Quality Metrics Driven Functional Verification for IP based SoC Design

Dissertation

A thesis submitted to the
Faculty of Computer Science, Mathematics and Electrical Engineering
of the University of Paderborn in partial fulfillment
of the requirements for the degree of Dr. ret. nat.

by

Tao Xie

Paderborn, October 2013

Supervisors:

1. Prof. Dr. rer. nat. Franz J. Rammig, University of Paderborn
2. Prof. Dr. rer. nat. Sybille Hellebrand, University of Paderborn

Abstract

System-on-a-Chip (SoC), centered at reuse of silicon Intellectual Properties (IPs) and characterized by separation of IP development and SoC system integration, becomes a dominant paradigm for designing electronic systems. Complexity of both IP and SoC system design grows exponentially and challenges the functional verification of these designs. In this context, we consider it a necessity to have a systematic management of verification quality by applying quantitative metrics. Therefore, the dissertation has the general goal of establishing a beyond-state-of-the-art, *metrics-driven verification methodology* that i) employs automated methods to efficiently improve the verification quality measured under such metrics and ii) extends the application of these metrics to accommodate emerging SoC system-level design language. *Mutation analysis* is the focused metric in this research for developing new methods. It has a unique, complex test generation problem to detect (*kill*) an error-injected design (called a *mutant*).

At IP level, verification handles designs in traditional hardware description languages (HDLs) and *mutant-targeted automatic test generation* is the main objective. Firstly, random simulation is considered appropriate for achieving a primary level of verification quality under mutation analysis, where we see the specific problem that random test generation becomes inefficient as being not metrics-tailored. An *adaptive random simulation* method is developed. Based on a modeling of random tests with Markov chain and constraints, the simulation process is continuously steered by a heuristic towards tests that are regarded more efficient in killing mutants. The experiments show that this adaptive simulation is effective of having more mutants killed with less simulation.

Secondly, with a portion of the mutants expected to be un-killed after random simulation, we solve the problem of further generating tests that kill each individual mutant. A *search-based test generation* method is developed, using real simulation results to guide an iterative process of finding a target test. An objective cost function is defined specifically for HDL mutation analysis, which calculates the progress of a test killing a mutant. In the experiments, the cost function, when used to equip a local search algorithm, delivers consistent performance for steering the search towards mutant-killing tests.

At SoC system level, an *IP-XACT mutation analysis* framework is developed, assuming IP-XACT as the default language for SoC integration. Here, first, since IP-XACT designs as XML data are not simulatable, a simulation engine for IP-XACT, in the form of an IP-XACT-to-SystemC generator that incorporates Transaction-Level Modeling, is built as the verification basis. Second, IP-XACT mutation operators are defined by compiling a table of possible error injections on the IP-XACT schema. The experiments, using an Eclipse-based tool implementation, shows that the proposal is practical and enables verification of IP-XACT SoC designs as well as quality measurement of such verification via mutation analysis.

Contents

CHAPTER 1: Introduction

The chapter presents the general research challenges that motivate this thesis.

1.1. Functional Verification Challenge

In the research area of *Electronic Design Automation* (EDA), *functional verification*, where the *functional correctness of a design is verified against its specification*, is widely regarded as the bottleneck of development and facing unsolved challenges [14] [15] [16]. Along the years, various automation techniques are proposed to tackle verification challenges. However, since, on the one hand, the increase of design complexity seems unstoppable and, on the other hand, new design languages and paradigms emerge alongside this complexity increase, novel verification methodology has always been needed to accommodate the changes.

For example, the functional verification of a micropcessor design should verify whether the design correctly executes sequences of instructions that are specified by its instruction set architecture (ISA). As the complexity of microprocessors increases following the *Moore's Law* – from the first commercial Intel 4004 processor containing about 2,300 transistors [17] to many over 1 billion nowadays, the design's state space that we need to verify increases *exponentially*, which is known as the *state-space explosion* problem in verification. This then suppresses the amount of design that we can verify.

A *design productivity gap* is depicted in **Figure 1.1** [18], which refers to the ever expanding gap between the Moore's Law and design productivity, i.e. between the number of gates, or transistors that *can* be manufactured into a single chip and the number of gates that we are actually able to accomplish in a chip design project, described in gates-per-day. The *International Technology Roadmap for Semiconductors* (ITRS) updates this graph every two years in their *design chapter*, as a high-level view of electronics design challenges. The use of semiconductor *Intellectual Properties* (IP) is also mentioned in this graph as a productivity promotion, which makes the gap not worse. IP-reuse will be a topic of next section.

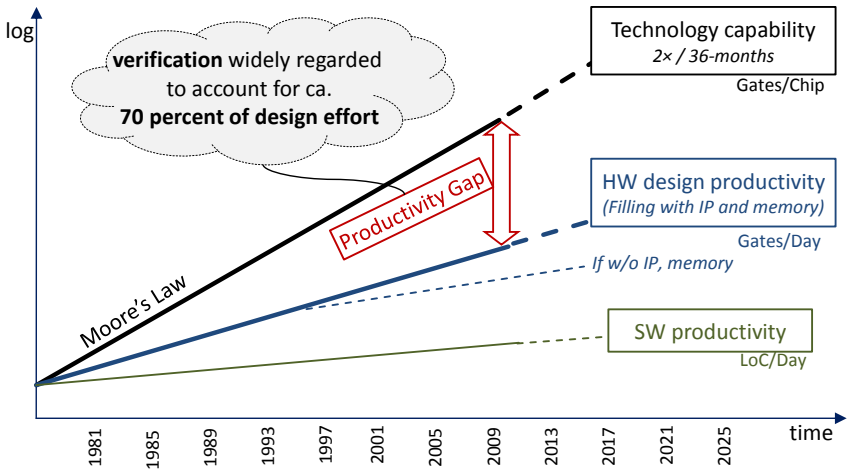


Figure 1.1 International Technology Roadmap for Semiconductors (ITRS) 2011: productivity gap [18].

In many occasions [14] [15] [19] [20], the effort spent on verification is estimated to account for 70% of the entire design activity, if not more. Considering that verification occupies a constant and large portion of design effort, we may also find a *verification gap* contained in the overall productivity gap.

In this context, a more specific question can be asked:

When can we say that the verification is done?

Accordingly, we may define the *verification closure* problem as finding *a point that we are certain of incompleteness and incorrectness no longer existing in the design under verification*. On the one hand, this confidence is partly a *subjective* matter. On the other hand, it is our research task to find an objective and systematic solution. For this, we may further consider two questions:

- How can we *effectively measure* the completeness, or thoroughness, or quality of our verification?
- How can we *efficiently improve* the verification quality under such measurement?

In this work, we use simulation for functional verification. We consider building a simulation-based verification methodology that i) relies on well-established coverage metrics to systematically manage the simulation quality and ii) employs novel methods for automatic simulation tests generation that targets the metrics. Therefore, we call this *quality metrics driven functional verification*.

In particular, we intend to leverage a well-researched, state-of-the-art metric for HDL (Hardware Description Language) simulation: *mutation analysis*, which has been implemented by, for example, a recent EDA tool *Certitude* [21] [22] [23] from Synopsys.

Exactly meant as an aid to answer the verification closure problem, mutation analysis gives *a quantitative, objective quality measure on simulation tests, by injecting artificial but typical errors into a design under verification, and assessing how many of these errors can be revealed by the tests.* The individual metric points are called mutants.

Further, the thesis is focused on functional verification. *No* observation on non-functional properties is considered, such as *power* or *performance*.

1.2. System-on-a-Chip Challenge

Nowadays, we are seeing increasingly more electronic systems in the form of *System-on-a-Chip* (SoC, or System-on-Chip), where a system is built into a single *integrated circuit* (IC) chip, instead of on a *printed circuit board* (PCB).

CoreConnect, as shown in **Figure 1.2** is an on-chip bus architecture proposed around 2000 by IBM for SoC integration [24], which is widely used ever since. The Processor Local Bus (PLB) bus provides separate 32-bit address and up to 128-buit data buses. With a fully synchronous architecture, PLB can be connected with multiple masters and slaves. High-throughput system cores, such as microprocessors, memory controllers, and Direct Memory Controller (DMA), are supported by PLB. Other peripheral cores such as a UART (Universal Asynchronous Receiver/Transmitter) controller can be connected to the low-bandwidth On-chip Peripheral Bus (OPB). Another Device Control Register (DCR) bus is intended specifically for register data move between a microprocessor and configuration registers of other components, so as to free the bandwidth of PLB.

