given a combinational function with N inputs, it can be transformed into any other combinational function of N inputs. Therefore, its test can be guaranteed by applying the 2^N distinct input combinations. This approach is not feasible when N is large, even if it could be applied if the function is defined as an interconnection of subfunctions. The exhaustive general fault model could then be used effectively to test these subfunctions. Models for several basic functional modules can be exploited for testing larger functions. For example, a common functional module often found in digital circuits is the decoder. Its can be described as a design with N inputs and 2^N outputs. Each input pattern activates exactly one output line corresponding to the input address. In [74], a functional fault model for decoders has been defined by analyzing all single transistor-level faults. This fault model considers three alternatives:

1. The selected output is not the correct one.
2. In addition to the correct line, an incorrect line is activated.
3. No output is selected.

It has been proven that this simple model can effectively represent all of the potential physical shorts and opens in the gate description of a decoder. A similar approach has been applied to study fault models for multiplexer [55], another basic building block. This component is composed by: N data inputs, log_2N control inputs and one output. The control inputs select the data input value that has to be copied on the output. The defined fault model for this component summarizes its faulty behaviors as:

1. It is not possible to drive a 0 or a 1 on all the data inputs.
2. It is not possible to drive a 0 or a 1 on all the control inputs.

Similar functional fault model have been defined for other basic functional blocks. Even though the complexity of a microprocessor is several orders of magnitude higher than the one of the previous examples, a functional fault models have been derived at the register transfer in [59]. It has been defined by representing the microprocessor as a set of functions: register decoding, data transfer, data manipulation, and instruction sequencing. Then, a functional fault model has been described for each of these functions. This approach has been further improved in [57] by hierarchical decomposition of instructions to micro-instructions and micro-instruction to micro-orders. The defined fault model for generic microprocessors can be represented as combination of the follow functional fault models:

- F*ault model for the register decoding function.* When the register R_i should be selected:
  1. The R_j is selected instead of R_i.
  2. The R_j and R_i are both selected.
  3. No register is selected.
- F*ault model for the data transfer function.* It is defined as:
  1. Any line can be stuck at 0 or 1.
  2. Any pair of lines *i,j* can be coupled.

- F*ault model for the data manipulation function.* No specific fault model is provided. In fact, it is assumed that the complete test set for any given ALU can be easily determined.
- F*ault model for the instruction sequencing function.* The presence of a fault can determine one of the following behavior:
  1. One or more micro-orders can be inactive.
  2. Usually inactive micro-orders become active.
  3. A set of microinstructions is active in addition to, or instead of, the normal microinstructions.

The presented approach includes several advantages:

- H*igh-level of abstraction.* It provides a functional fault model even for higher functional models.
- L*ow complexity.* Providing fault models for high-level block, it makes tractable complex systems.
- G*eneral approach.* It allows derivation of tests for complex design (e.g., microprocessors), even if implementation details are not known. In fact, this approach focus on the functionality instead of its specific implementation.

Behavioral faults

Behavioral fault models are used in [73] to represent complex failures in VLSI designs. Faults are simulated by deliberately introducing faulty values for state/-timing parameters or replacing correct parts of the design with faulty versions. The severe limitation of this approach is the correct selection of the fault model, which represents the actual failures. The most suitable methodology is based on the adoption of a library of fault models of complex devices that are based on actual failures. This approach has been extended in [71, 72] by a set of fault models based on the failure modes of the language constructs of a generic hardware description language. The C programming language is adopted to describe hardware with assurances that its language constructs may be extended to other hardware description languages. The presented fault models attempt to identify a link between hardware description language constructs and potential hardware faults. The main features of the proposed behavioral fault models are:

- S*equential state faults.* The states of a sequential component can be alternatively expressed by either an integer, boolean or a real variable. The fault can manifest itself by permanently assuming either the value V_1 or V_2, where V_1 and V_2 represent the lower and upper bounds of the logical value system.
- F*unction call faults.* Two different failures can be manifested by a function call, which can permanently return either V_1 or V_2, where V_1 and V_2 represent the lower and upper bounds of the function return value range.
- *f*`or` c*onstruct faults.* In the construct `for` *(C)* $\{S\}$, the set of statements S can be either not executed at all or infinitively execute regardless of the condition C.
- *s*`witch` c*onstruct faults.* Faulty `switch` *(id)* constructs may select alternatively: any value in the variable *(id)* range, none or all the specified cases.

- `if` *(C)*`then` *(S_1)* `else` *(S_2) statement faults.* The faulty behavior of this construct can force the execution of either S_1, S_2 independently from the value of the condition C or S_1 and S_2 are executed when C evaluates to false and true respectively.
- T*he assignment faults.* The assignment $X = Y$ may fail, such that X remains unchanged or assumes the lower or upper bounds of the value system, or X assumes the lower or upper bounds depending on a probability function.

Though relationships to possible hardware faults are proposed, there is no detailed analysis to justify these assertions. A further shortcoming of these models is the restriction to the lower bound or upper bound of the value system. Multiple bit signals must all be stuck at 0 or stuck at 1 rather than allowing for only a single stuck line.

Model perturbation

The fault models based on the concept of *model perturbation* have been introduced for the first time [69]. These fault models are based on the alteration of the HDL description. This approach has been customized for VHDL descriptions in [66]. Eight behavioral fault classes have been identified:

- S*tuck-then.* It represents a failure of the `if-then-else` construct to ever execute the `else` statements.
- S*tuck-else.* It represents a failure of the `if-then-else` construct to ever execute the `then` statements.
- A*ssignment control.* It represents a failure of the VHDL assignment operator to assign a new value to a signal.
- D*ead process.* No statement within the process will never be executed.
- D*ead clause.* It represents a failure of the VHDL `CASE` construct to execute one of the alternative sequences of statements (clauses).
- M*icro-operation.* It represents a failure of an operator to perform its intended function. The operator may fail to any other operator in its class.
- L*ocal stuck-data.* It represents failure of a signal or variable to have the correct value. The local stuck-data fault is restricted to the expression into which it is mapped.
- G*lobal stuck-data.* It represents failure of a signal or variable to change value within the device model.

Later, the former fault classes have been subdivided into two broad categories [65]: control faults and micro-operation faults. Control faults alter the sequences of the executed micro-operation. This group contains the former: stuck-then, stuck-else, assignment control, dead process and dead clause. Whereas micro-operation faults invert single micro-operations with others (e.g., `AND` $\rightleftharpoons$ `OR, INC` $\rightleftharpoons$ `DEC and ADD` $\rightleftharpoons$ `SUB`). The crucial aspects of this model concerns the selection of the micro-operation to perturb [75] and to determine whether any of these faults can actually occur in hardware.

The concept of equivalent faults has been applied to the previous fault model in [64] in order to define a new fault model. Stuck-then/stuck-else faults can be removed from the behavioral fault list if stuck-at faults are defined for unnamed signals corresponding to the conditional expressions of the IF statement. Likewise, a micro-operation fault for a logic operator is detected by a test for a stuck-at fault on one of its arguments. Finally, a dead-clause fault is equivalent to an assignment control fault under the assumption of a single behavioral fault model. The new behavioral fault model renames stuck-at faults to behavioral stuck-at faults. Assignment control faults are renamed behavioral stuck-open faults and micro-operation faults for arithmetic or relational operators are renamed micro-operation faults. Applying these equivalence reductions, the new fault model is composed by three fault types:

- B*ehavioral stuck-at fault.* Each bit of signal, virtual signal, a fan-out stem, or a fan-out branch can be permanently stuck-at logic 1 or 0.
- B*ehavioral stuck-open fault.* The value of the source expression of an assignment statement is not correctly transferred to its target.
- M*icro-operation fault.* An arithmetic or a relational operator is faulted to another operator.

This fault model has been effectively applied to define the *B-algorithm* presented in [64]. The main concerns of this model are related to the effective correlation to actual hardware faults.

2.2.5 Fault Injection

Fault injection is a technique that aims at modifying the behavior of a digital design in order to simulate the effect of a potential fault. It is mainly applied for validating fault tolerant systems and to develop detailed fault simulators. The faulty behavior is explicitly induced by the artificial modification of the design behavior. Fault injection techniques for hardware description can be classified in three main categories:

Hardware Implemented Fault Injection. It is performed directly at physical level by either modifying the environment surrounding the hardware (e.g., heavy ion radiation, electronic interferences) or altering the values on the design pins.

Software Implemented Fault Injection. The goal of these techniques is to reproduce at software level the faulty behavior deriving from software or hardware faults. These faults can be induced by the modification of the memory data or the modification of the executed code.

Simulated fault injection. The logic values of the simulated design are altered by modifying the simulator logic.

The simulated fault injection techniques are the most interesting techniques for the topic of this dissertation.

Figure 2.18 shows an accurate taxonomy for simulated fault injection techniques discussed in [76]. Fault injection techniques based on simulator commands have been presented in [77]. Simulator commands are exploited to modify the value of the model signals and variables without altering code description. HDL code modification techniques are based either on saboteurs [78,77] or on mutants of the model components [65,71]. Other techniques are based on the extension of the HDL language with specific data types and signals, or modifying the HDL resolution

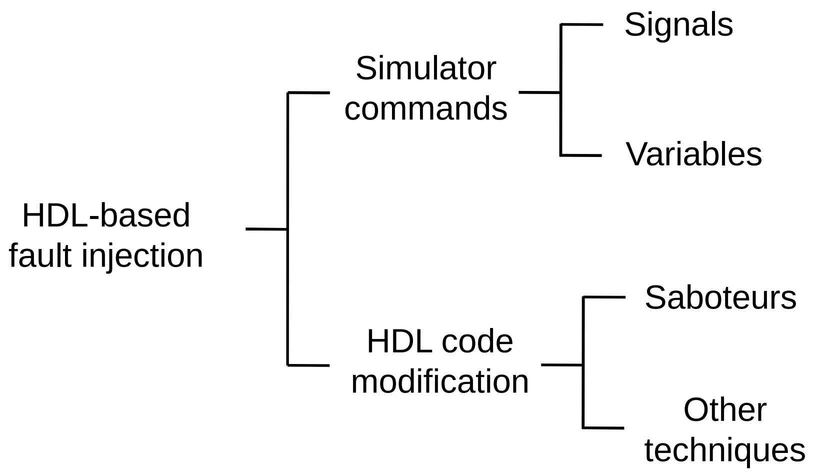


Fig. 2.18. A simulated fault injection taxonomy.

function [79,80]. The main disadvantage of these approaches is the requirement of *ad-hoc* HDL compilers and control algorithms to manage the language extensions.

Simulator commands

These techniques exploit the simulator commands to modify the value of the model signals and variables [77]. The fault injection campaign are described as script file loaded before the simulation start. In order to be able to modify the fault injection parameters (e.g., injection place, injection instant, fault duration, fault value, observation time) without altering the command code, they are defined via simulator macros [81].

Saboteurs

A saboteur is an artificial HDL component added to the original design [77]. The goal of the saboteur is the properties of the target signal (e.g., value, timing response) when the corresponding fault is injected. It presence does not affect the design behavior during the normal operation of the system. In [77] has been presented a saboteur taxonomy: serial simple, serial complex and parallel. In [76] this set has been extended with bi-directional saboteurs and some of the former versions have been adapted to be applicable to buses. The saboteur library defined in [76] contains:

- S*erial simple saboteur.* It interrupts the connection between an output (driver) and its corresponding receptor (input), modifying the reception value.
- S*erial simple bi-directional saboteur.* It has two input/output signals, plus a read/write input that determines the perturbation direction.
- S*erial complex saboteur.* It interrupts the connection between two outputs and their corresponding receptors, modifying the reception values.

- S*erial complex bi-directional saboteur.* It has four input/output signals, plus a read/write input that determines the perturbation direction.
- n-*bit Unidirectional simple saboteur.* It is used in unidirectional buses of n bits (address and control). It is composed of n serial simple saboteurs.
- n-*bit Bi-directional simple saboteur.* It is used in bi-directional buses of n bits (data and control). It is composed of n bi-directional serial simple saboteurs.
- n-*bit Unidirectional complex saboteur.* It is used in unidirectional buses of n bits (address and control). It is composed of n/2 serial complex saboteurs.
- n-*bit Bi-directional complex saboteur.* It is used in bi-directional buses of n bits (data and control).It is composed of n/2 bi-directional complex saboteurs.

Saboteurs are inserted on signals interconnecting sub-components of designs. A saboteur can be developed as a multiplexer [82], where the control input select the faulty or the fault free behavior.

2.3 Mutation Analysis

Mutation analysis has a rich and varied history, with major advances in concepts, theory, technology, and social viewpoints. This history begins in 1971, when Richard Lipton proposed the initial concepts of mutation in a class term paper titled "Fault Diagnosis of Computer Programs". It was not until the end of the 1970's, however, before major work was published on the subject [83, 84, 85]; the DeMillo, Lipton, and Sayward paper [85] is generally cited as the seminal reference.

PIMS [83, 86, 87, 88], an early mutation testing tool, pioneered the general process typically used in mutation testing of creating mutants (of Fortran IV programs), accepting test cases from the users, and then executing the test cases on the mutants to decide how many mutants were killed. In 1987, this same process (of add test cases, run mutants, check results, and repeat) was adopted and extended in the Mothra mutation toolset [89, 90, 91, 92], which provided an integrated set of tools, each of which performed an individual, separate task to support mutation analysis and testing. Because each Mothra tool is a separate command, it was easy to incorporate, and thus experiment with, additional types of processing. Although a few other mutation testing tools have been developed since Mothra [93, 94, 95], Mothra is likely the most widely known mutation testing system extant.

Despite the relatively long history of mutation testing, the software development industry has failed to employ it. The two primary reasons why industry has failed to use mutation testing are the inability to successfully integrate unit testing into software development processes, and difficulties with providing full and economical automated technology to support mutation analysis and testing. These two reasons for the lack of commercial success of mutation are primarily technological in nature. During the 1990s, a number of technological and theoretical advances were made in the application of mutation analysis and testing. Most of these advances are orthogonal, that is, they affect different aspects of mutation testing.

Before going into detail about these advances, a discussion of how mutation is used is given from a procedural point of view. Following that, a number of advances for applying mutation are discussed, which leads to a new process for how mutation can be applied. A test tool is presented that provides almost complete automation to the tester. A programmer submits a software module, and after a few minutes of computation, the tool responds with a set of test cases that are assured to provide the software with a very effective test, and a set of outputs that can be examined to find failures in the software. Furthermore, these input-output pairs can be used as a basis for debugging when failures are found. To be used by industry, this technology must be integrated with compilers, debuggers, and report generators.

2.3.1 The Mutation Analysis Process

Mutation analysis induces faults into software by creating many versions of the software, each containing one fault. Test cases are used to execute these faulty programs with the goal of distinguishing the faulty programs from the original program. Hence the terminology; faulty programs are mutants of the original, and a mutant is killed by distinguishing the output of the mutant from that of the original program.

Mutants either represent likely faults, a mistake the programmer could have made, or they explicitly require a typical testing heuristic to be satisfied, such as execute every branch or cause all expressions to become zero. Mutants are limited to simple changes on the basis of the coupling effect, which says that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults The coupling effect was first hypothesized in 1978 [85], then supported empirically in 1992 [96], and has been demonstrated theoretically in 1995 [97, 98].

Mutation analysis provides a test criterion, rather than a test process. A testing criterion is a rule or collection of rules that imposes requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of coverage; a set of test cases achieves 100% coverage if it completely satisfies the criterion. Coverage is measured in terms of the requirements that are imposed; partial coverage is defined to be the percent of requirements that are satisfied. Test requirements are specific things that must be satisfied or covered; for example, reaching statements are the requirements for statement coverage and killing mutants are the requirements for mutation. Thus, a test criterion establishes firm requirements for how much testing is necessary; a test process gives a sequence of steps to follow to generate test cases. There may be many processes used to satisfy a given criterion, and a test process need not have the goal of satisfying a criterion. In precise terms, mutation analysis is a way to measure the quality of the test cases and the actual testing of the software is a side effect. In practical terms however, the software is tested, and tested well, or the test cases do not kill mutants. This point can better be understood by examining a typical mutation analysis process.

When a program is submitted to a mutation system, the system first creates many mutated versions of the program. A *mutation operator*[2] is a rule that is applied to a program to create mutants. Typical mutation operators, for example,

[2] The terminology varies; they are also sometimes called mutant operators, mutagenic operators, mutagens, mutation transformations, and mutation rules [99]

replace each operand by every other syntactically legal operand, or modify expressions by replacing operators and inserting new operators, or delete entire statements. Figure 2.19 graphically shows a traditional mutation process. The solid boxes represent steps that are automated by traditional systems such as Mothra, and the dashed boxes represent steps that are done manually.

Next, test cases are supplied to the system to serve as inputs to the program. Each test case is executed on the original program and the tester verifies that the output is correct. If incorrect, a bug has been found and the program should be fixed before that test case is used again. If correct, the test cases are executed on each mutant program. If the output of a mutant program differs from the original (correct) output, the mutant is marked as being dead. Dead mutants are not executed against subsequent test cases.

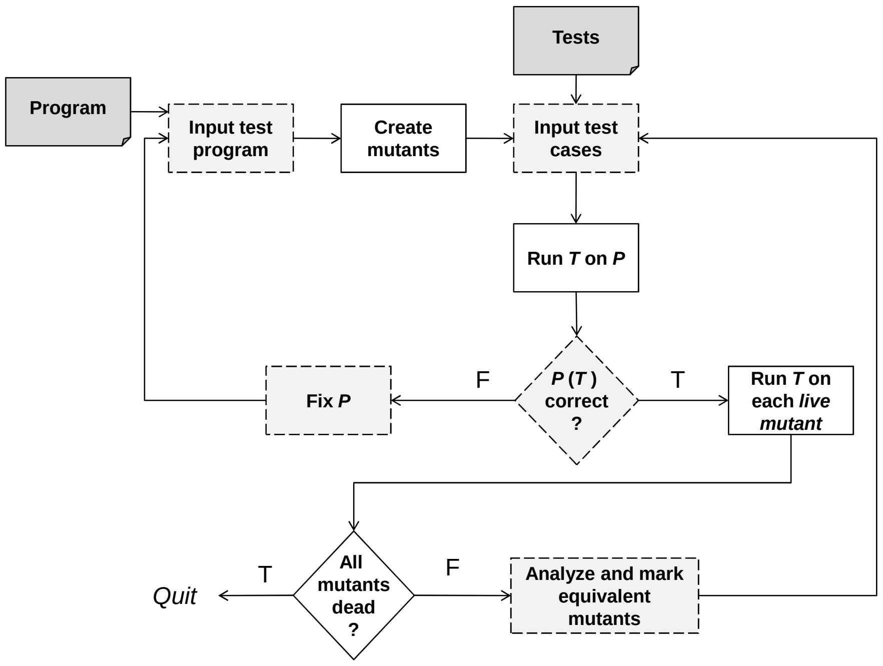


Fig. 2.19. Traditional Mutation Testing Process. Solid boxes represent steps that are automated and dashed boxes represent steps that are manual.

Once all test cases have been executed, a mutation score is computed. The mutation score is the ratio of dead mutants over the total number of non-equivalent mutants. Thus, the tester's goal is to raise the mutation score to 1.00, indicating that all mutants have been detected. A test set that kills all the mutants is said to be adequate relative to the mutants.

If (as is likely) mutants are still alive, the tester can enhance the set of test cases by supplying new inputs. Some mutants are functionally equivalent to the original program. Equivalent mutants always produce the same output as the original program, so cannot be killed. Equivalent mutants are not counted in the mutation score. Note that even if the tester has not found any faults by using the previous set of test cases, the mutation score gives some indication of the extent of the testing. Moreover, the live mutants point out inadequacies in the test cases. In most cases, the tester creates test cases to kill specific live mutants. This process of adding new test cases, verifying correctness, and killing mutants is repeated until the tester is satisfied with the mutation score. A mutation score threshold can be set as a policy decision to require testers to test software to a predefined level.

2.3.2 Using Mutation Analysis to Detect Faults

Many research papers about mutation have obscured the issue of how and when failures are found when using mutation. In standard IEEE terminology [100], a failure is an external, incorrect behavior of a program (an incorrect output or a runtime failure). A fault is the group of incorrect statements in the program that causes a failure. Failures in the software are detected when test cases are executed against the original program. The tester must decide whether the output of the program on each test case is correct. If the output is correct, the process continues as described above. If the output is incorrect, then a failure has been found and the process stops until the associated fault can be corrected. This leads to the fundamental premise of mutation testing, as coined by Geist [101]: in practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault.

2.3.3 Practical use of Mutation testing

One of the barriers to the practical use of mutation testing is the unacceptable computational expense of generating and running vast numbers of mutant programs against the test cases. The number of mutants generated for a software unit is proportional to the product of the number of data references and the number of data objects [102]. Typically, this is a large number for even small software units. Because each mutant program must be executed against at least one, and potentially many, test cases, mutation analysis requires large amounts of computation. This is shown in Figure 2.19 in the box labeled “Run test cases on each live mutant”. It is by far the most computationally expensive step in mutation testing.

The other barrier to more widespread use of mutation testing is the amount of manual labor involved in using this technique. For example, manual equivalent mutant detection is quite tedious and developing mutation adequate test cases can be very labor-intensive.

Recent advances show promise in bringing down both of these barriers. In the follow, first advances have been described for reducing the computational expense of mutation analysis and then review research work is presented, that has been successful in partially automating much of the labor intensive portions of mutation testing.

Reducing the Computational Cost of Mutation Analysis

Recall that the major cost of mutation analysis arises from the computational expense of generating and running vast numbers of mutant programs. Approaches to reduce this computational expense usually follow one of three strategies: *do fewer*, *do smarter*, or *do faster*.

- The *do fewer* approaches seek ways of running fewer mutant programs without incurring intolerable information loss.
- The *do smarter* approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs or seek to avoid complete execution.
- The *do faster* approaches focus on ways of generating and running each mutant program as quickly as possible.

Selective Mutation - a do fewer *approach*

Mothra used 22 mutation operators, of which the six most populous account for 40% to 60% of all mutants. This is typical of mutation systems - the goal was to include as much testing as possible by defining as many mutants as possible. These six mutants, and others, are in some sense redundant; that is, test sets that are generated to kill only mutants generated from the other mutant operators are very effective in killing mutants generated from the six. Wong and Mathur suggested the idea of constrained mutation to be applying mutation with only the most critical mutation operators being used [103]. This idea was later developed by Offutt et al. as an approximation technique called selective mutation that tries to select only mutants that are truly distinct from other mutants [102, 104].

Results showed that of the 22 mutation operators used by Mothra, 5 turn out to be *key* operators. In experimental trials, those five operators provided almost the same coverage as non-selective mutation, with cost reductions of at least four times with small programs, and up to 50 times with larger programs. The 5 sufficient operators are ABS, which forces each arithmetic expression to take on the value 0, a positive value, and a negative value, AOR, which replaces each arithmetic operator with every syntactically legal operator, LCR, which replaces each logical connector (AND and OR) with several kinds of logical connectors, ROR, which replaces relational operators with other relational operators, and UOI, which inserts unary operators in front of expressions. Future mutation systems will have the goal of minimizing the number of mutation operators - getting as much testing strength as possible with as few mutants as possible.

Mutant Sampling - a do fewer *approach*

First proposed by Acree [105] and Budd [106], in sampling only a randomly selected subset of the mutant programs are run. The effects of varying the sampling percentage from 10% to 40% in steps of 5% were later investigated by Wong [107]. A 10% sample of mutant programs, for example, was found to be only 16% less effective than a full set in ascertaining fault detection effectiveness.

An alternative sampling approach is proposed by Sahinoglu and Spafford [108] that does not use samples of some a priori fixed size but rather, based on a Bayesian sequential probability ratio test, selects mutant programs until sufficient evidence has been collected to determine that a statistically appropriate sample size has been reached.

Weak Mutation - a do smarter *approach*

Research systems such as Mothra execute mutant programs until they terminate, then compare the final output of the program with the output of the original program. Originally proposed by Howden [109], weak mutation is an approximation technique that compares the internal states of the mutant and original program immediately after execution of the mutated portion of the program. That is, weak mutation ensures that the necessity condition is satisfied, but not the sufficiency condition.

Weak mutation has been discussed theoretically [110, 111, 112] and studied empirically [113, 114, 115, 116]. Howden's original proposal stated that the states should be compared *after* the mutated statement, without elaborating on exactly when. Morell's concept of *extent* [110] and Woodward and Halewood's *firm* mutation [111] suggested that the comparison could be done at any point after the mutated statement.

The Leonardo system [117, 116], which was implemented as part of Mothra, did two things. It implemented a working weak mutation system that could be easily compared with strong mutation, and evaluated the extent/firm concept by allowing comparisons to be made at four different locations after the mutated component:

1. after the first evaluation of the innermost expression surrounding the mutated symbol,
2. after the first execution of the mutated statement,
3. after the first execution of the basic block that contains the mutated statement, and
4. after each execution of the basic block that contains the mutated statement (execution stops as soon as an invalid state is detected).

Experience with Leonardo indicated that weak mutation was able to generate tests that were almost as effective as tests generated with strong mutation, and that at least 50% and usually more of the execution time was saved. Moreover, it was found that the most effective point at which to compare the program states was after the first execution of the mutated statement.

Other do smarter *approaches*

Using novel computer architectures to distribute the computational expense over several machines represents another " do smarter" strategy. Work has been done to adapt mutation analysis systems to vector processors [118], SIMD machines [119], Hypercube (MIMD) machines [120, 121], and Network (MIMD) computers [122]. Because each mutant program is independent of all other mutant programs, communication costs are fairly low. At least one tool was able to achieve almost linear speedup for moderate sized program functions [120].

In another *do smarter* approach, Fleyshgakker and Weiss describe algorithms that improve the run time complexity of conventional mutation analysis systems at the expense of increased space complexity [123]. By intelligently storing state information, their techniques factor the expense of running a mutant over several related mutant executions and thereby lower the total computational costs. In the best case, these techniques can improve the speed by a factor proportional to the average number of mutants per program statement.

Schema-based Mutation Analysis - a do faster *approach*

Most mutation systems have worked by interpreting many slightly different versions of the same program. Although interpretation-based systems make the management of the mutant executions convenient, this conventional method has significant problems. Automated mutation analysis systems based on the conventional interpretive method are slow, laborious to build, and usually unable to completely emulate the intended operational environment of the software being tested. To solve these problems, Untch developed a new execution model for mutation, the Mutant Schema Generation (MSG) method [124, 94].

Instead of mutating an intermediate form, the MSG method encodes all mutations into one source-level program, a *metamutant.* This program is then compiled (once) with the same compiler used during development and is executed in the same operational environment at compiled-program speeds. Because mutation systems based on mutant schemata do not need to provide the entire run-time semantics and environment, they are significantly less complex and easier to build than interpretive systems, as well as more portable. Benchmarks show TUMS, an MSG-based prototype mutation analysis system, to be significantly faster than Mothra, with speed-ups as high as an orderof-magnitude observed.

Other do faster *approaches*

Another way of avoiding interpretive execution is the separate compilation approach, wherein each mutant is individually created, compiled, linked and run. The Proteum system [95] is an example of this approach. When mutant run times greatly exceed individual compilation/link times, a system based on such a strategy will execute 15 to 20 times faster than an interpretive system. When this condition is not met, however, a compilation bottleneck [121] may result.

To avoid compilation bottlenecks, DeMillo, Krauser, and Mathur developed a compiler-integrated program mutation scheme that avoids much of the overhead of the compilation bottleneck and yet is able to execute compiled code [93]. In this method, the program under test is compiled by a special compiler. As the compilation process proceeds, the effects of mutations are noted and code patches that represent these mutations are prepared. Execution of a particular mutant requires only that the appropriate code patch be applied prior to execution. Patching is inexpensive and the mutant executes at compiled-speeds.

Reducing Burdensome Manual Tasks

Manually developing test cases that are mutation adequate requires a great deal of effort. Additionally, determining which mutant programs are equivalent to the original program is a very tedious and error-prone activity. Progress has been made on partially automating both of these tasks and is described next.

Automatic Test Data Generation

One of the most difficult technical tasks in testing software is that of generating the test case values needed to satisfy the testing criterion. In his dissertation [91], Offutt developed a technique called constraint-based test data generation (CBT), which creates test data that comes reasonably close to satisfying mutation. CBT is based on the observation that a test case that kills a mutant must satisfy three conditions. The first is that the mutated statement must be reached; this is called the reachability condition. The second condition requires the execution of the mutated statement to result in an error in the program's state; this is called the necessity condition. The third condition, the sufficiency condition, states that the incorrect state must propagate through the program's computation to result in an output failure. Godzilla is a test data generator that uses constraint-based testing to automatically generate test data for Mothra [92].

Godzilla describes these conditions as mathematical systems of constraints. Reachability conditions are described by constraint systems called path expressions. Each statement in the program has a path expression that describes all execution paths through the program to that statement. The path expression is an assertion that is true if the statement is reached. The necessity condition is described by a constraint that is specific to the mutant operator and requires that the computation performed by the mutated statement create an incorrect intermediate program state. Because expressing the sufficiency condition as a set of constraints requires knowing in advance the complete path a program will take (in general, undecidable), Godzilla does not attempt to automatically satisfy this condition directly.

Godzilla conjoins each necessity constraint with the appropriate path expression constraint. The resulting constraint system is solved to generate a test case such that the constraint system is true. Experimentation [125] has verified that constraint-based testing creates test cases that kill over 90% of the mutants for most programs. CBT uses control-flow analysis, symbolic evaluation, and information about mutants to create the constraints, and a constraint satisfaction technique called domain reduction to generate test values.

CBT suffers from several shortcomings that prevent it from working in some situations and hamper its applicability in practical situations. Many of these shortcomings stem from weaknesses associated with symbolic evaluation and include problems handling arrays, loops, and nested expressions. Godzilla occasionally fails to find test cases, and for some programs it fails a large percentage of the time. This is partly because of problems with the technique, partly because of insufficiently general approaches to handling expressions, and partly because Godzilla employed relatively unsophisticated search procedures.

More recently, a test data technique called the dynamic domain reduction procedure was developed to address most of these problems [126, 127]. The dynamic domain reduction procedure (DDR) uses part of the CBT approach, and also draws from Korel's dynamic test data generation approach [128, 129] and symbolic evaluation. It uses a direct *domain reduction* method for deriving values, rather than function minimization methods as used by Korel or linear programming-like methods as used by Clarke [130]. Korel's dynamic method [129] executes a program along one specific path by starting with a particular input. When a branching point is reached, if the current inputs will cause the appropriate branch to be taken, the inputs will remain the same. If a different branch is required, then the inputs are dynamically modified to take the correct branch using function minimization. DDR also works by choosing a specific path, but there are no initial values, and the values are derived in-process from initial input domains.

Unlike dynamic symbolic evaluation [131,132], DDR creates sets of values that represent conditions under which a path will be executed. Thus, the results of dynamic symbolic evaluation attempt to represent all possible values that will execute a given path, while dynamic domain reduction only results in a small set of possible values. While this is more limited, it is also more practical for real programs.

The dynamic nature of DDR, which combines analysis of the software with satisfaction of constraints and test data generation, allows better handling of arrays and expressions. DDR also incorporates a sophisticated back-tracking search procedure to partially solve a problem that caused previous methods to fail. Because of the historical basis, the DDR procedure will always work when CBT does, and also in many cases when CBT does not.

The DDR procedure walks through the program control flow graph, generating test data along the way. Each input variable is initially given a large set of potential values (its domain) and, as branches are taken in the control flow graph, the domains for the variables involved in the predicates are reduced so that the appropriate predicates would be true for any assignment of values from the domain. When choices for how to reduce the domains must be made, a search process is initiated and choices are systematically made to try to find a choice that allows the subsequent edges on the path to be executed. When the procedure is finished, the remaining values for the variables' domains represent sets of test cases that will cause execution of the path. If any variable's domain is empty, the search process failed due to one of two possible reasons. One, the path is infeasible, so no satisfying values could be found. Two, it was very difficult to find values that execute the path; this could be because the constraints were too complicated or there are relatively few inputs that will execute the path.

Partial Automatic Equivalent Mutant Detection

A major problem with practically applying mutation is that of equivalent mutant programs. Equivalent mutants can be thought of as "dead-weight" in the testing process - they do not contribute to the generation of test cases, but require lots of time and attention from the tester. Equivalent mutants have traditionally been detected by hand, which is very expensive and time-consuming, and restricts the practical usefulness of mutation testing.

Although recognition of equivalent programs is in general undecidable [133], the idea of using compiler-optimization techniques to recognize some if not most equivalent mutants was suggested by Baldwin and Sayward in 1979 [134]. This technique was tried in a limited way by hand in Tanaka's 1981 thesis [135]. Offutt and Craft [136] refined, extended, and implemented the Baldwin and Sayward suggestions in a tool that was integrated with Mothra. This led directly to the idea of using constraint-based testing to detect equivalent mutants, which was implemented in a tool that detected almost 50% of the equivalent mutants [137, 138].

The constraint-based technique uses mathematical constraints to automatically detect equivalent mutants. The general idea is that if a constraint system that is created to kill a mutant is infeasible, then that mutant is equivalent. Although recognizing infeasible constraints is a difficult problem that cannot be solved in general, heuristic approximations have been developed that are quite effective. This approach also subsume all of the previous compiler-optimization techniques.

Hierons, Harman, and Danicic have gone one step further and use program slicing to detect equivalent mutants [139]. This approach in turn subsumes the constraint-based technique.

Unfortunately, no automated system will be able to detect all equivalent mutants, thus to complement the technique of recognizing equivalent mutants, we suggest that the remaining equivalent mutants can be safely ignored. Although this requires the tester to be willing to accept less than full mutation coverage, results indicate that the loss will not usually be significant, and the testing will still be more effective than testing with most other testing techniques. Although this approach is not completely satisfying from a theoretical point of view, it is an eminently practical engineering solution to a practically impossible problem.

2.4 Co-simulation

Co-simulation strategies allow to simulate and verify HW/SW embedded systems before the real platform is available. In this field, there is a large variety of approaches, that rely on different communication mechanisms to implement an efficient interface between the SW and the HW simulators. In the follow Section 2.4.1 presents the main aspects of co-simulation frameworks in a co-design scenario, afterwards Section 2.4.2 summarizes existing work on HW/SW co-simulation.

2.4.1 Co-design scenario

Embedded systems are mostly heterogeneous devices that comprise components as general purpose processors, DSP, custom ASICs and FPGAs. A tough obstacle in designing such systems is actually their partitioning in the hardware and software parts that are developed and tested with totally different tools and methodologies. Furthermore, hardware and software cannot be developed independently, since their interaction is a key point of the system behavior. In fact, a software developer needs to be aware of the underling hardware features to write effective programs (e.g., precise timing of the instructions, organization of the memory hierarchy), while hardware designers must know the characteristics of the application (e.g., memory access patterns, synchronization among tasks), in order to provide optimized components. Moreover, hardware components and software routines must be properly interfaced to warrant the correct behavior of the system.

In this scenario, it is convenient to adopt a co-design strategy, since it allows to develop both sides of the system (hardware and software) concurrently, thus avoiding the design loops where hardware and software are developed in succession and then refined several times until the convergence to a satisfactory system is met [140]. In such a co-design flow, a key component is a *co-simulation* tool that allows to verify the hardware, the software, and their interaction. Within a co-simulation environment, in fact, both hardware and software are evaluated concurrently, each one at the most convenient abstraction level. For this reason, a large number of co-simulation frameworks have been developed by academic groups and EDA vendors. However, while they all propose valuable solutions for HW/SW co-simulation, they lack a comprehensive methodology which addresses *all* the following aspects.

- *Minimal knowledge of the current design.* Frequently, an existing design must be extended without a complete knowledge of the basic components. This is due to IP-core protection and/or to the presence of legacy modules. Then, the methodology must be minimally intrusive for the description of HW and SW components, thus allowing the co-simulation of the new modules with the current design without requiring free access to the implementation details of the latter.
- *Heterogeneous co-simulation.* An homogeneous strategy allows HW and SW components to be modeled and simulated by using the same description language, and the same simulator. This is useful in the early definition of the system functionality. However, it abstracts away the real nature of the mixed HW/SW system, and it is generally not suited to model the design after HW/SW partitioning. Moreover, homogeneous co-simulation cannot be easily applied to incrementally extend an existing system. Thus, heterogeneous approaches are preferable, since they allow a more flexible framework. In this case, SW components are implemented by using a high-level programming language (e.g., C/C++) and they run on an instruction set simulator (ISS) or on a real programmable device. On the contrary, HW components are modeled by hardware description languages (e.g., VHDL, SystemC), and they are validated by using the appropriate simulators.
- *Flexibility with respect to the SW execution environment.* Generally, an ISS is adopted to simulate the SW components of the system, when the real programmable device is not yet available. Thus, the co-simulation framework must be able to co-simulate HW modules, described by using some hardware description languages, with a SW program running on both an ISS or a programmable device mounted on a real board. Using a unique interface mechanism allows a seamless replacement of the ISS with the real board, thus significantly simplifying the prototyping of the system.
- *Timing-accurate co-simulation.* In order to obtain realistic estimation of the system performance a timed synchronization between the SW and the HW domains is required. This represents a challenging task when different simulators are used for SW and HW components, basically because they do not share a common clock. Therefore, the co-simulation methodology must integrate a mechanism to exchange timing information and keep the synchronization between the ISS/board and the HW simulator.

2.4.2 Classification of co-simulation methodologies

Several co-simulation frameworks have been proposed in the literature [141, 142, 143, 144, 145, 146, 147, 148]. In spite of the variety of architectural targets, performance efficiency and description languages, we can classify these different solutions into two main categories: *homogeneous* and *heterogeneous* co-simulation frameworks. Notice that in this work we stick to the conventional distinction between homogeneous and heterogeneous schemes, based on the homogeneity or heterogeneity of the description languages used.

Homogeneous frameworks use a single engine for the simulation of both HW and SW components. The Ptolemy [141] and Polis [142] frameworks are pioneering works in that direction. In these approaches, homogeneity is achieved by abstracting away the distinction between hardware and software parts that are described as functional blocks. Homogeneous frameworks simplify the design modeling and they provide good simulation performance. However, they are suitable only in a very initial phase of the design, prior to HW/SW partitioning. Conversely, heterogeneous frameworks ensure a more accurate tuning between HW and SW components. Most of these frameworks essentially address the same problem: how to efficiently link event-driven hardware simulators and cycle-based instruction set simulators. Earlier HW/SW co-simulation frameworks [145, 146, 147] are mainly focused on multi-language system descriptions, i.e., a HDL for hardware, and a programming language for software. All these heterogeneous co-simulation solutions are quite similar since their main effort is focused on solving the issue of controlling and synchronizing two (or more) simulation engines. This heterogeneous style is suboptimal in terms of simulation performance and easiness of integration, but it was the only possible choice when VHDL or Verilog simulation was the highest possible level of abstraction for simulating hardware.

Some commercial tools, such as Mentor Graphics Seamless [144] and Synopsys Eaglei [143], also provide heterogeneous co-simulation capabilities. However, they allow HW/SW co-simulation at bus level, where each bus transaction involves all signals necessary to accomplish the bus function, thus degrading the co-simulation performance. The advent of design flows based on SystemC allowed the definition of efficient *semi-homogeneous* approaches [149, 150, 151, 152, 153], where the bus is abstracted to obtain a more efficient co-simulation. They are homogeneous from the language point of view, since both HW and SW are described by using C++. This definitely simplifies the implementation of the initial model as well as the subsequent HW/SW partitioning. However, these approaches are heterogeneous from the simulation point of view. In fact, HW components are simulated by using the SystemC simulation kernel, while SW programs run on an ISS or on a programmable device mounted on a real board. In this way, a more accurate performance estimation can be performed, since the heterogeneous model reflects the final embedded system. All these frameworks are based on two basic concepts:

- *Interprocess communication (IPC)*. It is used to realize the communication between the ISS/board, where the SW part runs, and the SystemC simulator, that models the HW part.

- *Bus wrapper.* It ensures synchronization between the SystemC simulation and the ISS/board, and it translates the information coming from the ISS/board into cycle-accurate bus transactions.

Most of these approaches [149, 150, 151] define a *custom interface* between the bus wrapper and the ISS/board. This makes the integration of new processor cores within the co-simulation framework harder, because the ISS needs to be modified to support the IPC primitives defined by the co-simulation system. This issue is addressed in [148], where a standardized interface between bus wrapper and ISS is proposed. It is based on the remote debugging primitives of GDB [154]. In this way, any ISS/board that can communicate with GDB (that is, basically any) can also become part of a system-level co-simulation framework. The approach of [148] still suffers from some performance bottlenecks, since the ISS/board and the SystemC simulators evolve in lock-step (synchronization is driven by the host operating system via IPC).

Another drawback of the previous co-simulation methodologies is represented by the lack of timing synchronization. This heavily limits their application as a design performance evaluation tool. Simulation performance was the strongest limitation to the development of effective timing accurate co-simulation strategies. This is particularly true for heterogeneous approaches, where a significant overhead is imposed by the effort of keeping the synchronization between HW and SW parts [155, 145, 146, 147]. Although homogeneous frameworks may make this task more manageable, homogeneity is usually achieved by abstracting away the distinction between hardware and software, making the very notion of time quite imprecise [141, 142].

Only recently, some approaches have addressed this performance bottleneck by explicitly targeting the timing accuracy of co-simulation.

One class of approaches borrows ideas from the theory of distributed synchronization, using the similarity between timed co-simulation and distributed event-driven simulation algorithms [156,157,158,159,160]. All these approaches follow an *asynchronous* paradigm, in which each simulator manages its local time, and the local times evolve at different speeds. They differ, however, in how the overhead required by synchronization is managed. One solution consists of using simulation rollbacks, when one simulator receives a past event from the other simulator [156]. Another solution relies on a proper alignment between the local clocks and global clock, when rollback is not possible. In this way, the synchronization occurs only when events are exchanged [157, 158, 159, 160].

A different class of approaches is based on the construction of a timing model for software, obtained by attaching timing annotations to the ISS (for instance, an execution time in clock cycles for each executed instruction). Thus, timing synchronization between software and hardware is achieved by using the accumulated delays for the software, and the clock cycle information provided by a HDL simulator for the hardware [161, 162, 163]. These solutions have two main drawbacks which limit an easy applicability: they are closely related to the specific implementation of the native OS, and the delay annotations depend on the processor where the SW runs (if the core model changes, delay annotations must be rewritten).

A final observation applies to all the timing-accurate HW/SW co-simulation solutions cited above. They are explicitly targeted for the interaction of two (or more) co-simulation engines, and are thus not suitable for a virtual prototyping context, which requires the timing-accurate interfacing of a simulator for HW and a SW program running on an real board, possibly hosting an OS. In other terms, when a real board is used instead of an ISS, it is necessary a synchronization between *actual* (i.e., not *simulated*) time (on the board side where SW is executed) and simulated time (on the host computer side where an HW model is simulated). This requirement rules out many of the possibilities previously reviewed; options such as simulation rollback or the use of instruction-based timing models, for instance, are either infeasible or inadequate. In fact, the real-time execution of the software on the board is based on the synchronization given by the hardware timer which monotonically increases its value. Moreover, the board may include some hardware devices which synchronizes their work by exploiting the timer value, thus rollback cannot be implemented, since it would require the rollback of the behavior of such real hardware components.

2.5 Constraint solving

Constraint solving and constraint programming [164] are a paradigm that is tailored to search problems. The main application areas are those of planning, scheduling, timetabling, routing, placement, investment, configuration, design and insurance. Constraint programming incorporates techniques from mathematics, artificial intelligence and operations research, and it offers significant advantages in these areas since it supports fast program development, economic program maintenance, and efficient runtime performance.

Basically, a constraint solving problem is composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take. The task is to assign a value to each variable satisfying all the constraints.

The direct representation of the problem, in terms of constraints, results in short, simple programs that can be easily adapted to changing requirements. The integration of these techniques into a coherent highlevel language enables the programmer to concentrate on choosing the best combination for the problem at hand. Because programs are quick to develop and to modify, it is possible to experiment with ways of solving a problem until the best and fastest program has been found. Moreover more complex problems can be tackled without the programming task becoming unmanageable.

Constraints differ from the common primitives of other programming languages in that they do not specify a step or sequence of steps to execute but rather the properties of a solution to be found. The constraints used in constraint solving are of various kinds, e.g. those used in constraint satisfaction problems and those solved by the simplex algorithm. Constraints are usually embedded within a programming language or provided via separate software libraries.

2.5.1 ECLiPSe: a Constraint Logic Programming System

Constraint logic programming (CLP) combines logic, which is used to specify a set of possibilities explored via a very simple in-built search method, with constraints, which are used to minimize the search by eliminating impossible alternatives in advance.

The programmer can state the factors which must be taken into account in any solution (the constraints), state the possibilities (the logic program), and use the system to combine reasoning and search. The constraints are used to restrict and guide search. The whole field of software research and development has one aim, viz. to optimize the task of specifying and writing and maintaining correct functioning programs.

A constraint logic program is a logic program that contains constraints in the body of clauses. An example of a clause including a constraint is:

```
A(X,Y) :- X + Y > 0, B(X), C(Y).
```

In this clause, `X + Y > 0` is a constraint; `A(X,Y), B(X)`, and `C(Y)` are literals like in regular logic programming. Intuitively, this clause tells one condition under which the statement `A(X,Y)` holds: this is the case if `X + Y` is greater than zero and both `B(X)` and `C(Y)` are true.

Like in regular logic programming, programs are queried about the provability of a goal, which may contain constraints in addition to literals. A proof for a goal is composed of clauses whose bodies are satisfiable constraints and literals that can in turn be proved using other clauses. Execution is done by an interpreter, which starts from the goal and recursively scans the clauses trying to prove the goal. Constraints encountered during this scan are placed in a set called constraint store. If this set is found out to be unsatisfiable, the interpreter backtracks, trying to use other clauses for proving the goal. In practice, satisfiability of the constraint store may be checked using an incomplete algorithm, which does not always detect inconsistency.

One of the main ambitions of Constraint Programming is the separation of Modeling, Algorithms and Search. This is best characterized by two pseudo-equations. The first one is paraphrased from Kowalski [165]:

```
Solution = Logic + Control
```

and states that have to solve a problem by giving a logical, declarative description of the problem and adding control information that enables a computer to deduce a solution.

The second equation

```
Control = Reasoning + Search
```

is motivated by a fundamental difficulty faced when dealing with combinatorial problems: we do not have efficient algorithms for finding solutions, we have to resort to a combination of reasoning (via efficient algorithms) and (inefficient) search.

Then every constraint program can be considered as an exercise in combining the 3 ingredients:

Logic The design of a declarative Model of the problem.

Reasoning The choice of clever Constraint Propagation algorithms that reduce the need for search.

Search The choice of search strategies and heuristics for finding solutions quickly.

ECLiPSe (ECLiPSe Common Logic Programming System) [166,167] is a Prolog based system whose aim is to serve as a platform for integrating various Logic Programming extensions, in particular Constraint Logic Programming (CLP). The kernel of ECLiPSe is an efficient implementation of standard (Edinburgh-like) Prolog as described in basic Prolog texts. It is built around an incremental compiler which compiles the ECLiPSe source into WAM-like code, and an emulator of this abstract code.

ECLiPSe was initially developed at the European Computer-Industry Research Centre (ECRC) in Munich, and then at IC-Parc, Imperial College in London until the end of 2005. It is now an open-source project, with the support of Cisco Systems.

ECLiPSe is a CLP system intended for:

- general programming tasks, especially rapid prototyping,
- problem solving using the available solver libraries and the CLP paradigm,
- development of new constraint solvers based on the existing solvers and employing ECLiPSe's lower-level language features.

The ECLiPSe system consists of:

- a runtime core,
- a collection of libraries,
- a modeling and control language,
- a development environment,
- interfaces for embedding into host environments
- interfaces to third-party solvers.

ECLiPSe offers several different libraries for handling symbolic and numeric constraints. The standard constraint solver offered is the fd (finite domain) solver, which applies constraint propagation techniques developed in the AI community. ECLiPSe supports finite domain constraints via the ic library. This library implements finite domains of integers, and the usual functions and constraints on variables over these domains. Others used libraries are the range library, the ria (real number interval) library, and finally the eplex (MIP) library.

The main benefit of constraint logic programming over other platforms for solving combinatorial problems is in the closeness between the conceptual model and the design model. ECLiPSe takes full advantage of this by offering facilities to choose different annotations of the same conceptual model to achieve design models which, whilst syntactically similar, can have radically different behavior.

Let consider the conceptual model for the map coloring example illustrated in Figure 2.20 into a design model which uses the finite domain constraint handler of ECLiPSe. As a toy example let us write a program to color a map so that no two neighboring countries have the same color. In constraint logic programs, variables start with a capital letter (eg A), and constants with a small letter (eg red).

The design model is encoded as shown in Figure 2.21. The problem involves four decisions, one for each country. These are modeled by the variables A, B, C and D. Countries is just a name for the list of four variables. Each decision variable, in this problem, has the same set of choices, modeled as possible values

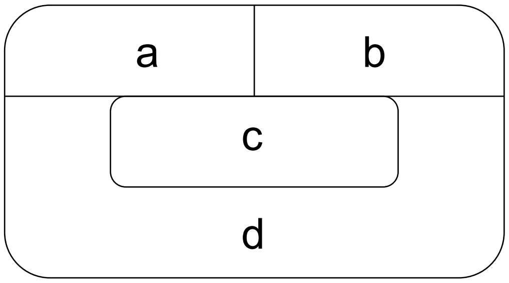


Fig. 2.20. Map coloring example.

```
:- lib(fd).

coloured(Countries) :-
     Countries= [A, B, C, D],
     Countries :: [red, green, blue],
     ne(A,B), ne(A,C), ne(A,D), ne(B,C), ne(B,D), ne(C,D),
     labeling(Countries).

ne(X,Y) :- X##Y.
```

Fig. 2.21. A finite domain CLP program for map coloring example.

for the variables (red, green and blue). There are six constraints, each of which is modeled by the same relation (ne meaning not equal to). The design model extends the conceptual model in four ways.

1. The ECLiPSe finite domain library is loaded (using `:- lib(fd)`).
2. An explicit finite domain is associated with each decision variable (using `Countries :: [red, green, blue]`).
3. The finite domain built-in disequality constraint is used to implement the ne constraint (using `ne(X,Y) :- X##Y`). `##` is a special syntax for disequality used by the finite domain constraint solver.
4. This program includes a search algorithm, that is invoked by the goal `labeling(Countries)`. As we shall see later, this predicate tries choosing, for each of the variables A, B, C and D in turn, a value from its domain. It succeeds when a combination of values has been found that satisfies the constraints.

Naturally this is a toy example, and it is not always so easy to turn a conceptual model into a design model. Nevertheless constraint logic programming, and in particular ECLiPSe, have made a lot of progress in achieving a close relationship between the conceptual model and the design model.

2.5.2 NuSMV: a Model Checking System

NuSMV is a symbolic model checker originated from the reengineering, reimplementation and extension of *SMV*, the original BDD-based model checker developed at CMU [168]. The NuSMV project aims at the development of a state-of-the-art symbolic model checker, designed to be applicable in technology transfer projects: it is a well structured, open, flexible and documented platform for model checking, and is robust and close to industrial systems standards [169].

The first version of NuSMV, basically implements BDD-based symbolic model checking. The new version of NuSMV (*NuSMV2*) inherits all the functionalities of the previous version, and extend them in several directions. The main novelty in NuSMV2 is the integration of model checking techniques based on propositional satisfiability (SAT) [170]. SAT-based model checking is currently enjoying a substantial success in several industrial fields (see, e.g., [171], but also [172]), and opens up new research directions. BDD-based and SAT-based model checking are often able to solve different classes of problems, and can therefore be seen as complementary techniques.

```
MODULE main
VAR
  bit0 : counter_cell(1);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
SPEC
  AG AF bit2.carry_out

MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value) := value + carry_in mod 2;
DEFINE
  carry_out := value & carry_in;
```

Fig. 2.22. A simple NuSMV program.

NuSMV is able to process files written in an extension of the SMV language. The only data types provided by the original SMV language are booleans, bounded integer subranges, and symbolic enumerated types. Moreover, NuSMV2 allows for the definitions of bounded arrays of basic data types.

The description of a complex system can be decomposed into modules, and each of them can be instantiated many times. This provides the user with a modular and hierarchical description, and supports the definition of reusable components. Each module defines a finite state machine. Modules can be composed either synchronously or asynchronously using interleaving. In synchronous composition a single step in the composition corresponds to a single step in each of the components. In asynchronous composition with interleaving a single step of the composition corresponds to a single step performed by exactly one component. The NuSMV input language allows to describe both deterministic and non deterministic systems.

A NuSMV program can describe both the model and the specification. Figure 2.22 gives a small example of a NuSMV program. The example in Figure 2.22 is a model of a 3 bit binary counter circuit. It illustrates the definition of reusable modules and expressions. The module `counter_cell` is instantiated three times, with names `bit0`, `bit1` and `bit2`. The module `counter_cell` has a formal parameter `carry_in`. In the instantiation of the module, actual signals (`1` for the instance `bit0`, `bit0.carry_out` for the instance `bit1` and `bit1.carry_out` for the instance `bit2`) are plugged in for the formal parameters, thus linking the module instance to the program (a module can be seen as a subroutine). The property that we want to check is "invariantly eventually the counter count till 8" which is expressed in CTL using the design state variables as "`AG AF bit2.carry_out`".

It is also possible to specify the transition relation and the set of initial states of a module by means of propositional formulas, using the keywords `TRANS`, and `INIT` respectively. This provides the user with a lot of freedom in designing systems. In Figure 2.23 there is an equivalent definition of module `counter_cell` using propositional formulas.

```
MODULE counter_cell(carry_in)
VAR
  value : boolean;
INIT
  value = 0
TRANS
  next(value) <-> ((!value &  carry_in) |
                   ( value & !carry_in))
DEFINE
  carry_out := value & carry_in;
```

Fig. 2.23. An equivalent definition of module `counter_cell` of the example described in Figure 2.22.

With respect to SMV, the input language of NuSMV has been extended to allow for the specification of properties expressed in LTL and the specifications of invariants.

NuSMV has both a batch and an interactive mode. The batch mode offers a built-in method for dealing with the system in which computations are activated according to a fixed predefined algorithm. The batch mode provides an interaction with the system that is essentially the same provided by SMV. In the interactive mode computation steps are activated by commands executed by a command line interpreter, thus allowing for the modification of the predefined model checking algorithm. The definition of the interactive shell required the decomposition of the model checking algorithm into small basic computation steps (e.g., parsing,

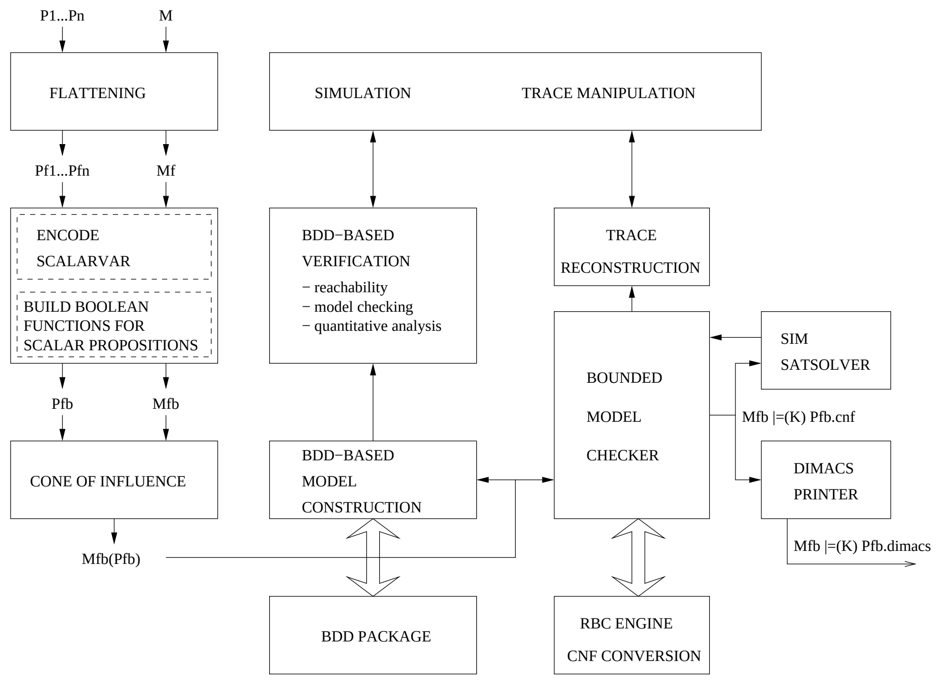


Fig. 2.24. The internal structure of NuSMV2.

model construction, reachability analysis, model checking). In the current version of the system, each of them corresponds to a command that implements a different functionality.

A high level description of the internal structure of NuSMV2 is given in Figure 2.24. An SMV file is processed in several phases. The first phases require to analyze the input file, in order to construct an internal representation of the system. NuSMV2 neatly separates the input language in different layers, of increasing complexity, that are incrementally eliminated. The construction starts from the modular description of a model M and of a set of properties $P_1, \ldots, P_n$. The first step, called *flattening*, performs the instantiation of module types, thus creating modules and processes, and produces a synchronous, flat model M^f, where each variable is given an absolute name. The second step, called *boolean encoding*, maps a flat model into a boolean model M^{fb}, thus eliminating scalar variables. This second step takes into account the whole SMV language, including the encoding of bounded integers, and all the set-theoretic and arithmetic functions and predicates. It is possible to print out the different levels of the input file, thus using NuSMV2 as a flattener. The same reduction steps are applied to the properties P_i, thus obtaining the corresponding flattened boolean versions P_i^{fb}. In addition, by means of the *cone of influence* reduction [173], it is possible to restrict the analysis of each property to the relevant parts of the model $M^{fb}(P_i^{fb})$. This reduction can be extremely effective in tackling the state explosion problem.

The preprocessing is carried out independently from the model checking engine to be used for verification. After this, the user can choose whether to apply BDD-based or SAT-based model checking.

In the case of BDD-based model checking, a BDD-based representation of the Finite State Machine (FSM) is constructed. In this step, different partitioning methods and strategies [174] can be used. Then, different forms of *BDD-based verification* can be applied:reachability analysis, fair CTL model checking, LTL model checking via reduction to CTL model checking, computation of quantitative characteristics of the model.

In the case of SAT-based model checking, NuSMV2 constructs an internal representation of the model based on a simplified version of *Reduced Boolean Circuit* (RBC), a representation mechanism for propositional formulae. Then, it is possible to perform SAT-based *bounded model checking* of LTL formulae [170]. Given a bound on the length of the counterexample, a LTL model checking problem is encoded into a SAT problem. If a propositional model is found, it corresponds to a counterexample of the original model checking problem. NuSMV2 represents each SAT problem as an RBC, that is then converted in CNF format and given in input to the internal SAT solver. Alternatively, the SAT problems can be printed out in the standard *DIMACS* format, thus allowing for the stand-alone use of other SAT solvers. With respect to the tableau construction in [170], enhancements have been carried out that can significantly improve the performances of the SAT solver [175]. In bounded model checking, NuSMV2 enters a loop, interleaving problem generation and solution attempt via a call to the SAT solver, and iterates until a solution is found or the specified maximum bound is reached.

NuSMV uses *SIM* [176] as the internal SAT solver. SIM is a SAT solver based on the Davis-Logemann-Loveland procedure. The features provided by SIM can produce dramatic speed-ups in the overall performances of the SAT checker, and thus of the whole system (see e.g., [177,171] for a discussion). It is currently under development a generic interface to SAT solvers to allow for the use of new state of the art SAT solvers like e.g. CHAFF [178].

The different properties that are checked on a FSM are handled and shown to the user by a property manager, that is independent of the model checking engine used for the verification. This means that it is possible for the user to decide what solution method to adopt for each property. Furthermore, the counterexample *traces* being generated by both model checking modules are presented and stored into a unique format. Similarly, the user can *simulate* the behavior of the specified system, by generating traces either interactively or randomly. Simulation can be carried out both via BDD-based or SAT-based techniques.

NuSMV2 is distributed with an OpenSource license [179], that allows anyone interested to freely use the tool and to participate in its development.

3

Motivation and Goals

The complexity of designs continues to rise, driven by technology advances, while time-to-market imposes always shorter time. Moreover, the increasing of design complexity implies that design validation becomes one of the most costs dominating phase in design production.

A verification phase is mandatory after each step of the digital system design flow to avoid the propagation of errors between the abstraction levels. The first important phase is the functional validation of the design at higher abstraction levels. In this phase, the presence of design errors must be detected, avoiding to propagate them to lower abstraction levels, saving time and money. In this context, a valuable solution for the functional validation is represented by dynamic verification which exploits simulation-based techniques to stimulate the whole design under validation (DUV). To achieve dynamic verification, test sequences are generated and then simulated on the DUV. To generate test sequences Automatic Test Pattern Generators (ATPGs) are used, chosen in relationship with the abstraction level where we want to apply verification. Thus, at the highest abstraction levels functional ATPGs are used, while at the lowest abstraction levels gate-level ATPGs are exploited.

Nowadays, very efficient logic-level ATPGs are available while high-level automatic test pattern generators are still in a prototyping phase. However, moving from lower-level to functional-level would provide a great benefit since functional descriptions are more tractable than lower-level ones, and early detection of design errors is mandatory for the success of a design.

Therefore, this thesis presents a functional ATPG framework, shown in Figure 3.1, able to generate efficient test sequences. To define such a king of ATPG, four main problems have to be treated, that can summarize the goals of this thesis as follows:

1. defining a model to represent the DUV;
2. defining a functional deterministic ATPG engine;
3. defining a high-level fault model to guide the pattern generation and to quantify the effectiveness of generated test sequences;
4. defining an efficient simulation engine.

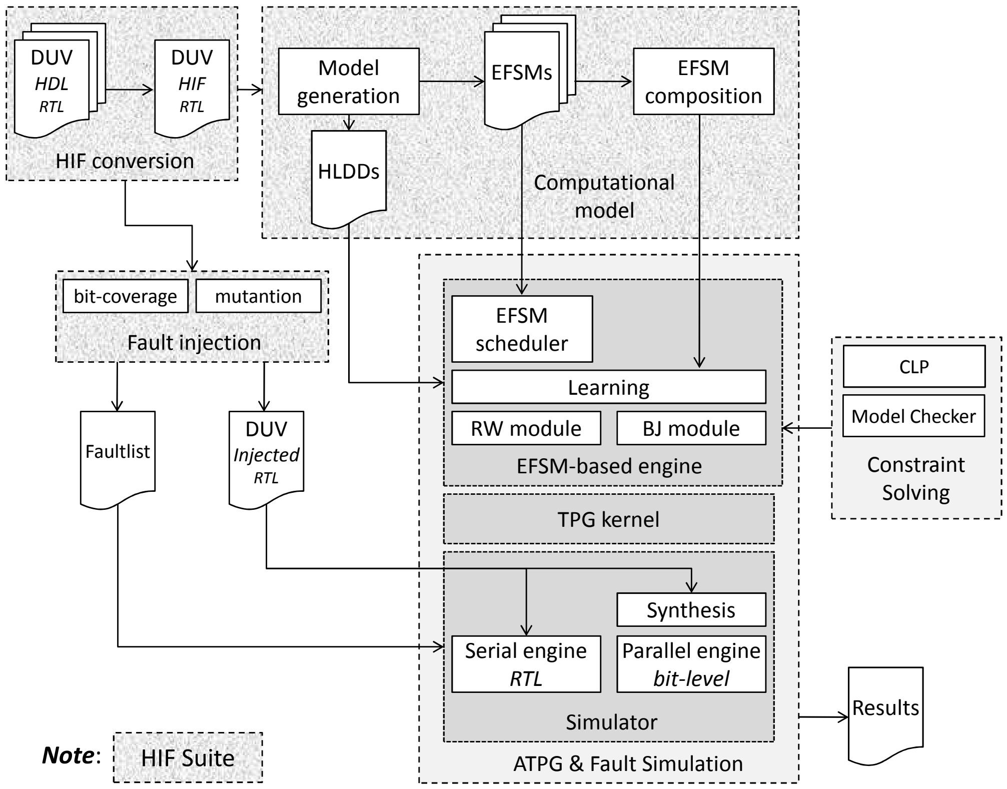


Fig. 3.1. Methodology flow.

3.1 Defining a model to represent the DUV

The main subject of this thesis is the definition and the practical implementation of a deterministic ATPG able to uniformly traverse the DUV state space and to cover hard-to-detect faults. Then, the first concern is to improve the traversal procedure of all the possible execution paths of the design description. In fact, differently from random-based ATPGs, a deterministic ATPG exploits the information embedded in the description of the DUV to explore its state space. Thus, its efficiency depends both on the adopted heuristics and on the model selected to represent the DUV.

The Extended Finite State Machine (EFSM) is an FSM that implicitly memorizes the values of the DUV registers, into transitions. Therefore EFSMs allow a compact representation of the DUV state space, reducing the risk of state explosion typical of more traditional FSMs. Moreover, the proposed variant of EFSM manages properly both synchronous and asynchronous modules in a uniform way.

In the current work, the EFSM model is adopted to describe the DUV. Then, a digital system is represented as a set of concurrent EFSMs, one for each process of the DUV. Many EFSMs can be generated starting from the same design description. However, despite from their functional equivalence, they can be more or less easy to be traversed. In the following sections a procedure to obtain a particular form of EFSM which is easier to be traversed. In the achieved EFSM model probabilities are uniformly distributed between the transitions out-going from a state, so that the ATPG can move more uniformly across the paths of the EFSM.

Finally, theoretical basis of EFSM composition has been proposed. The aim of composition is to improve functional ATPG whose effectiveness and efficiency may be limited when separate EFSMs are used to model the DUV.

3.2 Defining a functional deterministic ATPG engine

The proposed functional ATPG framework generates the test sequences by deterministically satisfy guards of the EFSMs transitions to traverse the DUV states space. The proposed ATPG exploits the EFSM model to guide a constraint solver during the DUV traversal end exploit the following techniques.

- A constraint solving strategy is adopted to deterministically generate test vectors that satisfy the guard of the EFSM transitions selected to be traversed.
- A two-step ATPG engine is defined which exploits the solver to traverse the DUV state space. At first, a random walk-based approach is used to cover the majority of easy-to-traverse transitions. Then a backjumping-based mode is used to activate hard-to-traverse transitions. In both modes, learning is exploited to get critical information that improves the performance of the ATPG.
- A scheduling algorithm has been defined to sort the EFSMs and deal with multi-process DUVs.
- As an alternative to the scheduling approach, the aim of proposed EFSM composition is to improve functional ATPG whose effectiveness and efficiency may be limited when separate EFSMs are used to model the DUV.
- A combined approach, involving EFSM and HLDD, is proposed to deterministically explore the design state space by using justification, learning and backjumping techniques.

3.3 Quantifying the effectiveness of generated test sequences

To evaluate the efficiency of this framework it is mandatory to adopt some metrics that guide the pattern generation and measure the performance of the developed deterministic ATPG. The bit coverage fault model is chosen, since it is related to design errors and it unifies into a single metrics the well known metrics concerning statements, branches and path coverage. Moreover, bit coverage allows to perform an automatic injection technique and shows a high correlation between gate-level stuck-at faults at different levels of abstraction.

Due to the fact that a device described in a hardware description language (HDL) at the functional level can be assimilated to a software program, functional faults are viewed as software faults. Therefore, this thesis proposes also the adaption of Mutation Analysis originally proposed for software testing. For this purpose, a set of mutation operators dedicated to HDL is selected and considered as a functional fault model.

Both these metrics are used to identify portions of the circuit not exercised by simulation. Test generation is guided by the goal of causing each mutant to fail or fault to be detected. Moreover, after all patterns have been generated, a fault coverage or mutation score is computed. It can be viewed as a reliability assessment for the design under validation, in the sense that higher the coverage/score, the higher our confidence that the design does not contain error.

The design manipulation, both computational model generation and fault injection, is build on the top of HIF Suite. The HIF Suite is a set of tools and application programming interfaces that provide support for modeling and verication of hardware and software systems.

3.4 Defining an efficient simulation engine

Verification via fault injection and fault simulation is a widely adopted technique to evaluate the correctness of a design implementation. However, the complexity of industrial designs and the huge number of faults that must be injected into them require efficient fault simulators, in order to make verification via fault simulation an affordable task. To optimize fault simulation performances, some parallelization techniques have been proposed at bit level. On the contrary, they have not been fully exploited at functional level, where functional fault models, instead of bit-level ones, are considered. Thus, this thesis analyzes the impact of such parallelization techniques on functional faults. In particular, possible issues are presented together with optimizations that can be implemented to speed up the simulation.

4

HDL Manipulation Infrastructure

The *HIF Suite* is a set of tools and application programming interfaces (APIs) that provide support for modeling and verification of HW/SW systems. The core of HIF Suite is the *HDL Intermediate Format* (*HIF*) language upon which a set of front-end and back-end tools have been developed to allow the conversion of HDL code into HIF code and vice versa. Thus, HIF Suite allows designers to manipulate and integrate heterogeneous components implemented by using different hardware description languages (HDLs). Moreover, HIF Suite includes tools, which rely on HIF APIs, for manipulating HIF descriptions in order to support code abstraction/refinement and post-refinement verification.

4.1 Introduction

The rapid development of modern embedded systems requires the use of flexible tools that allow designers and verification engineers to efficiently and automatically manipulate HDL descriptions throughout the design and verification steps.

From the modeling point of view, nowadays, it is common practice to define new systems by reusing previously developed components, that can be possibly modeled at different abstraction levels (TLM, RTL, etc.) by means of different hardware description languages like VHDL, SystemC, Verilog, etc.. Such an heterogeneity requires to either use co-simulation and co-verification techniques [180], or convert different HDL pieces of code into an homogeneous description [181]. However, co-simulation techniques slow down the overall simulation, while manual conversion from an HDL representation to another, as well as manual abstraction/refinement from an abstraction level to another, are not valuable solutions, since they are error-prone and time consuming activities. Thus, both co-simulation and manual refinement reduce the advantages provided by the adoption of a reuse-based design methodology. To avoid such disadvantages, this thesis's methodologies and tools exploit HIF Suite, a closely integrated set of tools and APIs for reusing already developed components and verifying their integration into new designs. Relying on the HIF language, HIF Suite allows system designers to convert HW/SW design descriptions from an HDL to another and to manipulate them in a uniform and efficient way. In addition, from the verification point of view, HIF Suite is intended to provide a single framework that efficiently supports many fundamental activities like transactor-based verification, mutation analysis, automatic test pattern generation, model checking, etc. Such activities generally require that designers and verification engineers define new components (e.g., transactors), or modify the design to introduce saboteurs, or represent the design by using mathematical models like EFSM, decision diagrams, Petri nets, etc. Nowadays there are no tools in the literature that integrate all the previous features in a single framework. HIF Suite is intended to fill in the gap.

An overview of the main features of HIF Suite is presented in Section 4.2, while the HIF core-language is described in Section 4.3. HIF-based conversion tools are presented in Section 4.4, while tools for extraction of mathematical models are described in Chapter 5, and a fault injection tool for verification is summarized in Chapter 7. Finally, concluding remarks are discussed in Section 4.5

4.2 HIFSuite Overview

Figure 4.1 shows an overview of HIF Suite features and components. The prototypal version of the majority of HIF Suite components have been developed in the context of three European Projects (SYMBAD, VERTIGO and COCONUT).

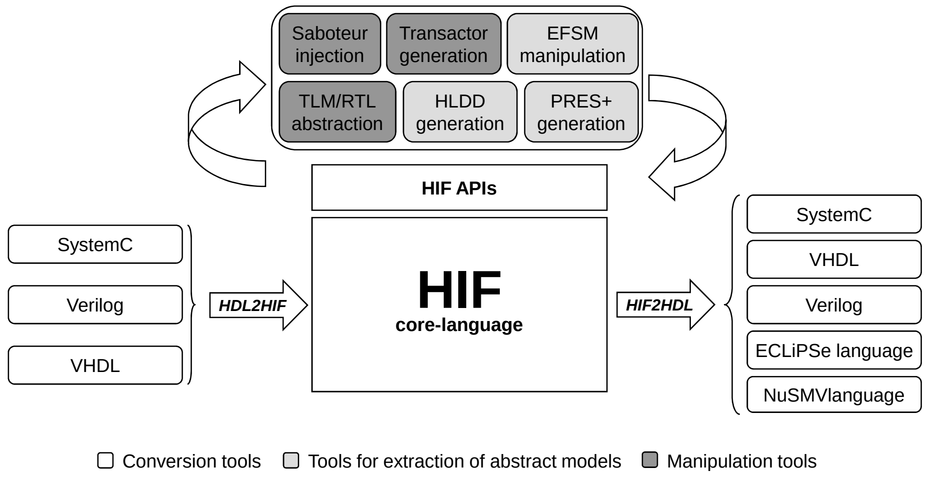


Fig. 4.1. HIF Suite overview

HIF Suite is composed of:

- *HIF core-language*: a set of HIF objects corresponding to traditional HDL constructs like, for example, processes, variable/signal declarations, sequential and concurrent statements, etc. (see Section 4.3).
- A set of front/back-end conversion tools (see Section 4.4), i.e.:

  - *HDL2HIF*: a front-end tool that parses VHDL and SystemC descriptions and generates the corresponding HIF representations. Extensions of such a tool for supporting also Verilog are under development.
  - *HIF2HDL*: a back-end tool that converts HIF models into VHDL, SystemC and Verilog code. The description languages of the NuSMV model checker [169] and the ECLiPSe constraint solver [166] are also supported.
- A set of APIs that allow designers to develop HIF-based tools to explore, manipulate and extract information from HIF descriptions (see Section 4.3.3). HIF code manipulated by such APIs can be converted to the HDLs supported by *HIF2HDL*.
- A set of tools developed upon the HIF APIs which extract abstract models from HIF code to support modeling and verification of HW/SW systems, i.e.:
  - *Phase1*: a tool that automatically EFSM models from HIF descriptions.
  - *HIF2HLDD*: a tool that automatically extracts HLDD models from HIF descriptions[1]. Both EFSM and HLDD models are a key components of the ATPG proposed in this thesis, in particular Chapter 5 describes methodology for EFSM and HLDD generation starting from a design description.
  - *HIF2PRES*: a tool that automatically extracts PRES+ (Petri-net based Representation of Embedded Systems) models from HIF descriptions[2]. A PRES+ model is a design representation based on Petri-nets, augmented with constructs that make them suitable for capturing important features of real-time embedded system designs [182]. Thus, PRES+ model is efficiently used for model checking of properties expressed in a timed temporal logic, and the approach is particularly suitable for, but not restricted to, models at an high-level of abstraction, such as transaction level.
- A set of tools developed upon the HIF APIs which manipulate HIF code to support modeling and verification of HW/SW systems, i.e.:
  - *ACIF*: a tool that automatically injects saboteurs into HIF descriptions. Saboteur injection is important to evaluate dependability of computer systems [183]. In particular, it is a key ingredient for verification tools that relies on fault models, like the automatic test pattern generators proposed in this thesis, tools that measure the fault coverage or evaluate the quality of testbenches through mutation analysis.
  - *TGEN*: a tool that automatically generates transactors. Verification methodologies based on transactors allow an advantageous reuse of testbenches, properties and IP-cores in TLM-RTL mixed designs, thus guaranteeing a considerable saving of time [184]. Moreover, transactors are widely adopted for the refinement (and the sub-sequence verification) of TLM descriptions towards RTL components [185].
  - *A^2T*: a tool that automatically abstracts RTL HIF code to TLM HIF code. Even if transactors allow designers to efficiently reuse RTL IPs at transaction level, mixed TLM-RTL designs cannot completely benefit of the effectiveness provided by TLM. In particular, the main drawback of IP reuse via

[1] *HIF2HLDD* has been developed in cooperation with the Department of Computer Engineering of the Tallinn Technical University (Estonia).

[2] *HIF2PRES* has been developed by the Embedded Systems Laboratory (ESLAB) of the Linkoeping University (Sweden).

transactor is that the RTL IP acts as a bottleneck of the mixed TLM-RTL design, thus slowing down the simulation of the whole system. Therefore, by using A^2T, RTL IPs can be automatically abstracted at the same transaction level of the other modules composing the design, to preserve the simulation speed typical of TLM, without incurring in tedious and error-prone manual abstraction [186, 187].

The main features of HIF core-language and HIF-based tools are summarized in next Sections.

4.3 HIF Core-Language and APIs

HIF stands for HDL Intermediate Format. It is an HW/SW description language structured as a tree of objects, similarly to XML. Each object describes a specific functionality or component that is typically provided by HDLs. However, even if HIF is quite intuitive to be read and manually written, it is not intended to be used for manually describing HW/SW systems. Indeed, it is intended to provide designers with a convenient way for automatically manipulating HW/SW descriptions as reported in Figure 4.1.

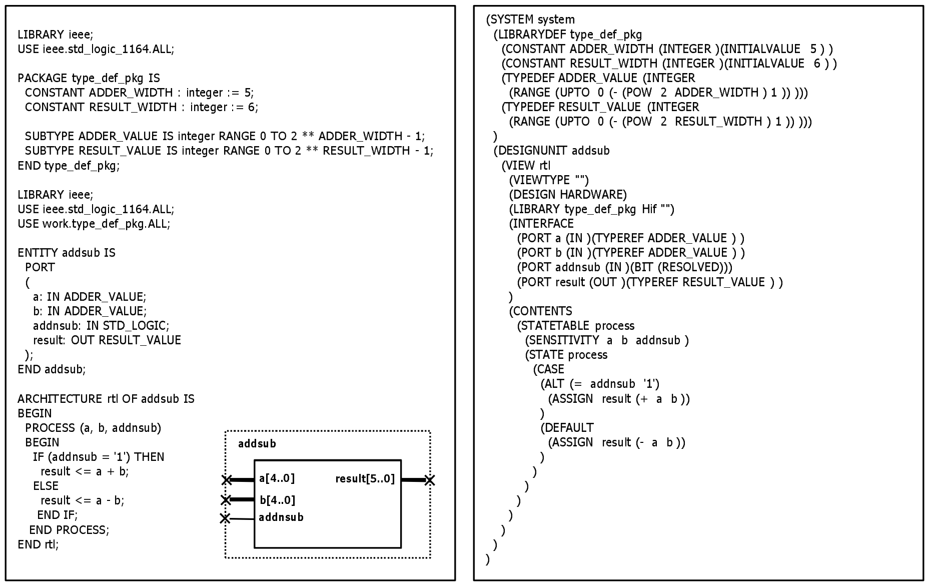
The requirements for HIF are manyfold, it must be capable to represent:

- system-level and transaction-level descriptions with abstract communication between system components;
- behavioral (algorithmic) hardware descriptions;
- register transfer level hardware descriptions;
- hardware structure descriptions;
- software algorithms.

To meet these requirements, HIF includes concepts which are inspired by different languages. Concerning RTL and behavioral hardware descriptions, HIF is very much inspired to VHDL and SystemC. Moreover, some constructs for the representations of algorithms (e.g., pointers and templates) have been taken from C/C++ programming language. The combination of these different features makes HIF a powerful language for HW/SW system representation.

4.3.1 HIF Basic Elements

HIF is a description language structured as a tree of elements, similarly to XML (see Figure 4.2). It is very much like a classical programming language, i.e., a typed language which allows the definition of new types, and includes operations like assignments, loops, conditional executions, etc. Moreover, since HIF is intended to represent hardware, it also includes typical low-level HDL constructs, as for example bit slices, which are globally the same as in VHDL. Finally, concerning the possibility of structuring a design description, the HIF language allows the definition of components and subprograms. To look at similarities between HIF and traditional HDLs, let us consider Figure 4.2. On the left side, a two-input parameterized adder/subtractor VHDL design is shown, while the corresponding HIF representation generated by *HDL2HIF* is depicted on the right.



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE type_def_pkg IS
  CONSTANT ADDER_WIDTH : integer := 5;
  CONSTANT RESULT_WIDTH : integer := 6;

  SUBTYPE ADDER_VALUE IS integer RANGE 0 TO 2 ** ADDER_WIDTH - 1;
  SUBTYPE RESULT_VALUE IS integer RANGE 0 TO 2 ** RESULT_WIDTH - 1;
END type_def_pkg;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.type_def_pkg.ALL;

ENTITY addsub IS
  PORT
  (
    a: IN ADDER_VALUE;
    b: IN ADDER_VALUE;
    addnsub: IN STD_LOGIC;
    result: OUT RESULT_VALUE
  );
END addsub;

ARCHITECTURE rtl OF addsub IS
BEGIN
  PROCESS (a, b, addnsub)
  BEGIN
    IF (addnsub = '1') THEN
      result <= a + b;
    ELSE
      result <= a - b;
     END IF;
   END PROCESS;
END rtl;
```

```
(SYSTEM system
  (LIBRARYDEF type_def_pkg
    (CONSTANT ADDER_WIDTH (INTEGER )(INITIALVALUE  5 ) )
    (CONSTANT RESULT_WIDTH (INTEGER )(INITIALVALUE  6 ) )
    (TYPEDEF ADDER_VALUE (INTEGER
      (RANGE (UPTO  0 (- (POW  2  ADDER_WIDTH ) 1 )) )))
    (TYPEDEF RESULT_VALUE  (INTEGER
      (RANGE (UPTO  0 (- (POW  2  RESULT_WIDTH ) 1 )) )))
  )
  (DESIGNUNIT addsub
    (VIEW rtl
      (VIEWTYPE "")
      (DESIGN HARDWARE)
      (LIBRARY type_def_pkg  Hif "")
      (INTERFACE
        (PORT a (IN )(TYPEREF ADDER_VALUE ) )
        (PORT b (IN )(TYPEREF ADDER_VALUE ) )
        (PORT addnsub (IN )(BIT (RESOLVED)))
        (PORT result (OUT )(TYPEREF RESULT_VALUE ) )
      )
      (CONTENTS
        (STATETABLE process
          (SENSITIVITY  a  b  addnsub )
          (STATE process
            (CASE
              (ALT (=  addnsub  '1')
                (ASSIGN  result (+  a  b ))
              )
              (DEFAULT
                (ASSIGN  result (-  a  b ))
              )
            )
          )
        )
      )
    )
  )
)
```

Fig. 4.2. (a) A VHDL design description. (b) The textual format of the corresponding HIF representation.

As special feature, HIF offers the possibility to add supplementary information to language constructs in form of so-called *properties*. A list of properties can be associated to almost every syntactic constructs of the HIF language. Properties allow designers to express information for which no syntactic constructs are included in the HIF grammar, and therefore they give a great flexibility to the HIF language. For example, the fact that a signal `s` has to be considered as a clock signal can be expressed by adding a property `signal_type` to the signal declaration as follows:

```
(SIGNAL s (BIT) (PROPERTY signal_type clock)).
```

4.3.2 System Description by using HIF

The top-level element of a system represented by an HIF description is the `SYSTEM` construct (see Figure 4.2(b)). It may contain the definition of one or more *libraries* which define new data types, constants and subprograms, and the description of *design units*. An HIF description may also contain a list of *protocols*, which describe communication mechanisms between design units.

Design units are modeled by `DESIGNUNIT` objects which define the actual components of the system. A design unit may use types, constants and subprograms defined in libraries included in the `SYSTEM` construct.

An `INTERFACE` object gives the link between a design unit and the rest of the system. An *interface* can contain ports, parameters, and accesses which represent a more abstract form of communication connections. In particular, `ACCESS` objects allow to link high-level communication channels to a design unit.

The same design unit can be modeled in different ways inside the same system by using *views*. For example, we can model different views of the same design unit at different abstraction levels. Thus, a VIEW object is a concrete description of a system component. It includes the definition of an interface by which the component communicates with the other parts of the system. Moreover, a view may include libraries and local declarations. The internal structure of a view is described in details by means of the CONTENTS construct.

A CONTENTS object can contain a list of local declarations, and either a list of state tables which describe sequential processes (in this case the view is called *behavioral*), or a list of component instances and nets which connect such instances (in this case the view is called *structural*). Furthermore, a CONTENTS object can contain a set of *concurrent actions* (also called *global actions*), i.e., assignments and procedure calls which assign values to a set of signals in a continuous manner.

Behavioral View

In HIF, behaviors described by sequences of statements (i.e., processes) are expressed by *state tables*.

A STATETABLE object defines a process, whose main control structure is an extended finite state machine, and the related sensitivity list. Note that state tables can describe sequential as well as combinational processes: in the case of a combinational process, the sensitivity list must contain all the signals which are read in the state table. The entry state of the state machine can be explicitly specified. Otherwise, the first state in the state list is considered as entry state.

STATE objects included in the state table are identified by a unique name, and they are associated to a list of instructions called *actions* (i.e., assignments, conditional statements, etc.) to be sequentially executed when the HIF model is converted into an HDL description for simulation.

Structural view

Structural descriptions, where more components are instantiated and connected each other are modeled by using the INSTANCE and the NET constructs.

An INSTANCE object describes an instance of a design unit. More precisely, an INSTANCE object refers to a specific view of the instantiated design unit.

A NET object contains either a list of access references or a list of port references. Nets are used to express connectivity between interface elements of different design unit instances (i.e., system components).

Concurrent actions

Concurrent actions, which correspond to concurrent assignments and concurrent procedure calls of VHDL, are modeled by GLOBALACTION objects.

Concurrent assignments are used to assign a new value to the target (which must be a signal or a port) each time the value of the assignment source changes. Similarly, concurrent procedure calls are used to assign a new value to signals mapped to the output parameters each time one of the input parameters changes its value.

Support for TML constructs

```
basic_response< DATA_TYPE > response;


basic_request< ADDRESS_TYPE,
                DATA_TYPE > request;


sc_port<tlm_blocking_put_if<basic_response<DATA_TYPE>>> OUTPUT_PORT;


sc_port<tlm_blocking_get_if<basic_request<ADDRESS_TYPE,
                                           DATA_TYPE > > > INPUT_PORT;
```

```
(VARIABLE response (TYPEREF basic_response
        (TYPETPASSIGN DATA_TYPE_RSP (TYPEREF DATA_TYPE ) ) ) )
 (VARIABLE request (TYPEREF basic_request
        (TYPETPASSIGN ADDRESS_TYPE_REQ (TYPEREF ADDRESS_TYPE ) )
        (TYPETPASSIGN DATA_TYPE_REQ (TYPEREF DATA_TYPE ) ) ) )
(VARIABLE OUTPUT_PORT (TYPEREF sc_port
        (TYPETPASSIGN SC_PORT_TYPE (TYPEREF tlm_blocking_put_if
        (TYPETPASSIGN BLK_IF_PUT_TYPE (TYPEREF basic_response
        (TYPETPASSIGN DATA (TYPEREF DATA_TYPE ) ) ) ) ) ) ) )
(VARIABLE INPUT_PORT (TYPEREF sc_port
        (TYPETPASSIGN SC_PORT_TYPE (TYPEREF tlm_blocking_get_if
        (TYPETPASSIGN BLK_IF_GET_TYPE (TYPEREF basic_request
        (TYPETPASSIGN ADDRESS (TYPEREF ADDRESS_TYPE ) )
        (TYPETPASSIGN DATA (TYPEREF DATA_TYPE ) ) ) ) ) ) ) )
```

Fig. 4.3. (a) Typical TLM interface with unidirectional channels and blocking calls in SystemC. (b) The corresponding HIF representation.

Transaction level modeling is becoming an usual practice for simplifying system-level design and architecture exploration. It allows the designers to focus on the functionality of the design, while abstracting away implementation details that will be added at lower abstraction levels. The prime example of transaction level modeling implementation is the SystemC TLM library, which exploits the extension capability of C++ language.

The HIF language supports SystemC TLM in the form of C++ constructs, by using, in particular, *pointers* and *templates*. Figure 4.3 presents a typical TLM interface with unidirectional channels and blocking calls in SystemC and the corresponding HIF representation. The SystemC interface definition exploits nested C++ templates which are preserved in the HIF description. The HIF language provides two keywords to support templates: `TYPETP` and `TYPETPASSIGN`. `TYPETP` is used for declaration of objects of template type. Instead `TYPETPASSIGN` is used for instantiation of templatized object as shown in Figure 4.3. Finally, in HIF the declaration of pointers is represented by using the POINTER object as follows: `(POINTER type {property})`.

4.3.3 HIF Application Programming Interfaces

HIF Suite provides the HIF language with a set of powerful C++ APIs which allow to explore, manipulate and extract information from HIF descriptions. There are two different subsets in HIF APIs: the *HIF core-language APIs* and the *HIF manipulation APIs.*

HIF core-language APIs

Each HIF construct is mapped to a C++ class which describes specific properties and attributes of the corresponding HDL construct. Each class is provided with a set of methods for getting or setting such properties and attributes.

For example, each assignment in Figure 4.2(b) is mapped to an `AssignObject` which is derived from `ActionObject` (see Figure 4.4). This class describes the assignment of an expression to a variable, a register, a signal, a parameter or a

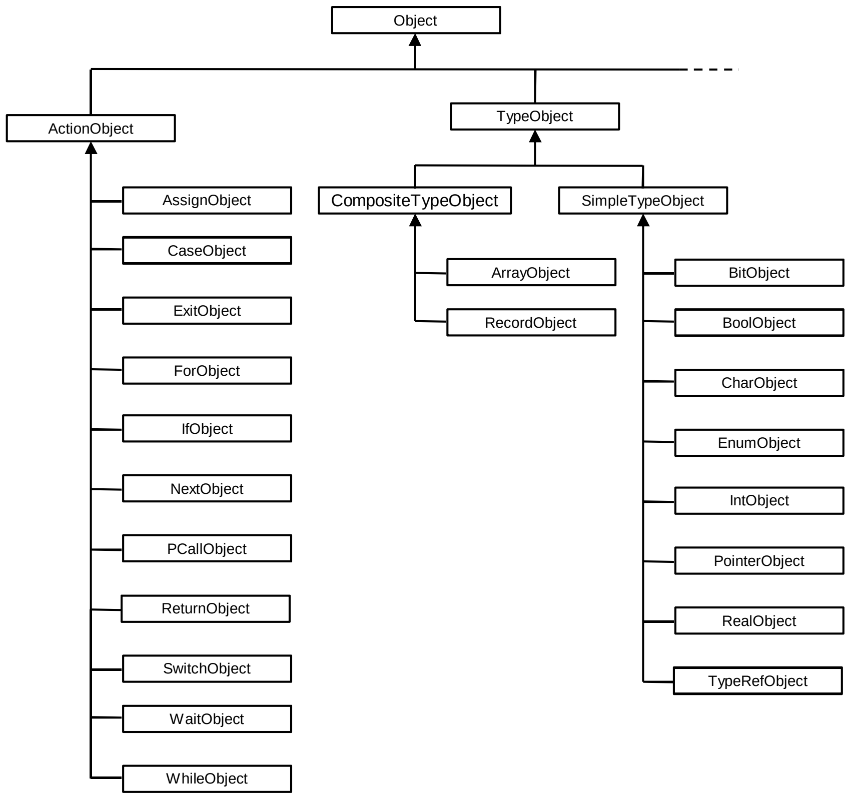


Fig. 4.4. A share of HIF core language class diagram

port, and it has two member fields corresponding to the left-hand side (target) and the right-hand side (source) of the assignment. Methods for setting and getting target and source are available.

The UML class diagram in Figure 4.4 presents a share of the HIF core-language APIs class diagram. `Object` is the root of the HIF class hierarchy. Every class in the HIF core-language APIs has `Object` as its ultimate parent.

HIF Manipulation APIs

The HIF manipulation APIs are used to manipulate the objects in HIF trees and they are exploited by the tools described in Chapter 5 and Chapter 7.

The first step for HIF manipulation consists of reading the HIF description by the following function:

```
Object* Hif::File::ASCII::read(const char* filename).
```

This function loads the file and build the corresponding tree data structure in memory. An analogous writing function allows to dump on a file the modified HIF tree:

```
Hif::hif_query query;
query.set_object_type(NameNode);        // search for NameNode
query.set_name("state");                // search for string "state"
std::list<Node*>* found_object = Hif::search(base_object, query);
```

Fig. 4.5. Search function usage example.

`char Aif::File::ASCII::write(const char* filename, Object* obj).`

Once the HIF file is loaded in memory, many APIs are available to navigate the HIF description, the most important ones are listed hereafter.

- *Search function.* The search function finds the objects which match criteria specified by the user. It searches the target objects starting from a given object until it reaches the bottom of the HIF tree (or the max depth, if the corresponding parameter is set). For example, the search function can be used to find out all variables which match the name `state` starting from `base_object`, as in Figure 4.5.
- *Visitor design pattern.* In object-oriented programming and software engineering, the visitor design pattern is generally adopted as a way for separating an algorithm from an object structure. A practical result of this separation is the ability to add new operations to existing object structures without modifying these structures. In fact, the visitor design pattern is very useful when there is a tree-based hierarchy of objects and it is necessary to allow an easy implementation of new features to manipulate such a tree. The HIF APIs provide visitor techniques in two forms: as an interface which must be extended to provide visitor operators, and as an `apply()` function. In the first case, a virtual method is inserted inside the HIF object hierarchy, which simply calls a specific-implemented visiting method on the object passed as parameter. The passed object is called *visitor* and it is a pure abstract class. Hence, the programmer has to extend such a visitor to visit and manage the HIF tree, by implementing the desired visiting methods, in accordance with its goals. On the contrary, the `apply()` function is useful to perform an user-defined function on all the objects contained in a subtree of a HIF description. The signature for the apply function is the following:

  `void Hif::apply (Object *o,char(*f)(Object *,void *),void *data).`

- *Compare function.* It provides designers with a way to compare two HIF objects and the respective subtrees. Its signature is the following:

  `static char compare (Object *obj1, Object *obj2)`

- *Object replacement function.* It provides designers with a way for replacing an object and its subtree with another one. Its signature is the following:

  `int Hif::replace(Object* from, Object* to)`

4.4 Conversion Tools

The conversion tools are organized into two main groups, according to their functionality, as follows:

- *HDL2HIF*: it contains front-end tools to convert code from traditional HDL languages to HIF. Currently, *HDL2HIF* supports conversions from VHDL and SystemC, which are implemented respectively in the submodules *VHDL2HIF* and *SC2HIF*. Extensions for supporting Verilog are under development.
- *HIF2HDL*: it contains back-end tools to convert HIF code back into VHDL (*HIF2VHDL*) or SystemC (*HIF2SC*). Extensions for supporting Verilog, and the languages of NuSMV and ECLiPSe are under development.

4.4.1 HDL2HIF

The *HDL2HIF* tools have a common structure, that can be summarized as follow:

- A pre-parsing module, which performs basic configuration operations and parameter parsing, and which selects the output format (readable plain text or binary).
- A parser, based on the GNU *Bison* tool, which creates an abstract syntax tree (AST) of the input code. Such an AST is composed of XML objects with dedicated tags.
- A core module which converts the AST into an HIF-objects tree. The conversion process is based on a recursive algorithm that exploits a pre-ordered visit strategy on the tree nodes.
- A post-conversion visitor, which refines the generated HIF tree according to the input language.
- A final routine, which dumps on file the HIF tree.

4.4.2 HIF2HDL

The *HIF2HDL* tools have been implemented by exploiting the HIF visitors. The structure of HIF2HDL tools can be summarized as follows:

- A pre-parsing module, which sets up the conversion environment, performs basic configuration operations, parses the parameters, and sets the output language.
- A set of refinement visitors, which perform operations to allow an easier translation of HIF trees, according to the output language. For instance, in VHDL it is possible to specify the bit value 1 by writing `'1'`, but in SystemC `'1'` is interpreted as a character, and thus a cast to the *sc_logic* type is required. To solve this problem, a visitor has been implemented to wrap the constant object `'1'` with an *sc_logic* cast object into the HIF tree.
- A module, which dumps on temporary files a partial conversion of the HIF code into the target language. Such a module has been implemented to solve problems of consistency between the order adopted to visit the HIF AST and the order needed to print out the code in the target language. Hence, to avoid

RTL constructs	**SC2HIF/HIF2SC**	**VHDL2HIF/HIF2VHDL**
Modules	x/x	x/x
Processes	x/x	x/x
Threads	x/x	
Functions	x/x	x/x (with restrictions)
Procedures	x/x	x/x (with restrictions)
Module Instances	x/x	x/x
Named port binding	x/x	x/x
Ports	x/x	x/x
Basic types	x (partially)/x	x/x
Derived types	x (partially)/x	x/x
Signals	x/x	x/x
Variables	x/x	x/x
Constants	x/x	x/x
Ports reading/writing	x/x	x/x
Wait statements	x/x	x/x
Posedge/negedge attributes	x/x	x/x
Pointers and references	x/x	
Operators on basic types	x/x	x/x
Assignment statements	x (with restrictions)/x	x/x
Conditional statements	x (with restrictions)/x	x/x
Loops	x (with restrictions)/x	x/x
Templates/generics	x (with restrictions)/x	x/x (with restrictions)

Table 4.1. RTL supported constructs.

a lot of complex checks, it is more convenient to firstly dump the output code directly in temporary files, and then to merge together the content of temporary files in the correct order.

- A post-visit module, which merges together the temporary files and creates the final output.

TLM constructs	**SC2HIF**	**HIF2SC**
sc_port / sc_export	x	x
get() / put() / peek()	x	x
nb_get() / nb_put() / nb_peek()	x	x
basic_status_write() / basic_status_read()	x	x
basic_slave_base	x	x
All the TLM templetized interfaces		x
Pointers and references	x (partially)	x
Instance method call	x	x
Instance method call via pointer dereferencing		x

Table 4.2. SystemC TLM constructs supported by *SC2HIF* and *HIF2SC*.

4.4.3 Supported HDL Constructs

Table 4.1 lists the RTL constructs supported by the front-end tools *SC2HIF* and *VHDL2HIF*, and the back-end tools *HIF2VHDL* and *HIF2SC*. Moreover, Table 4.2 reports the SystemC TLM constructs supported by *SC2HIF* and *HIF2SC*. Extensions for supporting further TLM constructs are under development. According to such tables, HIFSuite can be used to convert VHDL code into SystemC descriptions and viceversa, but it is worth noting that such languages are not fully equivalent or "compatibles". Hence, the conversion tools raise errors/warnings when HIF constructs derived from VHDL (SystemC) code have not a corresponding SystemC (VHDL) mapping. For example, conversion from SystemC TLM code to RTL VHDL code is not possible, since it would require a synthesis process that is currently not implemented in HIFSuite.

4.5 Concluding remarks

This chapter presented an overview of the HIF Suite, a set of tools and APIs that relies on the HIF language. The ATPG framework described in this thesis exploits the following provided features:

- *Conversion from VHDL/SystemC/Verilog to HIF and viceversa.* Current front-end and back-end tools support RTL VHDL/SystemC/Verilog constructs, and the core part of TLM SystemC. Extensions for supporting other languages (e.g., NuSMV and ECLiPSe) are under development, and will be available soon. HIF descriptions generated by front-end tools are structured like syntax trees, thus it is easy to write algorithms that manipulate the nodes of the tree.
- *Merging of mixed VHDL/SystemC descriptions.* Systems described partially in VHDL and partially in SystemC, can be converted to the HIF representation, and then merged to obtain a unique final description in SystemC or in VHDL.
- *Extendibility* The HIF library engine is structured to be easily extended. A special HIF object, called ProperyObject, is provided to describe non-standard or new features of other HIF objects.
- *HIF code manipulation.* A set of HIF-based manipulation tools are provided. Such tools can be used into modeling or verification workflows that adopt different HDL languages, regardless which these HDL languages are. In particular, computational model generation (Chapter 5) and fault injection (Chapter 7) activities involve HIF Suite tools.

HIF Suite releases can be downloaded from *http://www.edalab.it/HIFSuite*.

5

Methodology: Computational model

Extended Finite State Machines (EFSMs) can be efficiently adopted to model complex designs without incurring in the state explosion problem typical of the more traditional FSMs. However, traversing an EFSM can be more difficult than an FSM because the guards of EFSM transitions involve both primary inputs and registers.

This study analyzes the hardness of traversing a EFSM according to the format of its transitions. Then, it presents a methodology to generate an EFSM which is easy to be traversed. Such EFSM can be efficiently exploited by the proposed functional deterministic ATPG (Chapter 6). The ATPG joins backjumping, learning, and constraint logic programming (CLP) to efficiently explore the whole state space of the design under validation (DUV). Moreover, an EFSM-composition theory has been proposed to visit more efficiently the space of the state of analyzed system. Finally, a technique to automatically generate *High-Level Decision Diagrams* is proposed. The combined use of HLDDs and EFMSs for functional ATPG is presented at the end of (Chapter 6).

This Chapter, after an introduction of the problem (Section 5.1) and a background of the EFSM model (Section 5.2), presents an alternative approach to generate an EFSM model which is easy-to-traverse. To accomplish the goal, it is selected a pseudo-deterministic ATPG with a minimal set of features, which constitutes the basis for more evolutes ATPGs. Then, we define a probabilistic-based analysis to classify two different kinds of hard-to-traverse (HTT) transitions: input-dependent HTT transitions and register-dependent HTT transitions (Section 5.3). Then, this thesis proposes a methodology to avoid such a kind of transitions. It consists of a set of subsequent manipulations of the input-dependent hard-to-traverse EFSM to generate a new EFSM, where the probability of traversing the transitions is more uniformly distributed (Section 5.4). On the contrary, the register-dependent HTT transitions can be avoided by applying the stabilization process. However, differently from [9], not all the inconsistent transitions must be stabilized, but only those with a not uniformly distributed probability to be traversed (Section 5.5). Note that, these transformations are only related to functional validation and they do not impact on synthesis. In the follow, a particular variant of EFSM is defined (Section 5.6) to manage properly both synchronous and asynchronous modules in a uniform way. Theoretical basis are proposed to perform EFSM composition by bounding state and transition growth (Section 5.7). The aim of composition is to improve functional ATPG whose effectiveness and efficiency may be limited when separate EFSMs are used to model the system (see Section 6.3). Finally, the alternative HLDD paradigm is proposed in Section 5.8. HLDDs are generated starting from EFSMs, such as methodology is explained in Section 5.9.

The work is theoretically supported and an experimental confirmation is reported to show the effectiveness of the approach on different benchmarks (Section 5.10).

5.1 Introduction

The Finite State Machine (FSM) paradigm represented one of the most used and useful mathematical formalism to describe sequential circuits. However, the more and more increasing complexity of modern circuits leads FSM representations to the explosion of states. A valuable alternative to FSMs is represented by the Finite State Machine with Datapath (FSMD) paradigm [8]. In this case, the FSM model is used to describe only the control part of the system, while the datapath is modeled as a composition of combinatorial blocks and registers. This allows to sensibly reduce the number of states of the control part. The FSMD has showed to be a very attractive paradigm for both modeling and synthesis of sequential circuits [8]. However, the clean distinction between control and datapath represents a problem when automatic test pattern generation is adopted to test the design [188].

Another valuable alternative to FSMs is represented by the Extended Finite State Machine (EFSM) paradigm [9], which preserves many characteristics of an FSM and reduces the state explosion problem. In this case, control and datapath are mixed, but the number of states is sensibly lower with respect to a corresponding FSM, since the EFSM does not require an explicit representation of internal registers. The EFSM paradigm can be very effective, from a design point of view, to describe concurrent systems; moreover it has not been widely applied for verification, and particularly for automatic test pattern generation. The reason of this flop depends on the difficulty of traversing an EFSM, which is a fundamental requirement to control and to observe faults. In fact, moving from a state of the EFSM to another depends on the value of primary inputs, but maybe also on the value of internal registers. Such a kind of EFSM is note as inconsistent EFSM [9].

Some approaches have been proposed to identify and remove inconsistencies [9, 189, 190]. However, all of them can lead to the explosion of states, if the EFSM contains a large number of conditions on registers. Note that, this is a typical situation producing hard-to-detect faults. On the contrary, this thesis has define a methodology to generate different kinds of EFSMs starting form the same HDL description and shows that each of them is composed of transitions with a more or less uniformly distributed probability of being activated.

In particular, this study proposes a procedure that generates a new EFSM, starting from a *Reference EFSM* (REFSM) that is exactly equivalent to the HDL description of DUV. This goal is accomplished by successive steps. First, a reference EFSM (REFSM) is generated which is exactly equivalent to the DUV behavioral description. Then, the REFSM is manipulated to obtain a different EFSM, called the *Largest EFSM* (LEFSM), which is more easy to be traversed. However, the LEFSM is not equivalent to the original DUV. Thus, the LEFSM is further manipulated to obtain another EFSM which preserves the traversing characteristics of the LEFSM, but it is exactly equivalent to the REFSM.

This EFSM is called *Stabilized Smallest EFSM* (S^2EFSM) (see Section 5.5). Its state space can be more easily explored by a deterministic functional ATPG (Chapter 6), and controllability and observability of faults is greatly improved reducing the number of hard-to-detect faults.

All known approaches [9,189,190] in literature work on a single process design description. There are two possible solutions to deal with multi-process designs: to generate a set of concurrent EFSMs, one for each process, or to compose the processes into a single EFSM. The first solution requires to define an appropriate EFSM scheduling algorithm to maximize the ATPG capability of exploring the whole state space by uniformly traversing the concurrent EFSMs, as described in Section 6.1.3. A second solution, based on composing the given processes into a single EFSM, is proposed in Section 5.7. To perform EFSM composition, *Extended Event FSM* (EEFSM) is introduced. The EEFSM paradigm is suitable to model process statements with a sensitivity list, typical of hardware description languages (HDLs), such as VHDL, Verilog or SystemC. The sensitivity list, i.e., the set of events that traverse the execution of a simulation model, is attached to every EEFSM. If an event in the sensitivity list of some EEFSMs is enabled, those EEFSMs are executed. The composition theory of EEFSMs has been developed according to different system topologies.

5.2 The EFSM model

The first step in designing a digital system consists of formalizing its functionality according to a computational model. Many design methodologies have been proposed in the literature; their computational models fall into three distinct categories [191]: (a) state-oriented, (b) activity-oriented, (c) structure-oriented. The first category represents the system as a set of transitions among states triggered by external events (e.g., FSMs). Thus, state-oriented models are suited for control systems. The second category is intended to describe the systems as a set of activities related by data and execution dependencies (e.g., dataflow graphs). Thus, it is suited for modeling data-dominated systems, where data pass from an activity to the other in a pipelined fashion. The third category is used to describe the system as an interconnection of basic components (e.g., block diagrams). Thus, structure-oriented models are less suited than the other categories to specify the functionality, but they are more convenient at lower abstraction levels (e.g., RTL), where component reuse is very common. Finally, many of the characteristics typical of the previously cited categories are merged into heterogeneous models (e.g., program-state machines) to describe different views of complex systems.

In this thesis, the proposed approach represents a digital system as a set of concurrent EFSMs, one for each process of the DUV. In this way, according to the below Definition 1, we capture the main characteristics of state-oriented, activity-oriented and structure-oriented models. In fact, the EFSM is composed of states and transitions, thus it is state-oriented, but each transition is extended with HDL instructions that act on the DUV registers. In this sense, each transition represents a set of activities on data, thus, the EFSM is a data-oriented model too. Finally, concurrency is intended as the possibility that each EFSM of the same DUV changes its state concurrently to the other EFSMs to reflect the concurrent execution of the corresponding processes. Data communication between concurrent EFSMs is guaranteed by the presence of common signals. In this way, structured models can be represented.

Definition 1 *An EFSM is defined as a 5-tuple $M = \langle S, I, O, D, T \rangle$ where: S is a set of states, I is a set of input symbols, O is a set of output symbols, D is a n-dimensional linear space $D_1 \times \ldots \times D_n$, T is a transition relation such that $T : S \times D \times I \to S \times D \times \ O$. A generic point in D is described by a n-upla $x = (x_1, ..., x_n)$; it models the values of the registers of the DUV. A pair $\langle s, x \rangle \in S \times D$ is called configuration of M.*

An operation on M is defined in this way: if M is in a configuration $\langle s, x \rangle$ and it receives an input $i \in I$, it moves to the configuration $\langle t, y \rangle$ iff $((s, x, i), (t, y, o)) \in T$ for $o \in O$.

The EFSM differs from the classical FSM, since each transition does not present only a label in the classical form $(i)/(o)$, but it takes care of the register values too. Transitions are labeled with an *enabling* function e and an *update* function u defined as follows.

Definition 2 *Given an EFSM $M = \langle S, I, O, D, T \rangle$, $s \in S, t \in T, i \in I, o \in O$ and the sets $X = \{x | ((s, x, i), (t, y, o)) \in T \text{ for } y \in D\}$ and $Y = \{y | ((s, x, i), (t, y, o)) \in T \text{ for } x \in X\}$, the enabling and update functions are defined respectively as:*

$$e(x, i) = \begin{cases} 1 \ \textit{if } x \in X; \\ 0 \ \textit{otherwise.} \end{cases}$$

$$u(x, i) = \begin{cases} (y, o) & \textit{if } e(x, i) = 1 \textit{ and } ((s, x, i), (t, y, o)) \in T; \\ undef. & \textit{otherwise.} \end{cases}$$

An update function $u(x, i)$ can be applied to a configuration $\langle s_1, x \rangle$ if there is a transaction $t : s_1 \to s_2$, labeled e/u, such that $e(x, i) = 1$. In this case we say that t can be *traversed* by applying the input i.

Definition 3 *Two EFSMs, M_1, M_2, are functionally equivalent if for each sequence of input values provided to M_1 and M_2, they provide the same sequence of output values.*

It is worth noting that different, but functionally equivalent, EFSMs can be extracted from the same DUV description. Figure 5.1 shows a simple DUV description, coded by using a hardware description language (HDL). A corresponding EFSM is showed in Figure 5.2.

5.3 Classification of EFSM Transitions

Differently from random-based ATPGs, a deterministic ATPG exploits the information embedded in the description of the DUV to explore its state space. Thus, its efficiency depends on the adopted heuristics, but also on the model selected to represent the DUV. For this reason, a goal of this thesis consists of formalizing a procedure to obtain a particular form of EFSM which is easy to be traversed by using a wide class of deterministic ATPGs. To characterize such an EFSM, an ATPG with a very basic heuristic is assumed as reference.

5.3.1 Reference ATPG

The reference ATPG tries to *uniformly move* across the transitions of the EFSM by exploiting the information provided by the enabling functions. On the contrary, a random ATPG tends to traverse only transitions whose enabling functions present a high probability of being satisfied.

The reference ATPG starts from the reset state of the EFSM, it randomly selects an out-going transition, and it tries to satisfy its enabling function by assigning values to inputs. When it successes, it moves to the corresponding destination state, it selects another out-going transition from this state, and so on. More formally, given the set T_{s_i} of transitions out-going from a state s_i, at step i, the ATPG algorithm works as follows:

1. Randomly choose a transition $t_{s_i} \in T_{s_i}$.
2. Check if the enabling function e of t_{s_i} can be traversed by assigning opportune values to inputs involved in e. (For example, let `x` be an input of the EFSM, the enabling function `x=0` can be traversed by assigning `0` to `x`. Otherwise, if `x` is an internal register, the satisfiability of the enabling function depends on the previous assignment to `x`, i.e., on the current configuration of the EFSM[1]).
3. If e is satisfiable, assign to inputs involved in e such opportune values. Otherwise, remove t_{s_i} from T_{s_i} and come back to step 1.
4. Generate random values for inputs not involved in e.
5. Simulate the obtained test vector, move across the transition t_{s_i}, and come back to step 1 to generate the next test vector.

The ATPG stops when the target fault coverage has been reached or the maximum allowed computation time has been expired.

5.3.2 ATPG Efficiency

The efficiency of this reference ATPG depends on the probability of satisfying the enabling function of the transitions. If such probabilities are uniformly distributed between the transitions out-going from a state, the ATPG can move more uniformly across the paths of the EFSM. Thus, different kinds of EFSM, derived from the same DUV, can provide different probabilities of reaching the same portion of code. Some definitions are needed at first to characterize such EFSM models.

[1] A configuration stores the status of the EFSM, i.e., the value of its internal registers (see the definition of EFSM in Section 5.2)

Definition 4 *Let $M = \langle S, I, O, D, T\rangle$ an EFSM, n the maximum length of test sequences generated by the ATPG, $t \in T$ a transition from $s_{k-1} \in S$ to $s_k \in S$, $\Gamma = \{\gamma = (s_0, \ldots, s_k) \mid \gamma$ is a path on M that traverses $t, s_i \in S$ for $0 \leq i \leq k, lenght(\gamma) \leq n\}$, $T_{s_i} = \{t_{s_i} | t_{s_i}$ is a transition out-going from $s_i \in S\}$, and $|T_{s_i}|$ the cardinality of T_{s_i}, the probability of traversing t is defined as follows:*

$$P(traversing\ t) = \sum_{\gamma \in \Gamma} P(traversing\ \gamma) \tag{5.1}$$

$$P(traversing\ \gamma) = \prod_{i=0}^{lenght(\gamma)-2} \frac{1}{|T_{s_i}|} \cdot P(s_i \rightarrow_{t_{s_i}} s_{i+1}) \tag{5.2}$$

where $P(s_i \rightarrow_{t_{s_i}} s_{i+1})$ is the probability that the configuration $\langle s_i, x\rangle$, originated by reaching s_i, allows to traverse the transition $t_{s_i} \in T_{s_i}$ from s_i to s_{i+1}.

Definition 5 *Let $M = \langle S, I, O, D, T\rangle$ an EFSM and t a transition of M, t is hard-to-traverse (HTT) if $P(traversing\ f) \simeq 0$. Otherwise, t is easy-to-traverse (ETT).*

Definition 6 *Let $M = \langle S, I, O, D, T\rangle$ an EFSM and γ a path on M, γ is hard-to-traverse (HTT) if it includes an HTT transition. Otherwise, γ is easy-to-traverse (ETT).*

Definition 7 *Let $M = \langle S, I, O, D, T\rangle$ an EFSM and f a fault in M, f is hard-to-detect (HTD) if all test sequences of f are HTT paths. Otherwise, f is easy-to-detect (ETD).*

The most important aspect of Definition 4 is represented by $P(s_i \rightarrow_{t_{s_i}} s_{i+1})$. If the enabling function e of the transition t_{s_i} involves only inputs, then $P(s_i \rightarrow_{t_{s_i}} s_{i+1}) = 1$ by using the reference ATPG. In fact, the ATPG can assign the opportune values to inputs to satisfy e. Such a kind of transitions is always ETT for the proposed ATPG. Otherwise, if e involves also registers, $P(s_i \rightarrow_{t_{s_i}} s_{i+1})$ depends on the choices made by the ATPG moving across previous transitions, and it can tend to 0, thus t_{s_i} becomes HTT. In particular, there are two kinds of HTT transitions: *input-dependent HTT transitions* and *register-dependent HTT transitions*. The hardness of the first kind of transitions depends on the presence of conditional statements, which involve inputs, into the update functions. On the contrary, the hardness of the second kind of transitions depends on the presence of enabling functions whose conditions involve registers.

5.3.3 Input-dependent HTT transitions

An EFSM with input-dependent HTT transitions can be modified to obtain a different EFSM without such transitions. Consider, for example, a fault f related to the block B10 of the HDL description of Figure 5.1, and the two equivalent EFSMs M and M' of Figure 5.2 and Figure 5.3. The path with the minimum length to activate f on M' is $\gamma' = (S_0, S_0, S_0, S_0)$ by traversing the sequence of transitions (t'_0, t'_3, t'_1). Once the ATPG has traversed the enabling function of t'_3, which is definitely ETT, it can randomly choose between 2^{32} values for `in1`, since