CoreConnect will be frequently used in our examples and case studies. Here, it gives a first illustration of what defines a SoC: *higher system integration on the chip level*.

Previously, system components are implemented as separate ICs and then interconnected on a PCB. Now, both computation and communication components are

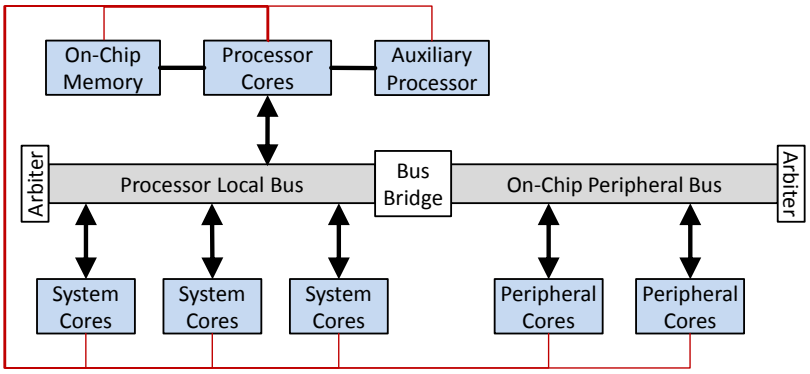


Figure 1.2 CoreConnect on-chip bus architecture [24].

integrated on-chip. They *comprise an integrated design, to be verified, synthesized, and then manufactured as a single chip*.

Arguably, SoC is *more of a design paradigm* than a perfect reality, since in the end, most systems still need to be embedded as PCBs.

Network-on-Chip (NoC) is another form of SoC, where system-level integrated components communicate with each other through on-chip network and routers. **Figure 1.3** shows the *FAUST (Flexible Architecture of Unified System for Telecom)* NoC [25] [26], which we have employed in a recent European research project *COCONUT* (*A Correct-by-Construction Workbench for Design and Verification of Embedded Systems*) [27]. 23 IP blocks are included and connected to a network of 20 nodes, resulting in a complexity of 8 M-gates.

Using a router-based, asynchronous network for communication, higher scalability and data throughput are expected. Therefore, it is intended for dataflow-intensive, especially 4G-radio-targeted applications. The chip is categorized as a SoC, as it integrates most system components that are previously off-chip now into a single IC, including, for example, an ARM microprocessor, memory controllers, radio communications such as OFDM and CDMA, and the network routers. More details on this NoC and its applications can be found in [25].

In the COCONUT project, a high-level, *Transaction-Level Modeling (TLM)* [28] based model of this FAUST NoC has been employed as a target platform, to develop a TLM based SoC design methodology. One of the project results is a TLM-based, RTOS

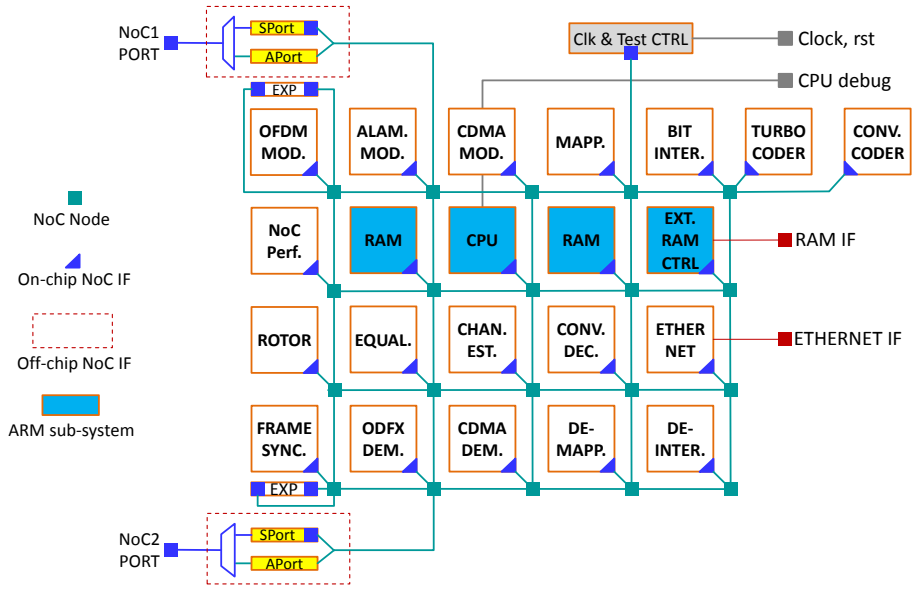


Figure 1.3 FAUST NoC [25]. A TLM model for this chip has been employed in the *COCONUT* project to create a TLM based SoC design methodology.

(real-time operating system)–aware SoC refinement flow [8]. The previously mentioned *Certitude* tool has also been involved in this project, for managing the verification quality part of the design methodology.

In this work, we do *not* particularly differentiate NoC and SoC, yet with an emphasis on traditional master-slave bus architectures.

SoC Design is Centered at IP-reuse

A prominent characteristic of SoC design is that *it is centered at component reuse*. Here, component means *design components*, instead of fabricated devices. These components are called semiconductor *Intellectual Property* (IP) in this context. Types of IPs include encryption/decryption cores, video/audio codecs, telecommunication network controllers – wired or radio-based, memory controllers, digital signal processors, general purpose microprocessors, and so on.

Around IP reuse, two roles can be defined for SoC development. One is *IP vendor*, whose task is to design and deliver an IP component for some specific functionality. The other one is *SoC integrator* that takes a wide range of IPs as input and integrates them into a complete system capable of hosting applications. At SoC integration phase, an IP can come either from an internal design group, or from an external IP vendor.

This IP-centered SoC design paradigm has its significant impact on verification. Besides the general *verification gap* from the increasing complexity of both IPs and SoCs, we face these particular challenges:

- *Separation of IP design and SoC system design leads to more stringent requirement on the quality of IP verification*. An IP design must be verified as thoroughly as possible before its delivery to any SoC integration phase, when the in-system debugging would become more difficult because of the SoC complexity, if not entirely impossible when the IP is provided as a black-box without source code.
- *Verification at SoC system level should accommodate new paradigms and languages for SoC design*. TLM is one example that we have just mentioned with regard to project COCONUT. *IP-XACT* is another XML-based, IEEE standard format specifically for describing IP reuse and SoC integration [29], which has been seeing increasing acceptance [30]. By aligning verification to SoC-specific languages, we will be able to *focus verification on system-level integration* and cope with the complexity of SoCs.

These general motivations will be further elaborated alongside the background presentation in next chapter, before we propose our methodology to meet the challenges.

1.3. Thesis Goal and Organization

Therefore, the thesis tries to provide one step towards solving the functional verification challenge, in the context of system-on-a-chip becoming a prevailing design paradigm.

We conclude this introduction chapter with the following considerations. We also present the concrete problems to be solved in the rest of the thesis.

- To meet the functional verification challenge, we consider systematic application and deployment of quality metrics to be a necessity. In particular, such application of metrics should consistently cover both IP and SoC system verification stages.
- We consider mutation analysis, as a well-researched, state-of-the-art testing technique, to be an advanced metric and the basis on which we build our verification methodology.
- We further consider that emerging system-level languages, such as IP-XACT and TLM, are used for SoC system design and, therefore, should be included in the methodology.

Problems

Test generation is the major problem that we encounter at the stage of IP design verification with mutation analysis.

- Considering that *random simulation* is a widely recognized technique for achieving a primary level of verification quality and should also be used for mutation analysis, we have the problem that random test generation becomes inefficient in the context of metrics-oriented simulation. It is because that i) initially, the random tests are usually *not* modeled for any specific metric and ii) a target metric also changes during simulation as a consequence of its subsets being satisfied. Moreover, mutation analysis is *simulation intensive*, which makes the problem more critical. Therefore, we consider an *adaptive* simulation necessary, able to consistently *steer* a random test generation process towards the mutation metric.
- Expecting a portion of the mutation analysis metric to be unsatisfied after random simulation, we face the problem of further generating tests to *kill* individual mutants. This test generation problem is unique to mutation analysis: tests are required to *reach a mutant, activate it,* and *propagate the erroneous behavior to design output*. Existing methods to the problem are based on symbolic manipulation and *not* as scalable as HDL simulation itself. We consider it necessary to develop a *non-symbolic, purely simulation-based* test generation method for HDL mutation analysis.

Moving to SoC system design, we focus on the following two sub-problems:

- IP-XACT designs as XML data are *not simulatable* and, therefore, present a barrier for us continuing the simulation-based, metrics-driven functional verification at SoC system-level. A simulation engine for IP-XACT SoC designs needs first to be built as the verification basis.
- Then, we find a general *lack of systematic metric* for SoC system verification. Specifically, if we require mutation analysis to be consistently applied also at system level, we should solve the problem of *enabling IP-XACT mutation analysis*, i.e. how IP-XACT design mutants can be created and simulated.

Solutions to these problems will *not* be limited to mutation analysis, but apply to other metrics in the general context of metrics-oriented IP and SoC verification too.

Our solution is called a *metrics-driven methodology*, as i) quantitative metrics are relied on for systematic measurement of verification thoroughness and quality, ii) automation methods are proposed to generate tests and improve such measured quality, and iii) for places where such metrics lack for IP-based SoC design, we tries to create one. The overall contribution can be stated as:

The thesis establishes a verification methodology that systematically manages and automatically improves the quality/thoroughness of a functional design verification process. In particular, it accommodates IP-based SoC design paradigm.

Organization

The thesis is then organized as shown in **Figure 1.4**:

- In Chapter 2, state-of-the-art techniques and methods for IP and SoC design are introduced as the background of our proposals. It follows a thread from *design*, to functional *verification* by simulation, to quality *metrics* for such simulation. In particular, *mutation analysis* as the focused metric is extensively discussed and compared to others.
- In Chapter 3, an overview of our proposals, which comprise *a quality-metrics driven functional verification methodology for IP-based SoC design*, is given.
- From Chapter 4 to Chapter 6, three components of the methodology are presented. Chapter 4 proposes an adaptive random simulation method, which uses mutation analysis results as on-line feedback to dynamically steer a random test generation process, so as to obtain an improved efficiency of mutation analysis. Chapter 5 proposes a search based test generation method for mutation analysis, where an

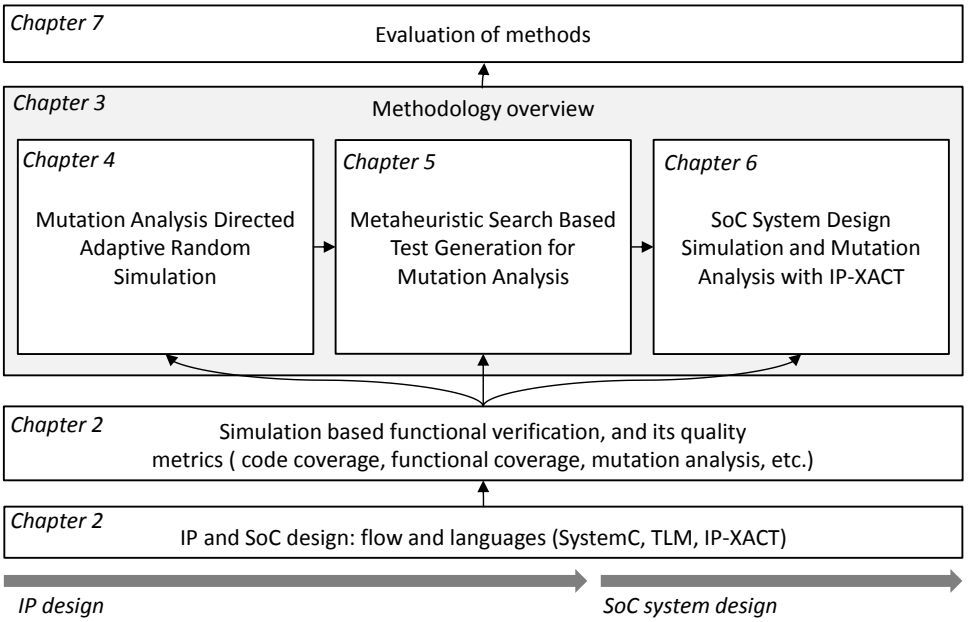


Figure 1.4 Thesis organization.

objective cost function, which is capable of guiding a metaheuristic search algorithm stepwise towards target tests that uncover a HDL mutant, is defined. These two methods are mainly for IP-level designs. In Chapter 6, an IP-XACT based SoC system design simulation and mutation analysis framework is proposed, to address the lack of systematic verification way at SoC system-level. The implementation of a prototype IP-XACT tool, based on Eclipse, is also presented.

- Literature directly related to our proposals is respectively discussed in Chapter 4 through 6.
- In Chapter 7, feasibility, effectiveness, and efficiency of the proposed verification methodology, based on simulation and mutation analysis, are investigated with real designs. IP-level test generation methods are evaluated with a microprocessor design. SoC system-level simulation methods are evaluated by exercising our IP-XACT tool with several CoreConnect/PowerPC SoC designs in TLM.
- In Chapter 8, we give conclusions on the thesis, also addressing some outlook from this research.

CHAPTER 2: Background

In this chapter, we give the background discussion necessary for the identification of what lacks in the state-of-the-art methods and techniques for IP and SoC designs, and further as the basis for our enhancement proposal. The chapter follows the thesis organization presented at the end of last chapter and is divided into three sections: *design*, *verification*, and *metrics for verification*.

- The whole background is unfolded based on a reference flow for IP-based SoC design, which is defined in Section 2.1.1. Advanced, state-of-the-art design techniques and methods are introduced by Section 2.1.2 and 2.1.3, which are focused on SystemC, Transaction Level Modeling, and IP-XACT.
- Discussion on functional design verification is limited to simulation, with common parts and approaches in HDL simulation introduced in Section 2.2. We define *quality metrics driven verification*, an approach that we follow for our verification methods, in Section 2.2.1.
- In Section 2.3, we discuss a wide range of metrics that can be employed in such *metrics driven verification*, with an emphasis on *mutation analysis* that will play a central role in our own methods.

Literature closely related to our contributions will be left to each corresponding chapter, for a better comparison. We further assume some mature languages and methods familiar to readers and not included in this discussion, such as tradition HDLs like VHDL and Verilog, designs at Register Transfer Level (RTL), and their simulation.

2.1. IP and SoC Design

2.1.1. A Reference Flow for IP-based SoC Design

In this Section, we introduce a reference design flow for IP-based SoC design, as shown in **Figure 2.1**. The purpose of the flow is threefold. First, it serves our definition of IP-

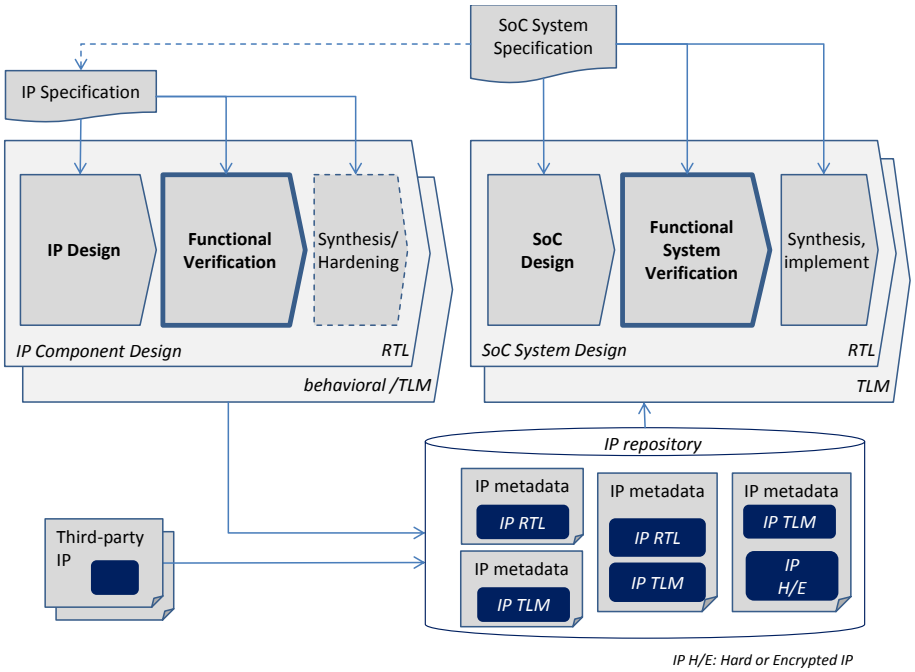


Figure 2.1 A reference IP-based SoC design flow.

based SoC design paradigm, in an abstract manner. Second, it constrains our discussion on design and verification, with regard to background, state-of-the-art methods, and what still lacks. Third, it is the basis flow upon which our proposal of a quality metrics driven verification methodology will be constructed, so that in the end we have an enhanced, integrated flow for IP-based SoC design.