```
if reset = '1' then
   state:=A;
else if clock'event and clock='1'
   case state is
      when A =>
B0          if in1!=0 then
B3             reg:=in1;
               out1<=1;
               out2<=1;
               state:=B;
            else
B4             reg:=in2;
               out1<=0;
               out2<=0;
               state:=C;
            end if
      when B =>
B1  B5      if reg=1
               out1 <= 0;
            else
    B6         out1 <=reg;
            reg:=reg*in1;
B7          if reg!=1
               out2<=reg*2;
               state:=A;
            else
B8             out2<=0;
               state:=B;
      when C =>
B2 B9       if in2!=0 then
               reg:=reg+in2;
               out1<=1;
               out2<=1;
               state:=C;
   B10      else
               if in1=0 then
   B11            out1<=reg;
                  out2<=reg/2;
               else
   B12            out1<=reg/2;
                  out2<=reg;
               end if
               state:=A;
   end case
end if
```

Fig. 5.1. A simple example of a FSMD.

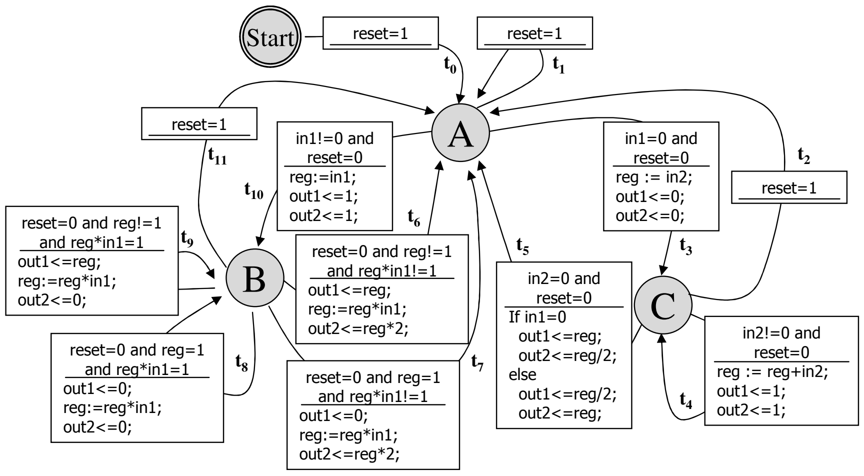


Fig. 5.2. An EFSM which models the HDL code of Figure 5.1.

`in1` is not involved in the enabling function. However, only if the ATPG sets `in1` at `0`, the register `state` assumes the value `C`, allowing to traverse the enabling function of t'_1. Thus, $P(S_0 \rightarrow_{t'_1} S_0) = 1/(2^{32}) \simeq 0$ and t'_1 is HTT. In particular,

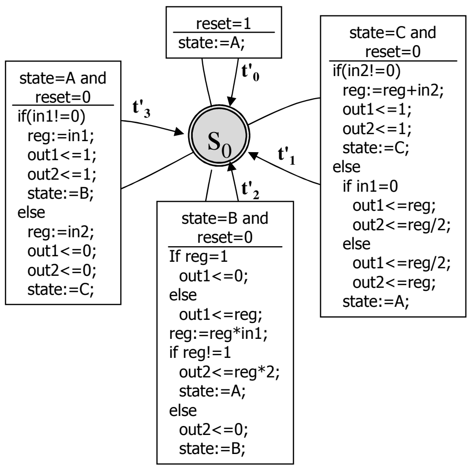


Fig. 5.3. Another EFSM for the code of Figure 5.1.

according to Definition 4 and considering paths of length 4:

$$\begin{aligned} &P(traversing\ t_1\ on\ M') = P(traversing\ \gamma') = \\ &= (1/4 \cdot P(t'_0)) \cdot (1/4 \cdot P(t'_3)) \cdot (1/4 \cdot P(t'_1)) \\ &= (1/4 \cdot 1) \cdot (1/4 \cdot 1) \cdot \left(1/4 \cdot 1/2^{32}\right) \simeq 0 \end{aligned}$$

Thus, f is HTD on M', because every path that activates f is HTT, since it must include t'_1. Now, consider M. In this case, the path with the minimum length to activate f on M is $\gamma = (Start, A, C, A)$ by traversing the sequence of transitions (t_0, t_3, t_5). In particular, $P(traversing\ t_5\ on\ M)$ is quite high, since the ATPG, running on M, realizes that, to activate t_3, `in1` must be set to `0`, and to activate t_5, `in2` must be set to `0`. Such information is extracted by analyzing the enabling functions of t_3 and t_5 without the need of complex backtracking-based or learning-based heuristics. According to Definition 4 and considering paths of length 4:

$$\begin{aligned} &P(traversing\ t_5\ on\ M) = P(traversing\ \gamma) = \\ &= (1 \cdot P(t_0)) \cdot (1/3 \cdot P(t_3)) \cdot (1/3 \cdot P(t_5)) = \\ &= (1 \cdot 1) \cdot (1/3 \cdot 1) \cdot (1/3 \cdot 1) = 1/9 \end{aligned}$$

Thus, γ is an ETT path, and f is ETD on M.

The previous example shows that input-dependent HTT transitions can be avoided by manipulating the EFSM description. Section 5.4 describe the methodology defined to manipulate the EFSM model.

5.3.4 Register-dependent HTT transitions

This kind of transitions is HTT for ATPGs which do not exploit a backtracking-based or learning-based heuristic. For example, consider a fault f in the block B5 of Figure 5.1. To activate f by using the EFSM model of Figure 5.2, the ATPG must move from A to B on t_{10} and then from B to B on t_8 or from B to A on t_7. Since `in1` is a 32-bit integer, the ATPG traverses t_{10} by fixing `reset` at `0` and by choosing between $2^{32} - 1$ values, different from `0`, for `in1`. Then, the ATPG can generate $2^{32} - 1$ different admissible configurations, when it moves on t_{10}. However, only the configuration where `reg=1` (obtainable by fixing `in1` at `1`) is valid to traverse t_8 or t_7. Thus, $P(B \rightarrow_{t_8} B) = P(B \rightarrow_{t_7} A) = 1/(2^{32} - 1) \simeq 0$ and t_8, t_7 are HTT. Consequently, faults related to the update function of t_8 and t_7 are HTD for ATPGs which implement a heuristic that exploits information local to the current configuration only. Note that, such faults are HTD also on M'.

This kind of HTT transitions can be avoided by applying the stabilization process presented in [192], even if in some particular cases it can lead to the explosion of states. However, this problem is limited, since stabilization is necessary only for register-dependent HTT-transitions. A register-dependent transition with a high probability of being traversed does not require to be stabilized. In Section 5.5 we show how stabilization is used to avoid register-dependent HTT transitions.

5.4 Avoiding Input-dependent HTT transitions

Input-dependent HTT transitions are due to the presence of conditional blocks embedded in the update functions of the EFSM. In fact, a conditional statement in an update function hides useful information from the reference ATPG, which is able to analyze only enabling functions. For example, t'_1 in Figure 5.3 is an input-dependent HTT transition because the ATPG cannot exploit the information hidden in the `if` statement included in the update function of t'_3. Thus, input-dependent HTT transitions can be avoided by making explicit, in the enabling functions, the conditional blocks embedded in the update functions. This requires a deep modification of the original EFSM, which consists of the following steps:

1. An EFSM is extracted starting from the HDL description of the DUV. Because many EFSMs can be equivalent to such a description, Section 5.4.1 proposes an algorithm to generate a particular kind of EFSM which allows us to standardize the following steps of the methodology. We call such an EFSM the *Reference EFSM* (REFSM).
2. The REFSM is modified to remove input-dependent HTT transitions as described in Section 5.4.2. The obtained EFSM is called *Largest EFSM* (LEFSM) because it is the EFSM without input-dependent HTT transitions with the largest number of states.
3. The LEFSM is optimized by grouping compatible transitions as described in Section 5.4.3. This process further increases the capability of the ATPG of uniformly traversing the STG of the EFSM. The obtained EFSM is called *Smallest EFSM* (SEFSM) because it is the EFSM without input-dependent HTT transitions with the smallest number of states.

5.4.1 Generation of the Reference EFSM

Given an HDL description of the DUV that reflects the FSMD template, the REFSM is obtained by applying an algorithm which is linear on the number of alternatives of the `case` statement.

The REFSM presents a very simple STG, it is composed of a unique state and n transitions, one for the slice of code executed when `reset=1`, and one for each alternative of the `case` statement. For example, Figure 5.3 shows the STG of the REFSM generated for the HDL description of Figure 5.1. The REFSM is equivalent to the HDL description, however, as shown in Section 5.3.2, it may include many HTT transitions.

5.4.2 Generation of the Largest EFSM

To avoid input-dependent HTT transitions, the REFSM must be modified as follows. Transitions, whose update functions include conditional statements, are substituted by subgraphs. Then, the update functions of each subgraph transition contain only assignments. Figure 5.4 shows the algorithm that implements this idea to generate the LEFSM. The conditional statements included in an update function of the REFSM are extracted, and for every condition two new transitions are generated: one for each truth value of the condition. The complexity of the LEFSM generation is linear with respect to the number of conditional statements included in the update functions of the REFSM.

Figure 5.5 presents the STG of the LEFSM obtained from the REFSM represented in Figure 5.3. While the REFSM is functionally equivalent to the initial HDL description, the LEFSM may require a larger number of clock cycles to behave exactly as the REFSM. Every transition t of the REFSM corresponds to a path on a subgraph of the LEFSM. For example, traversing the transition t_3' on Figure 5.3 corresponds to traversing the transitions t_1^L, t_2^L or t_1^L,t_3^L on Figure 5.5, according to the value assigned to `in1`. This may cause that a test sequence generated by considering the LEFSM looses its efficacy when simulated on the original HDL description. A not optimized solution to convert test sequences generated for the LEFSM in test sequences for the original description is proposed in [193]. However, such an approach does not guarantee that each fault tested on the LEFSM can be tested also on the original design. Thus, test generation on the LEFSM may produce a false sense of security. To avoid this problem, the LEFSM can be manipulated as described in the next section.

5.4.3 Generation of the Smallest EFSM

The SEFSM is obtained from the LEFSM by composing compatible transitions to possibly remove the intermediate state according to the following definition.

Definition 8 *Given a transition t_{ij} from S_i to S_j with enabling function e_{ij} and update function u_{ij}, and a transition t_{jk} from S_j to S_k with enabling function e_{jk} and update function u_{jk}, t_{ij} is compatible with t_{jk} if e_{ij} and e_{jk} are not conflicting, and u_{ij} does not contain assignments to variables involved in e_{jk}.*

```
build_LEFSM (REFSM $M^R$) {
 create the initial state $S_0^L$
 for each transition $t_j^R$ of $M^R$
   if the update function of $t_j^R$ does **not** contain
   conditional statements
     add a transition $t_j^R$ from $S_0^L$ to $S_0^L$ exactly equivalent to $t_j^R$
    else
     add a new state $S_k^L$
     add a transition $t_j^L$ from $S_0^L$ to $S_k^L$ such that
       the enabling function of $t_j^L$ is the same of $t_j^R$
     analyze_update_function(update_function_of_ $t_j^R$, $t_j^L$, $S_k^L$)
}

analyze_update_function(update_function $u_j^R$,transition $t_j^L$,state $S_k^L$){
 for each statements $i$ of $u_j^R$
   if $i$ is a conditional statement with condition $c$
     add a new state $S_c^L$
     add a transition $t_c^L$ from $S_k^L$ to $S_c^L$ with enabling function $c$
     analyze_update_function(then_body_of_$i$, $t_c^L$, $S_c^L$)
     add a transition $t_{\neg c}^L$ from $S_k^L$ to $S_c^L$ with enabling function $\neg c$
     analyze_update_function(else_body_of_$i$, $t_{\neg c}^L$, $S_c^L$)
   else if $i$ is an assignment
     add $i$ to the update function of $t_j^L$
}
```

Fig. 5.4. Algorithm to generate the LEFSM.

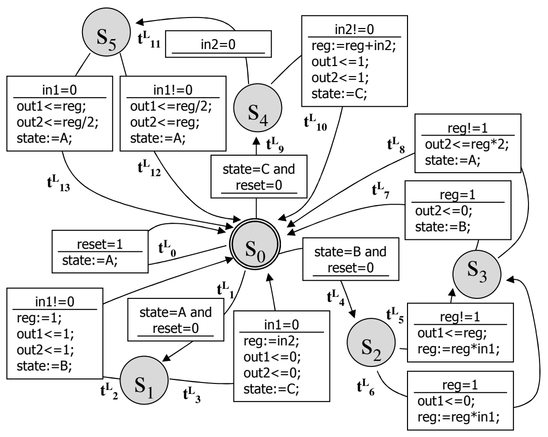


Fig. 5.5. LEFSM generated from the REFSM of Figure 5.3.

Now consider Figure 5.6. In (a), s_i has $m+n$ out-going transitions, m of which go to s_j, and s_j has $p+q$ out-going transitions, p of which go to s_k. If a transition t_{ij} from s_i to s_j is compatible with all transitions out-going from s_j, then t_{ij} can be removed and substituted by $p+q$ new transitions as shown in (b). In particular, for

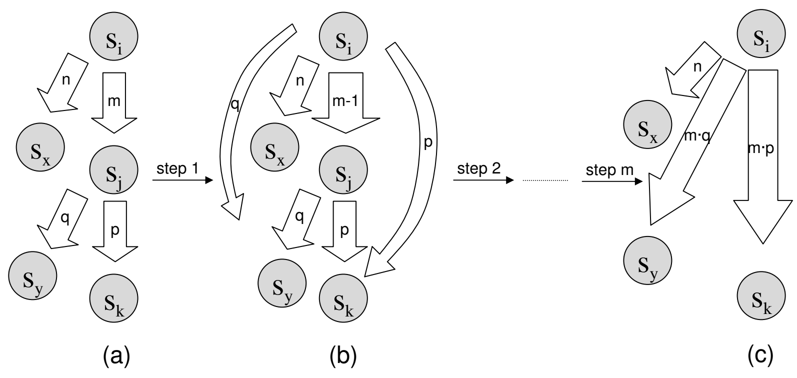


Fig. 5.6. Composition of compatible transitions.

each transition $t_{j\_}$ out-going from s_j a new transition $t_{i\_}$ is added from s_i to $s_\_$ such that: the enabling function of $t_{i\_}$ is $e_{i\_} = e_{ij} \wedge e_{j\_}$, and the update function of $t_{i\_}$ is $u_{i\_} = u_{ij} \cup u_{j\_}$. Besides, if all the transitions out-going from s_i are compatible with all the transitions out-going from s_j, then, after m steps, we obtain the situation shown in (c), where s_j has been removed.

The SEFSM is obtained by iteratively applying the previous transition collapsing procedure on the states of the LEFSM until no compatible transitions are found. The complexity of each iteration is given by the product of the number of transitionsin-going in S_j and the number of transitions out-going from S_j. Besides, there can be at maximum $|S^L|-1$ iterations, where S^L is the set of states of the LEFSM. Figure 5.7 shows the STG of the SEFSM obtained from the LEFSM represented in of Figure 5.5.

The SEFSM presents some advantages with respect to REFSM and LEFSM. First, it does not contain input-dependent HTT transitions. In fact, it derives from the LEFSM and the collapsing procedure does not introduce conditional statements in the update functions. Secondly, the SEFSM reduces the problem related to test sequence conversion which appears moving from the REFSM to the LEFSM. In fact, by applying the collapsing procedure, the number of states of the SEFSM is reduced with respect to the LEFSM. This reduces the size of the subgraph of the SEFSM that corresponds to a single transition of the REFSM. Finally, the probability of traversing the transitions of the SEFSM is more uniformly distributed than in the LEFSM. Thus, the reference ATPG can explore the state space more uniformly. This is shown by the next theorems and corollary.

Theorem 1 *Let M^L an LEFSM and $\langle M_0^S, ..., M_l^S, ..., M_n^S \rangle$ the sequence of EFSMs obtained from M^L by iterating the transition collapsing procedure (M_n^S is the SEFSM), and let s_i, s_j and s_k the states involved changing from M_l^S to M_{l+1}^S as shown in Figure 5.6(a). The probability of traversing a transition t out-going from s_k, reaching s_k starting from s_i, on M_{l+1}^S is greater than the same probability on M_l^S.*

Consider Figure 5.6. According to Definition 4 and the reference ATPG, the probability of traversing t depends on the probability of reaching s_k from s_i. In

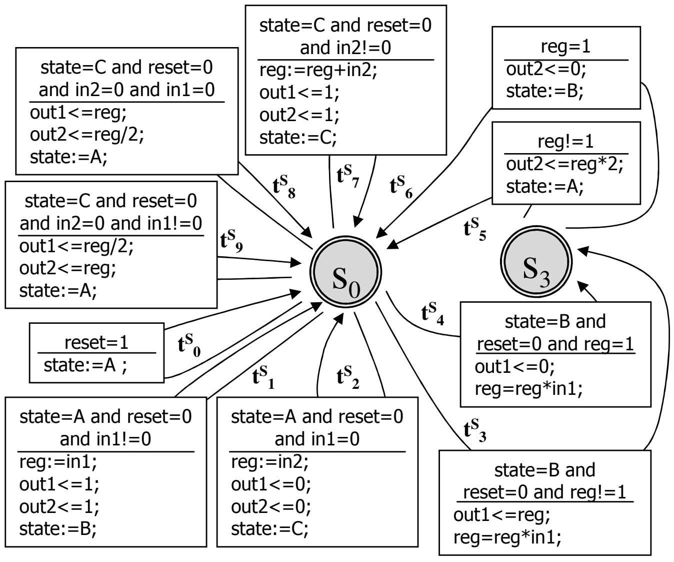

state=C and reset=0 and in2=0 and in1=0
out1<=reg; out2<=reg/2; state:=A;
state=C and reset=0 and in2!=0
reg:=reg+in2; out1<=1; out2<=1; state:=C;
reg=1
out2<=0; state:=B;
reg!=1
out2<=reg*2; state:=A;
state=C and reset=0 and in2=0 and in1!=0
out1<=reg/2; out2<=reg; state:=A;
reset=1
state:=A ;
state=A and reset=0 and in1!=0
reg:=in1; out1<=1; out2<=1; state:=B;
state=A and reset=0 and in1=0
reg:=in2; out1<=0; out2<=0; state:=C;
state=B and reset=0 and reg=1
out1<=0; reg=reg*in1;
state=B and reset=0 and reg!=1
out1<=reg; reg=reg*in1;
S0
S3
tS0
tS1
tS2
tS3
tS4
tS5
tS6
tS7
tS8
tS9

Fig. 5.7. SEFSM generated from the LEFSM of Figure 5.5.

Figure 5.6(a), this probability is $P^0_{ik} = \frac{m}{n+m} \cdot \frac{p}{p+q} \cdot X$, where X is the factor deriving from the presence of possible HTT transitions. X can only depend on the presence of register-dependent HTT transitions, since an LEFSM does not contain input-dependent HTT transitions. Thus, X is not removed by applying the collapsing procedure and it can be omitted in the following. After one step of the collapsing procedure (See Figure 5.6(b)), the probability of reaching s_k from s_i is

$$\begin{aligned} P^1_{ik} &= \frac{m-1}{n+m-1+p+q} \cdot \frac{p}{p+q} + \frac{p}{n+m-1+p+q} \cdot X \\ &= \frac{p}{p+q} \cdot \frac{m-1+p+q}{n+m-1+p+q} \cdot X \end{aligned}$$

But, $P^1_{ik} > P^0_{ik}$, in fact:

$$\begin{aligned} P^1_{ik} > P^0_{ik} &\Leftrightarrow \frac{p}{p+q} \cdot \frac{m-1+p+q}{n+m-1+p+q} > \frac{m}{n+m} \cdot \frac{p}{p+q} \\ &\Leftrightarrow \frac{m-1+p+q}{n+m-1+p+q} > \frac{m}{n+m} \\ &\Leftrightarrow 1 - \frac{n}{n+m-1+p+q} > 1 - \frac{n}{n+m} \\ &\Leftrightarrow n+m-1+p+q > n+m \Leftrightarrow p+q > 1 \end{aligned}$$

Then, $P^1_{ik} > P^0_{ik}$, because $p+q$ must be greater than 1, otherwise s_k cannot be reached from s_j. Besides, at the last step of the collapsing procedure (see

Figure 5.6(c)) the probability of reaching s_k from s_i is $P_{ik}^m = \frac{m \cdot p}{n+m \cdot q+m \cdot p} \cdot X$. But, $P_{ik}^m > P_{ik}^0$, in fact:

$$\begin{aligned} P_{ik}^m > P_{ik}^0 &\Leftrightarrow \frac{m \cdot p}{n+m \cdot q+m \cdot p} > \frac{m}{n+m} \cdot \frac{p}{p+q} \\ &\Leftrightarrow n+m \cdot q+m \cdot p < (n+m) \cdot (p+q) \\ &\Leftrightarrow n < n \cdot q + n \cdot p \Leftrightarrow p+q > 1 \end{aligned}$$

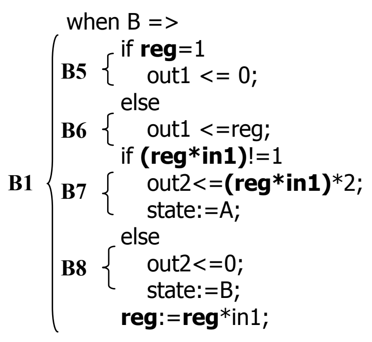
Then $P_{ik}^m > P_{ik}^0$. □

Theorem 2 *Under the same conditions of Theorem 1, the probability of traversing a transition t from s_i to s_x, on M_{l+1}^S is less than the same probability on M_l^S.*

The proof is similar to the proof of Theorem 1 and is omitted. From Theorem 1 and Theorem 2 derives the following property.

Corollary 1 *Changing from an LEFSM, M^L, to an SEFSM, M^S, by using the transition collapsing procedure, the probability of traversing transitions in M^S is more uniformly distributed than in M^L.*

5.4.4 Removal of Timing Discrepancies



```
when B =>
    if reg=1
        out1 <= 0;
    else
        out1 <=reg;
    if (reg*in1)!=1
        out2<=(reg*in1)*2;
        state:=A;
    else
        out2<=0;
        state:=B;
    reg:=reg*in1;
```

Fig. 5.8. Block B1 of Figure 5.1 has been modified to remove incompatibility between transitions of the LEFSM.

The transition collapsing procedure previously described may be not enough to make the SEFSM equivalent to the REFSM. This is due to the fact that some transitions are not compatible, thus they cannot be collapsed. According to Definition 8, a state s_j between two adjacent transitions, t_{ij} from s_i to s_j, and t_{jk} from s_j to s_k, cannot be removed when one of the following cases happens:

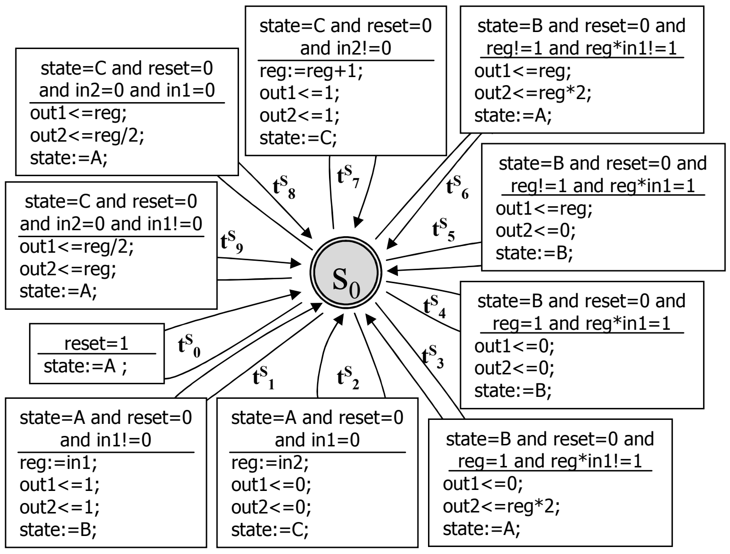

state=C and reset=0 and in2=0 and in1=0
out1<=reg; out2<=reg/2; state:=A;
state=C and reset=0 and in2!=0
reg:=reg+1; out1<=1; out2<=1; state:=C;
state=B and reset=0 and reg!=1 and reg*in1!=1
out1<=reg; out2<=reg*2; state:=A;
state=B and reset=0 and reg!=1 and reg*in1=1
out1<=reg; out2<=0; state:=B;
state=C and reset=0 and in2=0 and in1!=0
out1<=reg/2; out2<=reg; state:=A;
reset=1
state:=A ;
state=B and reset=0 and reg=1 and reg*in1=1
out1<=0; out2<=0; state:=B;
state=A and reset=0 and in1!=0
reg:=in1; out1<=1; out2<=1; state:=B;
state=A and reset=0 and in1=0
reg:=in2; out1<=0; out2<=0; state:=C;
state=B and reset=0 and reg=1 and reg*in1!=1
out1<=0; out2<=reg*2; state:=A;
S_0
t^S_0
t^S_1
t^S_2
t^S_3
t^S_4
t^S_5
t^S_6
t^S_7
t^S_8
t^S_9

Fig. 5.9. SEFSM for the code of Figure 5.1 after modification of B1.

1. the enabling functions of the transitions are conflicting;
2. the update function of t_{ij} updates the value of one or more registers involved in the enabling function of t_{jk}.

The first case happens when an update function of the REFSM contains conflicting conditional statements. If the conflicting conditions are nested, then the original DUV description contains a design error that must be removed before generating the LEFSM. Once the design error is removed, t_{ij} and t_{jk} becomes compatible and they can be composed removing the interleaved state. On the contrary, if the conditional statements are not nested, then the following observations can be applied.

By construction, if a state s of an LEFSM is the source of a transition with enabling function e, it is also the source of a transition with enabling function $\neg e$. Now, let t_{ij}, from s_i to s_j, and t_{jk}, from s_j to s_k, two conflicting transitions of an LEFSM generated by two not nested conditional statements of the same update function of an REFSM. Necessarily, there exist other two transitions t'_{ij}, from s_i to s_j, and t'_{jk}, from s_j to s_k, whose enabling functions are the negation of t_{ij} and t_{jk}. Thus, t_{ij} is compatible with t'_{jk}, and t'_{ij} is compatible with t_{jk}. In this case, the state s_j can be removed, t_{ij} is composed with t'_{jk}, and t'_{ij} is composed with t_{jk}. For example, let us consider transition t'_2 of the REFSM showed in Figure 5.3. Its update function contains the condition `reg=1` and the condition `reg!=1` which are conflicting, but not nested. When, the LEFSM is generated, such conditions become the enabling functions of transitions t^L_6, t^L_8, and their negation become the enabling functions of transitions t^L_5, t^L_7 (Figure 5.5). Thus, t^L_6 and t^L_5 could be composed to generate the SEFSM, first with t^L_4, and then, respectively, with t^L_7 and t^L_8 removing states s_2 and s_3.

Unfortunately, this composition cannot be completely performed because of the second case of incompatibility previously cited. In fact, the register `reg` is updated in t_5^L and t_6^L, and then it is used in the enabling functions of t_7^L and t_8^L. For this reason, the resulting SEFSM (Figure 5.7) has one more state than the REFSM. Thus, timing discrepancies cannot be completely removed.

The problem can be efficiently solved by modifying the block B1 of the original HDL description as showed in Figure 5.8. The update instruction for register `reg` has been moved from the beginning to the end of block `B1`, and each occurrence of `reg` included in B1 has been replaced by the expression `reg*in1`. In this way, the incompatibility between transition t_5^L, t_6^L and transitions t_7^L, t_8^L is removed, since `reg` is updated after the enabling functions of t_7^L and t_8^L are evaluated. The resulting SEFSM (Figure 5.9) is exactly equivalent to the REFSM, but it preserves the same facility to be traversed of the SEFSM of Figure 5.7.

From the previous considerations, the following rule of thumb can be extracted: *to generate an SEFSM exactly equivalent to the REFSM, the HDL descriptions of the DUV must be modified in such a way that, within a state, the update instruction of a register R must be put after each conditional statement that evaluates R.*

It is worth to note that we propose to modify the original DUV description, by exploiting the previous rule, only for testing purpose, not for synthesis or optimization aspects. The designer can implements the systems by following traditional design techniques. The original HDL description can be automatically converted into an exactly equivalent SEFSM that respects the previous rule. In this way, functional ATPG is performed on the SEFSM to identify design errors on the original description.

In this way, if a fault is detected on such an SEFSM, it is detectable also on the REFSM and the test sequence is the same.

5.5 Avoiding Register-dependent HTT transitions

The stabilization process proposed in [192] for stabilizing transition systems can be applied to avoid register-dependent HTT transitions. In fact, according to the terminology adopted in [192], a register-dependent HTT transition is similar to an unstable block transition.

Given an EFSM $E = \langle S, I, O, D, T \rangle$, a *block* is defined as a set of configurations of E that have the same (symbolic) state. A block transition graph (BTG) can be derived from E in the following way. The states of E represent the blocks, which become the nodes of the BTG. The transitions of E become the block transitions of the BTG. The labels of the block transitions are defined as follows. A transition t of an EFSM with enabling function $e(x,i)$ becomes a block transition t^B of the BTG with enabling function $e(x,i)|_i$, where $e(x,i)|_i$ represents the projection of $e(x,i)$ with respect to inputs only. Note that, this abstraction removes, from the enabling functions, all the expressions which involve comparisons on registers. Thus, the BTG is free from register-dependent HTT transitions by construction. According to this observation, the register-dependent HTT transitions of an SEFSM can be replaced with easy-to-traverse transitions by extracting the corresponding BTG. Unfortunately, abstracting away from registers can generate a non-deterministic

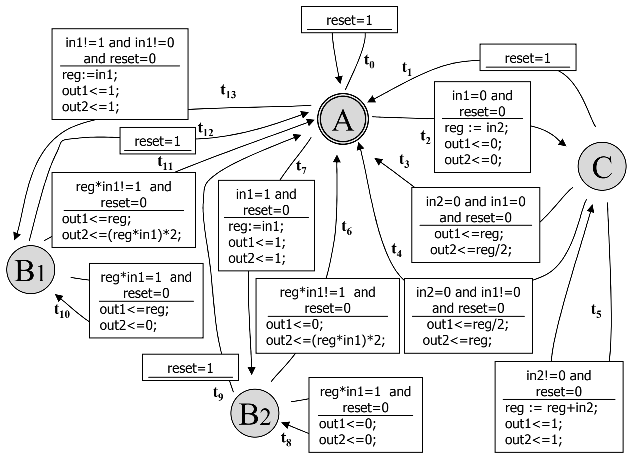


Fig. 5.10. S^2EFSM generated from the SEFSM of Figure 5.9.

BTG. In fact, a transition out-going from a block can have more than one destination block. Such a kind of block transitions is defined as *unstable*. The stabilization process has been proposed to remove the non-determinism of the BTG by splitting the blocks and the input symbols. The use of stabilization should be limited, since it can lead to the explosion of states [9]. Thus, generating the BTG, we applied the register abstraction only on register-dependent HTT transitions rather than on all the transitions involving registers.

Let us consider, for example, the STG of Figure 5.9. Clearly t_3^S is a register-dependent HTT transition, since the probability that `reg` assumes the value 1 is infinitesimal. The BTG corresponding to such an SEFSM consists of the same structure (1 block and 10 transitions), but the enabling functions of the block transitions, from t_0^S to t_9^S, become respectively:

```
reset=1;
reset=0 and in1!=0;
reset=0 and in1=0;
reset=0 and reg*in1)!=1;
reset=0 and (reg*in1)=1;
reset=0 and (reg*in1)=1;
reset=0 and (reg*in1)!=1;
reset=0 and in2!=0;
reset=0 and in2=0 and in1=0;
reset=0 and in2=0 and in1!=0
```

In particular, the enabling functions of block transitions t_3^S and t_6^S becomes undistinguishable (`reset=0 and reg*in1!=1`). By applying the stabilization process, we obtain the stabilized SEFSM of Figure 5.10 which present only ETT transitions.

Summarizing, a *Stabilized SEFSM* (S^2EFSM) does not contain input-dependent HTT transitions, nor register-dependent HTT transitions. Thus, moving from the

REFSM to the S^{2}EFSM definitely improves the capability of the reference ATPG to uniformly traverse the whole state space of the DUV.

5.6 Modeling system events

An *Extended Event FSM* (EEFSM) is an input/output finite state machine augmented by a set of *registers* that range over a finite alphabet and by a set of input/output *events* that trigger the transitions of the machine. Input and output events are used to model synchronization with a clock signal and the sensitivity list construct defined in hardware description languages. When a clock tick occurs or when an input signal in the sensitivity list changes its value, an event is generated and the EEFSM executes a transition. When changes occur only in input signals that do not belong to the sensitivity list, no events are generated and the state of the machine is not updated. This event-based semantics allows for a cleaner representation of the asynchronous composition of EEFSMs: when different components of the system react to different clocks, or when they are sensitive to different sets of signals, events are used to activate only the components that should trigger a transition, keeping silent the other ones.

The *register alphabet*, *the input alphabet*, and the *output alphabet* of an EEFSM are defined as tuples $R = R_1 \times \cdots \times R_n$, $I = I_1 \times \cdots \times I_m$, and $O = O_1 \times \cdots \times O_l$, where each R_j, I_j, and O_j is the finite alphabet of the j-th register, input signal and output signal, respectively. Given a generic alphabet $Q = Q_1 \times \cdots \times Q_n$, we define the set $Q^\perp = Q_1 \cup \{\perp\} \times \cdots \times Q_n \cup \{\perp\}$, where $\perp$ does not belong to any of the Q_i. The symbol $\perp$ will be used to represent the fact that a transition does not change the value of a register or of an output signal. Given a set of values $p \in Q$ and a set of values $t \in Q^\perp$, we define the operation of updating the values of p with respect to t as follows:

$$Update(p,t) = (q_1, \ldots, q_n),\ \ with\ q_i = \begin{cases} p_i \text{ if } t_i = \perp \\ t_i \text{ if } t_i \neq \perp \end{cases}$$

An *event alphabet* is a finite set E of *event symbols*. We extend every event alphabet E by adding a new symbol τ to model the fact that no events to which the EEFSM is sensitive occur. We define E^τ as the set $E \cup \{\tau\}$. EEFSMs are defined as follows.

Definition 9 *An Extended event finite state machine (EEFSM) is a tuple* $\mathcal{M} = \langle S, R, E, F, I, O, En, Upd, s_0, r_0 \rangle$ *where:*

- S *is a finite set of states;*
- $R = R_1 \times \cdots \times R_n$ *is a finite register alphabet;*
- E *is a finite input events alphabet;*
- F *is a finite output events alphabet;*
- $I = I_1 \times \cdots \times I_m$ *is a finite input alphabet;*
- $O = O_1 \times \cdots \times O_l$ *is a finite output alphabet;*
- En *is an* enabling function $S \times R \times E^\tau \times I \mapsto S$ *such that* $En(s_1, r, \tau, i) = s_1$ *for all* $r \in R$ *and* $i \in I$*;*
- Upd *is an* update function $S \times S \times R \times E^\tau \times I \mapsto R^\perp \times F^\tau \times O^\perp$ *such that* $Upd(s_1, s_2, r, \tau, i) = (\perp, \tau, \perp)$ *for all* $r \in R$ *and* $i \in I$*;*