In the figure, our key view of a typical IP-based SoC design flow is the *division and separation of IP design and SoC system integration*, which leads to two separate design phases. Main reasons for this *division and separation* are i) division between IP vendors and SoC integrators and ii) increasing complexity of SoC and larger integration.

It is often the case that for the assembly of a SoC design, the SoC integrator needs one or multiple components as IP from another specific component provider – or IP vendor. Separation of the IP design phase from the whole SoC design flow is straightforward. Even when a component is developed at the same place where the SoC should be assembled, because of the complexity of SoCs nowadays, it is reasonable that a "divide-and-conquer" paradigm is followed.

The specification for an IP does *not* necessarily comes from a SoC system specification. The IP specification defines a specific functionality for a SoC component without, or only partially, considering its final integration into a larger application scenario. An Instruction Set Architecture (ISA) for the implementation of a microprocessor IP can be viewed as a good example of such IP specification, which is quite independent from its final SoC

application, although the target SoC group, for intensive digital signal processing or as leisurely microcontroller, should have some impact on the selection of instruction set. Most importantly, in most cases, we start with the specification, design, and verification of an IP, before we embark on a SoC specification.

The design of an IP component – the first phase in the design flow – consists mainly of the design activity itself, the verification, and design synthesis as well as implementation.

- One important aspect of the flow, in both IP component and SoC system design phases, is the inclusion of a state-of-the-art design technique called *Transaction-Level Modelling* (TLM) [31] [28] [32] [8]. Basically, TLM is a design level with higher abstraction than traditional RTL. It is introduced in Section 2.1.3, together with a language called SystemC, in which TLM is typically conducted. RTL is still the major entry level for many design activities, in particular for IP level designs. Nevertheless, we will spare the space and not give introduction to the quite mature RTL methods and associated HDLs, like VHDL and Verilog. Basics of VHDL and Verilog can be found in [33] [34].
- For IP verification, we consider mainly the aspect of functional design verification, for example, whether a microprocessor design can correctly execute a test program from a specified ISA. Other non-functional properties like timing and power are not considered. Existing functional verification techniques, formal or simulation based, are outlined in Section 2.2, with slightly more focus on simulation based verification.
- The logic synthesis step is optional. There are generally three forms of IPs:
  - ***Soft*-IP**: the IP is provided as its source code.
  - ***Hard*-IP**: the IP is synthesized with a cell library to transistor layout format, for example GDSII [35], or even to a specific fabrication process. This is called an *IP hardening* process.
  - ***Encrypted*-IP**: the IP is provided with its source code, but encrypted. Later for the integration in a SoC, it is supposed to be decrypted by some specific accompanying tool.

  The advantage of a soft-IP is its flexibility for implementation. The advantage of a hard-IP, in contrast, is its predictability, because it is nearer to the implementation. IP hardening and IP protection by encryption are topics not focused in this work. Still, we assume that a hard and encrypted IP is always accompanied with a simulatable model for its integration in system design.
- Techniques on synthesis from a TLM design to RTL and automated abstraction from a RTL design to TLM exist, which can be found in literature [36] [37], for

example. Equivalence checking between RTL and TLM is another verification topic that is *not* covered by this work.

After the exhaustive verification of IP design, the IP is supposed to be delivered to a SoC system integrator, either in-house or a third-party vendor. In both cases, the IP should be imported in an IP repository [38] at the SoC integrator with metadata that document its possible and correct usage in a system integration, such as its on-chip connection interfaces, parameters, and reference to design files. An example later shows how proprietary metadata may look like in a Xilinx IP based SoC design environment for FPGA.

An IP repository may contain IPs in various forms. These include mainly RTL and TLM IPs in our discussion. If an IP is provided as a hard core, it is usually accompanied by a simulation model, say in TLM. Therefore, in a modern flow of IP-based SoC design, the IP metadata format should be capable of both RTL and TLM.

A successful shift to SoC system level can only secured by thorough verification of IP designs and their complete metadata. The system phase has similar steps as IP level – SoC design, verification, and synthesis/implementation.

- As a component based design paradigm, a SoC system description should mainly include the instantiation of IPs as components, their configuration, and their interconnection. The description language or format for this SoC integration further depends on the IP metadata format, since the metadata defines exactly the usage of IP in SoC. Later, we will show this dependence in the example of Xilinx SoC development environment, as well as in the introduction to IP-XACT standard.
- Inclusion of both RTL and TLM IPs implies another requirement that the SoC system design phase should also cover both RTL and TLM, and even an RTL/TLM mixed integration.
- We consider *system simulation* as a necessary step for verifying the functional correctness of a SoC system design, before any of its implementation. This step is also demonstrated in the Xilinx example. We will emphasize the provision of this system simulation as a significant gap for the IP-XACT standard. Although other system verification techniques, such as formal verification and emulation, should complement the simulation, they are *not* the target of our proposal on verification enhancement.
- Targeting a specific implementation technology, whether an ASIC implementation library or a FPGA device, the SoC design can be synthesized and implemented as an integrated circuit. In general, the *circuit testing* step is *not* included in our design flow. However, we will introduced briefly alike test methods that are applied in

circuit testing, such as fault-modeling and fault-aiming automated test generation, which can be compared to methods employed in our quality metric driven design verification.

- The design steps, not only here at SoC system level but also at IP level, can all be iterative. The functional verification certainly needs to be repeated, when a bug is revealed and then corrected in design.

This IP/SoC *division-and-separation* and the resulted two-phase design flow gives a significant impact on the verification aspect – the target of the thesis.

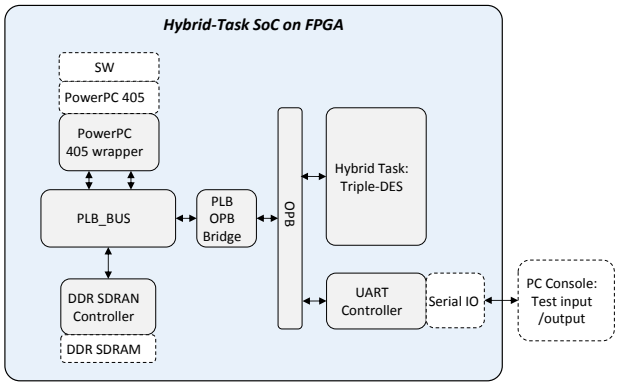
- Since that the design of an IP is separated from system integration, the IP design is required to be verified *as thoroughly as possible*. This thoroughness is only achievable through i) management of the verification process with quantitative, systematic quality metrics and ii) automated methods for improving these metrics.
- Then at the system level, the verification of the SoC design is required to be *focused on the system integration*, mainly as instantiation, configuration, and interconnection of IP *components*. Internal structure of the components may usually be not visible anymore. Any metric on verification quality should also consider a focus on integration.
- The flow implies that a design under verification is *not* always synthesizable, in both IP design and SoC system design.

We will propose our enhancement to this flow with a focus on verification and its quality, considering state-of-the-art design and verification techniques, which are to be introduced in the rest of this chapter. But before that, we present an example instance of the reference design flow.

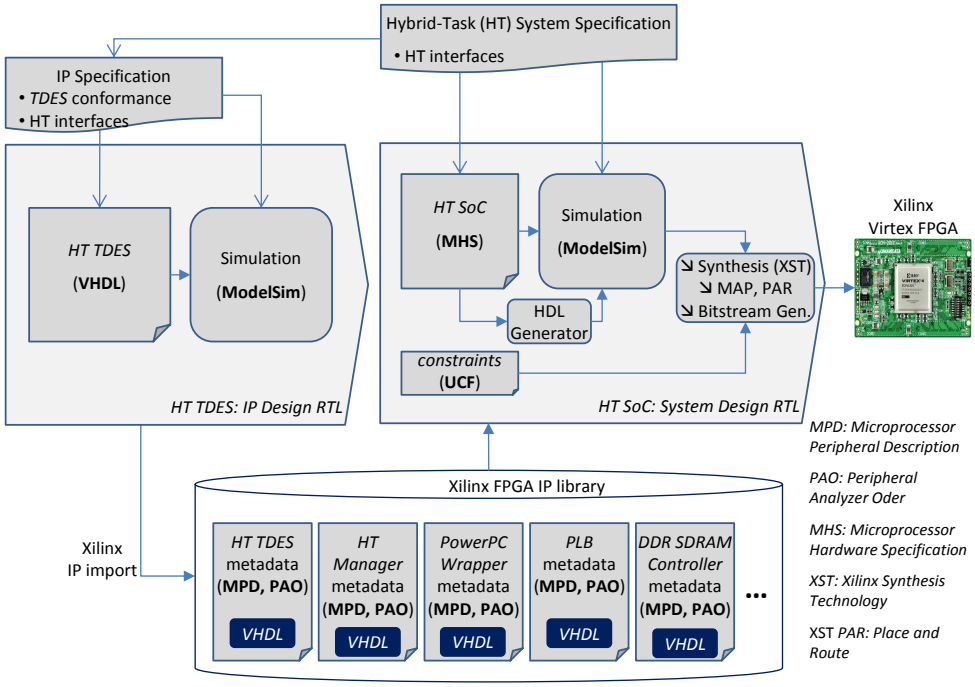
Example: Design of Hybrid-Task SoC with Xilinx FPGA Tools

With **Figure 2.2**, we present a *hybrid-task SoC*. The purpose is, less of presenting the IPs and system themselves, to show the design steps and, in particular, the languages and tools involved. It can be viewed as an instance of the reference design flow presented above. We choose the Xilinx development environment and tools for this example, as they indeed represent a typical and state-of-the-art IP-based SoC design flow, if we do not compare the circuit implementation stage.

We have developed this *Hybrid-task SoC* as a demo system to show the concept of *unified task scheduling and task migration* on a CPU-FPGA coupled platform [13] [11]. The idea is, for example with this hybrid triple-DES task, to enable a design flow with which we are able to obtain two copies of the triple-DES encryption, one for running on



a) The Hybrid-Task SoC



b) Design flow of IP component and SoC system

Figure 2.2 An instance of the reference flow: design of a Hybrid-Task SoC on Xilinx

CPU and the other one for running on FPGA, whose execution can be decided then at runtime by an operating system. The two copies of a so-called *hybrid task* have corresponding states, so that each of them can have its execution suspended, execution states extracted and retrieved, and the states restored to its counterpart for a seamless execution resumption. The reason for such a *hybrid-task migration* between CPU and FPGA can be, for example, some desired *load-balancing* on these two computation hosts.

Focusing on the hardware SoC part, the main system specification is certainly the provision of functionality and interfaces for task migration, such as suspending, resuming, and restarting task execution, as just mentioned.

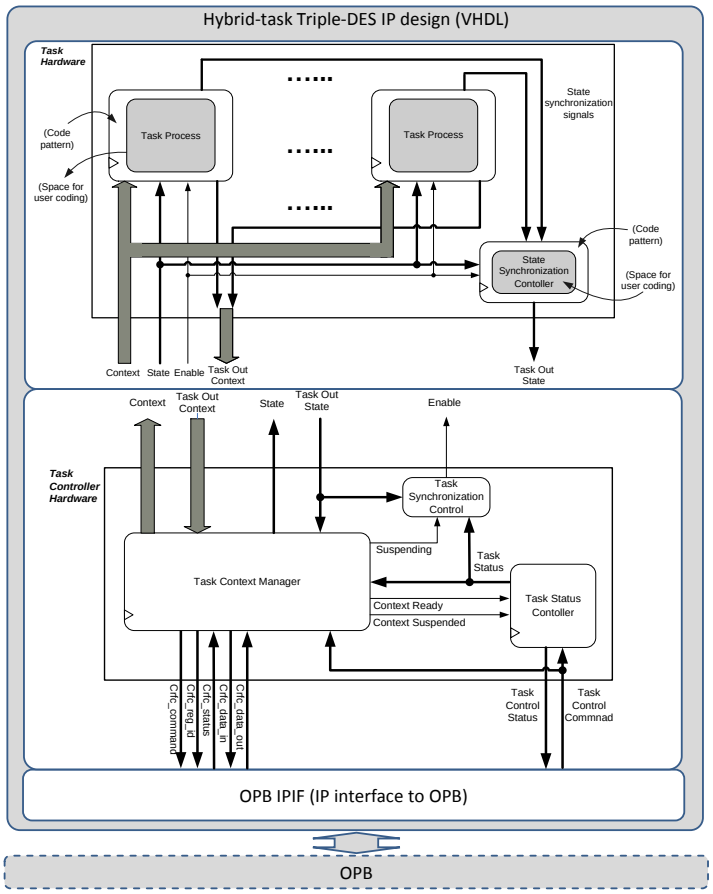


Figure 2.3 **Example IP design: *Hybrid-task Triple-DES*.** It is designed in VHDL, with functionality – encryption and task migration – simulated and verified with tool *ModelSim*.

Before the SoC integration, we made the design of two IP components, the *Hybrid-task Triple-DES* and the *Hybrid-Task Manager*, besides the PowerPC, memory controller, PLB/OPB buses and bridges, and UART controller directly from Xilinx IP repository.

Consider the design of *Hybrid-task Triple-DES* as the example for IP design phase. Two main expectation on the task are the conformance to the Triple-DES encryption standard and its implementation of the hybrid-task interfaces like *suspend* and *resume*.

As shown in **Figure 2.3**, the IP design is done in VHDL. With the DES encryption code taken and reused from open-source [39] and the OPB *IP Interface* (IPIF) generated by Xilinx tool, our design effort was mostly put on the hybrid-task controlling part. Then the IP design as a whole was simulated with tool ModelSim™. Correct production of encryption stream and response to task migration commands are thoroughly verified and debugged in ModelSim. ModelSim will be further mentioned in Section 2.2, with its capability of multi-language co-simulation.

After our best-effort verification, the IP should be packed with its metadata for later usage. In the Xilinx IP environment, this metadata consists mainly of two files – in two

```
## MPD file for triple-DES IP

BEGIN tripple_des
...
## Bus Interfaces
BUS_INTERFACE BUS = SOPB, BUS_TYPE = SLAVE, BUS_STD = OPB

## Generics for VHDL or Parameters for Verilog
PARAMETER C_BASEADDR = 0x00000000, DT = std_logic_vector, BUS = SOPB,
ADDRESS = BASE, PAIR = C_HIGHADDR, MIN_SIZE = 0x100
PARAMETER C_HIGHADDR = 0x0000ffff, DT = std_logic_vector, BUS = SOPB,
ADDRESS = HIGH, PAIR = C_BASEADDR
PARAMETER C_OPB_AWIDTH = 32, DT = INTEGER, BUS = SOPB
PARAMETER C_OPB_DWIDTH = 32, DT = INTEGER, BUS = SOPB
PARAMETER C_FAMILY = virtex2p, DT = STRING

## Ports
PORT OPB_Clk = "", DIR = I, SIGIS = Clk, BUS = SOPB
PORT OPB_Rst = OPB_Rst, DIR = I, SIGIS = Rst, BUS = SOPB
PORT OPB_ABus = OPB_ABus, DIR = I, VEC = [0:(C_OPB_AWIDTH-1)], BUS = SOPB
PORT OPB_DBus = OPB_DBus, DIR = I, VEC = [0:(C_OPB_DWIDTH-1)], BUS = SOPB
PORT OPB_RNW = OPB_RNW, DIR = I, BUS = SOPB
...
END
```

```
## PAO file for triple-DES IP

lib proc_common_v2_00_a proc_common_pkg vhdl
lib proc_common_v2_00_a family vhdl
lib proc_common_v2_00_a or_muxcy vhdl
lib proc_common_v2_00_a or_gate vhdl
lib proc_common_v2_00_a counter_bit vhdl
lib proc_common_v2_00_a counter vhdl
...
lib opb_ipif_v3_01_a write_buffer vhdl
lib opb_ipif_v3_01_a opb_bam vhdl
lib opb_ipif_v3_01_a opb_ipif vhdl
lib tripple_des_v1_00_a user_logic vhdl
lib tripple_des_v1_00_a tripple_des vhdl
lib tripple_des_v1_00_a dual_port_reg_ctrl vhdl
lib tripple_des_v1_00_a fifo_channel_rd vhdl
lib tripple_des_v1_00_a fifo_channel_wt vhdl
lib tripple_des_v1_00_a reg_bank vhdl
lib tripple_des_v1_00_a reg_ctrl vhdl
lib tripple_des_v1_00_a task_section_0 vhdl
lib tripple_des_v1_00_a task_section_1 vhdl
lib tripple_des_v1_00_a task_section_2 vhdl
lib tripple_des_v1_00_a test_Task vhdl
lib tripple_des_v1_00_a test_wrapper vhdl
```