```
M : process( i )
begin
  case state is
  when s0 =>
    if α ( i ) then
      A1(o); state := s1;
    else
      A2(o); state := s0;
    end if;
  when s1 =>
    if β ( i ) then
      B1(o); state := s0;
    else
      B2(o); state := s1;
    end if;
  end case;
end process;
```

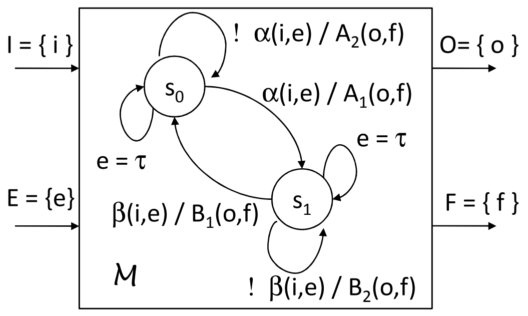


Fig. 5.11. An example of EEFSM and the corresponding HDL code.

- $(s_0, r_0) \in S \times R^{\perp}$ *is the reset configuration of* $\mathcal{M}$ [2].

Notice the event alphabets E and F are respectively companions of the alphabets I and O, in order to model asynchronous EEFSM composition (Section 5.7), where none, one or more EEFSMs may be triggered at a time by an event on a signal in the sensitivity list. It would not be sufficient to extend the alphabets I and O with the empty event τ to be I^{τ} and O^{τ}, because then one could not model the case of the same input feeding say two EEFSMs and being in the sensitivity list of only one of them. Indeed in that case the EEFSM with the alphabet in the sensitivity list should react to it with a regular transition, whereas the other EEFSMs should be inputted a τ signal and react with a silent transition (the state does not change and the output is empty).

Given a state s_1, registers value r, input event e and input symbol i, a transition from state s_1 to state s_2 occurs only if $En(s_1, r, e, i) = s_2$. When the transition is executed, the new values for the registers, the output event and the output symbol are given by $Upd(s_1, s_2, r, e, i)$. Note that, by the definition of EEFSM, when a τ input event is received, a τ output event is generated and the state, register values and output signals does not change. We usually represent EEFSMs as graphs where vertices are the states in S and where edges (s_1, s_2) are labeled with a pair $\gamma(r, e, i)/A(r, e, i)$ such that $En(s_1, r, e, i) = s_2$ if and only if $\gamma(r, e, i) = true$ and $Upd(s_1, s_2, r, e, i) = A(r, e, i)$. With a little abuse of notation, we call $\gamma(r, e, i)$ and $A(r, e, i)$ the *enabling function* and the *update function* of the transition (s_1, s_2). The semantics of an EEFSM is defined as follows.

Definition 10 *Given a word* $\alpha \in (E^{\tau} \times I)^*$ *, the corresponding trace (or computation) of an EEFSM* $\mathcal{M}$ *is a tuple* $(\sigma, \rho, \varepsilon, \beta) \in \left(S \times R \times F^{\tau} \times O^{\perp}\right)^*$ *such that:*

[2] Instead of introducing enabling and update functions, one could use a unique transition relation as in Sectior 5.2.

- $|\sigma| = |\rho| = |\varepsilon| = |\beta| = |\alpha|$
- *If* $En(s_0, r_0, \alpha(1)) = s'$ *and* $Upd(s_0, s', r_0, \alpha(1)) = (r', e', o')$*, then* $(\sigma(1), \rho(1), \varepsilon(1), \beta(1)) = (s', Update(r_0, r'), e', o')$*;*
- *For every* $2 \leq j \leq |\alpha|$*, let* $En(\sigma(j-1), \rho(j-1), \alpha(j)) = s'$ *and* $Upd(\sigma(j-1), s', \rho(j-1), \alpha(j)) = (r', e', o')$*, then* $(\sigma(j), \rho(j), \varepsilon(j), \beta(j)) = (s', Update(\rho(j), r'), e', Update(\beta(j), o'))$*;*

Given an EEFSM we define its *output function* $Out : S \times S \times R \times E \times I \mapsto F^{\tau} \times O^{\perp}$ as the restriction of *Upd* to $F^{\tau} \times O^{\perp}$. A particular case of EEFSM, is the Moore EEFSM, defined as follows.

Definition 11 *A Moore EEFSM is an EEFSM where the output function does not depend on input values,* $Out : S \times R \mapsto F^{\tau} \times O^{\perp}$*.*

A digital system can be represented as a collection of concurrent EEFSMs, one for each process. In this way, we capture the main characteristics of state-oriented, activity-oriented and structure-oriented models [191]. In fact, the EEFSM is composed of states and transitions, thus it is state-oriented, but each transition is extended with instructions that act on the registers. In this sense, each transition represents a set of activities on data, thus, the EEFSM is a structure-oriented model too. Finally, concurrency is intended as the possibility that each EEFSM changes its state concurrently with the other EEFSMs to reflect the concurrent execution of the corresponding processes. Data communication between concurrent EEFSMs is guaranteed by the presence of common signals. Figure 5.11 reports an example of EEFSM that can be extracted from the corresponding HDL code.

5.7 EFSM composition

The effectiveness and the efficiency of functional ATPGs based on deterministic strategies is influenced by the computational model adopted to represent the design under validation. In this context the EEFSM paradigm permits to model properly both synchronous and asynchronous system modules in a uniform way. The aim of composition is to improve functional ATPG whose effectiveness and efficiency may be limited when separate EFSMs are used to model the design under test. In this section the problem of composing two EEFSMs is depicted. Three types of composition has been identified: *Serial composition*, where outputs of the first EEFSM are inputs of the second one, *Parallel composition*, where the two EEFSMs may read the same inputs but compute in parallel different subsets of outputs, and *Feedback composition*, where the two EEFSMs are composed in a loop by feeding outputs of one as inputs to the other.

When composing two EEFSMs, can be assumed, without loss of generality, that they share the same registers. If it is not the case, we can always add to or both of them some "useless" registers that are never changed by the EEFSM and that do not affect the value of the output function nor of the transition function. Given a register alphabet $R = R_1 \times \cdots \times R_n$ and the corresponding extended alphabet $R^{\perp}$, we define the operation of merging two symbols $p, q \in R^{\perp}$ as follows:

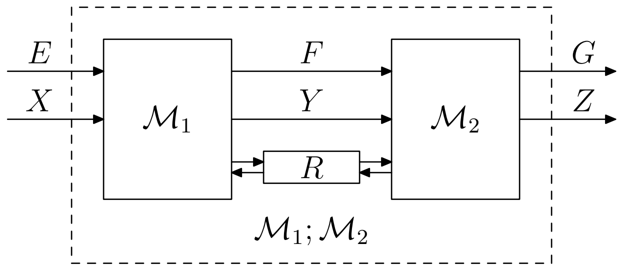


Fig. 5.12. Serial composition of EEFSMs.

$$Merge(p, q) = (r_1, \ldots, r_n), \ where \ r_i = \begin{cases} p_i \text{ if } q_i = \bot \\ q_i \text{ if } p_i = \bot \end{cases}$$

Note that $Merge(p, q)$ is not defined if p and q are such that there exist p_i and q_i with $p_i \neq \bot$ and $q_i \neq \bot$. This because we do not allow two EEFSMs to update simultaneously the value of the same register.

5.7.1 Serial Composition

The serial composition of two EEFSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ is denoted as $\mathcal{M}_1; \mathcal{M}_2$ and is portrayed in Figure 5.12.

Definition 12 *Given two EEFSMs* $\mathcal{M}_1 = \langle S_1, R, E, F, X, Y, En_1, Upd_1, s_0^1, r_0^1\rangle$ *and* $\mathcal{M}_2 = \langle S_2, R, F, G, Y, Z, En_2, Upd_2, s_0^2, r_0^2\rangle$, *their serial composition* $\mathcal{M}_1; \mathcal{M}_2 = \langle S_1 \times S_2, R, E, G, X, Z, En, Upd, (s_0^1, s_0^2), Merge(r_0^1, r_0^2)\rangle$ *is defined as follows:*

- $En((s_1, s_2), p, e, x) = (t_1, t_2)$ *where* $t_1 = En_1(s_1, p, e, x)$ *and* $t_2 = En_2(s_2, p, Out_1(s_1, t_1, p, e, x))$;
- $Upd((s_1, s_2), (t_1, t_2), p, e, x) = (q, g, z)$ *where* $(q_1, f, y) = Upd_1(s_1, t_1, p, e, x)$, $(q_2, g, z) = Upd_2(s_2, t_2, p, f, y)$, *and* $q = Merge(q_1, q_2)$.

Notice that the serial composition of two EEFSMs is not always defined, since we do not allow two EEFSMs to simultaneously change the value of the same register, and thus it is not guaranteed that $Merge(q_1, q_2)$ exists for any transition and any pair of EEFSMs.

5.7.2 Parallel Composition

The parallel composition of two EEFSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ is denoted as $\mathcal{M}_1 \parallel \mathcal{M}_2$ and is portrayed in Figure 5.13.

Definition 13 *Given* $\mathcal{M}_1 = \langle S_1, R, E, G, X, Y, En_1, Upd_1, s_0^1, r_0^1\rangle$ *and* $\mathcal{M}_2 = \langle S_2, R, F, H, X, Z, En_2, Upd_2, s_0^2, r_0^2\rangle$, *their parallel composition* $\mathcal{M}_1 \parallel \mathcal{M}_2 = \langle S_1 \times S_2, R, E \times F, G \times H, X, Y \times Z, En, Upd, (s_0^1, s_0^2), Merge(r_0^1, r_0^2)\rangle$ *is defined as follows:*

- $En((s_1, s_2), p, ef^\dagger, x) = (En_1(s_1, p, e, x), En_2(s_2, p, f, x))$;

† ef is an abbreviation for (e, f)

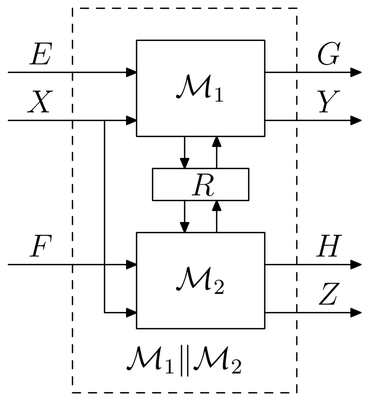


Fig. 5.13. Parallel composition of EEFSMs.

- *$Upd((s_1, s_2), (t_1, t_2), p, ef, x) = (q, gh, yz)$ where*
  $(q_1, g, y) = Upd_1(s_1, t_1, p, e, x)$,
  $(q_2, h, z) = Upd_2(s_2, t_2, p, f, x)$, and $q = Merge(q_1, q_2)$.

As in the case of the serial composition, the parallel composition of two EEFSMs is not always defined, since it is not guaranteed that $Merge(q_1, q_2)$ exists for any transition and for any pair of EEFSMs.

5.7.3 Feedback Composition

The feedback composition of two EEFSMs $\mathcal{M}_1$ and $\mathcal{M}_2$ is denoted as $\mathcal{M}_1 \times \mathcal{M}_2$ and is portrayed in Figure 5.14. Consider two EEFSMs $\mathcal{M}_1 = \langle S_1, R, G, H, U, V, En_1, Upd_1, s_0^1, r_0^1 \rangle$ and $\mathcal{M}_2 = \langle S_2, R, E \times H, G \times F, X \times V, U \times Z, En_2, Upd_2, s_0^2, r_0^2 \rangle$. To define their feedback composition $\mathcal{M}_1 \times \mathcal{M}_2 = \langle S_1 \times S_2, R, E, F, X, Z, En, Upd, (s_0^1, s_0^2), Merge(r_0^1, r_0^2) \rangle$ we have to solve two problems.

- As in the case of the serial and parallel composition, we have to guarantee that it is never the case that the value of a register is simultaneously changed by the two machines.
- Since the output of $\mathcal{M}_1$ is part of the input of $\mathcal{M}_2$, and vice versa, we have that the present values of h, v are a function of the present values of h, v:

$$\begin{aligned}(h, v) &= Out_1(s_1, t_1, r, g, u) \\ &= Out_1(s_1, t_1, r, Out_2^{g,u}(s_2, t_2, r, eh, xv));\end{aligned}$$

  and that the present values of g, u are a function of the present values of g, u:

$$\begin{aligned}(g, u) &= Out_2^{g,u}(s_2, t_2, r, eh, xv) \\ &= Out_2^{g,u}(s_2, t_2, r, e\, Out_1^{h}(s_1, t_1, r, g, u), \\ &\qquad x\, Out_1^{v}(s_1, t_1, r, g, u)).\end{aligned}$$

  Given s_1, s_2, r, e, and x, it is not guaranteed that there are h, u, g and v such that the above equations are satisfied.

The first problem can be solved as in the other cases of composition: given two update functions $Upd_1(s_1, t_1, r, g, u) = (r_1, h, v)$ and $Upd_2(s_2, t_2, r, eh, xv) = (r_2, gf, uz)$, we allow their simultaneous activation only if $Merge(r_1, r_2)$ is defined.

The second problem can be solved by three different strategies, with increasing level of generality and of algorithmic complexity.

- A first possibility is to restrict to the case when $\mathcal{M}_1$ or $\mathcal{M}_2$ is a Moore EEFSM. In this case one of the two output function does not depend on the inputs and thus the two above equations can be easily solved. For instance, if $\mathcal{M}_2$ is a Moore EEFSM, Out_2 does not depend on the inputs and thus $(h, v) = Out_1(s_1, t_1, r, Out_2^{g,u}(s_2, r))$.
- The second possibility is study the algebraic dependencies between the different input and output variables in U and V. Suppose that $U = U_1 \times \ldots \times U_n$ and $V = V_1 \times \ldots \times V_m$. The *dependency graph* of $\mathcal{M}_1 \times \mathcal{M}_2$ is a directed graph whose nodes are the U_i and the V_j and where we put an edge (U_i, V_j) if the value of V_j depends on the value of U_i in Upd_1 and an edge (V_j, U_i) if the value of U_i depends on the value of V_j in Upd_2. If this graph is acyclic then there are no algebraic loops in Upd_1 and Upd_2, and the composition is well defined.
- Finally, one can admit the possibility of loops in the dependency graph and still have a well-defined composition. It is known that a digital circuit can have loops or feedback paths and still be *combinational*, i.e., the values of the outputs depend on the current inputs only [194, 195]. When a circuit is cyclic, it is necessary to analyze it to determine whether it behaves combinationally or not. This analysis problem is usually solved by encoding the circuit in ternary-valued logic: zeros, ones and "undefined" values. Then the circuit is analyzed to determine whether it produces definite output values for every definite assignment of the input values. If this is the case then the circuit is combinational and loops behave correctly. Recently, BDDs and SAT-based techniques have been proposed to efficiently analyze cyclic circuits [196, 197]. The same approach can be applied to the feedback composition of EEFSMs to determine whether *En* and *Upd* are combinational even when there are loops in the dependency graph.

5.8 The HLDD model

HLDDs are graph representations of discrete functions that can be considered as a generalization of BDDs. Unlike in BDDs, where nodes are labeled by Boolean variables and edges hold only Boolean values, in HLDDs, any scalar variable values (e.g. integer, floating point, enumeration type, etc.) are allowed and edges are labeled by partitions of the domains of respective variables. HLDDs have proven an efficient model for simulation and fault modeling as they provide for fast evaluation by graph traversal and for easy identification of cause-effect relationships. In the following we present the definition of HLDDs and explain HLDD based modeling on a simple example.

Definition 14 *A HLDD representing a discrete function $y = f(x)$ is a directed non-cyclic labeled graph that can be defined as a quadruple $G = \langle M, E, X, D \rangle$,*

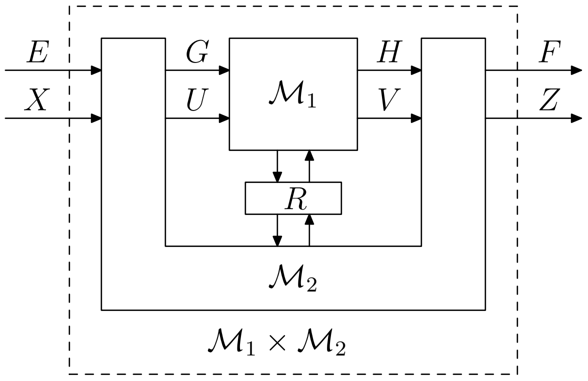


Fig. 5.14. Feedback composition of EEFSMs.

```
process (clock, reset)
  variable reg: integer range 32767 downto −32768;
  variable state: integer range 1 downto 0;
begin
  if reset = '1' then
    state := A;
    out1 <= 0;
    out2 <= 0;
  elsif clock'event and clock = '1' then
    case state is
    when A =>
      if in1 = 0 then
        out1 <= 0;
        out2 <= 0;
      else
        reg := in1;
        out1 <= 1;
        out2 <= 1;
      end if;
    when B =>
      if reg = 1 then
        out1 <= in1*2;
        out2 <= in1;
      else
        out1 <= reg;
        out2 <= reg*2;
      end if;
    end case;
  end if;
end process;
```

Fig. 5.15. A simple design suitable for HLDD generation.

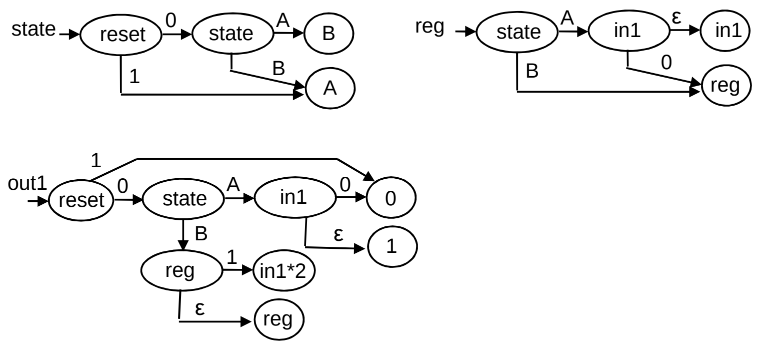


Fig. 5.16. HLDD models for variables *state*, *reg* and *out1* of example in Figure 5.15.

where M is a finite set of vertices (referred to as nodes), E is a finite set of edges, X is a function which defines the variables labeling the nodes and the variable domains, and D is a function on E. The function $X(m_i)$ returns a pair (x_i, X_i), where x_i is the variable letter, which is labeling node m_i and X_i is the domain of x_i. Each node of a HLDD is labeled by a variable. In special cases, nodes can be labelled by constants or algebraic expressions. An edge $e \in E$ of a HLDD is an ordered pair $e = (m_1, m_2) \in E^2$, where E^2 is the set of all the possible ordered pairs in set E. D is a function on E representing the activating conditions of the edges for the simulating procedures. The value of $D(e)$ is a subset of X_i, where $e = (m_i, m_j)$ and $X(m_i) = (x_i, X_i)$. It is required that $P_{m_i} = \{D(e)|e = (m_i, m_j) \in E\}$ is a partition of the set X_i. HLDD has only one starting node (root node), for which there are no preceding nodes. The nodes, for which successor nodes are missing are referred to as terminal nodes.

In HLDD models representing digital systems, the non-terminal nodes correspond to conditions or to control signals, and the terminal nodes represent operations (functional units). Register transfers and constant assignments are treated as special cases of operations. Simulation on HLDDs takes place as follows. Consider a situation, where all the node variables are fixed to some value. According to these values, for each non-terminal node a certain output edge will be chosen which enters into its corresponding successor node. Let us call such connections *activated edges* under the given values. Succeeding each other, activated edges form in turn *activated paths*. For each combination of values of all the node variables there exists always a corresponding activated path from the root node to some terminal node. We refer to this path as the *main activated path*. The simulated value of variable represented by the HLDD will be the value of the variable labeling the terminal node of the main activated path.

When representing systems by decision diagram models, in general case, a system of HLDDs rather than a single HLDD is required. During the simulation of HLDD systems, the values of some variables labeling the nodes of an HLDD are calculated by other HLDDs of the system.

Figure 5.16 presents an example of an HLDD for three variables, `state`, `reg` and `out1` corresponding to DUV shown in Figure 5.15.

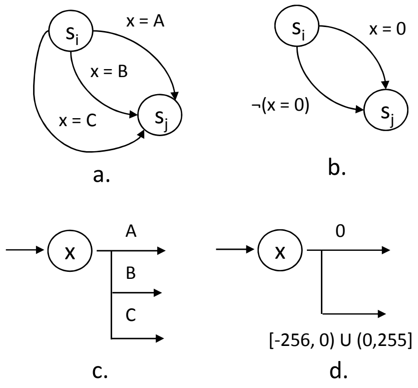


Fig. 5.17. EFSM states and transitions toward HLDD nodes and edges.

5.9 From EFSMs to HLDDs

At RT-level, DUV can be represented by a system of HLDDs, one for each variable occurring in the DUV. Therefore the HLDD model is a variable-based representation of the EFSM model described in Section 5.2. For example, Figure 5.16 shows the HLDDs of variables *state*, *reg* and *out1* occurring in the DUV of Figure 5.15. In this section, a procedure is proposed to automatically create HLDDs by traversing the EFSM model.

Given a variable occurring in the EFSM, the nodes and edges of the corresponding HLDD are defined by analyzing the enabling functions of the EFSM. Each enabling function may involve one or more atomic propositions (e.g., $x_1 + x_2 > 0$), referred as $A(x_1, x_2, \ldots, x_n)$ where x_i is a DUV variable, and boolean operators (e.g., $\wedge, \vee$). Without loss of generality, we consider, in the following, that atomic propositions involve only one variable.

Firs of all, let us consider the basic case where the enabling function of an EFSM transition involves only one atomic proposition. If the atomic proposition depends on the value of a variable x, whose type is an enumerative type, an HLDD node, labeled with x, is generated with as many out-going transitions as the number of values in the enumerative type domain of x (Figure 5.17.a, 5.17.c). Otherwise, if the atomic proposition $A(x)$ represents a generic condition, we generate an HLDD node, labeled with x, with two out-going transitions, labeled, respectively, with values of x domain which satisfy $A(x)$ and $\neg A(x)$ (Figure 5.17.b, 5.17.d).

Now, let us consider enabling functions consisting of two or more atomic proposition, for example $A(x)$ and $B(y)$. The corresponding portion of HLDD is shown in Figure 5.18.a and 5.18.b. Let $\mathcal{X}$ and $\mathcal{Y}$ respectively be the domains of variables x and y. Let also $\mathcal{A}$ and $\mathcal{A}^*$ be defined as $\mathcal{A} = \{v | v \in \mathcal{X} \wedge A(v) = \mathit{true}\}$ and $\mathcal{A}^* = \{v | v \in \mathcal{X} \wedge \neg A(v) = \mathit{true}\} = \mathcal{X} \setminus \mathcal{A}$. $\mathcal{B}$ and $\mathcal{B}^*$ can be defined likewise.

Given the HLDD portion corresponding to $A(x)$, the generation of the HLDD part corresponding to $\neg A(x)$, is straightforward, and it is shown in Figure 5.18.c.

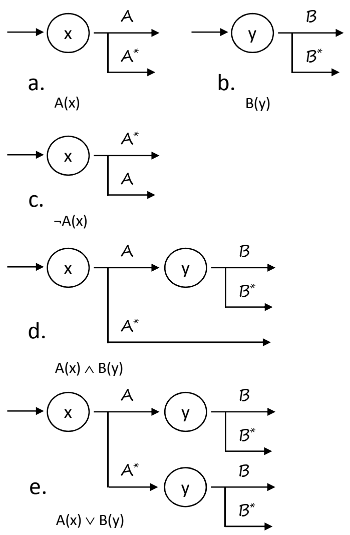


Fig. 5.18. Example of HLDDs derived from enabling functions involving more than one atomic proposition.

On the contrary, the portions of HLDD, depending on the conjunction and disjunction of two atomic propositions, i.e., $A(x) \wedge B(x)$ and $A(x) \vee B(x)$, are shown, respectively, in Figure 5.18.d and in Figure 5.18.e.

During the generation of the HLDD, a terminal node is produced when, traversing a path in the EFSM, we come back to the initial state. The terminal node is labeled with the value (constant, variable or expression) assigned to the target variable by traversing the corresponding path in the EFSM. If no value is assigned, the variable preserve the original value, that is, the node is labeled with the variable itself.

The complete HLDD representation of the DUV is obtained by iteratively applying the previous procedure for each variable of the DUV. The complexity of the HLDD generation is linear with respect to the number of transitions in the corresponding EFSM.

5.10 Experimental Results

Twofold experimental results have been performed to show the effectiveness of the proposed work. First the theoretical results related to the probability of traversing transitions on the different EFSMs are experimentally confirmed, as showed in Section 5.10.1. This analysis shows that moving from the REFSM to the S^{2}EFSM the probability of traversing the transitions becomes more uniformly distributed.

Secondly, functional validation framework based on the S^{2}EFSMs is applied to analyze their capability of detecting functional faults. These experimental results are presented in Section 5.10.2.

5.10.1 EFSM Traversing

The experimental confirmation of the theoretical analysis has been accomplished on the example reported in Figure 5.1. The conditional blocks of the benchmark are grouped according to their nesting levels. After the reference ATPG run for 100,000 clock cycles, the probability of activating the conditional blocks by using, respectively, the REFSM, the LEFSM, the SEFSM and the S^{2}EFSM is computed.

	REFSM		**LEFSM**		**SEFSM**		**S^{2}EFSM**	
Blocks	**Act**	**P**	**Act**	**P**	**Act**	**P**	**Act**	**P**
B0	50000	0.50	49976	0.55	44450	0.50	47094	0.47
B1	50000	0.50	16356	0.18	22216	0.25	31821	0.32
B2	0	0.00	24534	0.27	22234	0.25	20678	0.21
B3	50000	1.00	24930	0.50	22216	0.50	31638	0.67
B4	0	0.00	25046	0.50	22234	0.50	15456	0.33
B5	0	0.00	0	0.00	0	0.00	16121	0.50
B6	50000	1.00	16356	1.00	22216	1.00	15700	0.50
B7	50000	1.00	16356	1.00	22216	1.00	15934	0.50
B8	0	0.00	0	0.00	0	0.00	15887	0.50
B9	0	0.00	12188	0.50	7360	0.33	6780	0.33
B10	0	0.00	12260	0.50	14874	0.67	13898	0.67
B11	0	0.00	6076	0.50	7391	0.50	6925	0.50
B12	0	0.00	6184	0.50	7484	0.50	6973	0.50

Table 5.1. Probability of reaching conditional blocks.

SEFSM trans.	**Blocks**	**REFSM**	**LEFSM**	**SEFSM**	**S^{2}EFSM**
t0	**reset**	-	-	-	-
t1	**B0-B3**	0.5000	0.2750	0.2500	0.3149
t2	**B0-B4**	0.0000	0.2750	0.2500	0.1551
t3	**B1-B6-B7**	0.5000	0.1800	0.2500	0.0800
t4	**B1-B5-B8**	0.0000	0.0000	0.0000	0.0800
t5	**B1-B6-B8**	0.0000	0.0000	0.0000	0.0800
t6	**B1-B5-B7**	0.0000	0.0000	0.0000	0.0800
t7	**B2-B9**	0.0000	0.1350	0.0833	0.0700
t8	**B2-B10-B11**	0.0000	0.0675	0.0833	0.0700
t9	**B2-B10-B12**	0.0000	0.0675	0.0833	0.0700
Variance	-	0.0432	0.0111	0.0108	0.0058

Table 5.2. Probability of firing transitions.

Consider Table 5.1. Columns *Act* report the number of activations, and Columns *P* report the corresponding probabilities. It can be observed that on the REFSM the ATPG activates only blocks B0, B1, B3, B6, B7, because it cannot exploit information hidden in the update functions of the transitions, and moreover, t'_1 is a HTT transition. Moving from the REFSM to the LEFSM, and finally to the

SEFSM and S^{2}EFSM the capability of traversing the whole state space is greatly increased. However, block B5 and B8 are not activated on the LEFSM, nor on the SEFSM. This is due to the presence of register-dependent HTT transitions (t_6^L and t_7^L on the LEFSM, and t_3^S and t_4^S on the SEFSM). They are HTT because their enabling functions include the condition `reg=1`, which has an infinitesimal probability of being satisfied without exploiting learning or backtracking techniques. On the contrary, after the stabilization, register-dependent HTT transitions disappeared and all the conditional blocks can be easily traversed on the S^{2}EFSM.

A more detailed analysis can be performed by considering the probability of activating transitions on the EFSM, rather than blocks on the HDL code (Table 5.2). To compare the ATPG effectiveness on REFSM, LEFSM, SEFSM and S^{2}EFSM, the transitions of the SEFSM are grouped with respect to its unique state. Then, each transition t is mapped with the set of conditional blocks of the HDL code activated by traversing t. Thus, the probability of traversing one transition is represented by the product of the probabilities of activating the corresponding conditional blocks. This computation is performed by exploiting the probabilities of activating the conditional blocks collected for the REFSM, the LEFSM, the SEFSM and the S^{2}EFSM reported in Table 5.1. Finally, the variance of the probabilities of traversing the transitions is computed. The variance measures the distance from the mean value. More the variance is closer to 0, more the probabilities are uniformly distributed. It can be observed that the variance decreases moving from the REFSM to the S^{2}EFSM.

5.10.2 Fault Coverage

Name	PIs	POs	FFs	Gates	S^{2}EFSM Time (s.)
ex 16 bit	34	32	51	5358	0.014
vr1 16 bit	34	16	66	3069	0.013
vr2 16 bit	34	32	51	904	0.025
b04	13	8	66	650	0.029
b10	13	6	17	264	0.035
b11m	9	6	31	715	0.025
face_rec	34	32	100	1475	0.016

Table 5.3. Characteristics of benchmarks.

The efficiency of the proposed EFSM transformations, applied to automatic test pattern generation, has been evaluated by using the benchmarks described in Table 5.3. Columns report the number of primary inputs (*PIs*), primary outputs (*POs*), flip-flops (*FFs*) and gates (*Gates*). Finally, the last column reports the time required to automatically generate the S^{2}EFSM. Such benchmarks have been selected because they present different characteristics which allow us to analyze and confirm the effectiveness of the ATPG framework. *ex* is the example reported in Figure 5.1. Two versions of such an example are implemented, one with 16 bit-sized inputs and the other with 32 bit-sized inputs. The original HDL descriptions of *b04*, *b09* and *b11m* contain a large number of nested conditions on signals and registers of different size. *b04* and *b09* have been selected from the well known ITC-99 benchmarks suite [198], while *b11m* is a modified version of *b11* created by introducing a delay on some paths to make it harder to be traversed. Finally, *vr1*, *vr2* and *face_rec* contain conditional statements where one branch has probability $1 - \frac{1}{2^{32}}$ of being satisfied, while the other has probability $\frac{1}{2^{32}}$. Thus, they are very hard to be tested by a random ATPG. *vr1* and *vr2* have been selected from a set of internal benchmarks, while `face_rec` is a real industrial case. It is a module of a face recognition system, whose description is composed of two interacting processes that have been composed into a single EFSM.

		GA-ATPG			**PD-ATPG**		
Name	**B.C.**	**FC%**	**TV#**	**T (s.)**	**FC%**	**TV#**	**T (s.)**
ex 16 bit	867	13.8	2	129	44.3	23	92
ex 32 bit	1439	13.7	2	180	44.3	33	220
vr1 16 bit	461	49.7	11	35	50.1	16	33
vr1 32 bit	907	78.2	38	92	80.3	58	80
vr2 16 bit	601	3.83	2	202	42.4	19	117
vr2 32 bit	1182	1.1	4	421	41.7	17	177
b04	408	94.9	119	40	99.0	255	15
b10	216	87.0	175	47	94.1	154	43
b11m	725	37.0	149	60	39.2	120	57
face_rec	1041	0.9	35	1175	67.1	202	379

Table 5.4. Functional ATPG results.

The functional validation framework is independent from the adopted fault model. To evaluate its efficiency, a high-level fault model has been selected: the bit coverage fault model that will be deeply described in Section 7.1. According to the selected fault model, this study compared the performance of a genetic-based ATPG (*GA-ATPG*) [199] with the performance of the proposed pseudo-deterministic ATPG (*PD-ATPG*) based on the S^{2}EFSM. Table 5.4 shows, respectively, the number of bit coverage faults (*B.C.*), the fault coverage (*FC%*), the size of the test set (*TV#*), and the test generation time (*T*).

The very low fault coverage achieved by the GA-ATPG for *vr2* and *face_rec* is due to the presence of a transition out-going from the initial state, whose enabling function has an infinitesimal probability of being traversed by randomly fixing the values of PIs. On the contrary, the pseudo-deterministic ATPG efficiently exploits the constraint solver to generate the opportune PIs values by exploring the S^{2}EFSM model. This sensibly increases the achieved fault coverage for all benchmarks. Finally, note that, less accurate coverage metrics (e.g., line coverage) allow the proposed ATPG to achieve 100% of coverage, for every benchmark in Table 5.3, by exploiting S^{2}EFSMs.

5.10.3 EFSM and HLDD generation

Table 5.5 reports benchmark characteristic and generation times of EFMSs and HLDDs. For each benchmark an EFSM has been automatically generated as described in Section 5.4 and in Section 5.5, and for each variable of the EFSM the corresponding HLDD model has been automatically generated as described in Section 5.9. Columns of Table 5.5 report, for each benchmark, the number of code lines (*Lines*), the number of states (*States*) and transitions (*Trans.*) of the generated EFSM, the number of HLDDs derived from the EFSM (*HLDDs*), the number of nodes (*Nodes*) and edges (*Edges*) of the largest HLDD, and the time required for automatically generating the EFMS (GT_{EFSM}) and the corresponding HLDDs (GT_{HLDD}). As shown, generation times are almost negligible also for large benchmarks, like *dlx*.

		EFSM			**HLDD**		**Time (sec.)**	
DUV	**Lines**	**States**	**Trans.**	**HLDDs**	**Nodes**	**Edges**	**GT$_{EFSM}$**	**GT$_{HLDD}$**
ex1	56	6	11	5	14	13	0.028	0.024
b00	71	5	12	4	15	14	0.028	0.016
b01	107	17	34	3	36	35	0.036	0.020
b02	69	11	22	2	24	23	0.032	0.024
b04	96	20	38	12	40	39	0.044	0.096
b09	100	5	13	5	16	15	0.040	0.012
b10	190	32	64	11	66	65	0.053	0.124
b11m	110	12	32	5	35	34	0.044	0.024
b00z	74	7	16	4	19	18	0.036	0.020
fr	201	6	20	9	13	12	0.040	0.022
dlx	714	1057	1642	32	1646	1645	2.604	1.024

Table 5.5. Benchmark characteristics and EFSM/HLDD generation times.

5.11 Published contributions

This work has lead to the following publications [200, 201, 202]:

Giuseppe Di Guglielmo, Franco Fummi, Cristina Marconcini, and Graziano Pravadelli
EFSM Manipulation to Increase High-Level ATPG
In the Proceeding of "IEEE International Symposium on Quality Electronic Design (ISQED'06)"
San Jose, CA, USA, March 27-29, 2006, pp. 62-67

Anton Chepurov, Giuseppe Di Guglielmo, Franco Fummi, Graziano Pravadelli, Jain Raik, Raimund Ubar, Taavi Viilukas
Automatic generation of EFSMs and HLDDs for functional ATPG
In the Proceedings of "IEEE International Biennal Baltic Electronics Conference (BEC'08)"
Tallinn, Estonia, October 6-8, 2008, pp. 143-146

D. Bresolin, G. Di Guglielmo, F. Fummi, G. Pravadelli and T. Villa
The impact of EFSM Composition on Functional ATPG

In the Proceedings of "12th IEEE Symposium on Design and Diagnostics of Electronic Systems (DDECS'09)"
Liberec, Czech Republic, April 15-17, 2009

These papers presented a classification of hard-to-traverse transitions for the EFSM model. Then, such transitions can be replaced to obtain a new EFSM (S^2EFSM), where the probability of activating transitions is more uniformly distributed. The S^2EFSM is obtained by stabilizing a particular kind of EFSM (SEFSM) which is generated by linear-complexity manipulations. The potential state explosion induced by the stabilization is limited, since it is applied to a reduced number of HTT transitions.

This particular kind of EFSM which has been theoretically showed to allow a more uniform traversing of the DUV state space. Determinism is obtained by interfacing the proposed ATPG with a tool that adopts formal methods to solve the conditions of the enabling functions. In particular, the ATPG has been interfaced with both a CLP-based constraint solver and a SAT-solver. Experimental results showed that the effectiveness of the proposed ATPG compared with a genetic-based ATPG is evident. It greatly benefits from the fact that, by using the S^2EFSM model, all conditional statements included in the DUV are under its control.

An alternative paradigm, HLDD, is proposed to combine the beneficial properties of both HLDD-based and EFSM-based ATPG. The HLDDs are generated automatically starting from EFSMs by using the proposed technique.

6

Methodology: Automatic Test Pattern Generation

Random-based automatic test pattern generators (ATPGs) [203, 204] quickly achieve a quite high fault coverage at every abstraction level. However, the always rising complexity of modern circuits increases the number of corner cases where random-resistant faults reside. Thus, more evolved deterministic approaches, possibly based on learning [205] or backtracking [206], are needed to address such faults. These techniques exploit information on the structure of the DUV to traverse its state space. Many efficient deterministic ATPGs have been proposed at gate-level [207, 208, 209], where the system is described as an interconnection of primitive components. On the contrary, at functional level, the system is described in an algorithmic way, thus extracting structure information is more difficult [199]. For this reason, mathematical models, as the Finite State Machines (FSM) or the Extended Finite State Machines (EFSM) [9], can be adopted to apply deterministic techniques during functional testing.

In particular, the EFSM paradigm is preferred, since it allows a more compact representation of the state space with respect to the FSM one. Moreover, EFSMs are a very attracting formalism to be applied to automatic test pattern generation. A DUV description, based on states and transitions, can be easily adopted to fault modeling and state-space traversing. This approach can constitute the basis of any deterministic ATPG. A high-level fault model can be adopted to perturb the EFSM representation. Then, fault detection can be performed by traversing the EFSMs to activate the faults and propagate them to primary outputs. In this context, the EFSM structure permits to exploit deterministic algorithms to traverse the DUV, rather than random-based ones.

However, traversing an EFSM can be more difficult than traversing an FSM limiting the use of EFSMs in the ATPG context. In fact, moving from a state of an EFSM to another one depends on both primary inputs and internal registers. Thus, the presence of conditions involving registers on the guard of transitions imposes that already existent approaches, developed for traversing FSMs, cannot be easily adapted to EFSMs. Such a kind of EFSM is referred to as an inconsistent EFSM [9]. When the EFSM model is adopted to represent the DUV, its traversing is a critical operation to efficiently perform testing. In particular, traversing an EFSM is fundamental for functional ATPG, which requires to activate and propagate faults through the states of the high-level description [26].

Some papers, in the literature, propose EFSM-based techniques for high-level test generation [210,211,9]. However test generation techniques adopted for FSMs cannot be efficiently reused for EFSMs.

In [210, 211] different strategies are proposed to remove inconsistencies for reusing FSM-targeted ATPGs. However, the removal of inconsistencies can lead to the state space explosion if the HDL description contains a large number of conditions. A different approach is proposed in [9], where the authors present an orthogonalization process to extract an EFSM model from an HDL description. Then, a stabilization process is presented to improve the traversing of the EFSM. Finally, a breadth first search is used to generate a set of test patterns which cover all the transitions on the stabilized EFSM. The main limitations of this approach are represented by the complexity of the orthogonalization and stabilization process, which may lead to state explosion, and by the bread-first search, which is not targeted to fault detection. All previous approaches work on a single EFSM, but this requires to merge multi-process designs into a unique process. This solution is tedious and error-prone.

In this context, the contribution of proposing a functional ATPG framework can be summarized as follow:

- exploit an easy-to-traverse EFSM model to guide a constraint solver during the DUV traversal to explore the DUV state space;
- introduce backjumping techniques to move through hard-to-traverse transitions avoiding the state explosion problem;
- generate functional test for multiprocess DUVs;
- define a EFSM-composition theory to improve EFSM-based functional ATPG whose effectiveness and efficiency may be limited when separate EFSM are used to model the DUV;
- implement combined approaches that mix different state space exploration techniques and different computational models to address different DUVs and different areas of the same DUV.

This Chapter is organized as follow. Section 6.1 describes the defined ATPG framework. In particular, Section 6.1.2 presents the implemented ATPG architecture and a pseudo-deterministic EFSM-based engine, namely *Random Walk*. Section 6.1.3 describes the scheduling algorithm defined to works on a set of concurrent EFSMs that models the DUV.

Section 6.2 describes the EFSM-based engine that relies on *Learning*, *Backjumping* and constraint logic programming to deterministically generate test vectors for traversing all transition of the EFSM and activate the fault injected into the DUV.

Section 6.3 compares the effectiveness of the proposed functional deterministic ATPG, according to whether it exploits scheduling of concurrent EFSMs or EFSM composition (presented in Section 5.7).

Finally, Section 6.4 describes an alternative ATPG engine which exploits both EFSM and HLDD representations. The goal is to combine the beneficial properties of both paradigms using EFSMs for targeting control FSM transition and variable-oriented HLDDs for targeting faults in the data variables, respectively.

6.1 ATPG architecture

This Section presents the functional ATPG framework which exploits the EFSM model to deterministically generate test sequences. A constraint solver is used to generate test vectors that allow traversing the state space of the design under test (DUV) deterministically and uniformly. This approach definitely increases the ability of the ATPG to observe and control hard-to-detect faults.

The Section is organized as follows. Section 6.1.1 presents problem and the state of the art. Section 6.1.2 describes the defined ATPG architecture. Section 6.1.3 describes the scheduling algorithm defined to explore DUVs composed by concurrent EFSMs. The achieved experimental results are reported in Section 6.1.4.

6.1.1 Introduction

In this work the main characteristics of the S^2EFSM are exploited to define a functional ATPG framework. The transitions of the S^2EFSM present the most uniformly distributed probability of being activated when such a pseudo-deterministic ATPG is used, as discussed in Chapter 5 .

The framework is composed of two main modules. The first parses the DUV description, generates the S^2EFSM and performs fault injection. The second is the ATPG engine, which pseudo-deterministically navigates the S^2EFSM to generate test sequences.

The determinism is obtained by interfacing with an external solver. Given a transition of the S^2EFSM, the solver is required to generate opportune values for PIs that enable the S^2EFSM to move across such a transition. The ATPG can be configured to interface with different kinds of solvers (SAT-solver, CLP solver, etc.). Moreover, it can exploit also model checking tools. Two different types of solvers have been interfaced with the ATPG and then compared, as showed by the experimental results in Section 6.1.4.

The main characteristics of the proposed framework are the following.

- The framework accepts VHDL, Verilog and SystemC behavioral descriptions, on which a set of concurrent EFSMs is automatically identified, one for each process of the DUV. Intuitively, an EFSM is a finite state machine that implicitly memorizes the values of the DUV registers into transitions. Thus, the use of the EFSM model allows a compact representation of the DUV state space that reduces the risk of state explosion typical of more traditional FSMs [192]:
- each EFSM is automatically manipulated to obtain a new EFSM which is more uniformly traversable by exploiting a deterministic navigation strategy. This allows reducing the use of time-consuming backtracking techniques;
- the deterministic navigation relies on constraint solving. The EFSM model can be effectively integrated to CLP solver or SAT solver, since the ATPG to invokes the constraint solver only when moving between EFSM states;
- the navigation of concurrent EFSMs is guaranteed by an opportune EFSM scheduling algorithm that aims at maximizing the ATPG capability of exploring the whole state space. In this way multiprocess DUVs can be uniformly traversed.

The S^2EFSM is referred simply as EFSM in the following.

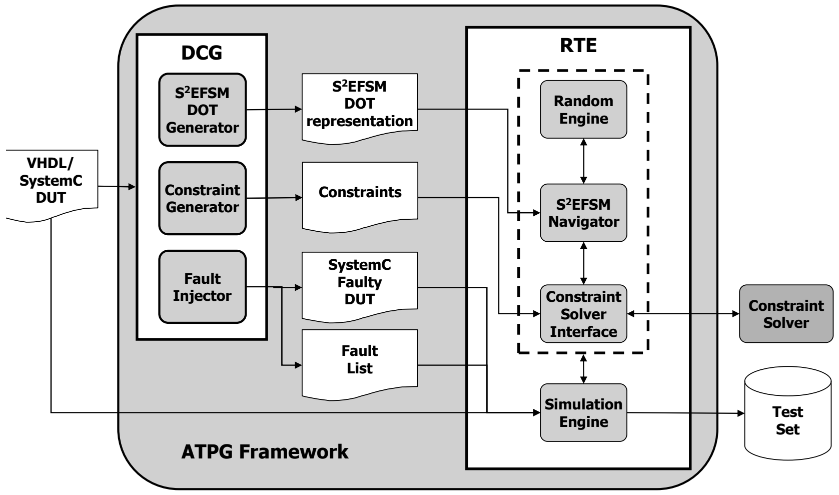


Fig. 6.1. The ATPG framework.

6.1.2 Functional ATPG Framework

The EFSM model is specially suited to be used with ATPGs that generate test sequences by deterministically activating the enabling functions of the transitions. According to this observation, this work proposes a functional ATPG framework depicted in Figure 6.1. It has been implemented in C++/SystemC, however, it accepts VHDL, Verilog and SystemC descriptions of the DUV. The framework is composed of two main modules:

- **DUV-dependent component generator (DCG)**. Given an HDL functional description of the DUV, this module generates the *State Transition Graph* (STG) representations of the corresponding EFSMs (*EFSM STG Generator*), the SystemC faulty description of the DUV and the related fault list (*Fault Injector*), and the file containing a set of constraints involved in the EFSM enabling and updating functions (*Constraint Generator*). Moreover, if the original DUV is coded in VHDL or Verilog, a corresponding SystemC version is provided, since the simulation engine is based on SystemC simulation kernel.
- **Run-time engine (RTE)**. This module is composed of the *EFSM navigator*, the *Constraint-Solver Interface*, the *Random Engine*, and the *Simulation Engine*. The RTE navigates the STG representation of the EFSM to generate test sequences. An external constraint solver is used to generate values for PIs of the DUV which allow to fire the enabling functions of transitions that the *EFSM navigator* wants to traverse. Values for PIs, not involved in the enabling function, are provided by the *Random Engine*. Then, the generated test sequences are provided to the *Simulation Engine* which compares the behavior of the fault-free and faulty DUVs. Test sequences that highlight discrepancies between the POs of the fault-free and faulty DUVs constitute the final test set.

Next paragraphs describe each module in details.

DUV-dependent component generator (DCG)

The DCG relies upon an *HDL Intermediate Format* (HIF) which allows to generate a hierarchical syntactic tree of the DUV description. Each node of the tree is an instance of an HDL construct. For example, the root represents the whole design and it has two offsprings: the entity and the architecture; the entity offsprings are PIs and POs, and so on for the other constructs. The leaves represent the occurrences of basic constructs like constants, variables, data types, etc. A powerful API allows user to navigate the syntactic tree to extract information or to add new nodes, thus modifying the DUV. The *HIF Suite* is described more in depth in Section 4. The DCG analyzes the DUV description and generates such a syntactic tree which is used by its submodules: the *EFSM STG Generator*, the *Fault Injector* and the *Constraint Generator*.

EFSM STG Generator

This module is responsible to automatically generate a DOT representation of the STG of the EFSM. Such a DOT description is used by the *EFSM navigator* during the test generation phase.

DOT is part of *GraphViz*, an *OpenSource* project developed by the *AT&T* laboratories [212]. DOT is a language for a collection of scripting tools for graphs vectorial representation. The *Boost Graph Library* (BGL) of C++ supplies all the functions to write and read files in DOT format [213]. Thus, once an EFSM is generated it is possible to save the EFSM structure on a DOT file, see the example in Figure 6.2

```
digraph b00 {
        0[label="state0"];
        1[label="state3"];
        0->0 [label="transition_#1", reset=true];
        0->0 [label="transition_#1", reset=false];
        0->0 [label="transition_#2", reset=false];
        0->1 [label="transition_#3", reset=false];
        0->1 [label="transition_#4", reset=false];
        1->0 [label="transition_#5", reset=false];
        1->0 [label="transition_#6", reset=false];
        0->0 [label="transition_#7", reset=false];
        0->0 [label="transition_#8", reset=false];
        0->0 [label="transition_#9", reset=false];
}
```

Fig. 6.2. *DOT* Description of the design in Figure 5.7.

Fault Injector

The *Fault Injector* is responsible to generate a faulty description of the DUV. It performs fault injection by inserting *saboteurs* into the DUV description. As, described in Section 2.2.5, generally, a saboteur is a special component added to the original model. The mission of this component is to alter the value, or timing characteristics, of one or more signals when a fault is injected. On the contrary, the saboteur remains inactive during the normal operation of the system. In this case, saboteurs are functions which can supply the correct or the faulty value of the target object (constants, variables, signals, and conditions) depending on the value of a control signal.

The way a saboteur perturbs a DUV description depends on the adopted fault model. Indeed, the *RTE* is independent from the fault model, thus, changing the implementation of saboteur functions allows us to easily modify the fault model semantics. Every occurrence of signals, variables and constants and every condition of the functional level description can be replaced by an opportune saboteur. Moreover, a saboteur for every language type, i.e., bit, integer, standard logic, boolean, etc, can be defined. Faults are enumerated and a `bit_vector`-type port, named `fault_port` is added to the DUV. The fault port drives all control signals. The number of elements of the fault port equals the number of injected faults. In this way every fault is associated to a unique element of the fault port. A fault is activated (deactivated) by the presence of a `‘1’` (`‘0’`) in the corresponding element of the fault port. Thus, a fault port, with all elements fixed to `‘0’` does not activate any fault. If the single fault assumption is desired, all configurations of the fault port with more than one element fixed to `‘1’` are not taken into account.

The adopted fault injection technique will be more deeply described in Chapter 7.

Constraint Generator

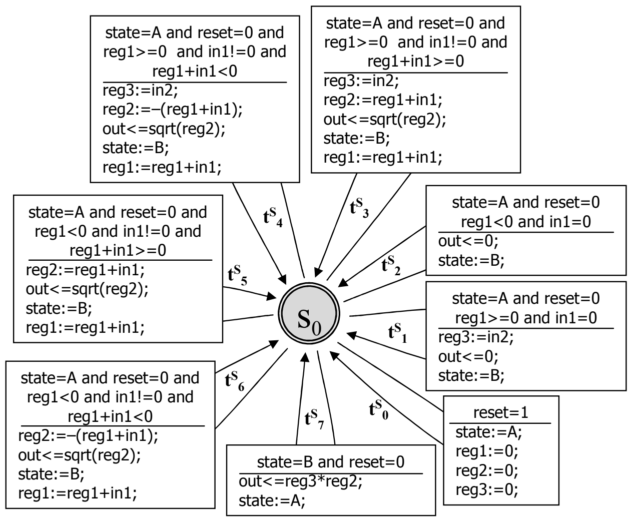

state=A and reset=0 and reg1>=0 and in1!=0 and reg1+in1<0
reg3:=in2; reg2:=-(reg1+in1); out<=sqrt(reg2); state:=B; reg1:=reg1+in1;
state=A and reset=0 and reg1>=0 and in1!=0 and reg1+in1>=0
reg3:=in2; reg2:=reg1+in1; out<=sqrt(reg2); state:=B; reg1:=reg1+in1;
state=A and reset=0 reg1<0 and in1=0
out<=0; state:=B;
state=A and reset=0 reg1>=0 and in1=0
reg3:=in2; out<=0; state:=B;
reset=1
state:=A; reg1:=0; reg2:=0; reg3:=0;
state=B and reset=0
out<=reg3*reg2; state:=A;
state=A and reset=0 and reg1<0 and in1!=0 and reg1+in1<0
reg2:=-(reg1+in1); out<=sqrt(reg2); state:=B; reg1:=reg1+in1;
state=A and reset=0 and reg1<0 and in1!=0 and reg1+in1>=0
reg2:=reg1+in1; out<=sqrt(reg2); state:=B; reg1:=reg1+in1;
S_0
t^S_0
t^S_1
t^S_2
t^S_3
t^S_4
t^S_5
t^S_6
t^S_7

Fig. 6.3. Example of EFSM.

```
bool eval_func (LookInDesign * lid, int trans, TestVector* tvs,
map<string, pair<int, int> >* tv_nos, pipestream& soolver) {
  int offset = 0;
  int name_size = 0;
// STEP 1
  /* Retrieving the value of the reset signal, its value
     depends on the simulation, thus it cannot be forced */
  sc_signal< sc_bit >* reset = lid->retrieve_reset();
  /* Retrieving the value of register reg1, its value depends
     on the simulation, thus it cannot be forced */
  Lrt_signal_var< sc_int<32> >* lrt_sig_reg1 =
  (Lrt_signal_var< sc_int<32> >*)lid->retrieve_lsig("reg1");
  /* Retrieving the value of register state, its value
     depends on simulation, thus it cannot be forced*/
  Lrt_signal_var< sc_bit >* lrt_sig_state =
  (Lrt_signal_var< sc_bit >*)lid->retrieve_lsig("state");
  switch (trans) {
      ...
      case 3:
// STEP 2
      /* Evaluating the static part of the enabling function,
         if it is not satisfied the transition cannot be fired */
      if ((lrt_sig_state->read() != A )||((reset->read() != 0 ))
          return (false);
      /* The dynamic part of the enabling function is evaluated */
      /* in1 is an input port, its value can be forced to fire
          the enabling function of transition 3 */
      map<string, pair<long int, long int> > * vars =
      new map<string, pair<long int, long int> >();
      map<string, string> * values = NULL;
      (*vars)["in1"] = make_pair(-32768, 32767);
      /* A string representing the constraint  in1!=0 and
         reg1>=0 and reg1+in1>=0  is generated */
      string exp = "(!(in1=0))&("+int_to_string(lrt_sig_reg1->read())
      +">=0)&(("+int_to_string(lrt_sig_reg1->read())+"+in1)>=0)";
      /* A constraint solver is invoked to solve the constraint*/
      values = solver->get_variable_value(vars, exp);
// STEP 3
      /* If the constraint cannot be satisfied the
         transition cannot be fired */
      if (values->size() == 0)
          return (false);
      /* If the constraint is satisfied the test vector is modified
         by assigning to in1 the value provided by the solver */
      sc_int<32> in1 = string_to_int((*values)["in1"]);
      offset =(*tv_nos)["in1"].first;
      for (int i = 0; i < in1.length(); i++) {
          tvs->force_bit((char)(in1[in1.length()-i-1] + '0'),
          offset++);
      }
      return (true);
      ...
  }
}
```

Fig. 6.4. Example of `eval_func`.

The *Constraint Generator* automatically creates a C++ function (referred as `eval_func` in the following) to allow the RTE to evaluate the enabling and updated functions constraints when the EFSMs are navigated. Such a function is constituted by a single `case` statement with one alternative for each transition of the S^2EFSM. For example, Figure 6.4 shows the slice of code of the `eval_func` related to transition t_3^S of the EFSM of Figure 6.3(d). The `eval_func` declares the following parameters:

- `lid`: a reference to the SystemC code of the DUV. It is used to retrieve the values of internal registers to evaluate the enabling function of the transition that must be fired;
- `trans`: the identification number of the transition to be fired;
- `tvs`: a test vector generated by the *random engine* of the *RTE*. The *RTE* initially fills the vector with random values. Then, when the `eval_func` is invoked, `tvs` is modified accordingly with the values provided by the constraint solver. In particular, if the enabling function of the transition `trans` is satisfied, the part of `tvs` related to the PIs involved in the enabling function is modified accordingly to the values returned by the constraint solver (see Section 6.1.2);
- `tv_nos`: a reference to a data structure which retains information about the position of each PI in `tvs`;
- `solver`: a reference to the *constraint solver interface* module which allows to use a constraint solver.

The `eval_func` works as follows when it is invoked by the *EFSM Navigator*:

1. At each simulation cycle, the simulation state is frozen and the values of internal registers are retrieved. Then, the evaluation of the enabling function of the transition to be traversed starts;
2. Some conditions of the enabling function can be evaluated without invoking a constraint solver, i.e., those which involve only internal registers and constants. If such conditions are not satisfied, the `eval_func` return `false`. Thus, the transition cannot be fired, and the *EFSM Navigator* must choose a different one. Otherwise, conditions which involve PIs are extracted from the enabling function, and a constraint solver is called.
3. If the constraint solver provides a solution, then the test vector is modified by fixing the values of PIs involved in the constraint with the values returned by the solver. Otherwise, false is returned like in step 2.

Run-time engine (RTE)

The RTE is a SystemC library which can be linked to the constraint description, and to the faulty and fault-free descriptions of the DUV to obtain a single executable. This definitely improves the performance of the test pattern generator. The RTE involves three main submodules described in the following: the *EFSM Navigator*, the *Constraint Solver Interface* and the *Simulation Engine*.

EFSM Navigator

The *EFSM Navigator* is the heart of the ATPG engine. In this paragraph a pseudo-deterministic approach is proposed, referred as *Random Walk*. A more sophisticated approach is described in Section 6.2. The Random Walk uniformly navigates the STG of the EFSM, and it interacts with the *Random Engine* and with the *Constraint Solver Interface* to generate test sequences. The DOT document describing the STG of the EFSM is parsed and an in-memory representation is generated. Then, the *EFSM Navigator* exploits the information provided by the enabling functions of the EFSM to increase the capability of detecting random resistant faults. In particular, it tries to uniformly move across the transitions of the EFSM. On the contrary, a random ATPG tends to traverse only transitions whose enabling functions present a high probability of being satisfied. Starting from a reset condition, the *EFSM Navigator* randomly selects a transition, then it tries to satisfy its enabling function by assigning values to the PIs of the DUV. When it succeeds, it provides the input vectors to the *Simulation Engine* which performs a simulation cycle to update the internal registers of the DUV, and to check for fault detection. Then, the *EFSM Navigator* selects another transition, and so on.

It is worth to note that the *EFSM navigator* can be applied to traverse whatever kind of EFSM, not only S^2EFSMs. However, the *EFSM navigator* is particularly suited to be applied to S^2EFSMs, since their transitions have a more uniformly distributed probability of being activated (see Section 5.4.3 and Section 5.5). More formally, given the set T_{s_i} of transitions out-going from a state s_i, at step i, the *EFSM Navigator* works as follows:

1. Randomly choose a transition $t_{s_i} \in T_{s_i}$;
2. check if the enabling function e of t_{s_i} is satisfiable, i.e., if it can be fired by assigning opportune values to inputs involved in e. For example, let `x` be an input of the EFSM, the enabling function `x=0` can be fired by assigning `0` to `x`. On the contrary, if `x` is an internal register, the satisfiability of the enabling function depends on the previous assignment to `x`, i.e., on the current configuration of the EFSM[1]. This step is performed by invoking a constraint solver by means of the *Constraint Solver Interface* module;
3. assign to inputs involved in e such opportune values if e is satisfiable. Otherwise, remove t_{s_i} from T_{s_i} and come back to step 1;
4. generate random values for inputs not involved in e. To accomplish this task, the *Random Engine* is invoked;
5. simulate the test vector so obtained, move across t_{s_i}, and come back to step 1 to generate the next test vector.

The *EFSM Navigator* resets the EFSM state periodically according to a user parameter, and it finally stops when the target fault coverage has been reached or the maximum allowed computation time has been expired.

[1] A configuration stores the status of the EFSM, i.e., the value of its internal registers (cf. the definition of EFSM in Chapter 5)

Constraint Solver Interface

This module interfaces the *EFSM Navigator* to an external constraint solver through the `eval_func` previously described. While the `eval_func` is automatically generated for each DUV, this interface does not require to be modified when a different DUV is analyzed. To launch a solving session, the interface provides the `get_variable_value` function which is called by the `eval_func` (see Figure 6.4). Two different interface modules have been implemented: one is tailored for invoking a SAT-solver: *zChaff* [214], the other for invoking a constraint solver: *ECLiPSe* [215].

zChaff is invoked by supplying the negation of the constraint that represents the enabling function to be satisfied to fire the corresponding transition. If a counterexample is found, its input projection represents the set of values that PIs must assume to fire the transition. On the contrary, the enabling function cannot be satisfied, and the negative answer is reported to the *EFSM Navigator*.

The main problem related to the use of zChaff is due to the fact that different invocation of the solver on the same constraint provides the same counterexample. This reflects on the quality of the generated test sequences. They allow to uniformly navigate across the EFSM, but always with the same data. Thus, the achieved fault coverage can be very low for data-dominated circuits, even if all execution paths of the EFSM are exercised. Consider, for example, the following slice of code:

```
if (data >= 0)
  out = data;
```

Every time zChaff is invoked to solve the constraint `data >= 0` , it provides the same value for `data`, for example `0`. Now, let us suppose the presence of a fault on the statement `out = data` which, when it is active, forces `out` to assume the value `0`. Such a fault can never be detected! On the contrary, zChaff would provide a value greater than `0` very likely, if it was able to provide a different value for `data` each time it is called to solve the same constraint.

The problem can be avoided by using ECLiPSe instead of zChaff. ECLiPSe is a Constraint Logic Programming system which can be adopted to search solutions for symbolic and numeric constraints. Given a satisfiable constraint, it can provide its whole solution space, not just a single solution. Moreover, a random value within the solution space can be easily extracted. The use of ECLiPSe is particularly efficient, since there exist a C++ programming library which can be directly linked to the other modules of the ATPG framework. In this way, no interprocess communication mechanism (like pipe, sockets, etc.) is required at all.

Simulation Engine

This module interacts with the *EFSM Navigator*, each time a transition of the EFSM is fired, to move the simulation of the DUV one step ahead. The *Simulation Engine* has two purposes: updating the internal registers of the DUV when a transition is traversed (i.e., executing its update function), and checking for fault detection. In particular, the *Simulation Engine* works as follows:

1. A fault `f` is activated by opportunely acting on the saboteur control signals of the faulty DUV;

2. the faulty and fault-free DUVs are reset;
3. each time a vector `t` is provided by the *EFSM navigator*, `t` is applied to the PIs of the faulty and fault-free DUVs. Then, a simulation cycle is performed. In this way, internal registers are updated;
4. the POs of the faulty and fault-free DUVs are compared. If a discrepancy is observed:
   a) `f` is marked as detected;
   b) the sequence `s` constitutes by all the simulated test vectors starting from the last reset are saved in the test set;
   c) the faulty and fault-free DUVs are reset;
   d) for performance reasons, `s` is simulated for all undetected faults if some of them can be tested by `s` too.
5. on the contrary, if no discrepancies are observed, the simulator waits for another test vector to be simulated, then it resumes from step 3. If the maximum length for the test sequence is reached without any fault detection, the test sequence is discarded, and the simulator resumes from step 1 by selecting a new fault.

The complexity of industrial designs and the huge number of faults that must be injected into them require efficient fault simulators, in order to make verification via fault simulation an affordable task. To optimize fault simulation performances, some parallelization techniques have been proposed at gate level. On the contrary, they have not been fully exploited at RTL, where functional fault models, instead of gate-level ones, are considered. Thus, Chapter 8 analyzes the impact of such parallelization techniques on functional faults. In particular, possible issues are presented together with optimizations that can be implemented to speed up the simulation.

6.1.3 Multi-Process Scheduling

This thesis proposes a method to represent a digital system as a set of concurrent EFSMs, one for each process of the DUV. Concurrency is intended as the possibility that each EFSM of the same DUV changes its state concurrently to the other EFSMs to reflect the concurrent execution of the corresponding processes. Data communication between concurrent EFSMs is guaranteed by the presence of common signals. In this way, structured models can be represented.

The navigation of concurrent EFSMs is guaranteed by an opportune EFSM scheduling algorithm that aims at maximizing the ATPG capability of exploring the whole state space. In this way multiprocess DUVs can be uniformly traversed.

In particular, two reasons induce introducing a scheduling algorithm. A first motivation is due to the fact that the same primary inputs may be involved in the enabling functions of the transitions of two or more EFSMs. For example, let us consider a DUV composed of two EFSMs, M and N. Moreover, let us suppose that at a certain simulation cycle, the ATPG selects the transition t^M from M and the transitions t^N from N. The enabling functions of t^M and t^N can be compatible or conflicting. In the first case, there exists a value assignment to primary inputs that satisfies the enabling functions of both t^M and t^N. Thus, primary inputs can be deterministically fixed to traverse both t^M and t^N. In the second case, such an assignment does not exist. Thus, t^M and t^N cannot be traversed concurrently, and one of them must be discarded and substituted with a different transition to remove the conflict. In this case, the scheduling algorithm is used to decide the priority of each EFSMs for fixing primary-input values. When a conflict happens, the transitions to be substituted are selected starting from the EFSMs with the lowest priority.

A second reason that motivates the use of a scheduling algorithm depends on the fact that not all EFSMs must be triggered at each simulation cycle. Think, for example, to an EFSM corresponding to a process whose sensitivity list's signals remain unchanged.

According to the previous considerations, a priority-based scheduling algorithm has been implemented. In this way, at each simulation cycle, the ATPG generates the test vector by deterministically fixing the primary inputs in order to traverse the transition of the highest-priority EFSM. Then, the primary inputs not involved in such a transition, are deterministically assigned according to the transition of the next-priority EFSM, and so on, until all primary inputs are fixed.

To implement such a policy, the scheduler relies on a two-level-queue mechanism without feedback. The queue with the highest priority permanently includes EFSMs extracted from clock-sensitive processes. Such EFSMs are simulated at each clock cycle. The second queue permanently includes EFSMs extracted from asynchronous processes that are triggered by signals modified by the processes of the first queue. Within each queue, the EFSMs are sorted by assigning a dynamic priority computed as the sum of a constant value, F, and an aging factor A.

The value F assigned to each EFSM is inversely proportional to the number of inputs included in its enabling functions. Larger is the number of inputs involved in the enabling functions, lower is the value F assigned to the EFSM, and later the EFSM is navigated. In this way, the DUV exploration is more uniformly distributed. In fact, if an EFSM, which involves many inputs on the transition-to-be-traversed, is scheduled first, its decision on inputs values definitely constraints the behavior of the remaining EFSMs. This may cause an incomplete exploration of the DUV. Analogously, the aging factor has been introduced to avoid that the behavior of low-priority EFSMs is always constrained by decisions taken by high-priority EFSMs. Initially, the value A is 0 for all EFSMs. Then, each time an EFSM is forced to discard and substitute the transition to be traversed (because the values assigned to inputs by a higher-priority EFSM do not satisfy its enabling function) the aging factor of EFSM is incremented of a constant quantity. On the contrary, the aging factor is reset to 0 after the EFSM reaches the highest priority. Thus, sooner or later, every EFSM becomes the highest-priority one.

6.1.4 Experimental Results

The efficiency of the proposed ATPG framework has been evaluated by using the benchmarks described in Table 6.1 where columns report the number of primary inputs (*PIs*), primary outputs (*POs*), flip-flops (*FFs*), gates (*Gates*) and the number of bit coverage faults (*B.C.*. Such benchmarks have been selected because they present different characteristics which allow to analyze and confirm the effectiveness of the ATPG framework. *ex1* is the example reported in Figure 6.3.

Name	PIs	POs	FFs	Gates	B.C.
ex1 16 bit	34	16	66	3069	461
ex1 32 bit	66	32	130	10754	907
b00 16 bit	34	32	51	904	601
b00 32 bit	66	64	99	1692	1182
b04	13	8	66	650	408
b10	13	6	17	264	216
b11m	9	6	31	715	725
b00z	66	64	99	11874	1439
fr	34	32	100	1475	1041
dlx	29	31	25	232	1167

Table 6.1. Characteristics of benchmarks.

	GA-ATPG				PD-ATPG				
Name	FC%	SC%	TV#	T	FC%	SC%	TV#	CLP T	SAT T
ex1 16 bit	49.7	85.7	11	35 s.	50.1	92.9	16	33 s.	51 m.
ex1 32 bit	78.2	85.7	38	92 s.	80.3	92.9	58	80 s.	aborted
b00 16 bit	3.83	26.7	2	202 s.	28.5	87.0	26	152 s.	60 m.
b00 32 bit	1.1	26.7	4	421 s.	48.7	87.0	66	360 s.	aborted
b04	94.9	98.0	119	40 s.	99.0	100.0	255	15 s.	51 m.
b10	87.0	66.7	175	47 s.	93.0	69.7	194	47 s.	41 m.
b11m	37.0	80.0	149	60 s.	39.0	82.2	145	54 s.	64 m.
b00z	13.7	31.0	2	180 s.	44.3	75.9	33	220 s.	aborted
fr	0.86	13.3	35	1175 s.	70.4	86.7	375	557 s.	135 m.
dlx	35.1	50.7	180	216 s.	46.7	63.9	145	211 s.	60 m.

Table 6.2. ATPG results.

Two versions of such an example have been implemented, one with 16 bit-sized inputs and the other with 32 bit-sized inputs. *b04*, *b09* have been selected from the well known ITC-99 benchmarks suite [198]. *b11m* is a modified version of *b11*, included in the same suite, created by introducing a delay on some paths to make it harder to be traversed. The original HDL descriptions of *b04*, *b09* and *b11m* contain a high number of nested conditions on signals and registers of different size. *b00*, *b00z* and *fr* contain conditional statements where one branch has probability $1 - \frac{1}{2^{32}}$ of being satisfied, while the other has probability $\frac{1}{2^{32}}$. Thus, they are very hard to be tested by a random ATPG. In particular, *b00* and *b00z* are internal benchmarks, while `fr` is a real industrial case, i.e., it is a module of a face recognition system, whose description is composed of two interacting processes. Finally, *dlx* is the controller of the well known RISC processor.

Section 6.1.2 reports that the framework is independent from the adopted fault model. To evaluate its efficiency, a high-level fault model has been selected. It is the bit coverage fault model that will be deeply described in Section 7.1. Moreover, statement coverage has been computed too.

Table 6.2 reports a comparison between the performance of a genetic-based ATPG, which outperforms a pure random-based ATPG, (*GA-ATPG*) and the performance of the proposed ATPG (*PD-ATPG*). In particular, the Table shows the fault coverage (*FC%*), the statement coverage (*SC%*), the size of the test set

(*TV*#), and the test generation time (*T*, *CLP T*, *SAT T*, by using respectively the GA-ATPG, the PD-ATPG with ECLiPSe, and the PD-ATPG with zChaff). The time to obtain the EFSMs from the HDL code is negligible with respect to the test generation time (few seconds). For some benchmarks zChaff was unable to solve constraints, and it aborted. On the contrary, when it succeeded, the achieved fault coverage was equal to the fault coverage achieved by using ECLiPSe. However, ECLiPSe always outperformed zChaff from the execution time point of view.

6.2 Deterministic EFSM-based engine

This Section describes the functional ATPG engine that explores deterministically the DUV state space by exploiting an easy-to-traverse extended FSM model. The ATPG engine relies on *Learning*, *Backjumping* and constraint logic programming techniques to deterministically generate test vectors for traversing all transitions of the EFSM. Testing of hard-to-detect faults is thus improved.

This Section is organized as follows. Section 6.2.1 introduce the problem and summarizes related previous works. Section 6.2.2 presents the defined deterministic ATPG engine, namely *FATE*. Finally, Section 6.2.3 reports experimental results.

6.2.1 Introduction

Many high-level ATPGs have been proposed to generate effective test sequences [216, 217, 218, 199, 219, 220, 221]. Generally, high-level ATPGs can be divided in two main categories: random-based and deterministic. The first set adopts simulation-based strategies guided by genetic algorithms or other probabilistic-based techniques [218, 199]. Generally, they rely on high-level fault models or coverage metrics which require to accordingly instrument and simulate a HDL description (e.g., SystemC, VHDL, Verilog, etc.) of the design under test (DUV). These ATPGs are fast, and they allow to quickly achieve a high coverage for easy-to-test designs. However, they tend to generate a large number of test sequences and they unlikely cover corner cases on complex DUVs. On the contrary, deterministic ATPGs exploit mathematical strategies tailored to allow a complete exploration of the DUV state space [216, 217, 219, 220, 221].

The development of a functional ATPG requires to deal with four basic aspects:

1. the formalism used to model the DUV (described in Chapter 5);
2. the algorithm to take decisions to move from a state to another one during DUV state exploration (e.g., genetic algorithms [222], SAT-solving [220], constraint logic programming [221], linear programming [220], etc.);
3. the strategy to deterministically reach particular states of the DUV representing corner cases (e.g., learning [223], justification [219], backtracking [217], backjumping [224], etc.);
4. the metrics to evaluate the quality of generated test sequences, i.e. transition coverage [9], path coverage [25], statement coverage [25], fault coverage [216] (the adopted metrics are described in Chapter 7).

Such ATPGs are more effective to cover corner cases than random-based ones, but their execution time tends to increase proportionally to the complexity of the adopted deterministic strategy. Moreover, deterministic ATPGs can be unable to complete the testing session, when applied to complex systems that lead the DUV model to the explosion of states.

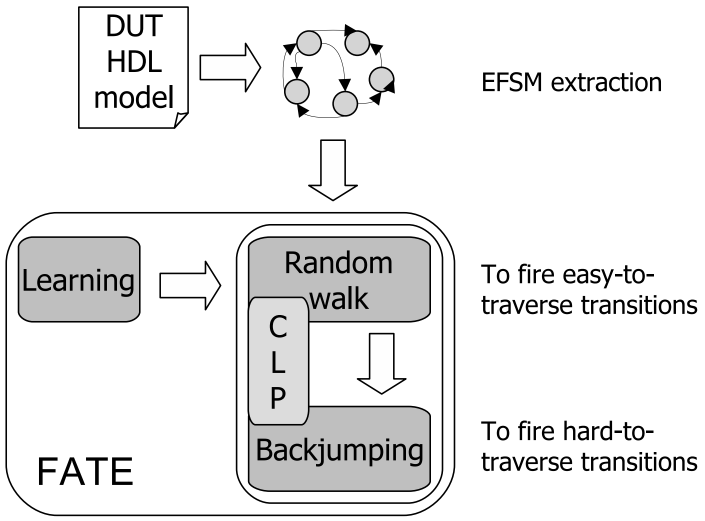

DUT
HDL
model
EFSM extraction
Learning
Random walk
C
L
P
Backjumping
FATE
To fire easy-to-traverse transitions
To fire hard-to-traverse transitions

Fig. 6.5. The FATE flow.

In this context, this thesis presents a functional deterministic ATPG, *FATE*, which addresses the previous aspects as follows (see Figure 6.5):

1. The EFSM paradigm is used to model the DUV. In particular, FATE works on a special kind of EFSM whose transitions present a quite uniformly distributed probability of being deterministically traversed, see Chapter 5;
2. a constraint logic programming-based (CLP) strategy is adopted to deterministically generate test vectors that satisfy the guard of the EFSM transitions selected to be traversed;
3. a two-step ATPG engine is implemented which exploits CLP to traverse the DUV state space: first, the *Random Walk*-based approach is used to cover the majority of easy-to-traverse (ETT) transitions; then a *Backjumping*-based mode is used to activate hard-to-traverse (HTT) transitions. In both modes, *Learning* is exploited to get critical information that improves the performance of the ATPG;
4. transition coverage is used to verify the goodness of the proposed ATPG, since 100% transition coverage represents a necessary condition for fault coverage and for more accurate coverage metrics. Then, the high-level bit coverage fault model has been adopted, see Chapter 7, to measure the functional coverage of the generated test sequences, which have been also simulated at gate-level on stuck-at faults to check fault coverage on logic-level implementations.

There are few papers in the literature propose ATPG based on the EFSM model. However, as already introduced in this Chapter, test generation techniques adopted for FSMs cannot be efficiently reused for EFSMs.

In [9] a breadth first search is used to generate a set of test patterns which covers all the transitions on the stabilized EFSM. The main limitations of this approach are represented by the complexity of the orthogonalization and stabilization processes, which may lead to state explosion. Moreover, the breadth-first search-based approach is surely less efficient than strategies based on learning and backjumping. In fact, these methods improve the performance of the ATPG by avoiding the starvation of DUV exploration in areas of the state space very far from the desired target. In particular, Backjumping is a special kind of backtracking strategy, also known as non-chronological backtracking, which rollbacks from an unsuccessful situation directly to the cause of the failure. Thus, it is more efficient than backtracking. In fact, the basic backtracking algorithm rollbacks to the most recent decision point before proceeding to a different search direction [225]. However, there is no guarantee that the most recent decision point is the source of failure. Thus, backtracking, differently from backjumping, may require many rollbacks before solving the conflict.

Many papers propose to use backtracking at gate-level to search for a path that propagates a target value to a target net. On the contrary, to the best of author knowledge, only the MIX-PLUS ATPG [226] exploits Backjumping for gate-level test generation. However, the approach of MIX-PLUS is different from what proposed in this work for the following reasons:

- MIX-PLUS uses Backjumping at gate-level, while FATE uses it at functional level;
- MIX-PLUS needs to generate and dynamically maintain an implication graph for using backjumping. However, the size of such a graph can exponentially increase for complex sequential circuits where justification is applied for several time frames. On the contrary, FATE statically generates a list that reports which transitions of the EFSM update the value of each register. The size of such a list is fixed and it equals at maximum $R*T$, where R is the total number of registers and T is the total number of transitions of the EFSM (in the extreme case where each register is updated in each transition).

The use of the implication graph is mandatory at gate-level, since the gate-level netlist does not include information to bind a conflict point directly to its cause. Such a problem is solved at functional level by FATE. In fact, the adoption of the EFSM model, joint to an accurate Learning phase, allows FATE to:

1. deterministically backjumps to the source of failure when a transition, whose guard depends on a previously set register, cannot be traversed;
2. modify the EFSM configuration to satisfy the condition on the register;
3. successfully come back to the target state to traverse the transition. In this way, FATE can efficiently traverse EFSMs without requiring stabilization.

6.2.2 The FATE ATPG Engine

The proposed ATPG works on the set of concurrent EFSMs representing the DUV in a three-step fashion. First, an off-line Learning phase is performed on the EF-

SMs to collect information about location of registers within enabling and update functions. Then, in the second phase, a pseudo-deterministic test pattern generation approach is adopted to uniformly traverse easy-to-traverse transitions. During this phase, information on state and transition reachability is also learned. Finally, in the third phase, the ATPG exploits information collected in the previous steps, by means of a backjumping-based approach, to traverse transitions that have not been traversed yet.

In the last two phases, the ATPG exploits the information provided by the enabling functions of the EFSMs to uniformly move across the transitions of each EFSM of the DUV. In this way, the capability of traversing HTT transitions is increased. On the contrary, a random ATPG tends to traverse only transitions whose enabling function presents a high probability of being satisfied by assigning random values to primary inputs.

Phase 1: Learning

The set of EFSMs representing the DUV is generated according to the approach described in Chapter 5, starting from a functional description of the DUV. During the EFSM generation, the following information is collected for each register *reg*:

- the set of transitions T^e_{reg} whose enabling functions involve *reg*;
- the set of transitions T^u_{reg} whose update functions update *reg*;
- the set of registers, and the corresponding locations in enabling and update functions, which behave as counters.

The previous information is useful to deterministically traverse register-dependent HTT transitions by exploiting Backjumping, as described in Section 6.2.2.

Moreover, a function *eval_func* is generated for each EFSM of the DUV. The *eval_func* allows the ATPG to evaluate at run-time the enabling functions of the transitions to be traversed. Each function consists of a single `case` statement with one alternative for each transition of the corresponding EFSM. When the function is invoked by the ATPG, the alternative related to the transition to be traversed executes as follows:

1. At each simulation cycle, the simulation state is frozen and the values of DUV internal registers are retrieved. Then, the evaluation of the enabling function of the transition to be traversed starts.
2. Some conditions of the enabling function can be evaluated without invoking a constraint solver, i.e., those which involve only internal registers and constants. If such conditions are not satisfied, the *eval_func* returns `false`. Thus, the transition cannot be traversed, and the ATPG must either choose a different transition, if it is running in Random-Walk mode (Section 6.2.2), or remove the cause of unsatisfiability, if it is running in backjumping mode (Section 6.2.2). When conditions on registers are satisfied, a constraint solver is called on conditions which involve primary inputs.
3. If the constraint solver provides a solution for conditions on primary inputs, then the values returned by the solver are collected to compose the test vector. If the constraint solver fails, `false` is returned like in step 2.

Phase 2: Random Walk

During the random walk phase, the ATPG randomly walks across the transitions of the EFSMs representing the DUV. Thus, ETT transitions are very likely traversed.

Starting from a reset condition, the ATPG randomly selects a transition from each EFSM according to the scheduling policy described in [227]. Then, it tries to satisfy the enabling function of each selected transition by exploiting the constraint solver invoked by the *eval_func* previously described. When it succeeds, the values for primary inputs provided by the constraint solver are used to generate a test vector. Finally, a simulation cycle is performed, by using the generated test vector, to update the internal registers of the DUV and to move to the destination state. Then, another transition is selected, and the cycle repeats. More formally, given the set T_{s_i} of transitions out-going from a state s_i, at step i, the ATPG works as follows:

```
/* sequence number and sequence length counters */
integer seq_num := 0; integer seq_len := 0;

/* loop: exit on achievement of maximum number of sequences or
 transition coverage equals 100% */
while seq_num <= MAX_SEQ_NUM && transition_cov <= 100 do

  /* reset the DUV */
  reset_step();
  seq_len := 1;

  /* loop: exit on achievement of maximum number of
           test vectors for the current sequence */
  while seq_len <= MAX_SEQ_LEN do

    /* create an empty test vector */
    TestVector tv;

    /* Step 1, 2, 3: update the test vector during the
       Random Walk phase */
    update_tv(tv);

    /* Step 4: apply test vector to the DUV*/
    drive_PI(tv);
    clock_step();

    /* add test vector to the current test sequence */
    seq_len++;

  end while;

  /* add test sequence to the test set */
  seq_num++;

end while;

function update_tv(Testvector tv) {

/* Step 1: order the EFSMs according to the scheduling policy */
list<EFSM> efsm_list = Scheduler(scheduling_policy);

/* Step 2: for each efsm in the schedule */
for_each efsm in efsm_list do

  /*  if there is a transition outgoing the current state */
  while  efsm.has_transition() do

    /* Step 2.a: choose randomly a transition */
    Transition transition := efsm.pickup_transition();

    /* extract the transition enabling function */
    ef = transition.enabling_funcion;
    /* Step 2.b: evaluate the enabling function
         by using the CLP solver and accordingly fix the PIs */
    if  Constraint_solver.eval_func(ef) then

       /* Step 2.c: assign values provided by
            the constraint solver*/
       fix_value(tv, Constraint_solver.get_value());

       /* Step 3: generate random values for the
          PIs not involved in the enabling function */
       fix_randomly_value(tv, ef);

       /* exit from the while */
       break;

    end if;

  end while;

end for_each;
end function;
```

Fig. 6.6. Pseudo-code of the random walk phase.

1. Order the EFSMs according to the scheduling policy reported in [227].
2. For each EFSM:
   a) Randomly choose a transition $t_{s_i} \in T_{s_i}$.
   b) Call the *eval_func* described in Section 6.2.2 to check if the enabling function e of t_{s_i} is satisfiable, i.e., if it can be traversed by assigning opportune values to inputs involved in e. For example, let `x` be an input of the EFSM, the enabling function `x=0` can be activated by assigning `0` to `x`. On the contrary, if `x` is an internal register, the satisfiability of the enabling function depends on the previous assignment to `x`, i.e., on the current configuration of the EFSM. Note that, for DUV composed of more than one EFSM, the transitions selected at step (a) may require to assign conflicting values to the same primary inputs to be concurrently traversed. In this case, according to the scheduling policy, the EFSM with the highest priority wins, while the others must consider such primary inputs as internal registers whose value cannot be changed.
   c) Assign to primary inputs involved in e the values provided by the constraint solver, if e is satisfiable. Otherwise, remove t_{s_i} from T_{s_i} and come back to step 2.
3. Generate random values for primary inputs not involved in the enabling functions of the selected transitions. To accomplish this task, a random engine is invoked.

4. Invoke the simulation engine to simulate the test vector so obtained, move across the selected transitions, and come back to step 2 to generate the next test vector.

Each time a test vector is generated, the traversed transition is labeled with the pair <*test sequence number, test vector number*>. A list of pairs of parametric length is saved for each transition. In this way, the backjumping mode can exploits such lists to quickly recover the prefix of a test sequence which allows the ATPG to move from the reset state to an already visited target state.

The pseudo-code of the random walk algorithm is reported in Figure 6.6.

Phase 3: Backjumping

The ATPG automatically changes to the backjumping mode when the computation time assigned to the random walk expires, or no coverage improvement is provided for long time. Thus, the ATPG works as follows:

1. Collect the not yet traversed transitions in an ordered list. Not traversed transitions, out-going from a state already visited during the random walk phase, are inserted at the beginning of the list. Such transitions should be more easily traversable with respect to transitions out-going from states never reached[2].
2. Pick up a transition t from the beginning of the list. Does its enabling function involve only conditions on primary inputs?
   - If yes, retrieve the pair <*test sequence number, test vector number*> corresponding to the source state S_t of t. Then, load and simulate such a test sequence s up to the vector tv indicated in the pair. In this way, the DUV moves from the reset state to S_t. Finally, invoke the constraint solver to generate primary inputs values to traverse t, simulate the new test vector so obtained, and go to step 8.
   - If no, the enabling function of t involves conditions on registers. It could be the case that the enabling functions is satisfiable by the register configuration generated by simulating s[3]. Thus, in this case, generate values for primary inputs involved in the enabling functions by means of the constraint solver, simulate the so obtained test vector, and go to step 8. Otherwise go to step 3.
3. For sake of clarity, let us suppose that the enabling function of t is expressed in the conjunctive normal form (CNF), and that its unsatisfiability depends on clauses involving a single register reg[4]. Then, extract an already visited transition t_u from the set of transitions T^u_{reg} whose update function updates reg. If all transitions in T^u_{reg} have never been visited, then, froze the situation of t, move to step 2 and solve the reachability problem of transitions in T^u_{reg}, finally come back to the problem related to t.

[2] Note that, if the list is not empty, there is always at least a transition out-going from an already visited state.

[3] This may happen if tv is the last vector of s.

[4] If the unsatisfiability of t depends on more than one register, the backjumping procedure is repeated for each of them.

4. Retrieve the pair <*test sequence number, test vector number*> corresponding to the source state S_{t_u} of t_u, and accordingly load the test sequence to move from the reset state to S_{t_u}. Thus, the ATPG *Backjumps* from S_t to S_{t_u}.
5. Use the Dijkstra's shortest path search algorithm [228] to provide a path π from S_{t_u} to S_t starting with t_u, without worrying about the satisfiability of enabling functions involved in the path (the satisfiability of enabling functions of π is addressed in step 7). The weight required by the Dijkstra's algorithm for each transition, to guide the search heuristic in a greedy-fashion, is computed by considering the number of registers involved in the enabling function. Higher is such a number, lower is the weight assigned to the corresponding transition. This is motivated by the consideration that the hardness in satisfying an enabling function increases proportionally to the number of involved registers. Thus, a path whose transitions involve few conditions on registers is easier to be traversed.
6. Satisfy the enabling function of t_u according to the constraints derived from the enabling function of t as follows. Let us suppose that e_{t_u} is the enabling function of t_u and $e_t|_{\mathtt{reg}}^{t_u}$ is the part of the enabling function of t which involves the clauses depending on *reg*, where each occurrence of *reg* has been substituted with the right-side expression of the assignment that updates reg in the update function of t_u (see, for example, Figure 6.7). Invoking the constraint solver to satisfy the constraint $e_{t_u} \wedge e_t|_{\mathtt{reg}}^{t_u}$ allows us to obtain a test vector which satisfies e_{t_u} and sets the value of reg in such a way that when simulation reaches transition t, following π, its enabling function will be correctly activated. The last observation may be false if there is a transition $t'_u \neq t_u \neq t$ in π, such that t'_u updates reg after t_u did. In this case, the ATPG moves the problem from t_u to t'_u requiring a solution for $e_{t'_u} \wedge e_t|_{\mathtt{reg}}^{t'_u}$.
7. Satisfy the enabling function of transitions included in π by iteratively applying the constraint solver to generate the corresponding test vectors. The test sequence obtained by joining s, to move from the reset state to S_{t_u}, and the test vectors generated to traverse π allow the ATPG to traverse t.
8. Remove t from the list of untraversed transitions and come back to step 1 until either the list of untraversed transitions is empty, or computation time expires.

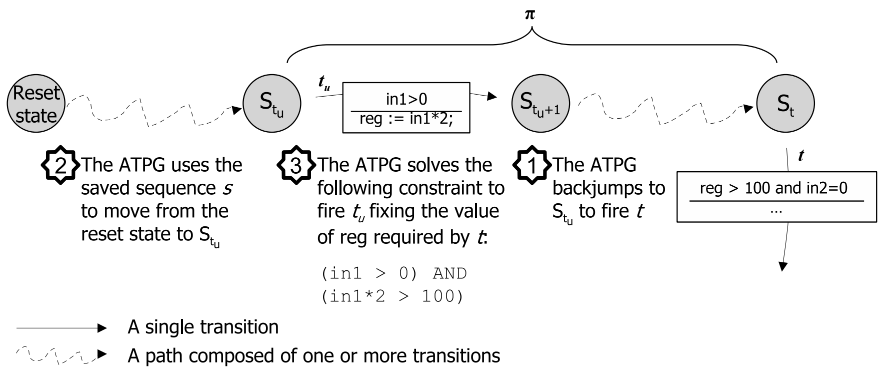

π
Reset state
S_{t_u}
t_u
in1>0
reg := in1*2;
S_{t_u+1}
S_t
t
reg > 100 and in2=0
...
2 The ATPG uses the saved sequence s to move from the reset state to S_{t_u}
3 The ATPG solves the following constraint to fire t_u fixing the value of reg required by t:
(in1 > 0) AND (in1*2 > 100)
1 The ATPG backjumps to S_{t_u} to fire t
A single transition
A path composed of one or more transitions

Fig. 6.7. The backjumping strategy.

```
/* Step 1: retrieve the list of unfired transition */

list<Transition> unfired_transition_list;

for_each efsm in efsm_list do

  /* order the list such that in front of it there
     are transitions exiting from states already
     visited during Random Walk phase */
  unfired_transition_list.append(efsm.get_unfired_transitions());

end for_each

for_each unfired_transition in unfired_transition_list do
  /* Step 2: pick up a transition and verify if its
             enabling function is satisfiable */

  /* retrieve subsequence from previously generated test set */
  pair<Tseq_index, Tvec_index> sub_sequence;
  sub_sequence := test_set.get_seq_until(unfired_transition);

  /* simulate the subsequence */
  reset_step();
  simulate(sub_sequence);

  /* extract the transitions enabling function */
  ef := unfired_transition.enabling_func;

  /* evaluate the enabling function by using the CLP solver
     and accordingly fix the PIs */

  if  Constraint_solver.eval_func(ef) then
      fix_value(tv, Constraint_solver.get_value());
      seq_len++;
  else
      /* if the enabling function is not satisfiable
         switch to the backjumping strategy */
      backjump(unfired_transition);
  end if;

end_for_each;

function backjump(Transition unfired_transition)
  /* Step 3: extract a transition which updates the
     register involved  in the current unfired transition */
  Register_ef_reg := unfired_transition.enabling_func.get_REGs();
  Transition update_transition;
  update_transition := get_transition_updating_reg(reg);
  /* Step 4: retrieve the subsequence from the reset state up to
     the source of the transition which updates the register */
  pair<Tseq_index, Tvec_index> sub_sequence;
  sub_sequence := test_set.get_seq_until(update_transition);

  /* Backjump from unfired transition to updating transition */
  reset_step(); simulate(sub_sequence);

  /* Phase 5: use Dijkstra s algorithm to generate a path
     from the updating transition to the unfired transition */
  list<Transition> path := dijkstra_sp(update_transition.target(),
                                       unfired_transition.source());

  /* Phase 6: compose enabling function of the unfired transition
   * with the update function of the updating transition */
  EnablingFunction ef_unfired := unfired_transition.enabling_func;
  EnablingFunction ef_update := update_transition.enabling_func;
  UpdateFunction uf_update := update_transition.update_func;
  Constraint constraint := compose(ef_unfired, uf_update);
  Constraint_solver.eval_func(constraint && ef_update);
  fix_value(tv, Constraint_solver.get_value());
  seq_len++;

  /* Phase 7: iteratively apply the constraint solver to generate
     the test vectors for the transition in the Dijkstra s path */
  for_each transition in path do
     /*   */
  end for_each;