Figure 2.4 Example IP metadata: MPD and PAO descriptions for IP *Triple-DES*.

formats called MPD and PAO – for each IP, as shown in **Figure 2.4**. For the triple-DES hybrid-task IP, the MPD (Microprocessor Peripheral Definition) file describes:

- OPB as its single bus interface.
- Its possible parameters, such as the base and high addresses when connected an OPB bus. They should be configured with new values during SoC integration, or use their default values when appropriate.
- Its ports at the component level. The ports are exposed here either with a mapping to the bus specification, for example here OPB signals, or for a later mapping to the SoC system ports.

Another complementary *PAO* (Peripheral Analyze Order) file further lists the paths to all files that consist the IP itself. These includes not only our VHDL design but also the dependent libraries. The *Order* in the format name PAO means that the synthesis dependences are implied by the order of the file listing. This IP metadata, as well as the IP verification, prepares our shift to the SoC system design phase.

The SoC integration is described in a Xilinx *MHS* file – Microprocessor Hardware Specification, as shown in **Figure 2.5**. The MHS format is mainly targeted at SoC integration with memory-mapped buses. In brief, it describes instantiations of IP components, their interconnections, and the corresponding configuration of their parameters.

Though in a concise form, together with the implied reference to IP metadata and the further referenced IP design files, the MHS file becomes a *complete* description of our SoC design.

To verify our hybrid-task SoC system described in MHS, the functional simulation of the whole system behavior was performed, before synthesizing and implementing the

```
## MHS for Hybrid-Task SoC

 PORT fpga_0_RS232_RX_pin = fpga_0_RS232_RX,
DIR = INPUT
 PORT fpga_0_RS232_TX_pin = fpga_0_RS232_TX,
DIR = OUTPUT
...

BEGIN ppc405
   PARAMETER INSTANCE = ppc405_0
   BUS_INTERFACE JTAGPPC = jtagppc_0_0
   BUS_INTERFACE IPLB = plb
   BUS_INTERFACE DPLB = plb
   ...
   PORT CPMC405CLOCK = sys_clk_s
END

BEGIN plb_v34
   PARAMETER INSTANCE = plb
   ...
   PORT PLB_Clk = sys_clk_s
END

BEGIN opb_v20
   PARAMETER INSTANCE = opb
   ...
   PORT OPB_Clk = sys_clk_s
END
```

```
BEGIN plb2opb_bridge
   PARAMETER INSTANCE = plb2opb
   PARAMETER C_RNG0_BASEADDR = 0x40000000
   PARAMETER C_RNG0_HIGHADDR = 0x7fffffff
   PARAMETER C_RNG1_BASEADDR = 0xfffe0300
   PARAMETER C_RNG1_HIGHADDR = 0xfffe03ff
   ...
   BUS_INTERFACE SPLB = plb
   BUS_INTERFACE MOPB = opb
   ...
   PORT PLB_Clk = sys_clk_s
   PORT OPB_Clk = sys_clk_s
END

BEGIN plb_ddr
   PARAMETER INSTANCE = DDR_SDRAM_1
   PARAMETER C_PLB_CLK_PERIOD_PS = 10000
   PARAMETER C_DDR_DWIDTH = 32
   PARAMETER C_DDR_AWIDTH = 13
   ...
   BUS_INTERFACE SPLB = plb
   PORT PLB_Clk_n = sys_clk_n_s
   PORT DDR_Clk90_in = ddr_clk_90_s
   PORT DDR_Clk90_in_n = ddr_clk_90_n_s
   PORT DDR_Addr = DDR_Addr
   PORT DDR_BankAddr = DDR_BankAddr
   PORT DDR_CASn = DDR_CASn
   ...
END
```

```
BEGIN opb_uartlite
   PARAMETER INSTANCE = RS232
   PARAMETER HW_VER = 1.00.b
   PARAMETER C_BAUDRATE = 38400
   PARAMETER C_DATA_BITS = 8
   PARAMETER C_ODD_PARITY = 0
   PARAMETER C_USE_PARITY = 0
   PARAMETER C_CLK_FREQ = 100000000
   PARAMETER C_BASEADDR = 0xFFFE0300
   PARAMETER C_HIGHADDR = 0xfffe03ff
   BUS_INTERFACE SOPB = opb
   PORT OPB_Clk = sys_clk_s
   PORT RX = fpga_0_RS232_RX
   PORT TX = fpga_0_RS232_TX
END

...

BEGIN tripple_des
   PARAMETER INSTANCE = tripple_des_0
   PARAMETER HW_VER = 1.00.a
   PARAMETER C_BASEADDR = 0x78600000
   PARAMETER C_HIGHADDR = 0x7860ffff
   BUS_INTERFACE SOPB = opb
END
```

Figure 2.5 Example SoC system design: MHS description for *Hybrid-task* SoC.

system onto FPGA. Since, on one side, the MHS description is not simulatable and, on the other side, the involved IPs are provided as VHDL models, Xilinx provides us a generation tool that transforms a MHS file into a VHDL model. With this generator, we were able to obtain a VHDL top netlist for the hybrid-task SoC and compile it together with all other IP models for a simulation, in which the system was tested and debugged.

After the simulation, with another *UCF* file – User Constraint File – that basically specifies the binding between the MHS described SoC ports and real FPGA pins, we went through the synthesis, mapping, place-and-route, and FPGA Bitstream generation steps. Software part of the system was also developed and we was finally able to run the how SoC with its software on FPGA and demonstrate the hybrid-task scheduling and migration concept. More details of the system are given in [13] [11].

2.1.2. SystemC and Transaction Level Modeling

This section provides necessary background on the *SystemC* language for hardware and system design, with an emphasis on *Transaction Level Modeling* that is unique to SystemC based SoC modeling and also a focus of this work at system level.

The SystemC language comes in the form of a C++ library, as shown in **Figure 2.6**, and therefore works with a standard C++ compiler such as GCC in a Linux environment. At its core, no different than most other HDLs, SystemC provides facilities for hardware description, simulation, and synthesis.

- A typical discrete-event driven simulation kernel is provided, for the modeling of concurrent hardware and system elements. Events can be timed and delta-timed,

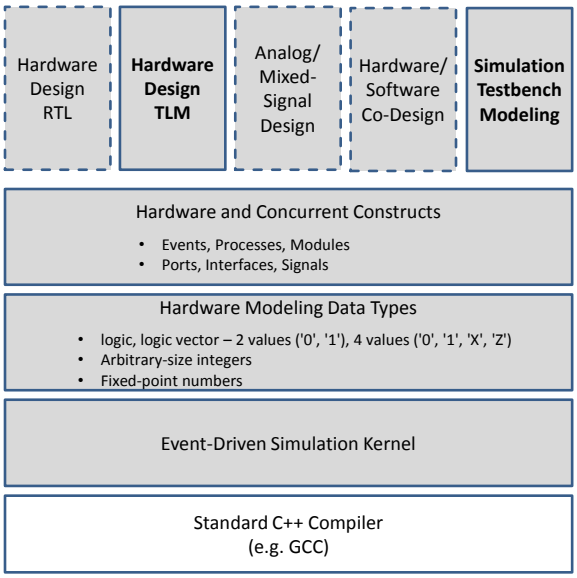


Figure 2.6 **SystemC language: core facilities.** Its usage for traditional RTL design, hardware/software co-design, and analog/mixed-signal design is *not* a focus in this work.

with default resolution of picosecond.

- Additional data types for hardware modeling are pre-defined, besides the standard C++ types.
- Then we have the core constructs for modeling hardware and concurrency. Concurrent processes can be defined as *sc_thread*, which should be made sensitive to some *sc_events*. The implementation is based on the *QuickThread* C++ threading library. Threads are encapsulated in *sc_modules*, similar to other traditional HDLs.
- Modules communicate with each other through ports and interfaces. **Figure 2.7** shows this mechanism. Basically, an interface class, inherited from *sc_interface*, should first be defined, specifying the communication services to be provided at this interface, for example, reading the value from a channel. Then this signature of communication should be implemented by a module, and accessed by another module through a port that is instantiated with this interface from template class *sc_port*. Since the first module implements the same interface as expected by the port of the second module, they are able to perform pre-defined communication during SystemC simulation, after their binding at initialization. As the fundamental mechanism for modeling communication in SystemC, this port-interface binding is used to implement not only the more abstract TLM but also RTL connections like *sc_signal*.

There are several reasons that we skip the detailed introduction to SystemC basics.

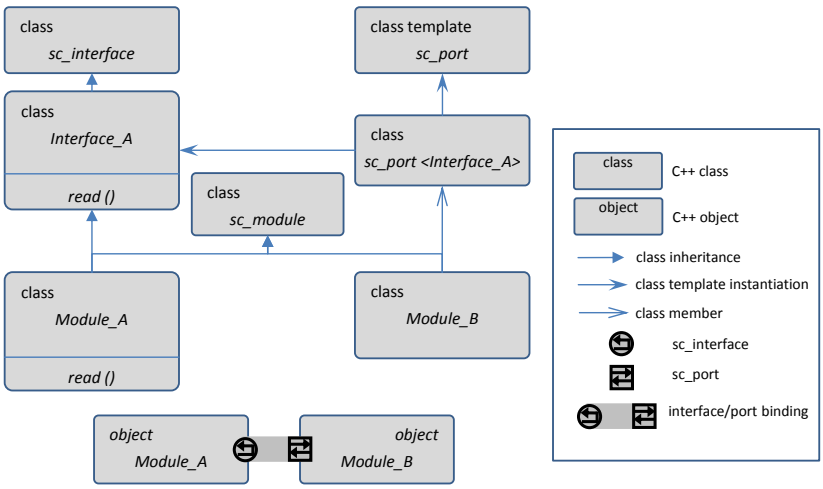


Figure 2.7 SystemC inter-module communication through *sc_interface*. It is provided by *sc_module*, required by *sc_port*, and bound at initialization.

Most importantly, for RTL design, SystemC provides modeling elements fundamentally no different than other HDLs. At IP level, we will consider designs at RTL or behavioral with traditional HDLs like VHDL. At system level, we will put an emphasis on TLM as a new, state-of-the-art domain.

Therefore, also assuming that our reader is *not* completely familiar with TLM, we introduce in the following not only the principle of TLM, but also an example of its application. The test-bench modeling capability of SystemC will be left to Section 2.2, in particular on *SystemC Verification Library*.

In essence, *with TLM in the context of SoC design we model on-chip communication between system components as function calls, which carries commands and data specific for that communication protocol*. Several function calls are grouped into a *TLM interface* as a class inherited from *sc_interface*, to be provided by a *sc_module* and accessed by another module through *sc_port*. Use of low-level signals for communication are mostly eliminated.

Figure 2.8 shows this principle of TLM. Processor Local Bus (PLB), a widely used on-chip communication protocol, is used for demonstration.

The upper part of the figure shows the block diagram and a write-transfer operation from the original PLB specification [40]. Structural and timing requirements for an RTL implementation are specified. A PLB bus transaction is defined on a bunch of signals.

The lower part draws us a picture how PLB transactions are modeled at TLM, from a PowerPC/CoreConnect based SoC design library [41]. Operations, *read* and *write*, from

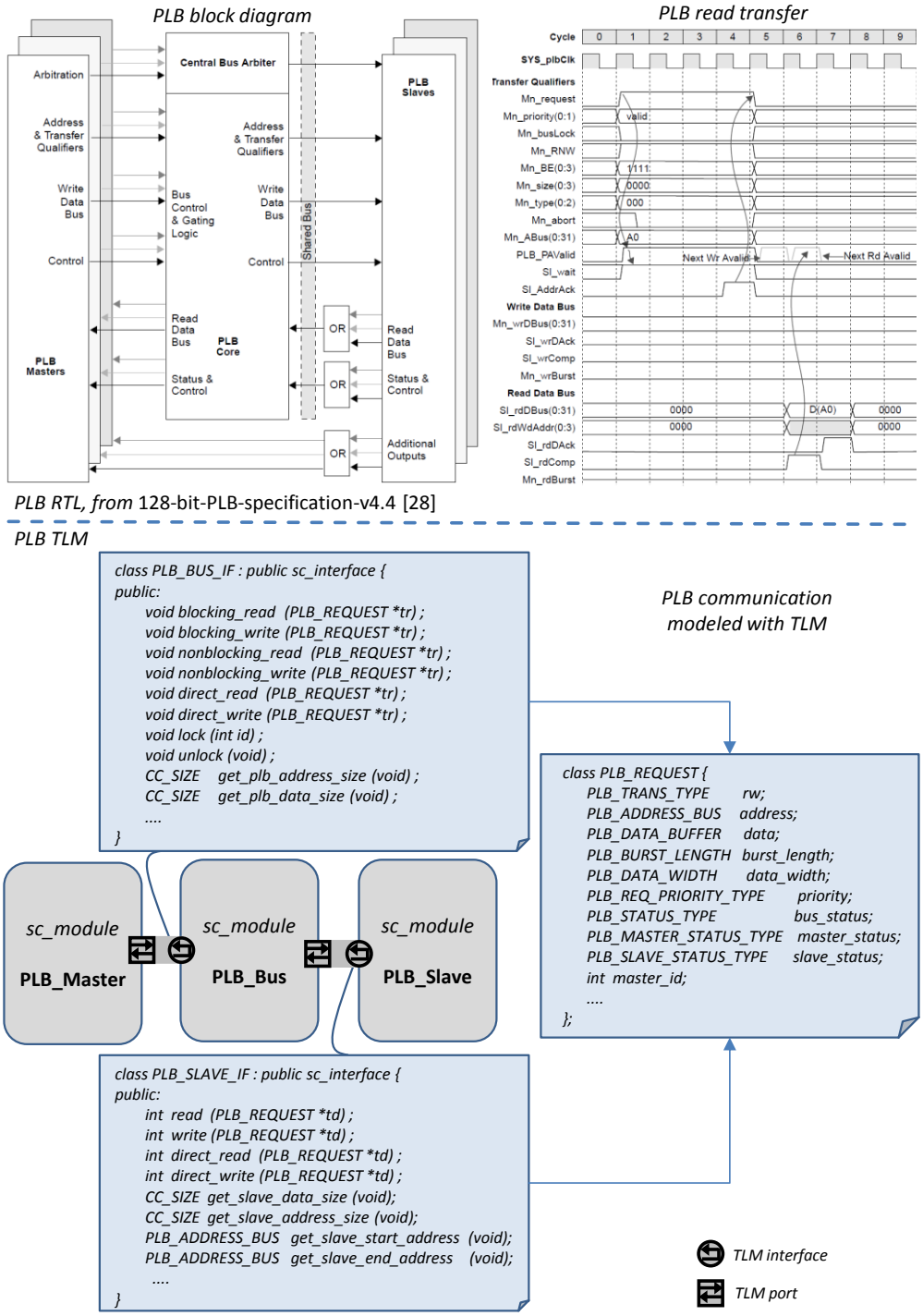


Figure 2.8 TLM principle: function calls to model SoC on-chip communications.

the PLB specification are abstracted as functional calls and grouped into two main TLM interfaces, *PLB_BUS_IF* between a PLB master and a PLB bus and *PLB_SLAVE_IF* between a PLB bus and a PLB slave.

PLB_REQUEST, which is carried by these function calls, is a protocol specific data structure and contains fields that exactly represents signals specified by the protocol. The communication parties, i.e. *PLB_BUS* and the corresponding ports, are responsible to maintain state-machines that match the communication protocol, according to PLB_REQUESTs that they send/receive through the TLM interfaces.

The rationale behind TLM is a *separation* of on-chip communication and computation in SoC design, supposing communication generally to be modeled in a more *abstract* manner than computation. By this, SoC IPs or components at various abstraction levels, RTL or behavioral, can all be encapsulated and integrated into a TLM communication platform. This enables the TLM based SoC modeling, simulation, and evaluation, which is a focus of our verification method at SoC system level.