end function;
```

Fig. 6.8. Pseudo-code of the backjumping strategy.

The pseudo-code of the backjumping strategy is reported in Figure 6.8. The backjumping-based approach allows the ATPG to traverse transitions not traversed during the random walk, without requiring a complete stabilization of the EFSM. However, the algorithm may fails when the register involved in the enabling function of the transition to be traversed behaves as a counter variable. For this reasons, the basic backjumping mode has been extended as reported in the next paragraph.

Addressing Counters.

Let consider the case of a DUV with a register *reg* implementing a counter, as reported in Figure 6.9. To traverse the target transition t, the ATPG backjumps to the transition t_u whose update function updates *reg*. Then, the path $\pi = t_u, \pi_t$ is provided by the Dijkstra's algorithm. However, directly traversing π, after reaching S_{t_u} from the reset state, would be useless to traverse t. In fact, the enabling function of t cannot be satisfied until the path $\pi' = t_u, \pi_c$ has been traversed at least five times (if reg is initially fixed to 0).

The problem arising with counters can be avoided by stabilizing the EFSM. However, the stabilization of a transition involving a counter represents the best way to incur on the state explosion problem as reported in [9].

Thus, this study proposes to extend the backjumping mode of the ATPG as follows. During the learning phase, all counter registers are statically identified, as well as the transitions whose update function contains them. This is quite easy, since, at functional level, a counter is detected each time an update function contains an assignment where reg appears in both the left and the right sides. Thus, at run-time, if transition t_u has been labeled as "counter inside", the Dijkstra's algorithm is invoked two times: one to search for a path from S_{t_u+1} to S_t, and the other to provide a path from S_{t_u} to S_{t_u} including t_u. Then, the constraint solver is exploited to compute how many times π' must be traversed before moving on π. Finally, steps 6 and 7 of the backjumping-based approach previously described are applied to generate test vectors that allow to traverse π and π'.

This approach overcomes the strategy proposed in [9] to avoid stabilization of counter-dependent transitions. In fact, in [9], the authors restrict the problem to the case where the update function t_u includes only counters of the form `reg := reg + c`, where `c` is a constant. Moreover, in [9] the incrementing loop of the counter must be composed of a single transition going from S_{t_u} to S_{t_u}. On the opposite, this approach, exploiting a constraint solver, allows us to solve more complex situations, i.e., think to a counter whose increment depends on several subsequent assignments distributed in a set of adjacent transitions.

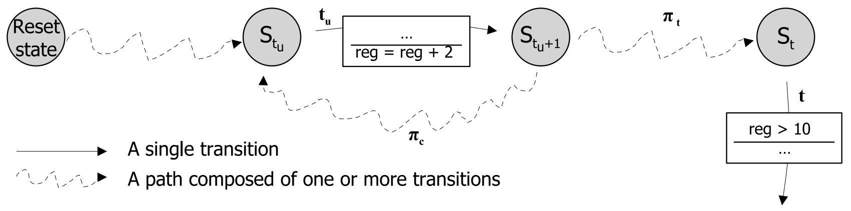


Fig. 6.9. Example of counter behavior.

6.2.3 Experimental Results

The efficiency of FATE has been evaluated by using the benchmarks described in Table 6.3, where columns report the number of primary inputs (*PIs*), primary outputs (*POs*), flip-flops (*FFs*) and gates (*Gates*). Column *Trns.* shows the number of transitions of the EFSM modeling the DUV and *GT (sec.)* the time required to automatically generate the S^{2}EFSM. Then, Column *BC* reports the number of bit coverage faults injected into the designs to check the fault coverage.

Such benchmarks have been selected because they present different characteristics which allow us to analyze and confirm the effectiveness of FATE. *b04*, *b10*

DUV	PIs	POs	FFs	Gates	Trns.	GT (sec.)	BC
ex1	66	32	130	10754	7	0.1	907
b00	66	64	99	1692	7	0.1	1182
b04	13	8	66	650	20	0.3	408
b10	13	6	17	264	35	0.3	216
b11m	9	6	31	715	20	0.2	725
b00z	66	64	99	11874	9	0.2	1439
fr	34	32	100	1475	10	0.2	1041
dlx	29	31	25	232	28	0.3	1167
diffeq	161	96	289	33510	4	0.9	3017
am2910	23	16	145	1598	543	3.1	5236
prawn	11	23	84	1996	191	1.5	3716

Table 6.3. Benchmarks properties.

	GA-ATPG				**PD-ATPG**				**FATE**			
DUV	**TC%**	**SC%**	**FC%**	**T (s.)**	**TC%**	**SC%**	**FC%**	**T (s.)**	**TC%**	**SC%**	**FC%**	**T (s.)**
ex1	71.4	85.7	78.2	3.3	85.7	92.9	80.3	2.9	100.0	100.0	96.0	3.1
b00	28.6	26.7	1.1	3.0	85.7	87.0	48.7	2.6	100.0	100.0	52.5	2.9
b04	80.0	90.2	94.9	23.2	85.0	95.0	99.0	8.7	100.0	100.0	99.0	9.1
b10	37.1	66.7	87.0	5.7	40.0	69.7	93.0	5.7	100.0	100.0	94.0	6.8
b11m	90.0	80.0	37.0	5.7	95.0	82.2	39.0	5.1	100.0	100.0	54.6	5.1
b00z	22.2	31.0	13.7	4.1	66.6	75.9	44.3	5.0	100.0	100.0	51.8	5.4
fr	20.0	13.3	0.86	10.3	80.0	86.7	70.4	4.9	100.0	100.0	84.0	5.2
dlx	50.0	50.7	35.1	3.3	60.7	63.9	46.7	3.2	100.0	100.0	59.5	3.4
diffeq	100.0	100.0	95.4	50.0	100.0	100.0	98.6	59.9	100.0	100.0	98.7	61.7
am2910	95.1	87.3	84.1	99.1	98.2	90.3	88.7	88.1	100.0	100.0	93.1	87.0
prawn	87.2	70.6	63.9	144.2	91.3	73.5	68.9	131.8	96.0	77.1	72.8	183.3

Table 6.4. Comparison between a GA-based ATPG, a pseudo-deterministic ATPG and FATE.

have been selected from the well known ITC-99 benchmarks suite [198]. *b11m* is a modified version of *b11*, included in the same suite, created by introducing a delay on some paths to make it harder to be traversed. The original HDL descriptions of *b04*, *b10* and *b11m* contain a high number of nested conditions on signals and registers of different size. *ex1*, *b00*, *b00z* and *fr* contain conditional statements where one branch has probability $1 - \frac{1}{2^{32}}$ of being satisfied, while the other has probability $\frac{1}{2^{32}}$. Thus, they are very hard to be tested by a random ATPG. In particular, *ex1*, *b00* and *b00z* are internal benchmarks, while *fr* is a real industrial case, i.e., it is a module of a face recognition system. *diffeq* is a data-dominated benchmark for solving differential equations. Finally, *dlx* is the controller of a RISC processor, *am2910* is a microprogram address sequencer, and *prawn* is an 8-bit microprocessor.

At functional level the effectiveness of FATE has been evaluated by comparing it to a genetic algorithm-based high-level ATPG, as shown below. Stopping criterion are defined in term of the number and length of the generated test sequences.

Genetic Algorithm vs. EFSM-based ATPG

FATE has been compared with a genetic algorithm-based high-level ATPG [229], which outperforms a pure random-based ATPG but it is not aware about the EFSM structure, and with a pseudo-deterministic ATPG [227], which uses only the random walk mode to traverse the DUV state space. Table 6.4 reports the transition coverage (*TC%*), the statement coverage (*SC%*), the fault coverage (*FC%*), and the test generation time (*T (sec.)*), by using respectively the genetic algorithm-based ATPG (*GA-ATPG*), the pseudo-deterministic ATPG (*PD-ATPG*), and the proposed ATPG (*FATE*). It can be observed that FATE outperforms both the GA-ATPG and the PD-ATPG. The very low transition coverage achieved by the GA-ATPG for some benchmarks is due to the presence of a transition out-going from the initial state, whose enabling function has an infinitesimal probability of being traversed by randomly fixing the values of primary inputs. Such a problem is partially solved by the PD-ATPG which is aware about the enabling functions of the EFSM, and definitely solved by the learning/backjumping-based ATPG that reaches 100% transition and statement coverage for all benchmarks. Then, also the achieved fault coverage for all benchmarks is sensibly increased.

6.3 EFSM composition vs EFSM scheduling

The impact of EFSM composition on functional ATPG is analyzed by considering the scheduling algorithm proposed in Section 6.1.3 and the ATPG proposed in Section 6.2. In particular, the ATPG works in a three-step fashion to traverse the set of concurrent EFSMs that represent the DUV. During test pattern generation, the enabling functions of the transitions belonging to different EFSMs of the same DUV, which have been selected to be fired in the same simulation cycle, may involve the same set of primary inputs. In some cases, these transitions cannot be traversed concurrently because the values to be assigned to the primary inputs are conflicting. Then, one or more of them should be discarded and substituted with different transitions to remove the conflict. To avoid this situation, the scheduling algorithm is used to decide the priority of the EFSMs in fixing the value of the primary inputs.

The scheduling implements an aging mechanism which allows each EFSM to be eventually the highest-priority one. This avoids the ATPG to starve in a subset of the DUV state space. Still, working on a set of EFSMs, the ATPG could take very long time to traverse some hard-to-fire execution paths in the DUV state space, because at each simulation cycle it has only a local view of the DUV. Let us consider, for example, a DUV composed of two EFSMs $\mathcal{M}_1$, $\mathcal{M}_2$, sharing a primary input `I` and a signal `S` both of type 32-bit integer (Figure 6.10). The signal binds an output line of $\mathcal{M}_1$ to an input line of $\mathcal{M}_2$; $\mathcal{M}_1$ is sensitive to `I`, while $\mathcal{M}_2$ is sensitive to both `I` and `S`. Suppose that the enabling and update functions of a transition t_1 in $\mathcal{M}_1$ are, respectively, `I!=10` and `S<=I` (the operator `<=` represents a concurrent assignment between signals), while the enabling functions of transition t_2 and t_3 in $\mathcal{M}_2$ are, respectively, `I==100` and `S==0`.

Firing concurrently t_1 and t_2, or t_1 and t_3 is very difficult, since the ATPG only considers an EFSM at a time to fix the value of primary inputs. In fact,

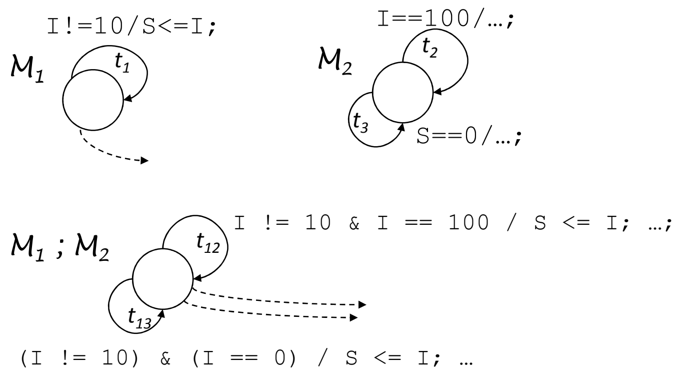

I!=10/S<=I;
$\mathcal{M}_1$
t_1
I==100/…;
$\mathcal{M}_2$
t_2
t_3
S==0/…;
$\mathcal{M}_1$; $\mathcal{M}_2$
t_{12}
I != 10 & I == 100 / S <= I; …;
t_{13}
(I != 10) & (I == 0) / S <= I; …

Fig. 6.10. Example.

if $\mathcal{M}_1$ has the highest priority, the constraint solver may choose among $2^{32} - 1$ possible values to satisfy `I!=10`, but only one of them (i.e., 100) allows to fire both t_1 and t_2. Similarly, only fixing `I` at 0, among $2^{32} - 1$ possible values, allows to fire concurrently t_1 and t_3. On the contrary, if $\mathcal{M}_2$ has the highest priority, firing concurrently t_2 and t_1 is straightforward, since the solver immediately chooses the value 100 for `I`, but firing concurrently t_3 and t_1 requires backjumping across different EFSMs, which is too time consuming when many EFSMs and many signals are involved. Thus, the probability of traversing the DUV execution path that concurrently involves t_1 and t_3 is almost 0, while traversing the execution path involving t_1 and t_2 becomes easy only when $\mathcal{M}_2$ gets the highest priority.

On the contrary, in case of EFSM composition, the ATPG has a global view of the unique EFSM that represents the DUV state space. Considering the previous example, according to the *Serial Composition* defined in Section 5.7.1, we can generate a single EFSM where t_1 and t_2 are merged in a transition t_{12} whose enabling function becomes `I!=10 & I==100`, while t_1 and t_3 are merged in a transition t_{13} whose enabling function becomes `I!=10 & I==0`. In this case, the constraint solver exploited by the ATPG immediately provides the correct value for `I`, when asked to solve either t_{12} or t_{13}.

Another limitation of relying on a EFSM scheduling policy to traverse the DUV is due to the fact that the ATPG, during the Random Walk phase, could be forced to discard many transitions at each simulation cycle. Let us consider the same EFSMs of the previous example, and suppose that the priority of $\mathcal{M}_2$ is higher than the priority of $\mathcal{M}_1$. When the ATPG selects t_2 as the transition to be fired during the random walk, it fixes `I` at 100. Then, it randomly selects a transition t_i to be fired in $\mathcal{M}_1$. However, it could happen that the enabling function of t_i imposes the condition `I!=100`. In this case, t_i must be discarded and substituted with a different transition. But again the new transition could require `I!=100`, and so on. The problem is due to the fact that the run-time composition of the enabling function of t_2 with the enabling function of the transition selected in $\mathcal{M}_1$ could generate an unsatisfiable constraint. This causes the ATPG to waste a lot of time calling the constraint solver till a satisfiable constraint is found.

DUT	PI	PO	P	FF	Gate	Trn	T (sec.)	BC
vr01	65	16	8	73	615	69	7.1	2168
ecc1	25	32	9	79	703	17	1.7	1022
ecc2	55	32	7	88	832	24	2.4	1302

Table 6.5. Benchmarks characteristics.

On the contrary, when two EFSMs are composed, all transitions generated by merging two enabling functions that originate an unsatisfiable constraint are statically discarded before running the ATPG. Thus, the composed EFSM contains only transitions whose enabling functions can be satisfied. This largely reduces the number of constraint solver calls.

As shown the EFSM composition approach is successful in supplying hard-to-fire execution paths to the ATPG and in pruning the DUV state space, but it suffers from one apparently unavoidable problem: the state explosion problem. This problem arise in systems composed of many concurrent processes. In general, the size of a composition may grow as the product of the sizes of the components. Because of this phenomenon, a design with a relatively small number of processes may have too many states and transitions for the proposed methodology to be directly useful. This risk is reduced by applying EFSM composition only to the EFSMs which involve not yet detected faults. This solution avoids both the state and transition growth and the drawbacks of the scheduling policy previously described. In any case, this is part of future works, while in this thesis the focus is showing that EFSM composition has a positive impact on functional test pattern generation.

6.3.1 Experimental Results

Experiments have been performed on industrial benchmarks provided by STMicroelectronics whose characteristics are reported in Table 6.5. Columns report the number of primary inputs (*PI*), primary outputs (*PO*), process statements (*P*)[5], flip-flops (*FF*) and gates (*Gate*), the time required to automatically generate the EFSM model (*T*), and the number of bit coverage faults (*BC*). *ecc1* and *ecc2* are modules which compute the error-correcting code of a sixteen-bit page of data, while *vr01* is a module of a face recognition system. All experiments have been carried out on an eight-processor Intel Xeon 2.8 MHz equipped with 8 GB RAM and 2.6.23 Linux kernel. CPU time has been computed with `time` command and `getrusage()` C-function by summing up *User* and *System* time.

Table 6.6 details the experimental results for the *vr01* benchmark. Columns report the composition degree (*Degree*), the transition coverage (*TC %*), the fault coverage (*FC %*), the test generation time in seconds (*T*), the number of generated test vectors (*TV*), the number of constraint solver invocations (*CSI*) and the time in seconds spent by the constraint solver (*CST*). In the table, each row corresponds to a different EFSM composition degree. The first row, degree 0, provides the results in the case no EFSM has been composed. In the second row,

[5] This represents the number of EFSMs generated for each benchmarks, one for each process.

Degree	**TC%**	**FC%**	**T (sec.)**	**TV**	**CSI**	**CST (sec.)**
0	80	54.10	228.384	15000	584935	182.7
1	80	54.10	201.345	15000	542523	162.1
2	80	54.10	187.340	15000	489031	159.3
3	93	71.80	134.918	15000	439413	103.4
4	93	71.80	132.332	15000	419321	100.0
5	93	71.80	103.431	15000	371391	73.1
6	100	98.90	13.926	256	16129	10.4
7	100	98.90	13.590	263	16089	10.1

Table 6.6. Experimental results: ***vr01***.

only two EFSMs have been composed among a total number of eight, and so on. The last row, degree 7, provides the results when all EFSMs have been composed in a single one. From degree 0 to degree 6, the ATPG exploits the scheduler, since more than one EFSM is used to model the system. When all EFSMs are composed, the scheduler is not necessary, so it is disabled. Results show that transition coverage, and consequently fault coverage, is in direct proportion to the composition degree. In particular, the scheduling methodology does not allow to achieve 100% transition coverage till 7 among 8 EFSMs (degree 6) are composed. The ATPG stops when either transition coverage is 100% or 15000 test vector have been generated. In addition, it is worth to note that *under the same transition coverage*, the constraint solver invocations, and the corresponding execution time, decrease by increasing the composition degree.

Table 6.7 and Table 6.8 report the experimental results for the *ecc1* and *ecc2* benchmarks. Only the cases corresponding to no composition and maximum composition degrees are reported. Concerning *ecc1*, the scheduling policy allows to achieve 100% transition coverage already at degree 0. However, when all EFSMs are composed (degree 8) we can note a great reduction (about 66%) of constraint solver invocations and a sensible decrement in terms of time spent by the constraint solver (about 26%). Concerning *ecc2*, the improvement given by EFSM composition is very relevant as in terms of transition and fault coverages as in terms of constraint solver calls.

The solver is invoked a lower number of times, with more complex constraints. The cut is in terms of time spent for solver activation, constraint submission and result returns.

The maximum number of constraint solver invocations can be determined in two general cases. In the case of N EFSM, whose enabling functions involve only primary inputs, or more in general assuming uniformly distributed the probability of being activated of each transition, the maximum number of constraint solver invocation is

$$CSI_{ub} = T \prod_{i=1}^{N} |M_i| \tag{6.1}$$

where M_i is the i-th EFSM, $|M_i|$ is the number of transitions in M_i, and $T = max_{i=1}^{N}|M_i|$.

If the enabling function of the EFSMs involves also register (and L counters) or more in general assuming not uniformly distributed the probability of being

Degree	TC%	FC%	T (sec	TV	CSI	CST (sec.)
0	100	87.7	4.852	884 / 1	11680	3.785
1	100	87.7	4.870	942 / 1	11332	3.945
2	100	87.7	4.548	865 / 1	9525	3.593
3	100	87.7	4.300	907 / 1	9138	3.397
4	100	87.7	4.184	968 / 1	8731	3.305
5	100	87.7	3.996	966 / 1	7767	3.317
6	100	87.7	4.002	998 / 1	7348	3.322
7	100	87.7	3.930	976 / 1	6175	3.223
8	100	87.7	3.612	973 / 1	5162	2.998

Table 6.7. Experimental results: ***ecc1*** benchmark.

Degree	TC%	FC%	T (sec	TV	CSI	CST (sec.)
0	71	32.1	312.32	15000	612042	291.30
1	71	32.1	301.42	15000	598125	278.32
2	84	65.2	287.10	15000	541001	256.15
3	84	65.2	210.10	15000	500143	198.35
4	84	65.2	198.12	15000	514313	187.10
5	100	97.4	18.24	312	49041	17.51
6	100	97.4	17.13	315	48984	16.98

Table 6.8. Experimental results: ***ecc2***.

activated of each transition, the maximum number of constraint solver invocation is

$$CSI_{ub} = max\{T, \Gamma\} \prod_{i=1}^{N} |M_i| \tag{6.2}$$

where C_j is the j-th counter of the description, $|C_j|$ is the size of the counter $((end - begin)/step)$, and $\Gamma = max_{j=1}^{L}|C_j|$.

6.4 Combined use of HLDD and EFSM

This section presents the *HLDD&EFSM Automatic Test Pattern Generator*. Traditionally, deterministic ATPGs exploit mathematical strategies tailored to allow a complete exploration of the DUV state space, thus covering corner cases, but they require a larger amount of timing and memory resources.

A possible way for limiting the resource consumption of deterministic ATPGs consists of implementing combined approaches that mix different state space exploration techniques and different computational models to address different DUVs and different areas of the same DUV. In this context, this Section presents a functional ATPG that relies on two paradigms: High-Level Decision Diagrams (HLDDs) and previously analyzed Extended Finite State Machines (EFSMs), and two ATPG engines which are based, respectively, on propagation and justification techniques across HLDDs, and Learning and Backjumping across EFSMs. Experimental results show that the joint use of such techniques allows improving the quality of generated test patterns and reduce the generation time.

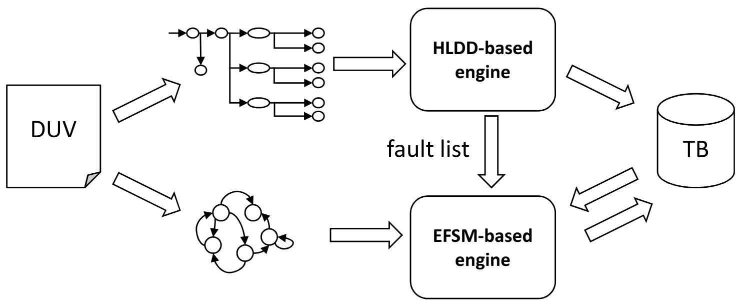


Fig. 6.11. The ATPG framework.

This Section is organized as follow. Section 6.4.1 describes the proposed ATPG framework. Section 6.4.2 summarizes the basis of the HLDD-based engine, while Section 6.4.3 proposes the integration between the FATE ATPG and the HLDD-based approach. Finally, in Section 6.4.4 reports experimental results.

6.4.1 The HLDD&EFSM ATPG Framework

Figure 6.11 shows the proposed ATPG framework that joins an HLDD-based engine with an EFSM-based engine. The framework measures the quality of generated test sequences according to the *bit coverage* fault model (see Chapter 7). The HLDD-based engine is first applied in order to detect untestable areas within the DUV. The tests are set up for variables and operations, which map to registers and functional units of the datapath of DUV, respectively. Subsequent to the setup, HLLD-based engine exploits propagation and justification techniques to traverse the HLDDs corresponding to the DUV. Test path activation constraints extracted during this process are satisfied by a constraint solver. Each variable is tested separately, one after another. Relying on HLDD models the engine is capable of finding all the consistent high-level test paths for a variable in DUV given enough time. If the test path constraints of all possible paths are non-satisfiable then the variable is considered to be untestable. Finally, when the HLDD-based engine have identified untestable areas (i.e. the set of untestable variables) of DUV, it will forward this list to the EFSM-based engine.

The EFSM-based engine exploits information provided by the HLDD exploration to traverse, via Backjumping, DUV areas that have not been explored. In particular, the EFSM-based engine addresses transitions of the EFSMs derived from the DUV that cannot be traversed by using the sequences generated during the HLDD exploration. The EFSM-based engine takes advantage of an external constraint solver to traverse hard-to-traverse transitions and it skips untestable areas marked by the HLDD-based engine. Therefore, the EFSM-based engine tries to increase the fault coverage without wasting time on untestable areas. This generally reflect on higher fault coverage and lower execution time, as reported in the experimental results.

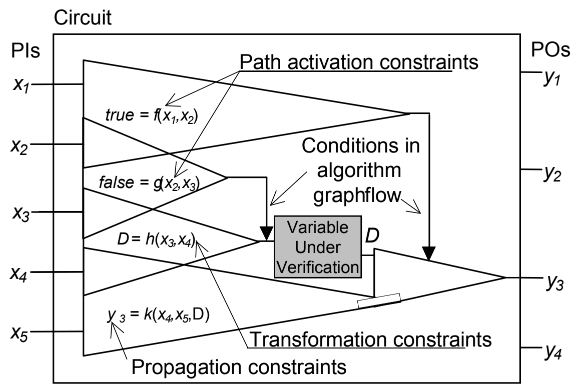


Fig. 6.12. HLDD-based path activation with constraint extraction.

6.4.2 The HLDD-based Engine

The test pattern generation algorithm implemented in the HLDD-based engine runs in two phases. During the first phase, a test path is activated to test a variable in the circuit and constraints required to activate it are extracted using HLDD models. At the second stage, the constraints are solved relying on the general purpose constraint solver ECLiPSe [215].

Test generation for the variable under validation (VUV) starts by setting the fault effect to VUV. In addition, a transformation constraint is created of current VUV. Subsequently, fault effect propagation follows. During the propagation stage we move forward in time (clock-cycles), fault effect is propagated towards primary outputs and path activation constraints are created whenever conditions in the control flow graph are traversed. Propagation is completed when we have obtained a state sequence transfering the fault effect to a primary output of the circuit.

After propagation, constraint justification begins. Justification moves backwards in time, starting from the clock-cycle, where propagation ended. During this process existing constraints are updated and additional path activation constraints are created. The process will be terminated when all the variables in all the constraints are primary inputs. Finally, constraints solving procedure is applied to the extracted constraints.

The test generation constraints considered in current paper can be divided into three categories: *Path Activation Constraints*, *Transformation Constraints* and *Propagation Constraints*. Path Activation Constraints correspond to the logic conditions in the control flow graph that have to be satisfied in order to perform propagation and value justification through the circuit. Transformation Constraints, in turn, reflect the value changes along the paths from the variable under validation to the primary inputs of the whole circuit. Finally, Propagation Constraints are necessary in order to calculate value transformation from the variable under validation until the primary output at the end of the currently activated test path. All three types of constraints can be represented by common data structures and manipulated by common procedures for creation, update, modeling and simulation.

Figure 6.12 explains the role of these constraints in test generation for a system variable on an example. In the Figure there are two path activation constraints: $true = f(x_1, x_2)$ and $false = g(x_2, x_3)$. The first one is necessary to propagate the value from the output of the module to the primary output y_3 of the circuit. The latter is required for justification of the variable under validation D. Both these constraints are extracted from the conditional nodes traversed in the HLDD for the FSM of the system during high-level path activation. In addition, the Figure presents a transformation constraint $D = h(x_3, x_4)$. This constraint represents the function for computing the value of the variable under validation based on the values of primary inputs of the circuit. Finally, there is a propagation constraint $y_3 = k(x_4, x_5, D)$ reflecting dependence of primary output y_3 on VUV and on primary inputs x_4 and x_5.

As it was mentioned above, the HLDD-based engine is applied in order to detect untestable areas within the DUV. The tests are set up for variables and operations, which map to registers and functional units of the datapath of DUV, respectively. Relying on HLDD models the engine is capable of finding all the consistent high-level test paths represented as a set of constraints similar to the ones in the example of Figure 6.12. If the test path constraints of all possible paths are non-satisfiable then the variable is considered to be untestable. In this paper a scheme is applied, where the HLDD-based engine marks untestable variables of DUV and will forward this list to the EFSM-based engine in order to improve efficiency and speed of overall test generation.

6.4.3 The EFSM-based Engine

The EFSM-based ATPG engine works in a three-step fashion traversing EFSMs that model the DUV. First, an off-line *Learning* phase is performed on the EFSMs to collect information about location of registers within enabling and update functions. Then, in the second phase, namely *HLDD-Test Simulation*, ETT transitions are traversed by using the test sequences generated by the HLDD-based engine. During this phase, information on state and transition reachability is also learned. Finally, in the third phase, the information collected in the previous steps are exploited to fire transitions that have not yet been activated, by means of a *Backjumping*-based approach.

In the *Backjumping* phase, the ATPG exploits the information provided by the enabling functions of the EFSMs to uniformly move across the transitions of each EFSM of the DUV. In this way, the capability of traversing HTT transitions is increased. On the contrary, a random ATPG tends to traverse only transitions whose enabling function presents a high probability of being satisfied by assigning random values to primary inputs. The EFSM-based engine changes to the Backjumping mode when it exhausts the test set provided by the HLDD-based engine. The Backjumping mode works as depicted in Section 6.2.2. Figure 6.13 summarize the HLDD&EFSM Backjumping strategy.

The Backjumping-based approach allows to fire transitions not traversed during the HLDD-Test Simulation phase, without requiring a complete stabilization of the EFSM.

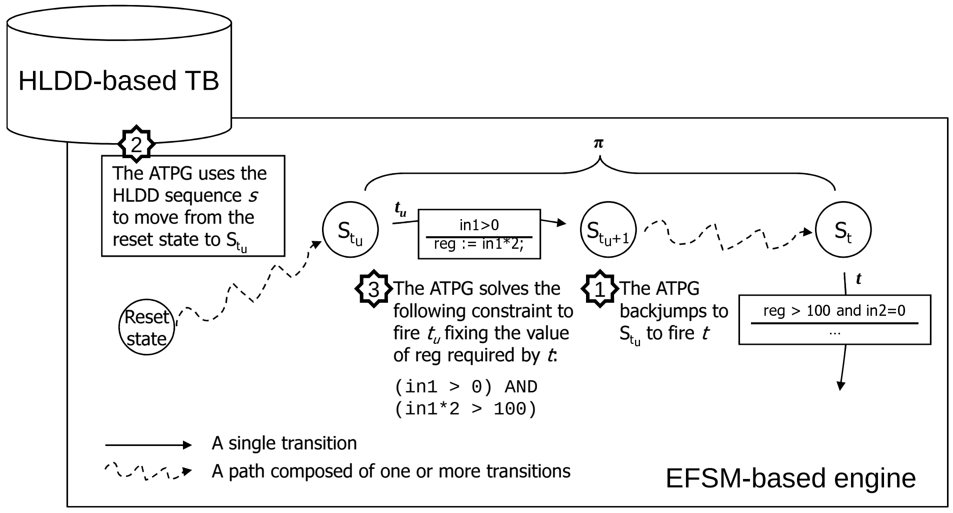

HLDD-based TB
2
The ATPG uses the HLDD sequence s to move from the reset state to S_{t_u}
π
t_u
S_{t_u}
in1>0
reg := in1*2;
S_{t_u+1}
S_t
t
Reset state
3
The ATPG solves the following constraint to fire t_u fixing the value of reg required by t:
(in1 > 0) AND (in1*2 > 100)
1
The ATPG backjumps to S_{t_u} to fire t
reg > 100 and in2=0
...
A single transition
A path composed of one or more transitions
EFSM-based engine

Fig. 6.13. The HLDD&EFSM Backjumping strategy.

DUV	PIs	POs	FFs	Gates	Trns.	GT (sec.)
ex1	66	32	130	10754	7	0.070
b00	66	64	99	1692	7	0.064
b04	13	8	66	650	20	0.168
b10	13	6	17	264	35	0.184
b11m	9	6	31	715	20	0.118
b00z	66	64	99	11874	9	0.084
fr	34	32	100	1475	10	0.182
dlx	29	31	25	232	28	0.212

Table 6.9. Benchmarks properties.

	PD-ATPG			**HLDD-EFSM-ATPG**		
DUV	**SC%**	**FC%**	**T (sec.)**	**SC%**	**FC%**	**T (sec.)**
ex1	92.9	80.3	2.9	100.0	96.0	2.6
b00	87.0	48.7	2.6	100.0	52.5	2.4
b04	95.0	99.0	8.7	100.0	99.0	7.3
b10	69.7	93.0	5.7	100.0	94.0	5.2
b11m	82.2	39.0	5.1	100.0	54.6	3.8
b00z	75.9	44.3	5.0	100.0	51.8	4.0
fr	86.7	70.4	4.9	100.0	84.0	4.1
dlx	63.9	46.7	3.2	100.0	59.5	2.7

Table 6.10. Comparison between a pseudo-deterministic ATPG and the HLDD&EFSM-based approach.

6.4.4 Experimental Results

The efficiency of the proposed ATPG framework has been evaluated by using the benchmarks described in Table 6.9, where columns report the number of primary inputs (*PIs*), primary outputs (*POs*), flip-flops (*FFs*) and gates (*Gates*). Such benchmarks have been selected because they present different characteristics which allow us to analyze and confirm the effectiveness of the proposed approach. *b04*, *b09* have been selected from the well known ITC-99 benchmarks suite [198]. *b11m* is a modified version of *b11*, included in the same suite, created by introducing a delay on some paths to make it harder to be traversed. The original HDL descriptions of *b04*, *b09* and *b11m* contain a high number of nested conditions on signals and registers of different size. *ex1*, *b00*, *b00z* and *fr* contain conditional statements where one branch has probability $1 - \frac{1}{2^{32}}$ of being satisfied, while the other has probability $\frac{1}{2^{32}}$. Thus, they are very hard to be tested by a random ATPG. In particular, *ex1*, *b00* and *b00z* are internal benchmarks, while *fr* is a real industrial case, i.e., it is a module of a face recognition system. Finally, *dlx* is the controller of the well known RISC processor.

The effectiveness of the proposed ATPG framework has been evaluated by comparing it to a pseudo-deterministic ATPG [227], which outperforms pure random and genetic algorithm-based high-level ATPGs. It uses a constraint solver to traverse the DUV state space but it does not exploit neither propagation and justification strategies on HLDDs nor backjumping on EFSMs. The Table 6.10 reports such a comparison. In particular, the Table shows the time required to automatically generate the HLDDs and the corresponding S^{2}EFSMs (*GT (sec.)*), the statement coverage (*SC%*), the fault coverage (*FC%*) and the test generation time (*T (sec.)*), by using respectively the pseudo-deterministic ATPG (*PD-APTG*), and the combination of the HLDD and EFSM-based ATPGs proposed in this work (*HLDD-EFSM-ATPG*).

It can be observed that the HLDD-EFSM-ATPG outperforms the PD-ATPG. The low statement coverage achieved by the PD-ATPG for some benchmarks is due to the presence of hard-to-traverse transition, whose enabling function has an infinitesimal probability of being traversed without backtracking or justification-based strategies. Such a problem is solved by the HLDD-EFSM-ATPG which exploits learning/backjumping technique. Indeed, the HLDD-EFSM-ATPG reaches 100% statement coverage for all benchmarks. Moreover, the fault coverage is sensibly increased for all benchmarks by adopting the HLDD-EFSM-ATPG. Finally, the test generation time is reduced thanks to the capability of the HLDD-based engine to identify untestable areas, which are skipped during the subsequent test generation phase performed by the EFSM-based engine.

6.5 Published contributions

The work presented in this Chapter has lead to the following publications [230, 231, 232, 202].

G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli
A Pseudo-Deterministic Functional ATPG based on EFSM Traversing

In Proceeding of "IEEE International Workshop on Microprocessor Test and Verification (MTV)"
Austin, TX, USA, November 3-4, 2005, pp. 70-75

Giuseppe Di Guglielmo, Franco Fummi, Cristina Marconcini, and Graziano Pravadelli
FATE: a Functional ATPG to Traverse unstabilized EFSMs
In Proceeding of "IEEE European Test Symposium (ETS'06)"
Southampton, UK, May 21-24, 2006, pp. 179-184

D. Bresolin, G. Di Guglielmo, F. Fummi, G. Pravadelli and T. Villa
The impact of EFSM Composition on Functional ATPG
In the Proceedings of "12th IEEE Symposium on Design and Diagnostics of Electronic Systems (DDECS'09)"
Liberec, Czech Republic, April 15-17, 2009

Giuseppe Di Guglielmo, Franco Fummi, Cristina Marconcini, and Graziano Pravadelli
Improving high-level and gate-level testing with FATE: A functional automatic test pattern generator traversing unstabilised extended FSM
Computers & Digital Techniques, IET
Volume 1, Issue 3, 2007, pp. 187-196

In this paper some functional ATPG framework has been presented. All proposed ATPGs exploit a particular kind of EFSM which has been theoretically showed to allow a more uniform traversing of the DUV state space. Determinism is obtained by interfacing with a tool that adopts formal methods to solve the conditions of the enabling functions. In particular, the ATPG has been interfaced with both a CLP-based constraint solver and a SAT-solver. Experimental results show that the first outperforms the second. Moreover, the effectiveness of the proposed ATPG compared with a genetic-based ATPG is evident. It greatly benefits from the fact that, by using the EFSM model, all conditional statements included in the DUV are under its control. This approach has been extended with a deterministic ATPG, namely FATE. The deterministic functional ATPG relies on Learning, CLP and backjumping to efficiently explore the DUV state space. The adoption of the EFSM joint to the Learning/Backjumping-based mechanism allows to accurately address HTT transitions, whose enabling function depends on registers, without requiring EFSM stabilization. Then, the impact of bounded EFSM composition on a functional ATPG has been analyzed. In particular, the presented work analyzed the performance of the ATPG working on a set of concurrent EFSM, by mean of Scheduling algorithm, and on a single EFSM, representing the same DUV. Finally, an high-level ATPG framework which exploits different computational models has been shown. The integration of the two engines (HLDD&EFSM ATPG) allows improving fault coverage, while testing time is reduced.

7 Methodology: Functional fault model and testbench quality

Defects of hardware system design can happen anywhere on die, on one or multiple layers, packages, boards etc. They can consume arbitrary areas and can have various electrical properties. Quality of designing circuits or systems may be evaluated by *defect coverage estimation* but, generally, it is not possible to measure this parameter directly because defects have to be modeled at the higher abstracted level as *faults*.

Defects are manifested in different manners: by changing a logical value on a node of a device under test, increasing the steady state supply current, changing timing specifications or discrepancy in other circuit parameters. The main goal of a fault modeling is to reduce the infinite set of possible defect behaviors into a finite set of faults. Fault models are used in test pattern generation to measure the test quality and bridge the gap between the physical reality and mathematical abstracts. Faults and fault models should mirror real defects in the circuit and system design, therefore defects have to be modeled in the right way and faults have to be classified. Moreover, modeling of faults is closely related to the modeling of the circuit and the corresponding degree or level of abstraction.

To increase the speed of fault coverage evaluation, high-level (functional or behavioral) fault models have been developed. High-level faults represent the effect of physical defects on the operation of a system represented on the functional or behavioral level. A high-level fault model can be considered good if the tests generated using this model provide a high coverage of stuck-at-faults or physical defects.

The main idea of the high-level fault modeling is to obtain from the high-level description of the system an incorrect version by introducing a fault into the system representation. This approach is called *model perturbation*. The model can be perturbed in certain ways, e.g. by truth-table modification, micro-operation modification etc. In some or other way, this idea is implemented in different high-level fault models, developed to change the behavior of the system by implementing and injecting some *saboteurs* in the HDL description.

Many high-level fault models [233, 64, 234, 235, 236, 237, 238, 150] and coverage metrics [239, 240] have been proposed in the past to guide the generation of functional tests. In some papers [234, 235, 236, 237], the authors highlight how the adopted high-level fault model is strictly correlated to the gate-level stuck-at fault model. The degree of correlation is measured as the percentage of gate-level stuck-at faults that are detected by fault simulating the test sequences generated at functional level.

Alternatively, in the design functional validation, high-level fault model based techniques are used to address the quality evaluation of the model checking process [241, 242]. In particular Assertion Based Verification methodologies requires effective automatic test pattern generator for investigating the capability of properties to identify functional perturbations of the design implementation [243].

Similarly to hardware, many techniques for analysis, testing and validation have been proposed to assist the software development process [244, 245, 246]. In general, there are three techniques to design test cases: *functional*, *structural* and *error* or *perturbation based*. These techniques are complementary to each other as they exercise different characteristics of the software being tested [247]; they are also the source of several testing criteria. One way to evaluate the quality of the test set is to use coverage measures derived from the required elements of a given testing criteria [248].

Mutation Analysis is a perturbation-based technique for software unit testing. It is used both to evaluate and generate the quality of a test set. The basic principle is to *mutate*, i.e. change, the program and then run the test suite on the *mutant*, i.e. the changed programs. If the test suite *kills*, i.e. detects, all mutants it provides confidence that the test suite covers the program sufficiently. However, if there remain mutants *live*, i.e. undetected, this can be seen as an indicator that the test suite might be inadequate to test the software. A *mutation operator* is the rule used to mutate a program. Figure 7.6 shows an example of a mutation where an arithmetic operator is exchanged for another. In this example the $-$ operator has been substituted for other arithmetic operators, likewise every behavioral construct in the program can have many associated mutations.

The fundamental hypothesis of mutation analysis is that if the program contains live mutants then the program also could contain real bugs (or coding mistakes) that cannot be found by the existing tests. If the testing is improved so as to kill live mutants, then these same tests can expose the vast majority of previously unknown bugs in the original program. There were several mutation analysis systems developed in the 1980s (e.g. Mothra for the FORTRAN language [246]), and experiments confirmed this hypothesis to be valid. In fact, experiments demonstrated this to be the most comprehensive measurement of quality of the tests for a given program. In the 1990s, researchers provided theoretical justifications explaining why mutation analysis is so effective [96, 97, 98].

Mutation analysis assumes the *competent programmer* hypothesis to be valid [249]. The design is considered to be largely correct, i.e. the majority of the code is assumed to not contain bugs. This is important because the mutation analysis assesses the ability of the verification environment to measure the quality of the current design implementation. When mutations are introduced, they take the design slightly out of specification.

Mutation testing uses mutation analysis to judge the adequacy of test data. The test data are judged adequate only if each mutant is either functionally equivalent to the original program or computes output different from the original program

```
function inject_fault_bit
  (object: bit; fault_code: integer; start_s0: integer;
   end_s0: integer; start_s1: integer; end_s1: integer)
   return bit is
variable res: bit; begin
  if (fault_code = start_s0) then
    res := '0';
  elsif (fault_code = start_s1) then
    res := '1';
  else
    res := object;
  end if;
  return res;
end;
```

Fig. 7.1. Saboteur VHDL function for bit operands.

on the test data. Inadequacy of the test data implies that certain faults can be introduced into the code and go undetected by the test data.

Therefore, validation of hardware design, through simulation and high-level fault models, and Mutation Analysis, originally proposed for software verification and testing, presents many analogies. This Chapter summarize the adopted *bit-coverage* fault model (Section 7.1) and the Mutation Analysis, oriented to functional validation (Section 7.2).

Bit-coverage and mutation operators, considered as functional fault model, can be exploited into the ATPG-fault-simulation engine proposed in this Chapter 8.

7.1 The Bit Coverage Fault Model

The first high-level fault model adopted in the proposed methodology is the *bit coverage* [250]. During fault simulation or test pattern generation, at most one bit coverage fault can be activated according to the following failure specification:

Bit failures. Each occurrence of variables, constants, signals or ports is considered as a vector of bits. Each bit can be stuck-at zero or stuck-at one.

Condition failures. Each condition can be stuck-at true or stuck-at false, thus removing some execution paths in the faulty representation.

Bit coverage is chosen since it is related to design errors [237, 238, 150] and it unifies into a single metrics the well known metrics [251] concerning statements, branches and conditions coverage. In addition, paths needed to activate and propagate faults from inputs to outputs of the DUV are also covered.

- ***Statement coverage.*** Any statement manipulates at least one variable or signal. The bit failures are injected into all variables and signals on the left hand and right hand side of each assignment. Thus at least one test vector is generated for all statements. To reduce the proposed fault model to statement

```
function inject_fault_bit_vector
  (object: bit_vector; fault_code: integer;
   start_s0: integer; end_s0: integer;
   start_s1: integer; end_s1: integer)
   return bit_vector is
variable left       : integer; variable right     : integer;
variable index      : integer; variable index_s0  : integer;
variable index_s1   : integer; variable length    : integer;
variable res_downto : bit_vector(end_s0 - start_s0 downto 0);
variable res_to     : bit_vector(0 to end_s0 - start_s0); begin
  left := object'left;
  right := object'right;
  length:= end_s0 - start_s0;
  if (left > right) then         -- vector range is downto
    res_downto := object;
    for index in lenght downto 0 loop
      index_s0 := index + start_s0;
      index_s1 := index + start_s1;
      res_downto(index) := inject_fault_bit(
                             object(index), fault_code,
                             index_s0, index_s0,
                             index_s1, index_s1);
    end loop;
    return res_downto;
  else                           -- vector range is to
    res_to := object;
    for index in 0 to lenght loop
      index_s0 := index + start_s0;
      index_s1 := index + start_s1;
      res_to(index) := inject_fault_bit(
                         object(index), fault_code,
                         index_s0, index_s0,
                         index_s1, index_s1);
    end loop;
    return res_to;
  end if;
end;
```

Fig. 7.2. Saboteur VHDL function for bit vector operands.

coverage it is thus sufficient to inject only one bit failure into one of the variables (signals) composing a statement. In conclusion, the bit coverage metric induces an ATPG to produce a larger number of test patterns with respect to statement coverage and it guarantees to cover all statements.

- ***Branch coverage.*** The branch coverage metric implies the identification of patterns which verify the execution of both the true and false (if present) paths of each branch. Modeling of the condition failures implies the identification of patterns which differentiate the true behavior of a branch from the false behavior, and vice versa. This differentiation is performed by making stuck

```
function inject_fault_integer
  (object: integer; fault_code: integer; start_s0: integer;
   end_s0: integer; start_s1: integer; end_s1: integer)
   return integer is
variable length : integer; variable res: integer; begin
  length := end_s0 - start_s0 + 1;
  res := to_int(inject_fault_bit_vector(
             to_bit_vector(object, length),
             fault_code, start_s0, end_s0,
             start_s1, end_s1));
  return res;
end;
```

Fig. 7.3. Saboteur VHDL function for integer operands.

```
 IF (data_in > rmax) THEN
   ack <= '1';


 IF (inject_fault_bool(
    inject_fault_integer(data_in, fault, 1438, 1445, 1446, 1453) >
      inject_fault_integer(rmax, fault, 1454, 1461, 1462, 1469),
      fault, 1470, 1470, 1471, 1471)) THEN
        ack <= inject_fault_bit('1', fault, 1472, 1472, 1473,1473);
```

Fig. 7.4. Fault-free and generated faulty VHDL code.

at true (false) the branch condition and by finding patterns executing the false (true) branch, thus executing both paths. In conclusion, the proposed bit coverage metric includes the branch-coverage metric.

- ***Condition coverage.*** The proposed fault model includes condition failures which make stuck at true or stuck at false any condition disregarding the stuck at values of its components.
- ***Path coverage.*** The verification of all paths of a SystemC method can be a very complex task owing to the possible exponential grow of the number of paths. The proposed fault model selects a finite subset of all paths to be covered. The subset of covered paths is composed of all paths that are examined to activate and propagate the injected faults from the inputs to the outputs of the design module within a given time limit.
- ***Block coverage.*** In [252] statement coverage has been extended by partitioning the code in blocks and by activating these blocks a fixed number of times. This block coverage criterion is included in the proposed fault model in the case the number of bit faults included in a block is larger than the number of times the block is activated. In fact, a test pattern is generated for each bit fault, thus the block including the fault is activated when the fault is detected.

Finally, bit coverage shows a high correlation between stuck-at faults at different levels of abstraction [237].

7.1.1 Bit-coverage Fault Injection

Bit coverage faults (faults in the following) are automatically injected into the DUV by using *UniVR Fault Injection Tool*. The UniVR Fault Injection Tool tool presented in [253] has been further enhanced and extended by exploiting the *HDL Intermediate Format* and the corresponding *HIF Suite* described in Chapter 4.

Fault injection is performed by inserting saboteurs into the HIF representation of the DUV. Generally, a saboteur is a special component added to the original model [76]. The mission of this component is to alter the value, or timing characteristics, of one or more signals when a fault is injected. The component remains inactive during the normal operation of the system. In this case, saboteurs are functions which can supply the correct or faulty value of the corresponding object depending on the value of a control signal. Every occurrence of signals, variables and constants and every condition of the functional level description is replaced by an opportune bit coverage saboteur.

We define a saboteur for every language type, i.e., bit, integer, standard logic, boolean, etc. Faults are enumerated and a integer-type port, named `fault` is added to the DUV. The `fault` port drives all control signals. Figure 7.1, Figure 7.7 and Figure 7.3 show respectively the saboteur function for VHDL `bit`, `bit_vector` and `integer` operands. Saboteurs for other data type are defined in a similar way referring to the bit case. Changing the definition of the saboteur for bit, the behavior of saboteurs for the other data types changes accordingly. The first parameter (`object`) of the saboteur is the target of the fault, `fault_code` is the value of the `fault` port and finally `start_s0-1` and `end_s0-1` show the range for `fault_code` to activate the stuck-at 0-1 on the target object.

The fault injection process generates a unique faulty description of the design that includes all bit coverage faults. Figure 7.4 shows an example of fault-free and faulty VHDL descriptions by using bit coverage saboteurs. It illustrates how the faults are recursively inserted in complex statements as an `if-then-else` statement. For example, to activate the fault stuck-at 0 on the third bit of the integer signal `rmax`, the signal `fault` must be set to 1456, since the range for faults stuck-at 0 on `rmax` is from 1454 to 1461. On the other hand, to activate the fault stuck-at true on the `if-then-else` condition the signal `fault` must be set to 1473.

After the fault injection phase, the fault list is depurated from redundant faults by using the methodology reported in [254].

7.2 The Mutation-based Fault Model

Assuming D, the design under validation, each alternate program, M, known as a mutant of D, is formed by modifying a single statement of D according to some predefined rules. Figure 7.5 gives an example of a twelve-mutant operators inherited from unit testing. Each of the mutant statements is executed at a time. The original design plus the mutant programs are collectively known as the *design neighborhood*, N, of D. Mutation analysis is a method of evaluating the adequacy of a set of test vectors for a system description. Informally, test vectors are considered

Type	**Description**
AOR	Arithmetic Operator Replacement
ABS	Absolute Value Insertion
CR	Constant Replacement
CVR	Constant for Variable Replacement
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
ODR	Operation for Delay Replacement
OSR	Operation for Skip Replacement
VCR	Variable for Constant Replacement
VR	Variable Replacement
UOI	Unary Operator Insertion
BOR	Bit Operator Replacement

Fig. 7.5. Software mutation operators set.

mutation-adequate for a design if they can distinguish the design from designs that differ from it by small syntactic changes. Test vectors are then measured by determining how many of the mutant designs produce incorrect output when executed. Each live mutant is executed with the test vectors and when a mutant produces incorrect output on a test vector, that mutant is said to be killed by that test vector and is not executed against subsequent test vectors. This shows that the current test vectors set is able to detect the faults represented by the dead mutants.

Two designs are functionally equivalent if they always produce the same output on every input. Some mutants are functionally equivalent to the original design and cannot be killed. Despite recent work in automating detection of equivalent mutants, this is usually done manually and is one of the greatest expenses of current mutation systems. A mutation score of a test set is the percentage of nonequivalent mutants that are killed by the test set. More formally, if a design has M mutants, E of which are equivalent, and a test set T kills K mutants, the mutation score is defined to be:

$$MS(D,T) = \frac{K}{(M-E)}.$$

A test set is *mutation-adequate* if its score is 100 percent (all nonequivalent mutants were killed). In practice, vectors sets that score above 95 percent on a mutation system tend to be difficult to create, but are effective at detecting faults.

Whereas in ordinary mutation analysis, which is often called *Strong Mutation Analysis*, a mutant M is considered killed by a test vector t only if the output $M(t)$ is different than $D(t)$, in *Weak Mutation Analysis*, a mutant is considered to be killed by t if the design state of M after some execution of the mutated statement is different than the program state of D at that same point.

It stands to reason that Weak Mutation adequacy is easier to check and probably less costly to use than Strong Mutation adequacy, since it is not necessary to execute a mutant completely, but only to the first point at which its internal state differs from that of the original program. In fact, because the states of a mutant

Original version	**Mutant**
$x\ :=\ z\ -\ 5;$	$x\ :=\ z\ +\ 5;$
	$x\ :=\ z\ *\ 5;$
	$x\ :=\ z\ /\ 5;$
	$x\ :=\ z\ MOD\ 5;$
	$x\ :=\ z\ .LEFTOP.\ 5;$
	$x\ :=\ z\ .RIGHTOP.\ 5;$

Fig. 7.6. Mutations (b) of an arithmetic expression (a).

and the original program can only diverge at the mutated statement, a more efficient approach is to execute the original program alone, saving each mutant of that statement in that state only. This speeds up program execution and saves memory resources.

7.2.1 Mutation Analysis using Program Schemata

The essence of this method lies in the creation of a specially parameterized program called the *metamutant*. Derived from D, the metamutant is compiled and runs at compiled-speeds. While running, the metamutant can be instantiated to function as any of the alternate programs found in N. Thus the metamutant is a type of *program schema*. A program schema is a template. As defined by Baruch and Katz [255], a schema syntactically resembles a program, but contains free identifiers, called abstract entities, in place of some program variables, datatypes identifiers, constants and program statements. A schema can be *instantiated* by providing appropriate substitutions for abstract entities. To explain how a metamutant is able to represent the functionality of a collection of mutants, a closer look at mutation analysis is necessary. Recall that for a program D, each mutant of D is formed as a result of a single modification to some statement of D. Each mutant in N differs from the original description in only one mutated statement. How these statements are altered is dictated by the modification rules used.

Consider the arithmetic operator replacement rule (AOR) which states that each occurrence of an arithmetic operator is replaced by each of the other possible arithmetic operators. Each operator is also replaced by special operators LEFTOP, RIGHTOP and SKIP. Where LEFTOP returns the left operand (the right is ignored) and RIGHTOP returns the right operand.

Applying this rule to the assignment statement of Figure 7.6(a) yields the six mutations in (b). These mutations could be *generically* represented as $x\ :=\ z\ ArithOp\ 5$; where $ArithOp$ is an abstract entity. The generic representation above can be recast as a syntactically valid statement $x\ :=\ AOrr(\ z, 5, 78)$; where the $AOrr$ function performs one arithmetic operation. A (simplified) implementation of the $AOrr$ function is given in Figure 7.7. The third argument, *78*, is simply a control value identifying where in the program the function is invoked. At run-time, a special global parameter selects which of the arithmetic operations to perform. A statement that has been changed to reflect such a generic form is said to have been metamutaded. A metamutation is a syntactically valid change

```
function AOrr
  (left_op: integer; right_op: integer; mut_index: integer)
  return integer is
begin
  case mut_index is
    when aoADD => return left_op + right_op;
    when aoSUB => return left_op - right_op;
    when aoMULT => return left_op * right_op;
    when aoDIV => return left_op / right_op;
    when aoMOD => return left_op MOD right_op;
    when aoLEFT => return left_op;
    when aoRIGHT => return right_op;
    others =>
      assert ( false )
        report "AOrr␣case␣out␣of␣range"
        severity warning;
      return 0;
  end case;
end;
```

Fig. 7.7. Simplified version of Arithmetic Operation function.

that embodies other changes and virtually all mutations can be represented by metamutations.

While generating the metamutant of D, a list of mutant descriptors is produced. This list details the alternative operations to be used at each change point in the program. Using this list, the metamutant can be dynamically instantiated to function as any of the mutants of D. A driver procedure invokes the metamutant and directs which mutants are to be instantiated. The driver takes care of such administrative matters as managing the test cases, handling exceptions, comparing mutant output to the original program output, and recording results. A common driver procedure is used for all metamutants.

The metamutant is not unduly greater in size than the original. Consequently compilation time is not a significant factor in measuring performance.

7.2.2 Mutation operators for HDL descriptions

Mutants generation is realized with a set of mutation operators. These operators are specific to HDL and allow to cover all paths, conditions, limit values and perturbation areas relevant to a given design. A big challenge is to avoid equivalent mutants, which increase the test time, reduce the usefulness of the mutation score, and complicate the designer's task. Some Mothra operator systematically produce equivalent mutants. For instance ABS replaces expressions and sub-expressions by their absolute values. Absolute values are identical to original values for unsigned vectors.

The HDL operators proposed in Figure 7.8 were designed to represent the most common errors that a designer might take. These operators model many types of design faults that can appear in a HDL description. For instance, a frequently

Type	Description
CLR	Constant Limit replacement. Test upper and lower boundaries of different registers. The same process is made for variables. The perturbation area is also simulated.
CNR	Comparable Array Name Replacement. Each array is replaced by one of the same type, but with a different name, present in the VHDL description.
CSR	Constant for Scalar Variable Replacement. Each variable and signal is replaced by a constant of the same type.
GRP	Generic Replacement. Simulate a misconnection. This operator is most interesting for structural designs.
SUR	Signed/Unsigned Replacement. Test signed and unsigned binary vectors.
VSAR	Variable and Signal Replacement. Test the bad variables and signals assignment resulting in synchronization errors and bad action sequences.
SAR	Signal Assignment Replacement. Test assignment to incorrect register.
SVIR	Signal and Variable Initialization Replacement. Generates incorrect initialization.
SSR	State Sequence Replacement. Modify the state sequence in a state machine.
LCR	Logical Constant Replacement. Each logical operator is successively replaced by others.
COR	Conditional Operator Replacement. Substitute all possible conditions.
LER	Level Replacement. Modify the sensitivity (high or low)

Fig. 7.8. HDL mutation operators set.

used method for the validation of designs is to test limit values. In this case, a perturbation area is simulated by increasing slightly the constants, variables or signals responsible of the boundary conditions.

Another example is the incorrect writing of data integers. To detect such errors, the operators assign the content of all arrays to all others array of the same type.

In general, the functionality is controlled by state machines. Some errors can produce an infinite loop or change the original state sequence of the machine. Mutations applied to state machines also test the branch coverage, modifying each condition by others.

8

Methodology: Fault simulation

Verification via fault injection and fault simulation is a widely adopted technique to evaluate the correctness of a design implementation. However, the complexity of industrial designs and the huge number of faults that must be injected into them require efficient fault simulators, in order to make verification via fault simulation an affordable task. To optimize fault simulation performances, some parallelization techniques have been proposed at gate level. On the contrary, they have not been fully exploited at RTL, where functional fault models, instead of gate-level ones, are considered. Thus, this Chapter analyzes the impact of such parallelization techniques on functional faults. In particular, possible issues are presented together with optimizations that can be implemented to speed up the simulation.

This Chapter, after an introduction of the problem (Section 8.1), presents the problems arising by applying parallel fault simulation to functional faults (Section 8.2). Section 8.3 explains some proposed optimizations to further increase parallel simulation performance. Finally, Section 8.4 reports experimental results and a comparisons between high-level simulation and bit-level parallel simulation.

8.1 Introduction

Fault simulation is used in test generation to determine the fault coverage of a test set. Given a circuit, a set of faults in the circuit, and a set of tests (the input stimuli), fault simulation is typically used to decide which faults can be detected by at least one test. New test vectors are then generated trying to cover the undetected faults, and fault simulation is performed again to determine the fitness of the new vectors. The cycle of test generation and fault simulation is repeated until a satisfactory test set is obtained [256]. In complex designs, the number of faults is very large, thereby stressing the importance of implementing fast and efficient fault simulators. Traditionally, *serial fault simulation* is the simplest method of simulating faults. It consists of transforming the model of the fault-free circuit N so that it models the circuit N_f created by fault f. Then N_f is simulated. The entire process is repeated for each fault of interest. Thus faults are simulated one at a time [257]. Even if this can be applied to industrial circuits, it is possible to explore the potentialities of parallel simulation to further increase performances. This possibility has led to the development of special purpose computer architectures for fault simulation [258, 259, 260] at gate level. These accelerators are designed to take into account the concurrencies that exist in fault simulation. Indeed, besides serial fault simulation, there are, at least, three further types of fault simulation: parallel [261], concurrent [262] and deductive [263, 264]. These techniques differ from serial method in two fundamental aspects:

- they determine the behavior of the design in presence of faults without explicitly changing the model of the design;
- they are capable of simultaneously simulating a set of faults.

In *parallel fault simulation*, the fault-free circuit and a certain number of faulty circuits are simultaneously simulated. The subset of faulty circuits is opportunely selected to avoid interactions among faults. *Concurrent fault simulation* is based on the observation that, generally, during a simulation session, the majority of signals/variables values in faulty circuits equal the values of corresponding signals/variables in the fault-free circuit. Finally, the *deductive fault simulation* simulates the fault-free circuit to determine all of the good values on the inputs and outputs. Then, using these values, the list of all faults that cause changes in the output are "deduced" from the input values and the gate function. These techniques may also be implemented in conjunction with distributed and parallel processing [265].

When applying such techniques, fault simulation is generally performed on gate-level circuits, and failures are approximated by using classical gate-level fault models (e.g. stuck-at, bridge, etc.). In particular, several algorithms have been developed, in the past, to address efficient gate-level fault simulation under the stuck-at fault model by using parallel [266], deductive [263], concurrent [267], parallel-valued list [268], differential [269], and parallel-differential [270] approaches.

Indeed, fault parallelization can be obtained also at functional (RTL) level, by running multiple instances of the design [271] on a parallel architecture-based machine. However, at RTL it is not possible to directly exploit word-level vectorization, as done at gate level. Nevertheless, gate-level simulation is definitely slower than RTL simulation to the additional information modeled at the gate level. Such conflicting considerations give rise to two main questions:

1. Can the *gate-level* parallelization produce better performance results than serial *functional* simulation?
2. If yes, are there some cases in which this advantage is higher than in other cases?

This Chapter tries to answer the previous questions. In particular, it discusses how to benefit from the use of *gate-level parallel simulation techniques* for simulating *functional-level faults*, along with the issues implied by porting parallel simulation from gate level to functional level. In this context, the rest of the Section refers to the concept of simulating functional faults by exploiting parallelization techniques on a gate-level netlist with the expression *parallel functional fault simulation.*

Figure 8.1 shows the implemented framework to compare serial RTL simulation with parallel gate-level simulation on the same set of functional faults. The RTL design under validation (DUV) is instrumented by injecting functional faults and

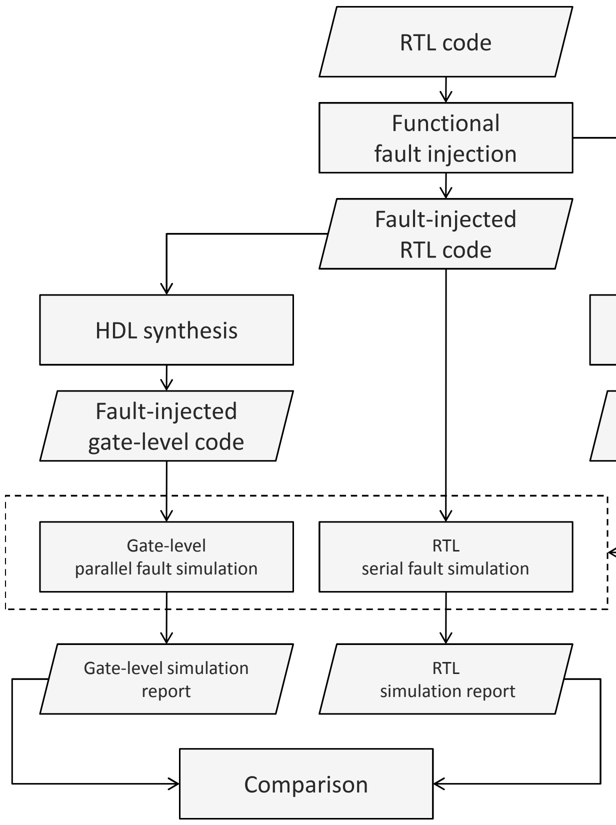

RTL code
Functional fault injection
Fault-injected RTL code
HDL synthesis
Functional ATPG
Fault-injected gate-level code
Testcases
Gate-level parallel fault simulation
RTL serial fault simulation
Gate-level simulation report
RTL simulation report
Comparison

Fig. 8.1. Simulation framework.

a set of testcases are generated by exploiting the functional FATE ATPG proposed in Section 6.2.2. Then, the instrumented DUV is synthesized to obtain a gate-level netlist. Finally, the testcases are simulated on both the RTL faulty designs, by exploiting a serial simulation engine, and the gate-level faulty netlist, by exploiting a parallel simulation engine. The fault propagation results is definitely the same, but the simulation time may sensibly vary as reported in the experimental results section.

8.2 Open issues

The parallel functional fault simulation is a complex task with many tradeoffs between design decisions. The main and most important ones are the followings:

- the adopted fault model, which shall be behavioral but synthesizable;
- the choice of which kind of functional fault parallelization shall be used;
- the parallel faults management engine and its integration with the parallel netlist;

- the simulation kernel and the adopted simulation language;
- the possible infinite simulation loop due to flipping bits.

Each of these problems is addressed in the next sections.

8.2.1 The functional fault model

In the literature there is a great variety of different fault models, and each of them has been developed to focus on particular design problems and design levels [56, 272]. When approaching the parallel simulation at functional level, not all the fault models are suitable. In fact the parallel simulation is implemented at gate level, and hence, the injected design must be synthesizable. This implies that the chosen fault model shall not introduce non-synthesizable constructs. Moreover, it is required that the injected designs have a new port (or variable), namely the *fault port*, which is used to select which fault to enable, because, after parallelization, this port shall be driven and managed by the parallel simulation engine (Figure 8.2).

For the experiments a couple of fault models have been chosen, the *Mutant Fault Model* and the *Bit Coverage Fault Model*, because they meet this requirement.

The bit coverage fault model [273] has been developed for the RT level, with the objective to correlate the RTL faults with the gate-level ones [274, 254]. This fault model is very similar to the traditional gate-level stuck-at fault model, because it sticks a bit either to zero or to one as stuck-at does. Moreover, it can also stick a condition to *true* or *false*. One of the main differences between these two fault models is that bit coverage is at the functional level, and it is designed to inject faults into variables, functions and operations, rather than gate-level nets. Hence, it has to perform complex injection operations because it has to deal correctly with all the RTL data types.

The mutant fault model is far more abstract than the bit coverage, and it is more focused on functional validation [249]. In the past testing based on mutation has been depicted as powerful but computationally expensive. This expense has prevented mutation from becoming widely used in practical situations, but recent engineering advances have provided techniques and algorithms for significantly reducing the cost of mutation testing [275, 276]. There are a lot of different kinds of possible code mutations: a statement can be deleted, a condition can be stucked at *true*, *false* or it can be negated, an arithmetic operator can be substituted with another one, and each boolean relation can be substituted with another one.

In the experiments, the usage of these two different fault models, which are also related with different verification objectives, points out that the explained techniques can be widely adopted, even in very different context.

8.2.2 Functional fault parallelization

Among all possible parallelization techniques, *vectorization* and *concurrency* have been chosen. The *deductive* technique has not been considered, because it requires to modify the simulation kernel. On the contrary, vectorization and concurrency can be implemented with a hardware design language and then simulated by using the designers preferred simulator.

Vectorization can be applied only to bit-blasted netlists. Bit-blasting is the term for breaking down each netlist element to its individual bit members. The vectorization is implemented by mapping each single bit to a vector of bits. Then each operation on bits is transformed into an operation on such vectors. It is done in order to associate each bit of the vector to a netlist copy, enabling the faults on all the copies but one, namely the "reference copy", which performs the normal computation. Inside the netlist it is required to transform each computation on bits to a computation on bit-vectors, i.e., to transform a logic operator to the corresponding bitwise one. This task is quite simple because all the usual languages support bitwise operations natively.

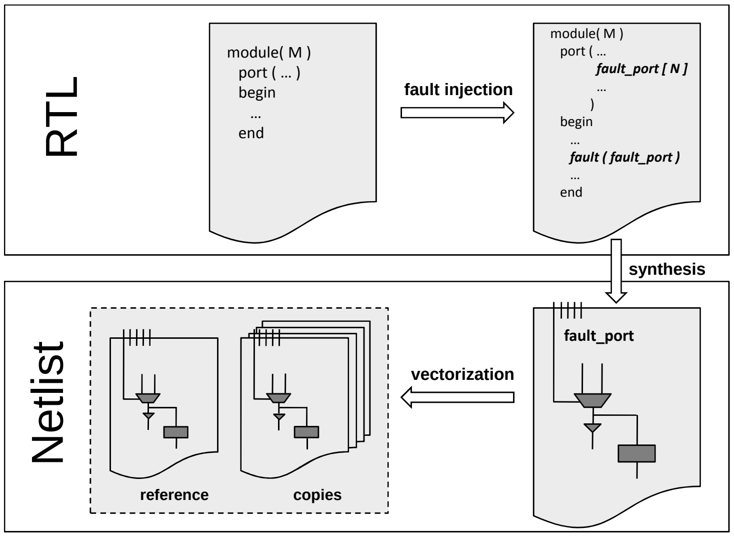


Fig. 8.2. Steps required to vectorize functional faults.

Figure 8.2 shows the steps required to implement this parallelism. In the proposed implementation, which is written in *C*, each vectorized bit have been mapped on a machine word, in order to use directly the machine instructions, avoiding the possible overhead of using more complex constructs. This mapping to a machine word allows switching from a 32-bit machine machine to a 64-bit one, to further increase performance, without any code changing.

The other kind of implemented parallelization that is the *concurrency*. It consist in enabling more than one fault per netlist copy. The main problem is that if two faults will impact on the same outputs or on the same registers, *a priori* they cannot be classified because it is not possible to understand their interaction and their effects in the simulation. To avoid this problem there are two possible strategies. The first strategy is to enable different faults as long they will not interact, i.e., their impacted registers and outputs will never interact. This check can be performed offline, but it can require some long computation time. Moreover in real-life netlists, each fault impacts on many registers, and hence this implies that only few faults could be enabled at the same time. The other solution is to just enable the faults without any offline check, but the netlist has to be instrumented in order to check eventual conflicts at runtime, i.e., during the simulation itself. This means that if a fault is going to conflict, it must be disabled before this will happen, and all the impacted registers and outputs must be reset to the reference netlist value. Moreover as soon as a fault is classified as propagated, it can also be disabled, in order to minimize the conflicts. This second strategy has been implemented.

In the rest of this Chapter, where not specified differently, the term “parallelism” refers to these two joint parallelization approaches.

8.2.3 The parallel simulation engine

While the injected netlist must be synthesizable, this is not the case for the engine which checks the faults propagation, and manages the parallel faults simulation. It could be useful to synthesize also the management code in order to deploy it on an hardware accelerator [277], but due to its complexity and to the high cost of hardware accelerators, this idea could be addressed in some future work. For this reason, in the developed parallel simulation engine there are two main conceptual modules: the parallelized netlist, and a wrapper. The problem is that it is not easy to integrate these parts, due to their different abstraction levels:

- The wrapper has to read the inputs to be passed to the netlist, but such inputs are designed for the RT level. Hence, the wrapper has to split them into single bits, and then to vectorize them, in order to be applicable to the bit-blasted and vectorized ports of the netlist.
- The outputs of the netlist are bit-blasted and vectorized. Hence, the wrapper has to compare each bit to check differences between the reference netlist and the injected netlist copies. A naive approach could considerably increment the simulation time: hence, it is required to implement an optimized algorithm which uses bitwise operations and low-level bit manipulation functions.
- The registers must be taken into account for fault concurrency on the same netlist copy. In fact, each low-level register has been substituted with a function which implements the register itself but also checks for fault collisions. Moreover, when a collision happens, one of the colliding faults must be disabled, restoring its mutated registers to the reference copy value.
- The fault port must be vectorized and split as each other netlist input. Moreover a mechanism must be implemented to enable multiple faults on the same netlist copy. A lot of fault injection mechanisms use an integer for the fault port type, in order to identify which fault to inject. For the simulation purposes it is better to use a fault port with a bit for each fault, and hence its vectorization and management is as simple as every other input port.

8.2.4 The simulation kernel and the simulation language

Usual hardware description languages (HDLs), like VHDL, Verilog and SystemC, are event driven. Their advantages are simplicity, and the existence of a lot of tools that have been developed to deal efficiently with them. But at gate level, the number of events is substantially higher with respect to RTL, especially with a bit-blasted netlist. This implies that performance is degraded because most of the simulation time is used in managing and dispatching the events, instead of simulating the design. To avoid this problem, it is necessary to switch to a simulation schema that is not event-driven but cycle-based. In fact cycle-based simulation is targeted to deal with few (or no) events, but it increases the computation complexity. In other words, each event must be translated into an equivalent expression, and such expressions must be evaluated following a special order, which must lead to the same computation results as the event driven model. This is not only a choice related with the simulation engine performance, but also a choice about which language to adopt: on one hand, HDLs are event-driven but they are widely used by designers; on the other hand, a language like C guarantees high computational performances, but it is not suitable for a large part of designs. The proposed solution exploits a translator, which has as input a standard HDL netlist, and it automatically generates a netlist written in C, which uses a cycle-based-like technique. The generated C netlist has to follow the schema depicted in Figure 8.3, in order to have the correct simulation results. The main point of this hybrid schema is that delta cycles are emulated by cycling through the netlist, but they are performed only when there is a new register value to be propagated. This optimizes the iterations, grouping all the events together. In fact the logic is computed following a top-down order, i.e., the output of a gate is computed before computing other gates impacted by this output: this reordering removes the need for the events, and hence all the values are directly propagated, instead of requiring a delta-cycle, like happens, for instance, for signals in usual netlists. Instead, the registers preserve the need of two phases, one of computation and one of propagation of the new values: this is why the writing on their output nets is performed *at once*. The outputs are checked only at the end of delta cycles, in order to avoid to check values during eventual glitching.

8.2.5 The flipping bit problem

Suppose that a fault negates the value of a net which is driven by a register, and that, this faulted value, impacts on such a register. Then it is possible that, according with the proposed cycle-based schema, the value of the register is updated, creating a delta-cycle, but then the fault negates the output bit, and hence it changes again the value of the register, creating another delta-cycle, and so on. This faulted bit, named *"flipping bit"*, creates an infinite loop. To avoid this problem, a mechanism have been implemented to detect such a fault and disable it. The idea is to count the number of delta-cycles, and if it becomes greater than a prefixed threshold, the simulation goes on, but a special tracking code is activated. This code tracks the value of each register for another prefixed number of iterations. At the end of this tracking time, it checks which registers had continuously changed their values, and hence, it disables the fault which impacts on them.

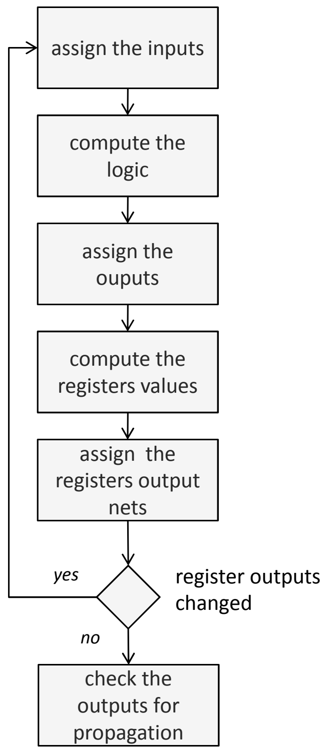

assign the inputs
compute the logic
assign the ouputs
compute the registers values
assign the registers output nets
yes
register outputs changed
no
check the outputs for propagation

Fig. 8.3. The simulation algorithm.

8.3 Optimizations

Having resolved open issues as reported in the previous section, and implemented the basic parallel simulation engine, it is possible to investigate some different ideas to further optimize the performance:

- optimizing the inputs management;
- optimizing the mux computations;
- splitting the netlist in logic cones;
- optimizing the flops computations;
- dealing with the compilers;
- adopting a four-values logic;
- exploiting the function inlining.

All these optimizations have been implemented in the proposed parallel simulation engine, and each of them is detailed in the following sections.

8.3.1 Optimized inputs management

The wrapper must split and vectorize all the input values of the netlist. This operation is quite slow, thus a good idea consists of checking if an input is changed, and in this case to recalculate only the associated input port values.

8.3.2 Mux computation optimization

Each mux can be seen as a sort of conditional construct. Then the mux can be rewritten as an *if* construct, into which the *guard* will be the selector, the *then* branch will normally compute the mux result, while the *else* branch will just propagate the input to be propagated in the case of the selector holds zero. The idea is that, if all the vectorized bits of the selector are at zero, then there is no need of computation. Moreover. it is possible also to encompass inside the *else* branch all the logic which impacts only on the zero-propagated mux input. In this way, it is possible to avoid many computations.

8.3.3 Splitting the netlist logic cones

Recomputing all the logic expressions at each simulation iteration is very time consuming. But usually only some expressions must be recomputed, because large parts of the netlist do not change their values. Hence, it is possible to divide the logic into different groups, which are classified with respect to the inputs and outputs of the logic. The main groups are:

- Logic driven only by inputs: this logic must be updated only when an input changes. During the delta-cycles this group will never be recomputed, because the inputs will preserve their values.
- Register driven logic: this logic is driven only by registers. Hence, it must be computed only during delta-cycles.
- Data logic: this logic impacts only the data input signals of flip-flops. Then it must be computed only when there is a “rising edge” on the clock.
- Mixed logic: this is all the logic which is not inside any other set. This logic must be always recomputed, and hence its computation cannot be optimized.

Each split part is written encompassed by an appropriate conditional construct, so it shall be recomputed only when it will be really required.

8.3.4 Optimizing the flops computations

The flop values are updated only on a clock “rising edge”. But usually large groups of flops are driven by the same clock signal, thus it is useless to perform a check for each of them. The idea is to encompass all of them inside a unique conditional construct, in order to minimize the checks. This implies that, if the flop has also asynchronous inputs, as it can be a reset signal, the flop implementation must be divided into both the conditional branches: the code to manage the synchronous signals be put inside the *then* branch only, while the asynchronous signals will be managed inside both the branches.

8.3.5 Dealing with the compiler

When dealing with low level circuit representations, like netlists, HDL compilers have to deal with the following problems:

- manage files with thousands of instructions and declarations inside the same scope;
- compile a large number of files in a short time.

Usually, for traditional HDLs this is not a problem, because they are languages with simple semantics and constructs.

However the proposed parallel simulation engine has been prototyped in SystemC, which is a C++ library, and which is usually compiled by software-oriented compilers, such as *GNU gcc*. To avoid the arising compilation problems, these steps have been performed:

1. Eliminate HDL code to avoid mixed C++/HDL language compiler and simulator, as, for instance, Modelsim. This speeds up the compile time.
2. Break the netlist implementation into many short functions. In fact, it is a software engineering error to implement C++ functions made by thousands of lines, because it ends up with unmaintainable code. For this reason, *gcc* is not able to compile such huge functions. Breaking the netlist in small functions avoids this *gcc* limitation.
3. Declare all the netlist variables as globals. This avoids the scope problems which could arise when splitting the netlist implementation into many functions.
4. Implement the parallel simulation engine in C. Note that, C is a very simple language in comparison with C++. In fact, C++ compilers have to deal with complex classes, and complex resolution rules. Switching to C code sensibly reduces the compiling time.

These optimizations reduced the compilation time by three orders of magnitude.

8.3.6 The four value logic

One of the most used HDL data types is the *logic* type, which is able to describe complex bit behaviors. At gate level, all the components are usually described by using such a type, and all gates and register behaviors are described taking into account all the possible logic values. As explained before, the parallel simulation engine is implemented by using the *C* language, to optimize computation performances, but such a language does not natively support logic types. To obtain the same behavior of a netlist described in an HDL, a logic with four values, i.e., *zero*, *one*, *unknown* and *high impedance* has been choosen. Each value has been described by using a couple of bits, and then all the logic gates have been encoded by using *Karneaugh maps*. This implementation is highly optimized, because it uses only few bitwise operations and it is faster than other strategies, like, for instance, the *lookup tables* used by SystemC.

8.3.7 Function inlining

A widely implemented software optimization is the inlining of functions. In *C* there are at least two ways to implement this optimization:

- implement the function as a preprocessor macro;

- using the keyword *inline*.

Using a macro guarantees that the function body will be inlined into the final code, but on the other hand, the final executable could suffer from "code bloating", i.e., it could be unnecessarily huge, causing a performances loss. This usually happens when the inlined function is very long. Instead using the appropriate keyword is safer, because it is a sort of hint for the compiler: in fact it is up to the compiler to decide whether to inline the function. The drawback is that the compiler may not inline the function, whereas the programmer knows that it really leads to a good optimization.

In the parallel engine there are a lot of small functions, each of them implementing a different kind of logic gate. Such functions are implemented in few lines of code, because they are highly compact and optimized using bitwise operators. Hence such functions have been re-implemented as macro-function, in order to avoid the overhead of their calls.

8.4 Experimental results

When a simulation is going to be performed, one of the factors which has major impact on the performances is the abstraction level at which the design is described. In fact, the simulation at RTL is much faster than the simulation at gate level, because, with less abstraction, a lot of new details must be taken into account during the simulation. In this case, interest is not on a pure simulation at gate level, but the objective is to check the faults propagation. In other words, even if the single run at gate level is slower than a single run at RTL, maybe the overall faults classification time is lower, thanks to the parallel approach that is not applicable at RT level.

To check if this is possible, some designs have been injected with both bit coverage and mutant fault models. The designs are a 12-bit microprogram sequencer, named *am2910* [278], a CPU core, named *hc11* [279], and few tests taken from the ITC-99 benchmark suite [280]. For each design 60 input sequences have been generated by using a genetic engine-based functional ATPG, each one composed of 100 test vectors. The faults have been classified both at RTL, using a serial simulation engine (SSE), and at gate-level, using the parallel simulation engine (PSE). Execution has been performed on an eight-processor Intel Xeon 2.8 MHz equipped with 8 GB of RAM and 2.6.23 Linux kernel. CPU time has been computed with the `time` command by summing up *User* and *System* time, and it is expressed in seconds. Experimental results are reported in table 8.1.

Columns report the name of the DUV (*Design*), the considered functional fault model (*FM*), the number of injected faults (*Fault #*), the simulation time achieved by using the RTL serial simulator (*SSE time*), the simulation time achieved by using the gate-level parallel simulator (*PSE time*), the achieved fault coverage (*FC %*), and the speed up gained by using the PSE simulator instead of the SSE simulator (*Speed up %*).

It is possible to note that for the majority of the designs, a good speedup has been gained by using the parallel fault simulator. On the *b04* design, using the bit coverage fault model, the parallel simulation is very slow with respect to RTL.

Design	**FM**	**Fault #**	**SSE time**	**PSE time**	**FC %**	**Speedup %**
b10	bitcov	244	9.647	7.504	91.0	22.21
b04	bitcov	398	2.816	14.194	99.0	-403.87
b10	mutant	185	25.319	7.984	81.1	68.46
b04	mutant	66	10.337	6.728	74.2	34.94
hc11	mutant	2245	811.115	204.341	49.8	74.81
am2910	bitcov	3608	755.82	505.23	83.3	33.15
am2910	mutant	2763	2219.46	636.98	76.5	71.3

Table 8.1. Experimental results: comparison between serial and parallel simulation.

The reason for this performance low is that all the faults have been propagated by the first testcase. This means that the parallelism has not been fully exploited, because there were too few faults to be simulated in simulation runs after the first.

Hence, in order to optimize the overall performances, it could be a good idea to use the parallelism at the beginning, and when the faults to be analyzed became few, return to RTL and use the serial simulation. This idea is shown in Figure 8.4.

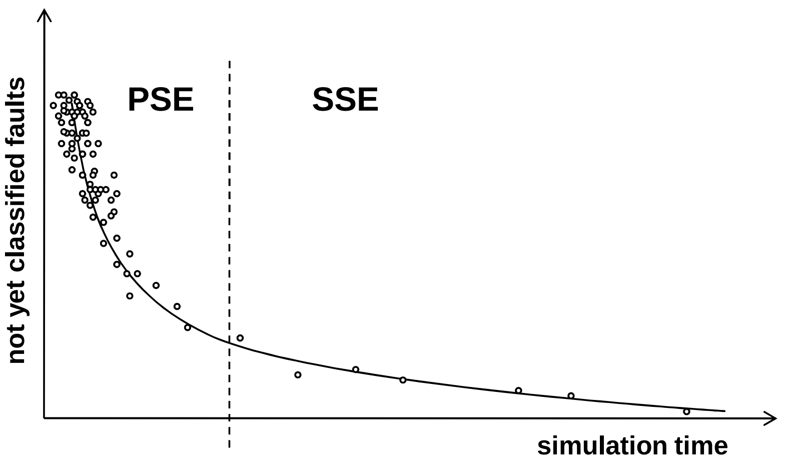


Fig. 8.4. Fault detection.

9

Application to Embedded Systems

This chapter is devoted to present how the proposed methodology can be efficiently applied to improve validation of embedded systems. The functional validation has assumed an essential role in the digital design, as the *functional approach* to the test generation permits to gain optimal results, sometimes better than the results provided by pattern generation techniques at lower levels [281,216,282,250,48,283].

A wide range of approaches for functional validation are based on automatic test pattern generation. In particular, early functional validation approaches were based on random-like-ATPG solutions. These algorithms produce effective sequences for easy-to-detects faults [203, 204]. Nevertheless, due to the increasing complexity of digital designs and the presence of hard-to-detect faults, these techniques require excessive execution times. Deterministic pattern generation algorithms deal very effectively with large systems at the expense of high amount of CPU time and memory.

Likewise, the environment description of the digital design is a key factor in efficiently generating patterns for hard-to-detect faults. In this context, the *co-simulation* is a design technique which provides the inter-communication between two modeling and simulation environments. The co-simulation aim is the integrated simulation of hardware and software systems [284]. In particular, the hardware is described by hardware description languages and simulated by HDL simulator, while the software environment typically is an *Instruction Set Simulator* (ISS) or an operating system emulator.

Traditionally, the ATPG approaches are applied to the design under validation: the digital system is stand-alone with respect to the environment. On the contrary, the proposed approach applies the ATPG techniques to the design under validation interacting with the software system, so the co-simulation is mandatory. As depicted in Figure 9.1, in the case of a validation approach based on ATPG and co-simulation, both the fault free and the faulty designs need to communicate with an independent instance of the software-environment simulator (*Simulator 1* and *Simulator 2*). The first instance of the software environment is co-simulated with the correct design (CDUV), while he second instance is co-simulated with the erroneous design (EDUV). Each of this executes an user defined application targeted the hardware device. The *Device Driver* allows higher-level applications

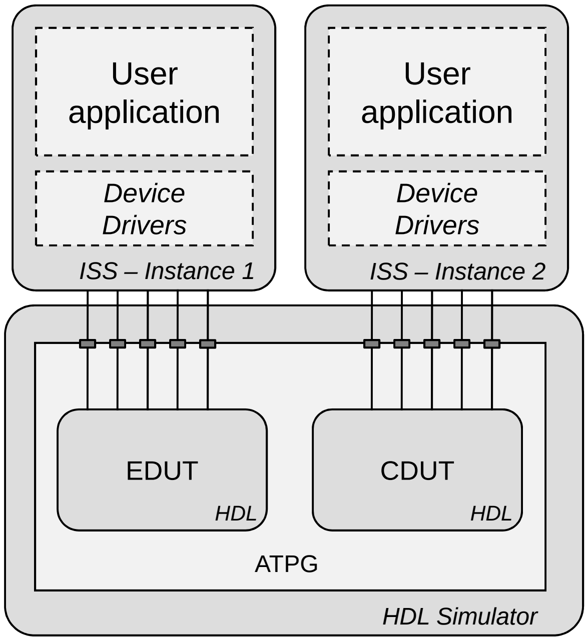


Fig. 9.1. Validation framework based on co-simulation and ATPG approach.

to interact with the hardware device. The hardware devices are modeled by HDL descriptions.

In the following sections the APTG framework, proposed in this thesis, is applied to a case study by extending the methodology to support the co-simulation approach. In particular, Section 9.1 summarize the case study platform. Section 9.2 introduces the *multi-instance co-simulation* basis. The proposed solution guarantees the integration of the ATPG and the ISS. Section 9.3 extends the proposed ATPG to permit the communication with independent instances of the software environment. The ATPG exploits the structural information of the device to explore deterministically the DUV state space, as described in Chapter 5 and in Chapter 6. As well, the adopted fault model and fault simulation techniques have been described in Chapter 7 and in Chapter 8.

9.1 Vertigo reference design platform

The design and validation methodology described in this thesis has been developed in the context of the Vertigo European Project [285]. The main goal of the Vertigo project was the development of a systematic methodology to combine a simulation-based approach (dynamic verification) together with formal methods (static verification) integrated into an IP-cores and platform based design flow, for the purpose of producing a software kit applied to the platform validation.

The system specification and the RTL code of the Project Reference Design Platform have been provided by *STMicroelectronics* [286]. The platform is composed of three main devices and memory elements interconnected by AHB e IPS buses, as reported in Figure 9.2. In particular, the *ECC* is an error correction code module for read-write operations on memory elements. The *CRC* is a cyclic-redundancy checking module which guarantees the correctness of the transferred streaming data. Finally the *DSPI* is a synchronous peripheral interface module which interconnects the system to serial ports.

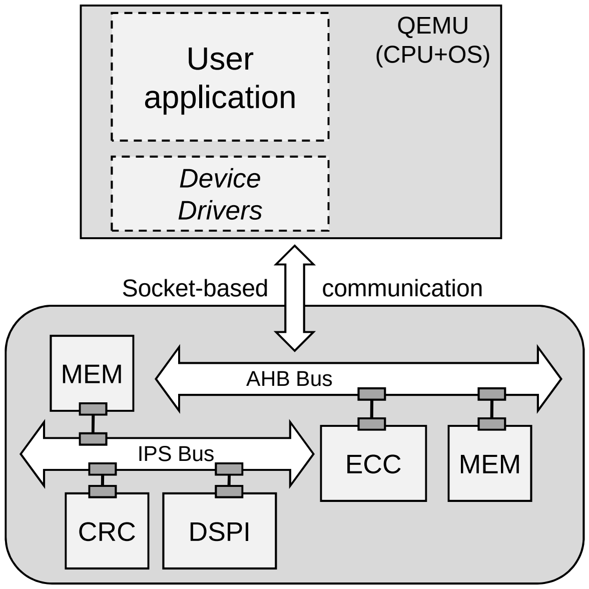


Fig. 9.2. The Vertigo platform.

One of projects aims was to identify a more comprehensive metric to assess the effectiveness of simulation patterns based on error simulation. This metric allows to measure the observability of the applied patterns, contrasted to the controllability achieved by code coverage. The fault injection techniques proposed in this thesis was one necessary feature. Moreover, STMicroelectronics have established 90% as the minimum target to sign off the Verification Environment. In addition, having the possibility to automatically generate new test patterns at RTL has been vital to speed up the achievement of the 90% Fault Coverage target.

The characteristics of each device are reported in Table 9.1: the *PIs* column reports the Primary Inputs, the *POs* reports the Primary Outputs and *FFs* reports the Flip Flops.

9.2 Mixing modeled hardware devices and applications

In the design of ever complex embedded systems, a major task is handling several platforms consisting of different processors and operating systems as well as a

Device	PIs	POs	Gates	FFs
ECC	25	32	993	79
CRC	56	34	9213	442
DSPI	25	21	1335	462

Table 9.1. Characteristics of main devices in the Vertigo platform.

large amount of HW devices such as memory, DSPs, I/O interfaces and ASICs. Instruction set simulators (ISS's) can be used to reproduce the behavior of target processors; their use offers several advantages, such as the flexibility of specifying different targets and the wide availability of standard development tools [287]. Even if some ISS has the capability to model simple HW components, such as memory and timers, in general, the simulation of HW components relies on the use of HW description languages and their simulation environments. Furthermore, in the last decade, HW/SW co-simulation has come in the mainstream [141, 142, 143, 144, 145, 146, 147, 148].

Figure 9.3 depicts different co-simulation strategies which can be used to simulate HW components:

- *host-mapping approach*: devices are mapped on the actual ones on the host machine;
- *model-level co-simulation approach*: devices are simulated by using HDL models and every driver controls the corresponding device through a dedicated channel connected to the corresponding HDL model;
- *tool-level co-simulation approach*: devices are simulated by using HDL models and synchronization between HW and SW simulations is done at tool level by exchanging messages through a single control channel.

In literature the model-level co-simulation approach has been addressed by [288]. In that work an ISS, e.g., QEmu [289], executes the application and the operating system, some HW components are mapped on the corresponding host devices while others are modeled in SystemC [290]. The communication between drivers and the corresponding devices modeled in SystemC is implemented through dedicated inter-process channels (i.e., sockets) leading to two main drawbacks:

1. HW/SW communication in case of SystemC-simulated devices is different from the final actual implementation since the designer has to put explicit socket calls in the driver implementation and in the SystemC device description;
2. in case of multiple SystemC devices the number of sockets between QEmu and SystemC may decrease simulation speed.

The proposed work aims at solving these issues by supporting HW/SW communication directly in the ISS and in the HDL simulator. The advantages are:

1. the way in which device drivers access HW devices is the same both in case of host-mapped components and HDL models;
2. a single inter-process channel is established between the ISS and the HDL simulator thus increasing the efficiency and scalability of the co-simulation framework which can handle several CPUs connected to many HDL models.

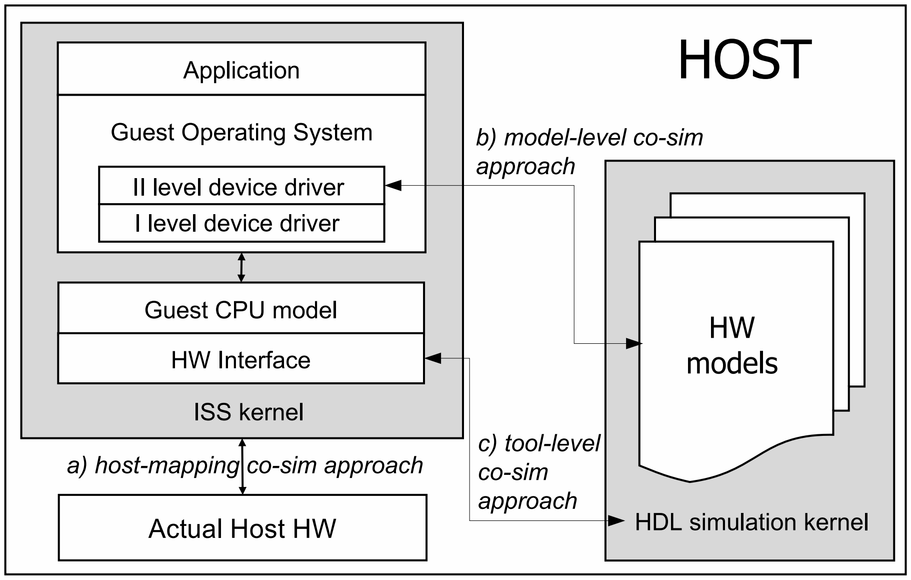


Fig. 9.3. Co-simulation strategies.

In the follow, Section 9.2.1 introduces the basic co-simulation architecture and the requirements of the work. Section 9.2.2 describes the main contribution of the paper.

9.2.1 Co-simulation architecture

Co-simulation is a methodology for the accurate verification of mixed HW/SW systems. It allows to meet the requirements for fast HW prototyping and for early SW development, because high level HW models can be effectively inserted into the development flow. The framework described in this work uses SystemC to model the HW and QEmu to emulate the SW even if the methodology can be applied to other similar tools. The reasons of the choice are:

1. QEmu already supports the use of host-mapped devices;
2. SystemC supports the HW description at many abstraction levels;
3. QEmu source code is available and easy to understand and modify.

QEmu: SW simulation

QEmu [289] achieves good SW simulation speed by using dynamic code translation to map SW instructions of the guest CPU to the host CPU so that it behaves as an ISS. QEmu has two operating modes:

- *Full system simulation.* In this mode, QEmu simulates a full system (for example a PC), including one or several processors and various peripherals; QEmu exploits the components present on the host platform to map the guest components.

- *User mode simulation.* In this mode, QEmu can launch processes compiled for one CPU on another CPU. It can be used to simplify cross-compilation and cross-debugging. Several processors are supported, e.g., x86, PowerPC, ARM, 32-bit MIPS, Sparc32/64 and ColdFire (i.e., m68k).

SystemC: HW simulation

HW devices are modeled in SystemC and module interfaces (i.e., input/output ports) follow the rules presented in [291]. The communication between the SystemC simulator and the ISS is implemented by means of three new type of ports added to the SystemC library, i.e., `iss_in`, `iss_out`, `iss_inout` (in the following they are also grouped under the term `iss_port`) and `iss_interrupt`. The `iss_in` port is derived from the standard `sc_in` port and it is used to read data coming from ISS, the `iss_out` port extends the SystemC `sc_out` port and it allows to send data from SystemC to the ISS. These special ports can be used to model HW registers, thus allowing the ISS to read and write them. In the following text, we assume that HW registers are memory-mapped.

The connection between the two sides of the co-simulation is performed by binding specific addresses of the ISS memory space to SystemC `iss_port` contained in the HW models. When the CPU accesses some registers through their addresses, the SystemC kernel determines the corresponding special ports of the HW model. The link between SystemC ports and memory addresses in the ISS is implemented by using a binding table stored in the SystemC kernel.

Device driver structure

In actual embedded platforms, SW applications access HW devices through device drivers. The same mechanism is needed in a co-simulation model. According to good-practice rules [292] device drivers should follow a two-levels structure:

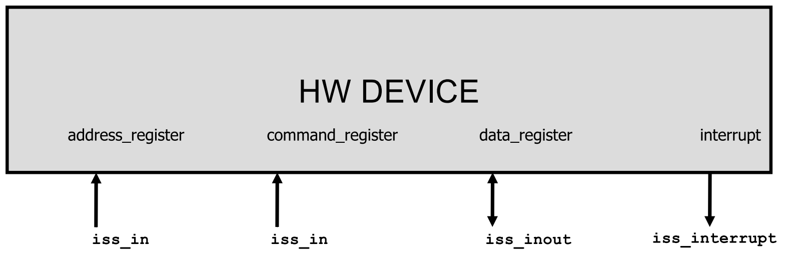
- The *II level device driver* contains the functions used by user applications to access the device (e.g., read, write, config, etc.). Each function implements a specific communication protocol according to the type of device.
- The *I level device driver* implements atomic operations used to access the device registers (such as `read` and `write`). These operations are invoked by the second level functions: the sequence of invocations forms the communication protocol. This level is equal for all devices except for some architectural choices (e.g., the addresses on which the device is mapped, interrupt handling, etc.).

Figure 9.4 shows an example of HW device and the corresponding driver code organized in two levels.

Co-simulation requirements

Since co-simulation is used for verification, the device drivers used in the co-simulation platform must be the same as on the actual operating system. This fact creates some requirements that must be met by the co-simulation architecture.

(a)



(b)

II level driver

```
int ioctl ( inode, file, command, arg ){
   switch ( command ) {
     case WRITE: return do_write ( arg );
     case READ:  return do_read  ( arg );
} }

int do_write ( arg ){
   sclink_write ( arg->address, config_value );
   sclink_write ( arg->address, config_write_value );
   sclink_write ( arg->address, arg->value );
}

int do_read ( arg ){
   return sclink_read ( arg->address );
}
```

I level driver

```
int init (){
   request_mem_region ( mem_base, 0xFF, "sclink" );
   init_registers ( mem_base );
   request_irq ( irq_num, sclink_interrupt_handler, … );
}

void sclink_write ( address, value ) {
   *address_register = address;
   *data_register = value;
   *command_register = 1;
   wait_event_interruptible ( irq_num, irq_flag!=0 );
}

void sclink_read ( address, value ) {
   *address_register = address;
   *command_register = 0;
   wait_event_interruptible ( irq_num, irq_flag!=0 );
   return *data_register;
}

int sclink_interrupt_handler ( irq, dev, *regs ){
   wake_up_interruptible ( irq_num );
}
```

Fig. 9.4. HW device (a) and the corresponding driver code (b).

- The *CPU - device communication* mechanisms must be managed. The way used to access I/O depends on the computer architecture, bus and devices being used. However, the main mechanism used to communicate with devices is through memory-mapped I/O (MMIO) according to which specific areas of CPU's addressable space are reserved for I/O. Each I/O device responds to the CPU's access of device-assigned address space. In the co-simulation architecture, access to MMIO regions must be managed by an external wrapper. Whenever the device driver accesses MMIO regions with read or write operations, the request must be forwarded to the simulated device and then the result must be brought back to the driver.

- Device drivers handle *interrupts*: thus, co-simulation must guarantee that interrupts risen by the SystemC device are forwarded to the ISS side of co-simulation.
- A device driver might contain *mutual exclusion* resources to avoid race conditions. Thus, co-simulation must manage concurrent access to the modules that handle communication with the SystemC side.

9.2.2 Co-simulation methodology

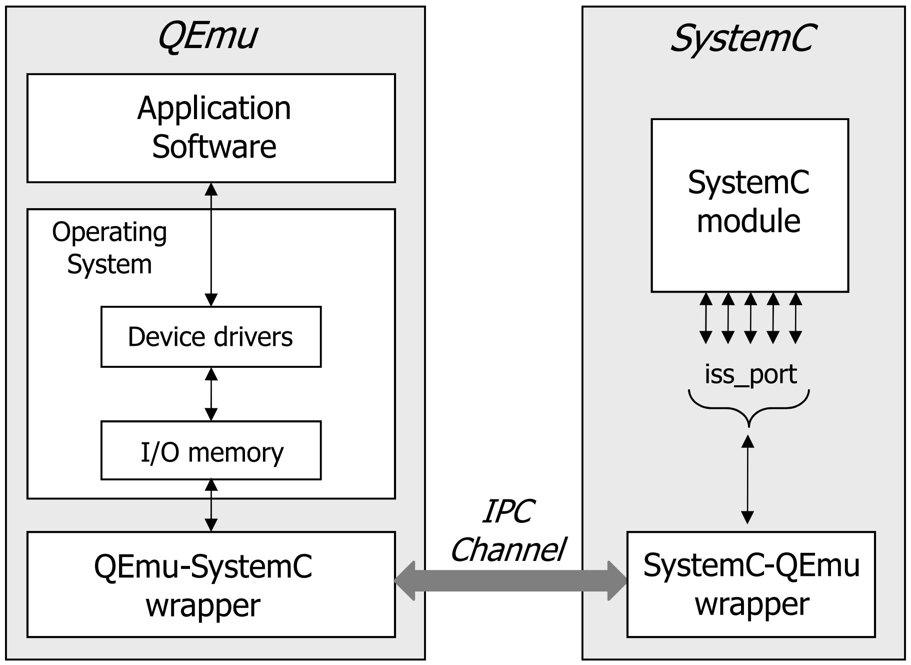


Fig. 9.5. QEmu-SystemC co-simulation schema.

Figure 9.5 shows how the SW application running on the QEmu simulator exchanges data with the SystemC simulator. An user application simply accesses the hardware devices by using their drivers, that read and write device registers through the I/O memory where the device is registered. Operations over this I/O memory pass through QEmu kernel virtualizing the hardware device implemented in SystemC. Communication with HW device, modeled in SystemC, is managed by SystemC kernel, suitably modified to support the co-simulation methodology. In order to implement this HW/SW simulator framework two steps are required:

- Modifications to the QEmu both to communicate with the SystemC simulator and to manage the HW device.

- Modifications to the SystemC simulator kernel. For the SystemC simulator, it is necessary to add the capability of reading and interpreting the messages coming from the QEmu side, as well as of sending interrupts to QEmu whenever the HW models generate them. These operations must be transparent to the designer who just writes the model by using the standard SystemC statements.

Communication between QEmu and SystemC simulator kernel is established by an inter-process channel (i.e., a socket) implementing the HW/SW interface in order to transmit synchronization messages.

SystemC-QEmu wrapper

The most meaningful parts of the SystemC - QEmu wrapper code are reported in Figure 9.6.

The SystemC - QEmu wrapper handles the SystemC side of co-simulation: it activates execution on the SystemC modules by setting the `iss_port` to the values received from QEmu through the socket.

Messages from QEmu are directed to four ISS ports on the SystemC side. The methods of the SystemC-QEmu wrapper are sensitive to these ports: whenever a data is received on a port, the corresponding method is called in order to update the wrapper registers and eventually trigger other events on the SystemC platform.

The wrapper consists of four main functions:

- `Read_iss_data_register`: this method is sensitive to the ISS data port. Whenever a new data is written to this port, the new value is saved in the data register;
- `Read_iss_address_register`: this method is sensitive to the ISS address port. Whenever a new data is written to this port, the new value is saved in the address register;
- `Read_iss_command_register`: this method is sensitive to the ISS command port. If a new data is written to this port, it means that the QEmu side has finished transmitting data and thus the SystemC side has already received the updated values for both data and address. The `read_iss_command_register` function updates the command register and it writes 0 to the ISS control port, to keep the QEmu side waiting. Then, the `read_iss_command_register` wakes up the entry method by notifying a `run_io_process` event: the entry function will process the QEmu request and write the result to the ISS data port.
- `Entry`: this function waits for a `run_io_process` event. Whenever such an event is notified, the function writes the values of data, address and command on an `ahb_transport` port to the SystemC platform. The SystemC AHB bus will receive the data and forward it to the corresponding device. Then, the `entry` function gets the execution result from the `ahb_transport` port and it writes it to the ISS data port. Finally, the ISS control port is updated to 1, to notify QEmu that execution on SystemC side is finished. This value will raise an interrupt on the QEmu side.

```
void read_iss_command_register () {
   iss_command_register->read  ( tmp, sizeof (int) );
   command_register = tmp;
   iss_command_register->write ( 0, sizeof (int) );
   run_io_process.notify();
}
void read_iss_address_register () {
   iss_address_register->read ( tmp, sizeof (int) );
   address_register = tmp;
}
void read_iss_data_register () {
   iss_data_register->read ( tmp, sizeof (int) );
   data_register = tmp;
}
void entry () {
   wait (run_io_process);
   switch (command_register) {
    case WRITE: request = set_request ( address_register,
                             data_register, WRITE );
                response = ahb_transport ( request );
    case READ:  request = set_request ( address_register,
                             data_register, READ );
                response = ahb_transport ( request );
                iss_data_register->write
                           ( response->data,sizeof(int) );
   }
   iss_control_register->write ( 1 ,sizeof (int) );
}
```

Fig. 9.6. SystemC - QEmu wrapper code.

QEmu-SystemC wrapper

The most meaningful parts of the Qemu - SystemC wrapper code are reported in Figure 9.7. The Qemu - SystemC wrapper has an important role in the co-simulation architecture: it manages accesses to MMIO regions assigned to devices, forwarding the requests to the SystemC side and bringing the result back to the device driver.

The wrapper consists of six main functions:

- `Update:` this function is used to raise an interrupt or to knock an interrupt down;
- `Init:` this function is used to initialize memory and I/O resources, to manage the addresses of the ISS ports on the SystemC side and to start the socket communication;
- `Restore:` it restores the socket communication;
- `Read:` this function is invoked whenever a read operation is performed by the driver on the I/O memory assigned to the device. This function prepares the data to be sent via socket to the SystemC side and it invokes the `cosim` function. Then, it returns the result received via socket;

- `Write`: this function is invoked whenever a write operation is performed by the driver on the I/O memory assigned to the device. This function prepares the data to be sent via socket to the SystemC side and it invokes the `cosim` function;
- `Cosim`: this function sends data to the SystemC side via socket. It is invoked by both the `read` and the `write` functions.

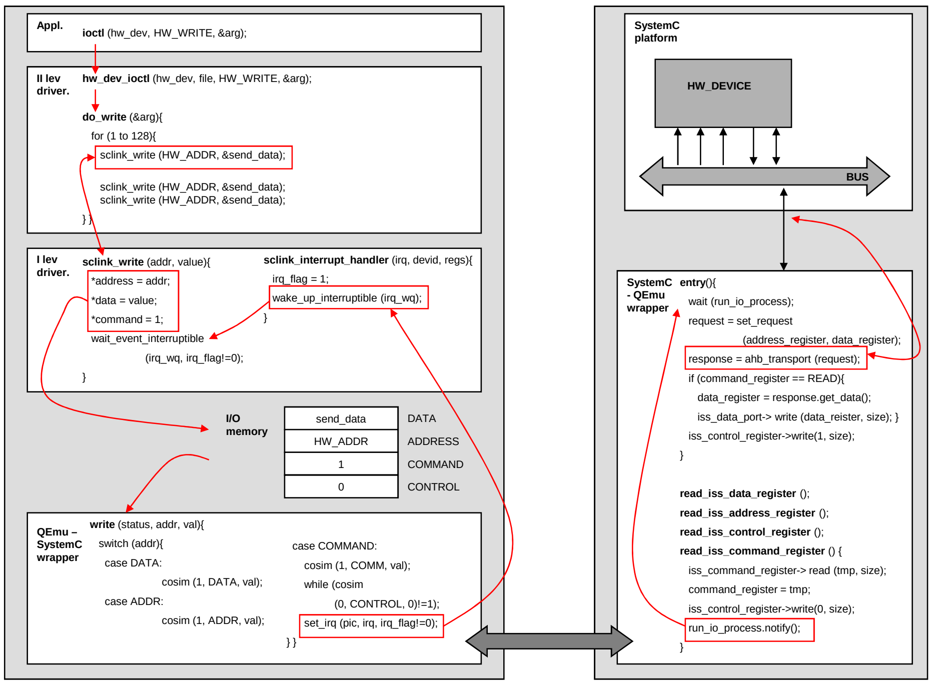
```
void update ( *state ){
   set_irq_new( pic, irq, state->flag != 0 );
}

int read ( *state, addr ){
   switch ( addr ) {
     case 0 : cosim ( 0, DATA_PORT, 0 );
              return state->data;
     case 1 : cosim ( 0, ADDR_PORT, 0 );
              return state->addr;
     case 2 : cosim ( 0, CONTROL_PORT, 0 );
     case 3 : state->irq_pending = 1;
              cosim ( 0, COMM_PORT, 0 );
} }

void write ( *state, addr, val ){
   switch ( addr ) {
     case 0 : cosim ( 1, DATA_PORT, val );
     case 1 : cosim ( 1, ADDR_PORT, val );
     case 2 : cosim ( 1, CONTROL_PORT, val );
     case 3 : cosim ( 1, COMM_PORT, val );
              while cosim (( 0, CONTROL_PORT, 0 ) != 1);
              state->flag = 1;
              update( *state );
} }

int cosim ( op, addr, data ) {
   msg = msg_create ( op, addr, data );
   send ( socket, msg, sizeof ( msg ), 0 );
   recv ( socket, msg, sizeof ( msg ), 0 ));
}
```

Fig. 9.7. QEmu - SystemC wrapper code.

Execution flow

The execution flow to access a simulated device is the following:

1. A user application wants to access the device. Thus, it invokes the `ioctl` function of the corresponding second level device driver.
2. The second level device driver implements the communication protocol with a certain number of invocations of the functions implemented by the first level device driver (`sclink_write` and `sclink_read`).
3. The first level device driver writes data to the MMIO locations assigned to the device. Then, it invokes the `wait_event_interruptible` function to suspend its execution until an interrupt is received.



```
ioctl (hw_dev, HW_WRITE, &arg);
```

```
hw_dev_ioctl (hw_dev, file, HW_WRITE, &arg);

do_write (&arg){
  for (1 to 128){
    sclink_write (HW_ADDR, &send_data);

    sclink_write (HW_ADDR, &send_data);
    sclink_write (HW_ADDR, &send_data);
} }
```

```
sclink_write (addr, value){
  *address = addr;
  *data = value;
  *command = 1;
  wait_event_interruptible
        (irq_wq, irq_flag!=0);
}

sclink_interrupt_handler (irq, devid, regs){
  irq_flag = 1;
  wake_up_interruptible (irq_wq);
}
```

send_data	DATA
HW_ADDR	ADDRESS
1	COMMAND
0	CONTROL

```
write (status, addr, val){
  switch (addr){
    case DATA:
          cosim (1, DATA, val);
    case ADDR:
          cosim (1, ADDR, val);
    case COMMAND:
      cosim (1, COMM, val);
      while (cosim
          (0, CONTROL, 0)!=1);
      set_irq (pic, irq, irq_flag!=0);
} }
```

```
entry(){
  wait (run_io_process);
  request = set_request
         (address_register, data_register);
  response = ahb_transport (request);
  if (command_register == READ){
    data_register = response.get_data();
    iss_data_port-> write (data_reister, size); }
  iss_control_register->write(1, size);
}

read_iss_data_register ();
read_iss_address_register ();
read_iss_control_register ();
read_iss_command_register () {
  iss_command_register-> read (tmp, size);
  command_register = tmp;
  iss_control_register->write(0, size);
  run_io_process.notify();
}
```

Fig. 9.8. Execution flow.

4. QEmu catches the accesses to the MMIO locations and invokes the functions of the QEmu-SystemC wrapper to forward the requests to the SystemC side of co-simulation. When the SystemC-QEmu wrapper receives a command value, the simulated device is activated.
5. As soon as the requested operation has been executed, the SystemC-QEmu wrapper sends an acknowledge message to the QEmu side of co-simulation. This message in interpreted as an interrupt: thus, the QEmu-SystemC wrapper functions notifies that execution is finished by rising an interrupt.
6. The interrupt is forwarded to the target CPU.
7. The target CPU invokes the interrupt handler function (`sclink_interrupt_handler`), that knocks the interrupt down and executes the `wake_up_interruptible` function.
8. The first level device driver resumes execution and it returns the result to the second level device driver.
9. As soon as the second level device driver has completed the communication protocol, it returns the final result to the user application, that resumes execution.

Figure 9.8 shows and clarifies the main steps of the execution flow.

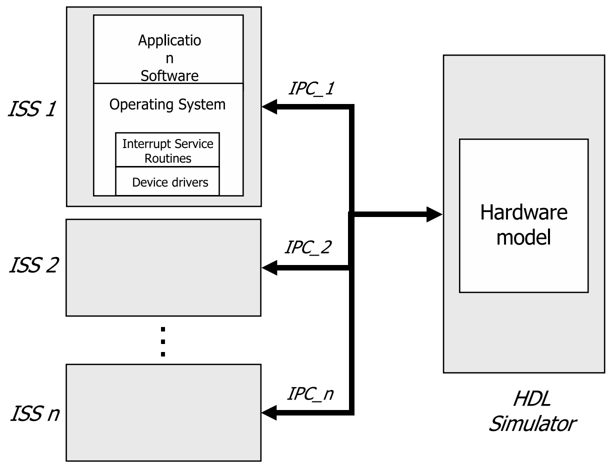


Fig. 9.9. General architecture of the multi-ISS co-simulation.

9.2.3 Multi-instance co-simulation

This section presents the SystemC simulator extensions to support a multi-ISS simulation. The multi-ISS co-simulation mechanism is based on the communication protocol established between the ISS and SystemC as described in Section 9.2.2.

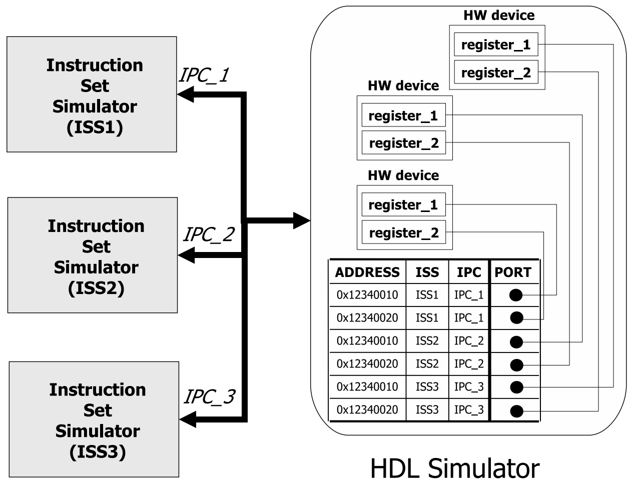
The communication between the HW simulator and the ISS is implemented by means of three new type of ports added to the SystemC library, i.e., `iss_in`, `iss_out` and `iss_interrupt`. The `iss_in` port is derived from the standard `sc_in` port and it is used to read data coming from ISS, the `iss_out` port extends the SystemC `sc_out port` and it allows to send data from SystemC to the ISS. These special ports can be used to model HW registers, thus allowing the ISS to read and write them. In the following text, we assume that HW registers are memory-mapped. The connection between the two sides of the co-simulation is performed by binding specific addresses of the ISS memory space to SystemC `iss_in` and `iss_out` ports contained in the HW models. When the CPU accesses some registers through their addresses, the SystemC kernel determines the corresponding special ports of the corresponding hardware model. The link between the SystemC port and the corresponding memory address in the ISS is implemented by using a binding table stored in the SystemC kernel.

To support multi-ISS simulation, the SystemC simulator has been modified to manage messages incoming from all ISS's. First of all, the constructor of previously described special ports (i.e., `iss_in` and `iss_out`) has been extended to obtain a string which identifies the ISS instance which can issue read/write requests. For example, an `iss_in` port has to be initialized as follows:

$$iss\_int * reg1 = iss\_in(0x12340000, ISS1)$$

Corresponding to this declaration, the following record is inserted into the binding table:

$$< reg1, 0x12340000, ISS1 >$$



ADDRESS	ISS	IPC	PORT
0x12340010	ISS1	IPC_1	●
0x12340020	ISS1	IPC_1	●
0x12340010	ISS2	IPC_2	●
0x12340020	ISS2	IPC_2	●
0x12340010	ISS3	IPC_3	●
0x12340020	ISS3	IPC_3	●

Fig. 9.10. The communication between different ISS instances and SystemC.

This mechanism allows to cluster HW models and ISS instances to model independent processing units containing a CPU and some related HW devices.

The SystemC kernel creates an IPC channel for each ISS, as shown in Figure 9.9. Since the SystemC kernel has to know the mapping between the ISS identification string and the IPC channel, during the setup phase, each ISS instance sends its identification string to SystemC which updates the binding table accordingly. An example of binding table is depicted in Figure 9.10; it shows the relationship between addresses, ISS identification strings and IPC channels. It is worth noting that the same address can be used for different registers connected to different ISS's since their memory spaces are disjoint.

Figure 9.11 shows the pseudo-code of the SystemC simulator to support multi-ISS simulation; bold text represents added code with respect to the pseudo-code described in Figure 9.11. Lines 1-4 implement the setup phase. After this phase, each ISS is able to send/receive data to/from the SystemC simulator. In the multi-ISS scenario the SystemC kernel could receive messages from different ISS's. Therefore each IPC channel has to be monitored to verify the presence of an ISS request (Line 6 and 7). Corresponding to each ISS request, the SystemC kernel has to reply to the proper ISS identified by the `iss_id` variable. Finally, the simulator extracts the next event from the queue to schedule it. In the multi-ISS case, SystemC has to find the ISS able to receive the response (Line 17).

9.3 Automatic Test Pattern Generation and Co-simulation

The methodology proposed in previous sections guarantees the intercommunication between SystemC simulation kernel and the multi-instance ISS. This section summarizes how to extend the reference ATPG to exploit co-simulation. The main problem arising in integration of the proposed ATPG and the ISS is related to time synchronization. In particular it is necessary to extend the framework previously

```
for (iss_id=0; iss_id < ISS_NUM; iss_id++) {
  create_IPC_channel(iss_id);
  update_binding_table(iss_id);
}
do {
  for (iss_id=0; iss_id < ISS_NUM; iss_id++) {
    if ( !channel[iss_id]->isEmpty() ) {
      receive(msg);
      if (SystemC_Time < msg.ISS_Time)
         add_timed_event(<operation,
                msg.ISS_Time, ISS_Port, iss_id>);
      else
         send_response_to_ISS(iss_id);
    }
  }
  event = extract_event_from_queue();
  if (event == "TIMED_EVENT")
     iss_response = find_iss_by_event(event);
     send_response_to_ISS(iss_response);
   else
     // NORMAL SystemC Kernel code
} while(…SystemC events…);
```

Fig. 9.11. SystemC procedure to support multi-ISS co-simulation.

described to gain timing accuracy. A mandatory requirement to implement the timing-accurate ISS/SystemC co-simulation is the notion of time inside the ISS and the ATPG. With this feature the local time of the ISS can be compared with the time of the SystemC simulator.

Time synchronization has to be guaranteed between ISS/SystemC-simulator kernels and between the ATPG and the simulated DUV. The first issue requires to correlate the ISS events to the SystemC kernel, in particular the simulation of the modules is clock-based, so a common clock signal is generated by the ATPG, both for the CDUV and the EDUV. Traditionally, the behavior of this signal is generated by a `clock_step()` method which exploits the SystemC method `sc_start()` as follow:

```
up()
sc_start(semiper);
down();
sc_start(semiper);
```

The clock value is set to 1, then the simulation time advances of a semi-period, then the clock value is set to 0 and the simulation advances of an other semi-period. The `sc_start` method resumes the SystemC scheduler from the time it had reached at the end of the previous call. The scheduler runs for the time passed as an argument (the semi-period), relative to the current simulation time. The `sc_start()` method does not support the ISS events to guarantee the synchronization between the HW/SW simulated environments, therefore a new SystemC method, `iss_clock_step`.

A co-simulated design has to wait for the commands coming from the ISS, this causes the follow synchronization problem between the ATPG and the DUV. The proposed solution exploits the *transaction* concept: the ATPG waits for the DUVs to complete the interaction with the ISS before to generate a new pattern. This solution require to define the following signals:

- `sc_signal <sc_bit> *iss_start_transaction`
- `sc_signal <sc_bit> *iss_end_transaction`
- `sc_signal <sc_bit> *iss_eend_transaction`

These signals are involved in the ATPG simulation cycle:

```
if(ISS_flag){
  iss_start_transaction->write((sc_bit)1); //drive start to a transaction
}
...
...
if (ISS_flag){
  while(iss_end_transaction->read()==(sc_bit)0){
  // wait until cdut transaction returns
  clock_step();
}
if(iss_eend_transaction->read()==(sc_bit)0) {
  fault_suspicious = true;
}
  iss_start_transaction->write((sc_bit)0);
  iss_end_transaction->write((sc_bit)0);
  iss_eend_transaction->write((sc_bit)0);
  clock_step();
}
```

If the ATPG is activated in *co-simulation mode* the `ISS_flag` is enabled; in this case the `iss_start_transaction` is set to 1 to identify the beginning of a new transaction. The `iss_end_transaction` and `iss_eend_transaction` flags are set to 1 by each DUV (faulty and fault free) when they complete their respective co-simulation cycle.

There is the possibility that the faulty DUV does not terminate due to an injected fault. A *termination threshold* is defined, if the faulty DUV exceeds this threshold the computation is terminated and the fault is marked as *suspicious*.

10 Conclusions

This thesis proposes a functional deterministic ATPG framework to perform functional validation of embedded design description.

The main problems of defining a deterministic ATPG have been presented and analyzed. An EFSM model to represent the DUV has been chosen and motivated and a strategy to deterministically generate test sequences by exploiting the EFSM model has been defined. Then, a high-level fault model has been adopted to quantify the effectiveness of the ATPG. Finally, a parallel fault simulation engine has been developed to improve framework efficiency. The proposed methodology has been extended to support co-simulation and validation of a complete industrial embedded system.

The main contributions presented by this works are:

1. The EFSM model has been chosen to model the DUV and an accurate analysis of its advantages have been performed and presented. This study presents a methodology to automatically manipulate the original EFSM that can be extracted form every functional design description to generate a particular kind of EFSM that is easy-to-traverse, so that the ATPG can uniformly navigate the whole DUV states space. The new EFSM is more uniformly traversable by exploiting a forward navigation strategy. This allows reducing the use of time-consuming backtracking.
2. A technique to automatically generate high-level decision diagrams (HLDDs) from EFSMs has been proposed. Then, these two paradigms are exploited inside a functional test pattern generator. The goal is to combine the beneficial properties of the above paradigms using EFSMs for targeting control FSM transitions and variable-oriented HLDDs for targeting faults in the data variables, respectively.
3. An ATPG framework has been defined that works on high-level description and efficiently exploit the features of this abstraction level. This framework is built on the top of the HIF Suite. The HIF Suite is a set of tools and APIs that relies on HIF language. HIF Suite provides designers and verification engineers with conversion from VHDL/Verilog/SystemC descriptions to HIF and viceversa, merging of mixed HDL descriptions and HIF code manipulation.
4. The navigation of concurrent EFSMs is guaranteed by an opportune EFSM scheduling algorithm that aims at maximizing the ATPG capability of exploring the whole state space. In this way multiprocess DUVs can be uniformly traversed.
5. The quality of test sequence generation is evaluated by using high-level fault models, the bit coverage and the mutation based fault model. In particular, this work presents a validation approach at a high description level, which is based on an adaptation of the mutation analysis technique. This technique was originally proposed for software testing.

6. The proposed ATPG can be configured to interface with different kinds of solvers, a CLP solver, a SAT-solver and a Model Checker. In fact the determinism is obtained by interfacing with a tool that adopts formal methods to solve the conditions of the enabling functions. In particular, the ATPG has been interfaced with both a CLP-based constraint solver, SAT-solver and Model Checker. The effectiveness of the proposed ATPG compared with a genetic-based ATPG is evident. It greatly benefits from the fact that, by using the EFSM model, all conditional statements included in the DUV are under its control.
7. A two-step ATPG engine is implemented which exploits constraint solving to traverse the DUV state space. At first, a random walk-based approach is used to cover the majority of easy-to-traverse (ETT) transitions. Then a backjumping-based mode is used to activate hard-to-traverse (HTT) transitions. In both modes, learning is exploited to get critical information that improves the performance of the ATPG. Testing of hard-to-detect faults is thus improved.
8. A new kind of EFSM triggered by events, called EEFSM, has been introduced. An Extended Event Finite State Machine is an finite state machine augmented by a set of registers that range over a finite alphabet and by a set of input/ output events that trigger the transitions of the machine. Input and output events are used to model synchronization with a clock signal and the sensitivity list construct defined in hardware description languages.
9. A theory to compose such EEFSMs has been defined. Then, the impact of bounded EEFSM composition has been analyzed on a functional ATPG, which exploits a constraint solver to deterministically traverse EEFSM transitions. In particular, the performance of the ATPG working on a set of concurrent EEFSMs and on a single EEFSM, representing the same DUV, have been compared. Experimental results showed that the ATPG benefits from EFSM composition, since this allows a global view of the DUV. This generally reflects on the achievement of higher transition and fault coverages, and on the reduction of constraint solver invocation which sensibly impacts on the the ATPG execution time.
10. The main issues arising to implement an efficient parallel simulation engine for functional faults has been analyzed. Moreover, this work describes a set of optimizations implemented to increase performance of parallel simulation. Experimental results show that even if the parallelization requires bit-level simulation, which is generally slower than high-level simulation, the overall simulation time is reduced as long as there are enough faults to be checked in parallel.
11. The proposed ATPG methodology has been integrated in a HW/SW co-simulation framework, based on the interaction of SystemC with a time-aware instruction set simulator. The co-simulation framework performs timing-accurate co-simulation and supports multiple ISS instances. The overall ATPG/co-simulation framework has been applied to an industrial case study.

The methodology presented in this thesis has given interesting results, but it opens still space for future works:

1. The EFSM composition methodology has been defined and implemented. A rigorous demonstration of correctness is necessary.
2. Actually the stimuli generation is transition oriented. It is necessary to extend the methodology supporting fault-oriented pattern generation. Therefore, if the design under validation is represented as a set of EFSMs, it is necessary to identify the relations between the high-level faults and the structure of the models, i.e. states and transitions.
3. The proposed ATPG methodology is dedicated to hardware verification. The next step consists of extending it to software testing. Software testing is aimed at evaluating the quality of a program or system and determining that it meets its required

results. Correctness testing and reliability testing are two major areas of testing. For is nature, the complexity with software is generally intractable and most of the defects in software are design errors. A further complication has to do with the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, the software may now work for a test case that it didn't work for previously. But its behavior on pre-error test cases that it passed before can no longer be guaranteed. To account for this possibility, testing should be restarted. Different problem should be accomplished. As first step, different faults models, specific for software, should be defined. Consequently an accurate analysis should be performed to identify the relationship between mutants and fault models for software. Another challenging tasks is to deal with the dependability of software. As software is executed, it should be checked if it is tolerant to problem in executions.

4. The CLP engine is based on semi-formal techniques that for their nature tend to increase the problem complexity and require time and resources. Optimization techniques have been already proposed that gives reduce the complexity for the solver. However, other heuristics for the paths script generation could be considered and compared to improve the engine effectiveness.
5. Further techniques, different from CLP, could be applied to generate the sequences. Solvers that can be interfaced trough C++ could be exploited by the functional ATPG.

11

Published Contributions

11.1 Journal papers

Giuseppe Di Guglielmo, Franco Fummi, Cristina Marconcini, Graziano Pravadelli
Improving high-level and gate-level testing with FATE: A functional automatic test pattern generator traversing unstabilised extended FSM
Computers & Digital Techniques, IET
Volume 1, Issue 3, 2007, pp. 187-196

11.2 International Conference

Davide Bresolin, Giuseppe Di Guglielmo, Franco Fummi, Graziano Pravadelli, Tiziano Villa
The impact of EFSM Composition on Functional ATPG
In the Proceedings of "12th IEEE Symposium on Design and Diagnostics of Electronic Systems (DDECS'09)"
Liberec, Czech Republic, April 15-17, 2009

Giuseppe Di Guglielmo, Franco Fummi, Mark Hampton, Graziano Pravadelli, Francesco Stefanni
The Role of Parallel Simulation on Functional Verification
In the Proceedings of "IEEE International High Level Design Validation and Test Workshop (HLDVT'08)"
Nevada, USA, November 19-21, 2008, pp. 117-124

Anton Chepurov, Giuseppe Di Guglielmo, Franco Fummi, Graziano Pravadelli, Jain Raik, Raimund Ubar, Taavi Viilukas
Automatic generation of EFSMs and HLDDs for functional ATPG
In the Proceedings of "IEEE International Biennal Baltic Electronics Conference (BEC'08)"
Tallinn, Estonia, October 6-8, 2008, pp. 143-146

Giuseppe Di Guglielmo
On the Validation of Embedded Systems through Functional ATPG
In the Proceedings of "IEEE Conference on Ph.D. Research in Microelectronics and Electronics (PRIME'08)"

Istanbul, Turkey, June 22-25, 2008, pp. 149-152

Giuseppe Di Guglielmo, Franco Fummi, Maksim Jenihhin, Graziano Pravadelli, Jaan Raik, Raimund Ubar
On the Combined Use of HLDDs and EFSMs for Functional ATPG
In the Proceedings of “5th IEEE East-West Design & Test International Symposium (EWDTS'07)”
Yerevan, Armenia, September 7-10, 2007, pp. 503-508

Giuseppe Di Guglielmo, Franco Fummi, Cristina Marconcini, Graziano Pravadelli
FATE: a Functional ATPG to Traverse unstabilized EFSMs
In Proceeding of ”IEEE European Test Symposium (ETS'06)”
Southampton, UK, May 21-24, 2006, pp. 179-184

Giuseppe Di Guglielmo, Franco Fummi, Cristina Marconcini, Graziano Pravadelli
Improving Gate-Level ATPG by Traversing Concurrent EFSMs
In Proceeding of ”IEEE VLSI Test Symposium (VTS'06)”
Berkeley, CA, USA April 30 - May 4, 2006

Giuseppe Di Guglielmo, Franco Fummi, Cristina Marconcini, Graziano Pravadelli
EFSM Manipulation to Increase High-Level ATPG
In Proceeding of ”IEEE International Symposium on Quality Electronic Design (ISQED'06)”
San Jose, CA, USA, March 27-29, 2006, pp. 62-67

Giuseppe Di Guglielmo, Franco Fummi, Cristina Marconcini, Graziano Pravadelli
A Pseudo-Deterministic Functional ATPG based on EFSM Traversing
In Proceeding of ”IEEE International Workshop on Microprocessor Test and Verification (MTV'05)”
Austin, TX, USA, November 3-4, 2005, pp. 70-75

11.3 PhD Forums

Giuseppe Di Guglielmo
On the Validation of Embedded Systems through Functional ATPG
IEEE International Design Automation & Test in Europe DATE (DATE'09) EDAA PhD Forum
Nice, France, April 20-24, 2009

Giuseppe Di Guglielmo
On the Validation of Embedded Systems through Functional ATPG
IEEE Conference on PhD Research in Microelectronics and Electronics (PRIME'08)
Istanbul, Turkey, June 22-25, 2008, pp. 149-152

Giuseppe Di Guglielmo
On the Validation of Embedded Systems through Functional ATPG
TTTC PhD at IEEE VLSI Test Symposium (VTS'08)
San Diego, CA, April 27 - May 1, 2008

References

1. J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, Norwell Massachusetts, 2000.
2. M.A. Breuer, M. Abramovici, and A.D. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, 1990.
3. P.J. Ashenden. *The VHDL Cookbook*. Kluwer Academic Publishers, first edition, 1990.
4. D.E. Thomas and P.R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, second edition, 1994.
5. Synopsys. *SystemC User's Guide*. 2002.
6. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):79–85, 1986.
7. I. Ghosh and M. Fujita. Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(3):402–415, 2001.
8. D.D. Gajski, N.D. Dutt, S. Allen, C.H. Wu, and Y.L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, first edition, 1992.
9. K.T.Cheng and A.S. Krishnakumar. Automatic generation of functional vectors using the extended finite state machine model. *ACM Trans. on Design Automation of Electronic Systems*, 1(1):57–79, 1996.
10. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.
11. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.
12. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *In Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.
13. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.
14. O. Lichtenstein and A. Pnueli. Checking that finite-state concurrents programs satisfy their linear specification. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–186, 1985.
15. E. Clark, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

16. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.
17. D. Brand. Verification of large synthesized designs. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pages 534–537, 1993.
18. A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 263–268, 1997.
19. W. Kunz and D. Pradhan. Recursive learning: a new implication technique for efficient solutions to cad problems - test, verification and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(9):1143–1158, 1994.
20. H. Cho, G.D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate FSM traversal based on space state decomposition. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 25–30, 1993.
21. H.J. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdds. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pages 130–133, 1990.
22. G. Cabodi, P. Camurati, and S. Quer. Implicit manipulation of equivalence classes for large finite state machines. *IEE Computer and Digital Techniques*, 145(6):395–402, 1998.
23. G. Cabodi, P. Camurati, and S. Quer. Improved reachability analysis of large finite state machines. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pages 354–360, 1996.
24. B. Wile, J. C. Goss, and W. Roesner. *Comprehensive Functional Verification - The Complete Industry Cycle.* Elsevier, 2005.
25. G.J. Myers. *The Art of Software Testing.* Wiley - Interscience, New York, 1979.
26. F. Ferrandi, M. Rendine, and D. Sciuto. Functional verification for systemc descriptions using constraint solving. In *Proceedings of IEEE Design Automation and Test in Europe (DATE)*, pages 744–751, 2002.
27. M.E. Amyeen, I. Pomeranz, and W.K. Fuchs. Theorems for efficient identification of indistinguishable fault pairs in synchronous sequential circuits. In *Proceedings of IEEE VLSI Test Symposium (VTS)*, pages 181–186, 2002.
28. H. Shi-Yu, C. Kwang-Ting, H. Chung-Yang, and F. Brewer. Aquila: An equivalence checking system for large sequential designs. *IEEE Transactions on Computers*, 49(5):443–464, 2000.
29. D. Moundanos, J.A. Abraham, and Y.V. Heskote. A unified framework for design validation and manufacturing test. In *Proceedings of IEEE International Test Conference (ITC)*, pages 875–884, 1996.
30. Q. Wu and M. Hsiao. Efficient sequential atpg based on partitioned finite-state-machine traversal. In *Proceedings of IEEE International Test Conference (ITC)*, pages 281–289, 2003.
31. S. Tasiran, F. Fallah, D.G. Chinnery, S.J. Weber, and K. Keutzer. A functional validation technique: Biased-random simulation guided by observability-based coverage. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 82–88, 2001.
32. B. Shaer, S.A. Al-Arian, and D. Landis. Partitioning sequential circuits for pseudoexhaustive testing. *IEEE Transactions on VLSI*, 8(5):534–541, 2000.
33. N. Kamiura, Y. Hata, and N. Matsui. Controllability/observability measures for multiple-valued test generation based on d-algorithm. In *Proceedings of IEEE International Symposium on Multiple-Valued Logic (ISMVL)*, pages 245–250, 2000.

34. S. Devadas, A. Ghosh, and K. Keutzer. An observability-based code coverage metric for functional simulation. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 418–425, 1996.
35. F. Fallah, S. Devadas, and K. Keutzer. Functional vector generation for HDL models using linear programming and 3-satisfiability. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 528–533, 1998.
36. F. Fallah, S. Devadas, and K. Keutzer. Occom: Efficient computation of observability-based code coverage metrics for functional verification. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 152–157, 1998.
37. F. Fallah, P. Ashar, and S. Devadas. Simulation vector generation from hdl descriptions for observability-enhanced statement coverage. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 666–671, 1999.
38. X. Yu, J. Wu, and E.M. Rudnick. Diagnostic test generation for sequential circuits. In *Proceedings of IEEE International Test Conference (ITC)*, pages 225–234, 2000.
39. W.E. Dougherty and R.D. Blanton. Using regression analysis for ga-based atpg parameter optimization. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 516 –521, 1998.
40. M.J. O'Dare and T. Arslan. Generating test patterns for vlsi circuits using a genetic algorithm. *Electronics Letters*, 30(10):778–779, 5 1994.
41. F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, and G. Squillero. Initializability analysis of synchronous sequential circuits. *ACM Transactions on Design Automation of Electronic Systems*, 15(2):249–264, 2002.
42. E.M. Rudnick and J.H. Patel. A genetic algorithm framework for test generation. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 16(9):1034–1044, 1997.
43. F. Corno, P. Prinetto, M. Rebaudengo, , and M. Sonza Reorda. Gatto: A genetic algorithm for automatic test pattern generation for large synchronous sequential circuit. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):991–1000, 1996.
44. E.M. Rudnick and J.H. Patel. Combining deterministic and genetic approaches for sequential circuit test generation. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 183–188, 1995.
45. F. Corno, M. Sonza Reorda, G. Squillero, and M. Violante. A genetic algorithm-based system for generating test programs for microprocessor ip cores. In *Proceedings of IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 195–198, 2000.
46. M. Boschini, X. Yu, F. Fummi, and E.M. Rudnick. Combining symbolic and genetic techniques for efficient sequential circuit test generation. In *Proceedings of IEEE European Test Workshop (ETW)*, pages 105–110, 2000.
47. M. Lajolo, L. Lavagno, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Behavioral-level test vector generation for system-on-chip designs. In *Proceedings of IEEE International High-level Design Validation and Test Workshop (HLDVT)*, pages 21–26, 2000.
48. F. Ferrandi, A. Fin, F. Fummi, and D. Sciuto. An application of genetic algorithms and bdds to functional testing. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 48–56, 2000.
49. M.S. Hsiao, E.M. Rudnick, and J.H. Patel. Fast static compaction algorithms for sequential circuit test vectors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 48(3):311–322, 1999.
50. E.M. Rudnick and J.H. Patel. Efficient techniques for dynamic test sequence compaction. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 48(3):323–330, 1999.

51. F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero. Effective techniques for high-level atpg. In *Proceedings of IEEE Asian and Test Symposium (ATS)*, pages 225–230, 2001.
52. M.S. Hsiao, E.M. Rudnick, and J.H. Patel. Dynamic state traversal for sequential circuit test generation. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):548–565, 2000.
53. G. Biasoli, F. Ferrandi, A. Fin, F. Fummi, and D. Sciuto. Behavioral test generation for the selection of bist logic. *Journal of System Automation (JSA)*, 47(10):821–829, 2002.
54. G. Harik. Linkage learning via probabilistic modeling in the ecga. Technical Report 99010, University of Illinois at Urbana-Champaign, 1999.
55. J.A. Abraham and V.K. Agarwal. *Test Generation for Digital Systems - Fault-Tolerant Computing: Theory and Techniques.* Prentice-Hall, first edition, 1985.
56. J.A. Abraham and K. Fuchs. Fault and error models for vlsi. *Proceedings of the IEEE*, 74(5):639–654, 5 1986.
57. D. Brahme and J.A. Abraham. Functional testing of microprocessors. *IEEE Transactions on Computers*, C-33(6):475–485, 1985.
58. S.M. Thatte and J.A. Abraham. A methodology for functional level testing of microprocessors. In *Proceedings of IEEE International Conference on Fault-Tolerant Computing (ICFC)*, pages 90–95, 1978.
59. S.M. Thatte and J.A. Abraham. Test generation for microprocessors. *IEEE Transactions on Computers*, 29(3):429–441, 1980.
60. M.C. Hansen and J.P. Hayes. High-level test generation using physically-induced faults. In *Proceedings of IEEE VLSI Test Symposium (VTS)*, pages 20–28, 1995.
61. J.P. Hayes. Fault modeling for digital mos integrated circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 3(3):200–207, 1984.
62. J.P. Hayes. A fault simulation methodology for vlsi. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 393–399, 1982.
63. T. Sridhar and J.P. Hayes. A functional approach to testing bit-sliced microprocessors. *IEEE Transactions on Computers*, 30(8):563–571, 1981.
64. C.H. Cho and J.R. Armstrong. B-algorithm: A behavioral test generation algorithm. In *Proceedings of IEEE International Test Conference (ITC)*, pages 968–979, 1994.
65. F.S. Lam J.R. Armstrong and P.C. Ward. *Test Generation and Fault Simulation for Behavioral Models - Performance and Fault Modeling with VHDL.* Prentice Hall, first edition, 1992.
66. P.C. Ward and J.R. Armstrong. Behavioral fault simulation in vhdl. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 587–593, 1990.
67. J.R. Armstrong. *Chip Level Modeling with VHDL.* Prentice Hall, first edition, 1989.
68. J.R. Armstrong. Chip level modeling with hdls. *IEEE Design and Test of Computers*, 5(1):8–18, 1988.
69. A.K. Gupta and J.R. Armstrong. Functional fault modeling and simulation for vlsi devices. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 720–726, 1985.
70. J.R. Armstrong. Chip level modeling of lsi devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 3(4):288–297, 1984.
71. S. Ghosh and T.J. Chakraborty. On behavior fault modeling for digital designs. *International Journal of Electronic Testing: Theory and Applications*, 2(2):135–151, 1991.
72. T.J. Chakraborty and S. Gosh. On behavior fault modeling for combinational digital designs. In *Proceedings of IEEE International Test Conference (ITC)*, pages 593–600, 1988.

73. S. Gosh. Behavior-level fault simulation. *IEEE Design and Test of Computers*, 5(3):31–42, 1988.
74. P. Banerjee. A model for simulating physical failures in mos vlsi circuits. Technical Report CSG-13, University of Illinois at Urbana-Champaign, 1985.
75. C.H. Chao and F.G. Gray. Micro-operation perturbations in chip level fault modeling. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 579–582, 1988.
76. J. Gracia, J.C. Baraza, D. Gil, and P.J. Gil. Comparison and application of different vhdl-based fault injection techniques. In *Proceedings of IEEE International Symposium on Defect and Fault Tolerance (DFT)*, pages 579–582, 2001.
77. E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: The mefisto tool. In *Proceedings of IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, pages 66–75, 1994.
78. A.M. Amendola, A. Benso, F. Corno, L. Impagliazzo, P. Marmo, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. Fault behaviour observation of a microprocessor system through a vhdl simulation-based fault injection experiment. In *Proceedings of IEEE European Design Automation Conference (EURO-DAC)*, pages 536 –541, 1996.
79. V. Sieh, O. Tschche, and F. Balbach. Verify: Evaluation of reliability using vhdl-models with embedded fault descriptions. In *Proceedings of IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, pages 32–36, 1997.
80. J.A. Profeta III T.A. DeLong, B.W. Johnson. A fault injection technique for vhdl behavioral-level models. *IEEE Design and Test of Computers*, 13(4):24–33, 1996.
81. J.C. Baraza, J. Gracia, D. Gil, and P.J. Gil. A prototype of a vhdl-based fault injection tool: Description and application. *Journal of System Automation (JSA)*, 47(10):847–867, 2002.
82. J. Gracia, J.C. Baraza, D. Gil, and P.J. Gil. Application of different vhdl-based fault injection techniques to the validation of a fault-tolerant microcomputer system. In *Proceedings of IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, pages 54–55, 2000.
83. T. Budd and F. Sayward. Users guide to the Pilot mutation system. In *technical report 114*, 1977.
84. R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3:279–290, July 1977.
85. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11:34–41, April 1978.
86. T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. The design of a prototype mutation system for program testing. In *In the Proc. of NCC, AFIPS Conference Record*, pages 623–627, 1978.
87. R. J. Lipton and F. G. Sayward. The status of research on program mutation. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 355–373, December 1978.
88. A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. In *technical report GITICS-79/08*, 1979.
89. A. J. Offutt and K. N. King. A Fortran 77 interpreter for mutation analysis. In *In the Proc. of 1987 Symposium on Interpreters and Interpretive Techniques*, pages 177–188, June 1987.
90. R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In IEEE Computer Society Press, editor, *In the Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, July 1988.

91. A. J. Offutt. Automatic Test Data Generation. In *PhD thesis, Technical report GIT-ICS 88/28*, 1988.
92. R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17:900–910, September 1991.
93. R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation.
94. R. H. Untch, M. J. Harrold, and J. Offutt. Schema-based mutation analysis.
95. M. E. Delamaro and J. C. Maldonado. Proteum-A tool for the assessment of test adequacy for C programs. In *in the Proc. of the Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, July 1996.
96. A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1:3–18, January 1992.
97. K. S. H. T. Wah. Fault coupling in finite bijective functions. *The Journal of Software Testing, Verification, and Reliability*, 5:3–47, March 1995.
98. K. S. H. T. Wah. A theoretical study of fault coupling. *The Journal of Software Testing, Verification, and Reliability*, 10:3–46, March 2000.
99. D. Wu, M. A. Hennell, D. Hedley, and I. J. Riddell. A practical method for software quality control via program mutation. In IEEE Computer Society Press, editor, *In the Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pages 159–170, July 1988.
100. IEEE. IEEE Standard Glossary of Software Engineering Terminology. ANSI/IEEE Std 610.12-1990, 1996.
101. R. Geist, A. J. Offutt, and F. Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing*, 41:550–558, May 1992.
102. A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5:99–118, April 1996.
103. W. E. Wong, M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Constrained mutation in C programs. In *In the Proc. of the 8th Brazilian Symposium on Software Engineering*, pages 439–452, October 1994.
104. A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *In the Proc. of the Fifteenth International Conference on Software Engineering*, pages 100–107. IEEE Computer Society Press, May 1993.
105. A. T. Acree. On Mutation. PhD thesis, 1980.
106. T. A. Budd. Mutation Analysis of Program Test Data. PhD thesis, 1980.
107. W. E. Wong. On Mutation and Data Flow. PhD thesis, December 1993.
108. M. Sahinoglu and E. H. Spafford. A bayes sequential statistical procedure for approving software products. In *In the Proc. of the IFIP Conference on Approving Software Products (ASP-90)*, pages 43–56, September 1990.
109. W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8:371–379, July 1982.
110. L. J. Morell. Theoretical insights into fault-based testing. In *In the Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pages 45–62. IEEE Computer Society Press, July 1988.
111. M. R. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *In the Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pages 152–158. IEEE Computer Society Press, July 1988.
112. J. R. Horgan and A. P. Mathur. Weak mutation is probably strong mutation. technical report SERC-TR-83-P, December 1990.

113. M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In *In the Proc. of the Eighth International Conference on Software Engineering*, pages 313–319. IEEE Computer Society Press, August 1985.
114. B. Marick. Two experiments in software testing. In *technical report UIUCDCS-R-90-1644*, November 1990.
115. B. Marick. The weak mutation hypothesis. In *In the Proc. of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 190–199. IEEE Computer Society Press, October 1991.
116. A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20:337–344, May 1994.
117. A. J. Offutt and S. D. Lee. How strong is weak mutation? In *In the Proc. of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 200–213. IEEE Computer Society Press, October 1991.
118. A. P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. In *technical report SERC-TR-14-P*, April 1988.
119. E. W. Krauser, A. P. Mathur, and V. Rego. High performance testing on SIMD machines. In *In the Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, pages 171–177. IEEE Computer Society Press, July 1988.
120. A. J. Offutt, R. Pargas, S. V. Fichter, and P. Khambekar. Mutation testing of software using a mimd computer. In *In the Proc. of 1992 International Conference on Parallel Processing*, pages II–257–266, August 1992.
121. B. Choi and A. P. Mathur. High-performance mutation testing. *The Journal of Systems and Software*, 20:135–152, February 1993.
122. C. N. Zapf. Medusamothra - a distributed interpreter for the mothra mutation testing system. In *M.S. thesis*, August 1993.
123. V. N. Fleyshgakker and S. N. Weiss. Efficient Mutation Analysis: A New Approach. In *In the Proc. of the International Symposium on Software Testing and Analysis (ISSTA 94)*, pages 185–195. ACM SIGSOFT, ACM Press, August 1994.
124. R. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *In the Proc. of the 1993 International Symposium on Software Testing, and Analysis*, pages 139–148, June 1993.
125. R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2:109–127, April 1993.
126. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction approach for test data generation: Design and algorithms. In *technical report ISSE-TR-94-110*, September 1994.
127. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction approach to test data generation. *Software-Practice and Experience*, 29:167–193, January 1999.
128. B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16:870–879, August 1990.
129. B. Korel. Dynamic method for software test data generation. *The Journal of Software Testing, Verification, and Reliability*, 2:203–213, 1992.
130. L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2:215–222, September 1976.
131. L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *The Journal of Systems and Software*, 5:15–35, January 1985.
132. R. E. Fairley. An experimental program testing facility. *IEEE Transactions on Software Engineering*, SE-1:350–3571, December 1975.
133. T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18:31–45, November 1982.

134. D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. In *research report 276*, 1979.
135. A. Tanaka. Equivalence testing for fortran mutation system using data flow analysis, master's thesis. 1981.
136. A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *The Journal of Software Testing, Verification, and Reliability*, 4:131–154, September 1994.
137. A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *In the Proc. of the 1996 Annual Conference on Computer Assurance (COMPASS 96)*, pages 224–236. IEEE Computer Society Press, June 1996.
138. A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *The Journal of Software Testing, Verification, and Reliability*, 7:165–192, September 1997.
139. R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification, and Reliability*, 9:233–262, December 1999.
140. G. De Micheli and R. Ernst. *Readings in Hardware/Software Co-Design.* Morgan Kaufmann, 2002.
141. J. Buck, S. Ha, E.A. Lee, and D. G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *International Journal in Computer Simulation*, pages 527–543, 2001.
142. F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, et al. *Hardware-software co-design of embedded systems: the POLIS approach.* Kluwer Academic Press., 1997.
143. http://www.synopsys.com/products. Technical report, Eaglei, Synopsys Inc.
144. http://www.mentor.com/seamless. Technical report, Seamless CVE, Mentor Graphics Inc.
145. C. Liem, F. Nacabal, C. Valderrama, P. Paulin, and A. Jerraya. System-on-chip co-simulation and compilation. *IEEE Design and Test of Computers*, 14(2):16–25, 1997.
146. C. Valderrama, F. Nacabal, P. Paulin, and A. Jerraya. Automatic VHDL-C Interface Generation for Distributed Cosimulation: Application to Large Design Examples. *Design Automation for Embedded Systems*, 3(2):199–217, 1998.
147. P. Coste, F. Hessel, P. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and A. Jerraya. Multilanguage design of heterogeneous systems. In *In the Proc. of IEEE International Workshop on Hardware-Software Codesign*, pages 54–58, 1999.
148. L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. SystemC cosimulation and emulation of multiprocessor SoC designs. *IEEE Computer*, 36(4):53–59, 2003.
149. J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli. Software timing analysis using HW/SW cosimulation and instruction set simulator. In *In the Proc. of the 6th International Workshop on Hardware/Software Codesign*, pages 65–69. IEEE Computer Society Washington, DC, USA, 1998.
150. M B Santos, F M Gonalves, I C Teixeira, and J P Teixeira. RTL-based functional test generation for high defects coverage in digital SoCs. In *Proceedings of IEEE ETW*, pages 99–104, 2000.
151. K. Lahiri, A. Raghunathan, G. Lakshminarayana, and S. Dey. Communication architecture tuners: a methodology for the design of high-performance communication architectures for systems-on-chips. In *In the Proc. of the ACM/IEEE Design Automation Conference*, pages 513–518. ACM New York, NY, USA, 2000.
152. F. Fummi, S. Martini, G. Perbellini, and M. Poncino. Native ISS-SystemC Integration for the Co-Simulation of Multi-Processor SoC. In *In the Proc. of IEEE Conference on Design Automation and Test in Europe*, pages 564–569, 2004.

153. I. Moussa, T. Grellier, and G. Nguyen. Exploring sw performance using soc transaction-level modelling. In *Proceedings of IEEE Design Automation and Test in Europe (DATE)*, pages 120–125, 2003.
154. http://www.gnu.org/software. Technical report, GDB.
155. A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, and O. Yamamoto. A hardware-software co-simulator for embedded system design anddebugging. In *In the Proc. of IEEE Asian and South Pacific Design Automation Conference*, pages 155–164.
156. S. Yoo and K. Choi. Optimistic Timed HW-SW Cosimulation. In *In the Proc. of Asia-Pacific Conference on Hardware Description Language*, pages 39–42, 1997.
157. W. Sung and S. Ha. Optimized timed hardware software cosimulation without roll-back. In *In the Proc. of the IEEE of Design Automation and Test in Europe*, pages 945–946. IEEE Computer Society Washington, DC, USA, 1998.
158. D. Kim, C. E. Rhee, Y. Yi, S. Kim, H. Jung, and S. Ha. Virtual synchronization for fast distributed cosimulation of dataflow task graphs. In *In the Proc. of the 15th International Symposium on System Synthesis*, volume 2, pages 174–179, 2002.
159. Y. Yi, D. Kim, and S. Ha. Virtual synchronization technique with OS modeling for fast and time-accurate cosimulation. In *In the Proc. of IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–6, 2003.
160. H. Muhr, R. Holler, and M. Horauer. A Heterogeneous Hardware-Software Co-Simulation Environment Using User Mode Linux and Clock Suppression. In *In the Proc. of IEEE/ASME International Conference*, pages 1–6, 2006.
161. G. Nicolescu, L. Gauthier, and A. Jerraya. Fast timed cosimulation of HW/SW implementation of embedded multiprocessor SoC communication. In *In the Proc. of IEEE International Workshop on High Level Design Validation and Test*, pages 79–82, 2001.
162. M. Bacivarov, S. Yoo, and A. Jerraya. Timed HW-SW cosimulation using native execution of OS and application SW. In *In the Proc. of IEEE High-Level Design Validation and Test Workshop, 2002*, pages 51–56, 2002.
163. S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. Jerraya. Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer. In *In the Proc. of IEEE Design Automation and Test in Europe*, volume 1, pages 550–555. IEEE Computer Society Washington, DC, USA, 2003.
164. Mark G. Wallace. Constraint programming. In *The Handbook of Applied Expert Systems.* CRC Press, 1997.
165. Robert Kowalski. Algorithm = logic + control. In *Communications of the ACM*, pages 424–436, 1979.
166. Krzysztof R. Apt and Mark G. Wallace. *Constraint Logic Programming using Eclipse.* Cambridge Univeristy Press, 2006.
167. http://www.eclipse-clp.org. Technical report, ECLiPSe.
168. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publ., 1993.
169. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *International Journal on Software Tools for Technology T ransfer (STTT)*, 2(4), March 2000.
170. A. Biere, A. Cimatti, E. .M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, 1999.
171. F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In *Proc. of CAV 2001*, LNCS, pages 436–453, 2001.

172. A. Borälv. A Fully Automated Approach for Proving Safety Properties in Interlocking Software Using Automatic Theorem-Proving. In S. Gnesi and D. Latella, editors, *Proc. of the Second International ERCIM FMICS*, Pisa, Italy, July 1997.
173. S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *Proc. COMPOS*, 1997.
174. R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient BDD algorithms for FSM synthesis and verification. In *Proc. IEEE/ACM International Workshop on Logic Synthesis*, Lake Tahoe (NV), May 1995.
175. A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In *Proc. WMCAI 2002*, number 2294 in LNCS, pages 182–195, 2002.
176. E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proc. of IJCAR 2001*, volume 2083 of *LNCS*, pages 347–363. Springer, 2001.
177. O. Shtrichman. Tuning SAT checkers for bounded model-checking. In *Proc. 12th International Computer Aided Verification Conference (CAV'00)*, 2000.
178. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the 39th Design Automation Conference*, June 2001.
179. The Open Source Organization. http://www.opensource.org.
180. Massimo Bombana and Francesco Bruschi. Systemc-vhdl co-simulation and synthesis in the hw domain. In *Proc. of IEEE DATE*, pages 106–111, 2003.
181. C. Cote and Z. Zilic. Automated systemc to vhdl translation in hardware/software codesign. In *Proc. of IEEE ICECS*, pages 717–720, 2002.
182. Luis Alejandro Cortés, Petru Eles, and Zebo Peng. Verification of embedded systems using a petri net based representatio n. In *Proc. of IEEE ISSS*, pages 149–155, 2000.
183. Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.
184. Felice Balarin and Roberto Passerone. Specification, synthesis and simulation of transactor processes. *IEEE Trans. on CAD*, accepted for future publication.
185. Alexandre Chureau, Yvon Savaria, and El Mostapha Aboulhamid. The role of model-level transactors and uml in functional prototyping of systems-on-chip: A software-radio application. In *Proc. of IEEE DATE*, pages 698–703, 2005.
186. Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. A methodology for abstracting rtl designs into tl descriptions. In *Proc. of IEEE MEMOCODE*, pages 103–112, 2006.
187. Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. Towards equivalence checking between tlm and rtl models. In *Proc. of IEEE MEMOCODE*, pages 113–122, 2007.
188. F. Fummi, M. Boschini, X. Yu, and E.M. Rudnick. Sequential circuit test generation using a symbolic/genetic hybrid approach. *Journal of Electronic Testing*, 17(3-4):321–330, 2001.
189. U. Uyar and A.Y. Duale. Modeling VHDL specifications as consistent EFSMs. In *Proc. of IEEE MILCOM*, pages 740–744, 1997.
190. U. Uyar and A.Y. Duale. Resolving inconsistencies in EFSM-modeled specifications. In *Proc. of IEEE MILCOM*, pages 135–139, 1999.
191. D. Gajski, J. Zhu, and R. Domer. Essential issue in codesign. Thecnical report ICS-97-26, University of California, Irvine, 1997.
192. D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proc. of ACM STOC*, pages 264–267, 1992.
193. F. Fummi, C.Marconcini, and G.Pravadelli. Functional verification based on the EFSM model. In *Proc. of IEEE HLDVT*, pages 69–74, 2004.

194. W. H. Kautz. The necessity of closed circuit loops in minimal combinational circuits. *IEEE Trans. Comput.*, 19(2):162–164, 1970.
195. R. L. Rivest. The necessity of feedback in minimal monotone combinational circuits. *IEEE Trans. Comput.*, 26(6):606–607, 1977.
196. Marc D. Riedel and Jehoshua Bruck. The synthesis of cyclic combinational circuits. In *Proceedings of the 40th Design Automation Conference, DAC 2003*, pages 163–168, Anaheim, CA, USA, June 2003.
197. John Backes, Brian Fett, and Marc D. Riedel. The analysis of cyclic circuits with boolean satisfiability. To appear, 2008.
198. High Time for High-Level Test Generation. Panel at IEEE ITC, 1999.
199. A. Fin and F. Fummi. Genetic Algorithms: the Philosopher's Stone or an Effective Solution for High-Level TPG? In *Proc. of IEEE HLDVT*, pages 163–168, 2003.
200. G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. EFSM Manipulation to Increase High-Level ATPG Efficiency. In *Proc. of IEEE ISQED*, pages 57–62, 2006.
201. A. Chepurov, G. Di Guglielmo, F. Fummi, G. Pravadelli, J. Raik, R. Ubar, and T. Viilukas. Automatic generation of EFSMs and HLDDs for functional ATPG. In *Proc. of IEEE International Biennal Baltic Electronics Conference (BEC'08)*, pages 143–146, October 6-8, 2008.
202. D. Bresolin, G. Di Guglielmo, F. Fummi, G. Pravadelli, and T. Villa. The impact of EFSM Composition on Functional ATPG. In *In the Proc. of 12th IEEE Symposium on Design and Diagnostics of Electronic Systems (DDECS'09)*, April 2009.
203. A.L. Courbis and J.F. Santucci. Pseudo-Random Behavioral ATPG. In *Proc. of ACM/IEEE GLSVSLI*, pages 192–195, 1995.
204. A. Yamani and E.J. McCluskey. Built-in Reseeding for Serial BIST. In *Proc. of IEEE VTS*, pages 63–68, 2003.
205. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.
206. R.M. Stallman and G.J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer Aided Circuit Analysis. *Artificial Intelligence*, 9:135–196, 1977.
207. M.K. Iyer, G. Parthasarathy, and K.T. Cheng. SATORI - a Fast Sequential SAT Engine for Circuits. In *Proc. of IEEE ICCAD*, pages 320–325, 2003.
208. B. Li, M.S. Hsiao, and S. Sheng. A novel SAT all-solutions solver for efficient preimage computation. In *Proc. of IEEE DATE*, pages 272–277, 2004.
209. C. Wang, S.M. Reddy, I. Pomeranz, X. Lin, and J. Rajski. Conflict Driven Techniques for Improving Deterministic Test Pattern Generation. In *Proc. of ACM/IEEE ICCAD*, pages 87–93, 2002.
210. R.M. Hierons, T.-H. Kim, and H. Ural. Expanding an extended finite state machine to aid testability. In *Proc. of IEEE COMPSAC*, pages 334–339, 2002.
211. A.Y. Duale and U. Uyar. A method enabling feasible conformance test sequence generation for EFSM models. *IEEE Trans. on Computers*, 53(5):614–627, 2004.
212. http://www.graphviz.org. Technical report, DOT Language and Graph Representation.
213. http://www.boost.org. Technical report, Boost C++ Libraries.
214. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Egineering an efficient sat solver. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 530–535, 2001.
215. M. Wallace and A. Veron. Two problems-two solutions: one system-ECLIPSE. In *IEE Colloquium on Advanced Software Technologies for Scheduling*, pages 1–3, 1994.
216. F. Ferrandi, F. Fummi, and D. Sciuto. Implicit test generation for behavioral vhdl models. In *Proc. of IEEE ITC*, pages 436–441, 1998.

217. I. Ghosh and M. Fujita. Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(3):402–415, 2001.
218. F. Corno, G. Cumani, Matteo Sonza Reorda, and G. Squillero. Effective Techniques for High-Level ATPG. In *Proc. of IEEE ATS*, pages 225–230, 2001.
219. L. Zhang, I. Ghosh, and M. Hsiao. Efficient Sequential ATPG for Functional RTL Circuits. In *Proc. of IEEE ITC*, pages 290–298, 2003.
220. L. Lingappan, S. Ravi, and N.K. Jha. Test generation for non-separable RTL controller-datapath circuits using a satisfiability based approach. In *Proc. of IEEE ICCD*, pages 187–193, 2003.
221. Fei Xin, M. Ciesielski, and I.G Harris. Design validation of behavioral VHDL descriptions for arbitrary fault models. In *Proc. of IEEE ETS*, pages 156–161, 2005.
222. Q. Wu and M.S. Hsiao. Efficient ATPG for design validation based on partitioned state exploration histories. In *Proc. of IEEE VTS*, pages 389–394, 2004.
223. M.K. Ier, G. Parthasarathy, and K.-T. Cheng. Efficient conflict-based learning in an RTL circuit constraint solver. In *Proc. of IEEE DATE*, pages 666–671, 2005.
224. S. Padmanabhuni. Extended analysis of intelligent backtracking algorithms for the maximal constraint satisfaction problem. In *Proc. of IEEE CCECE*, pages 1710–1715, 1999.
225. S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2002.
226. Xijiang Lin, Irith Pomeranz, and Sudhakar M. Reddy. Techniques for improving the efficiency of sequential circuit test generation. In *Proc. of ACM/IEEE ICCAD*, pages 147–151, 1999.
227. G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. Improving Gate-Level ATPG by Traversing Concurrent EFSMs. In *Proc. of IEEE VTS*, 2006.
228. E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
229. A. Fin and F. Fummi. Genetic algorithms: the philosophers stone or an effective solution for high-level TPG? In *Proc. of IEEE HLDVT*, pages 163–168, 2003.
230. G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. A Pseudo-Deterministic Funcional ATPG based on EFSM Traversing. In *In* Proc. of MTV, 2005.
231. G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. Improving high-level and gate-level testing with fate: a functional atpg traversing unstabilized efsms. *Computers & Digital Techniques, IET*, 1:187–196, May 2007.
232. G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. Fate: a functional atpg to traverse unstabilized efsms. In *Proc. of IEEE ETS*, 2006.
233. C. Marconcini. A Functional ATPG as a bridge between Functional Verification and Testing. In *Ph.D. thesis*, April 2008.
234. Chien-In Henry Chen and Tim H Noh. VHDL behavioral ATPG and fault simulation of digital systems. *IEEE Transaction on Aerospace and Electronic Systems*, 34(2):428–447, 1998.
235. F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero. An RT-level fault model with high gate level correlation. In *Proceedings of IEEE International High-level Design Validation and Test Workshop (HLDVT)*, pages 3–8, 2000.
236. Fulvio Corno, Paolo Prinetto, and Matteo Sonza Reorda. Testability analysis and ATPG on behavioral RT-level VHDL. In *Proceedings of IEEE ITC*, pages 753–759, 1997.
237. F. Ferrandi, F. Fummi, and D. Sciuto. Test generation and testability alternatives exploration of critical algorithms for embedded applications. *IEEE Transactions on Computers*, C-51(2):200–215, 2002.

238. Olga Goloubeva, G Jervan, Zebo Peng, and Matteo Sonza Reorda. High-level and hierarchical test sequence generation. In *Proceedings of IEEE HLDVT*, pages 169–174, 2002.
239. F Fallah, S Devadas, and K Keutzer. OCCOM-efficient computation of observability-based code coverage metrics for functional verification. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 20(8):1003–1015, 2001.
240. M.B. Santos, F.M. Gonalves, I.C. Teixeira, and J.P. Teixeira. Implicit functionality and multiple branch coverage (IFMB): a testability metric for RT-level. In *Proceedings of IEEE Internationa Test Conference (ITC)*, pages 377–385, 2001.
241. F. Fummi, G. Pravadelli, A. Fedeli, U. Rossi, and F. Toto. On the use of a high-level fault model to check properties incompleteness. In *In the Proc. of Formal Methods and Models for Co-Design, 2003. MEMOCODE'03. Proceedings. First ACM and IEEE International Conference*, pages 145–152, 2003.
242. A. Fedeli, F. Fummi, and G. Pravadelli. Properties Incompleteness Evaluation by Functional Verification. *IEEE Transactions on Computers*, pages 528–544, 2007.
243. L. Di Guglielmo, F. Fummi, and G. Pravadelli. Vacuity Analysis by Fault Simulation. In *In the Proc. of Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference*, pages 27–36, 2008.
244. N. G. Leveson and J. L. Stolzy. Safety Analisys using Petri Nets. *IEEE Transactions on Software Engineering*, 13(3):386–397, 1987.
245. J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., 1981.
246. B. J. Choi et al. The mothra tool set. In *In the Proc. of 22nd Hawai International Conference on Systems and Software*, January 1989.
247. R. S. Pressman. *Software Engineering - A Practitioner's Approach, (3rd edition).* McGraw-Hill, 1992.
248. V. Linnenkugel and M. Miillerburg. Test Data Selection Criteria for (Software) Integration Testing. In *In the Proc. of the First International Conference on Systems Integration*, April 1990.
249. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11:34–41, April 1978.
250. F. Ferrandi, F. Fummi, L. Gerli, and D. Sciuto. Symbolic functional vector generation for VHDL specifications. In *Proc. of IEEE DATE*, pages 442–446, 1999.
251. G J Myers. *The Art of Software Testing.* Wiley - Interscience, 1999.
252. F. Corno, P. Prinetto, and M. Sonza Reorda. Testability analysis and atpg on behavioral rt-level vhdl. In *Proceedings of IEEE International Test Conference (ITC)*, pages 753–759, 1997.
253. A. Fin, F. Fummi, and G. Pravadelli. Amleto: A multi-language environment for functional test generation. In *Proceedings of IEEE International Test Conference (ITC)*, pages 821–829, 2001.
254. F. Fummi, C. Marconcini, and G. Pravadelli. Logic-level mapping of high-level faults. *INTEGRATION, the VLSI Journal*, 38:467–490, 2004.
255. O. Baruch and S. Katz. Partially Interpreted Schemas for CSP Programming. *Science of Computer Programming*, 10:1–18, February 1988.
256. Chien-In Henry Chen. Behavioral test generation/fault simulation. *Potentials, IEEE*, 22(1):27–32, Feb.-Mar. 2003.
257. M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital systems testing and testable design.* IEEE Press, 1992.
258. G. Pfister. The Yorktown Simulation Engine. In *Proc. of 19th ACM/IEEE Design Automation Conf.*, 1982.

259. Y. M. Levendel, P. R. Menon, and S. H. Patel. Parallel fault simulation using distributed processing. In *Bell Syst. Tech. J.*, volume 62, no. 10, pages 3107–3130, Dec. 1983.
260. T. Blank. A survey of hardware accelerators used in computer aided design. *IEEE Design & Test Comput.*, Aug. 1984.
261. S. Seshu. On an Improved Diagnosis Program. *IEEE Trans. on Electronic Computers*, EC-12(2):76–79, Feb. 1965.
262. E. G. Ulrich and T. G. Baker. Concurrent Simulation of Nearly Identical Digital Networks. *Computer*, 7(4):39–44, April 1974.
263. D. B. Armstrong. A deductive method for simulating faults in logic circuits. In *IEEE Trans. Comput.*, volume C-21, no. 5, pages 464–471, May 1972.
264. H. C. Godoy and R. E. Vogelsberg. Single Pass Error Effect Determination (SPEED). *IBM Technical Disclosure Bulletin*, 13:3344–3443, April 1971.
265. P. C. Patton. Multiprocessors: Architectures and applications. *Computer*, 18(6):29–40, 1985.
266. E. W. Thompson and S. A. Szygenda. Digital logic simulation in a time-based, table-driven environment: part 2. parallel fault simulation. In *Comput.*, volume 8, pages 38–49, Mar. 1975.
267. S. Kyushik. Fault simulation with the parallel valued list algorithm. In *VLSI Syst. Design*, pages 36–43, Dec. 1985.
268. P. R. Moorby. Fault simulation using parallel valued lists. In *Proc. IEEE Int. Conf. Computer-Aided Design*, pages 101–102, 1983.
269. W. T. Cheng and M. L. Yu. Differential fault simulation - A fast method using minimal memory. In *Proc. Design Automation Conf.*, pages 424–428, June 1989.
270. T. M. Niermann, W. T. Cheng, and J. H. Patel. PROOFS: A fast memory-efficient sequential circuit fault simulator. In *IEEE Trans. Computer Aided Design*, volume I-1, pages 198–207, Feb. 1992.
271. C. Lpez, T. Riesgo, Y. Torroja, E. de la Torre, and J. Uceda. A method to perform error simulation in VHDL. *Design of Circuits and Integrated Systems Conference*, 1998.
272. S. Gosh and T. J. Chakraborty. On Behavior Fault Modeling for Digital Design. *Journal of Electronic Testing: Theory and Applications (JETTA)*, 2(2):31–42, June 1991.
273. F. Ferrandi, F. Fummi, L. Gerli, , and D. Sciuto. Symbolic functional vector generation for VHDL specifications. In *IEEE DATE*, pages 226–446, 1999.
274. F. Ferrandi, G. Ferrara, S.Sciuto, A. Fin, and F. Fummi. Functional test generation for behaviorally sequential models. In *Design, Automation, and Test in Europe*, pages 403–410, 2001.
275. A. Jefferson Offutt and Ronald H. Untch. *Mutation 2000: uniting the orthogonal.* Kluwer Academic Publishers, 2001.
276. A. Jefferson Offutt. A practical system for mutation testing: help for the common-programmer. In *International Test Conference*, pages 824–830, Oct. 1994.
277. A. Castelnuovo, A. Fedeli, A. Fin, F. Fummi, G. Pravadelli, U. Rossi, F. Sforza, and F. Toto. A 1000X Speed Up for Properties Completeness Evaluation. In *Proc. of IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 18–22, 27-29 Octorber 2002.
278. Advanced micro devices. 1978. the am2910, a complete 12-bit microprogram sequence controller.
279. Hc11. http://www.gmvhdl.com/hc11core.html.
280. High time for high-level test generation. Panel at IEEE ITC, 1999.
281. A. Fin and F. Fummi. A vhdl error simulator for functional test generation. In *Proceedings of IEEE Design Automation and Test in Europe (DATE)*, pages 597–600, 2000.

282. G. Buonanno, L. Ferrandi, F. Ferrandi, F. Fummi, and D. Sciuto. How an evolving fault model improves the behavioral test generation. In *Proceedings of ACM Great Lake Symposium on VLSI (GLSVLSI)*, pages 124–129, 1997.
283. K.T. Cheng and J.Y. Jou. Functional test generation for finite state machines. In *Intl. Test Conference*, pages 162–168, 1990.
284. F. Fummi, M. Loghi, G. Perbellini, and M. Poncino. SystemC co-simulation for core-based embedded systems. *Design Automation for Embedded Systems*, 11(2):141–166, 2007.
285. Vertigo - European Project. http://www.vertigo-project.eu.
286. STMicroelectronics. http://www.st.com.
287. G. Braun et al. A universal technique for fast and flexible instruction-set architecture simulation. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 23(12):1625–1639, December 2004.
288. M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina. Mixed SW/SystemC SoC Emulation Framework. In *ISIE 2007: International Symposium on Industrial Electronics*, pages 2338–2341, June 2007.
289. http://fabrice.bellard.free.fr/qemu. Technical report, QEMU Emulator.
290. IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual. In *IEEE Std 1666-2005*.
291. F. Fummi, G. Perbellini, M. Loghi, and M. Poncino. ISS-centric modular HW/SW co-simulation. In *In the Proc. of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI '06)*, pages 31–36, April 2006.
292. A. Rubini and J. Corbet. *Linux Device Drivers, 2nd Edition.* O'Reilly, June 2001.