Timing in design and verification is *not* a focus in this work. TLM interfaces can be implemented with different timing abstractions, cycle accurate or timing approximate – for example, whether the *read* operation in the PLB_BUS_IF is implemented with clock cycles strictly conforming to the original specification, or only in a functionally correct way. TLM wrapped computation can also be of different timing accuracy, regardless of communication timing. In [8], we have also proposed a system refinement process based on TLM, taking into consideration both software and hardware.

The contribution of this thesis at SoC system level will focus on the gap between design of system integration and TLM based functional system simulation, as well as the quality of such simulation.

Further, automated TLM extraction from RTL and TLM synthesis to RTL are both *not* considered in our approach, though the equivalence checking between these two levels can also be accounted as a task of functional verification. Interested readers can refer to [36] [37] [42], for example.

Example: TLM based SoC Design Experiment with ARM/AMBA

We have carried out this small design experiment shown in **Figure 2.9**, as a further demonstration of TLM design.

There are two inputs for the experiment. One is an ARM microprocessor model called *SWARM – SoftWare ARM* [43] [44]. The other one is a TLM design library for AMBA SoC architecture, called *CASI AMBA – Cycle Accurate Simulation Interface AMBA* [45].

Written in C++, *SWARM* models an ARM 7 processor that implements the ARMv4T architecture. When used as Instruction Set Simulator (ISS), it executes ARM instructions in a *cycle-accurate* manner. To be cycle accurate, it also models and simulates partially the microarchitecture of the processor, as shown in the figure. Further, it includes several basic

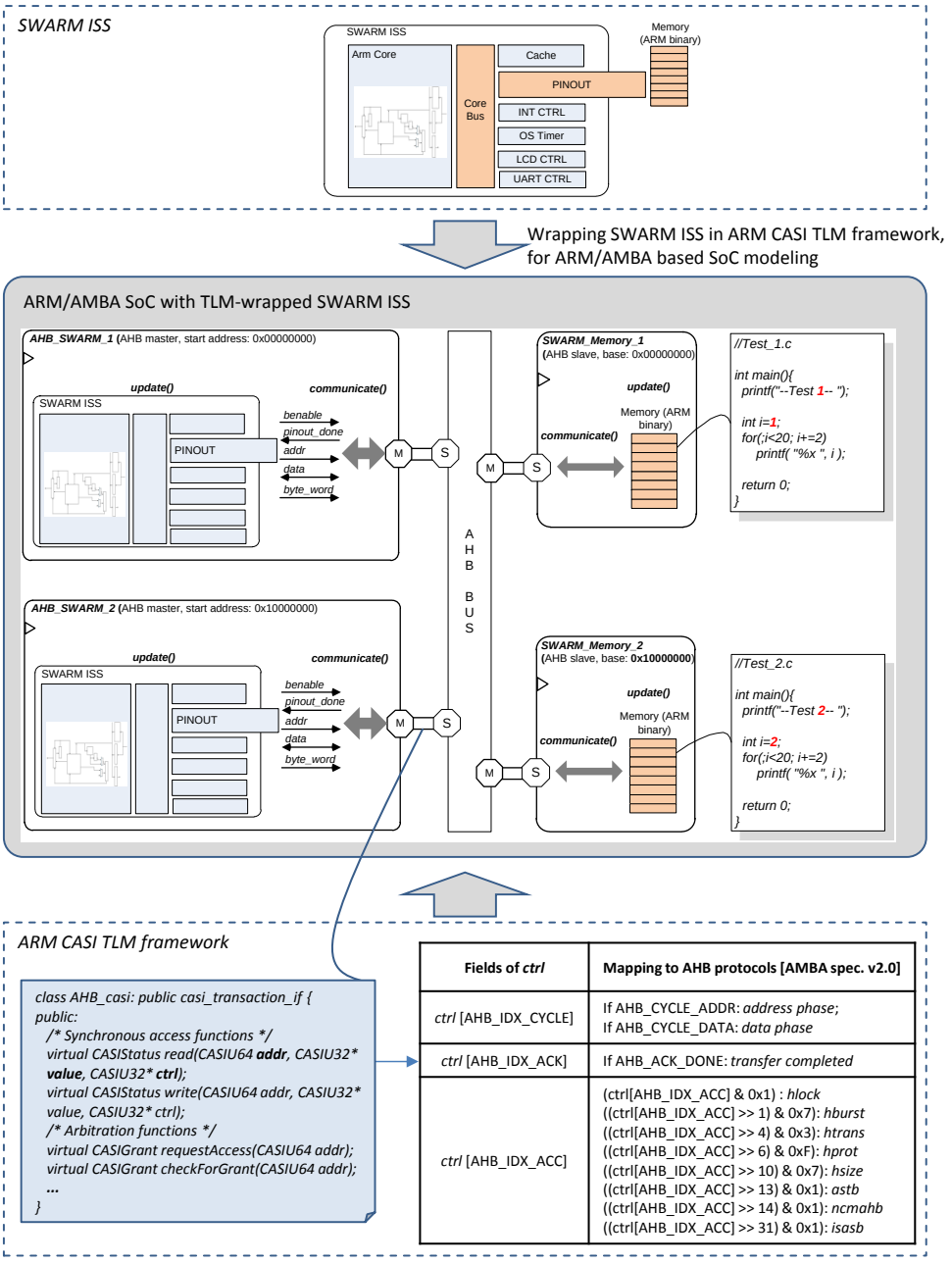


Figure 2.9 TLM based SoC design experiment. We have used ARM CASI TLM framwork to wrap SWARM ISS and model a basic ARM/AMBA system.

peripheral models connected to an internal bus, such as a cache with configurable size. It is able to run a porting of Linux.

SWARM has been used in several SoC research experiments such as [44] , because of its open-source nature and the popularity of ARM/AMBA SoC architecture. This also leads to our motivation of taking it as an ARM processor IP, packing it as a TLM component, and composing a TLM SoC demonstration.

For this, we find the *CASI AMBA* TLM library that is directly provided by ARM. As mentioned, the core of such TLM modeling is the abstraction of an on-chip communication protocol into TLM interfaces that consist of function calls. In the figure, we show a TLM interface from the *CASI AMBA* library, which abstracts AMBA AHB (Advanced High performance Bus) protocol. We see how the *read/write/requestAccess/checkForGrant* function calls represent the bus access and how the protocol signals are encapsulated and carried by these calls. As pointed out by the name, such CASI AMBA communication is modeled in TLM as cycle accurate to the original protocol specification, so that we can make accurate simulation, performance evaluation, and design exploration.

This CASI AMBA library does not include any concrete SoC component, except for the bus models. However, because of the advantage of TLM that provides a separation between on-chip communication and computation, we are able to wrap IP components at any level, RTL or behavioral, as TLM components and enable a TLM-based SoC system integration.

As **Figure 2.9** shows, using the CASI AMBA library we created a TLM wrapper for SWARM ISS, which converts its original memory-accessing *PINOUT* into the TLM AHB interface, and backward. We also wrapped a memory model as a TLM AHB slave, which loads ARM binary at its initialization.

The simple system was then integrated by instantiating an AHB bus from the library and at the same time attaching dual TLM SWARMs and two memory models to the bus. Two test programs were supposed to exercise this system integration in a simulation.

In the end, we were able to compile the whole TLM system with SystemC, compile the software programs with a cross-compiler *gnuarm-3.4.3*, and successfully simulate the system with dual-SWARM execution.

2.1.3. IP-XACT Standard for IP Reuse and SoC Integration

Verification depends on the language that is used for design. For IP-level designs, for example a microprocessor IP, we assume traditional HDLs or SystemC in use, either at RTL or behavioral. For SoC system level, we try to propose a systematic verification framework based on a standard IP reuse and SoC integration language, or format, called *IP-XACT*. In this sense, IP-XACT is our HDL at SoC system level.

The IP-XACT standard has been made IEEE 1685-2009 [29] in 2010, with the effort initiated even earlier by the *SPIRIT* consortium, formed by several major semiconductor and EDA tool vendors. It intends to provide standardization support to an IP-based SoC design flow, such as that previous example on Xilinx FPGA design environment. It is exactly the formats of IP metadata and their integration – MPD, PAO, and MHS in the

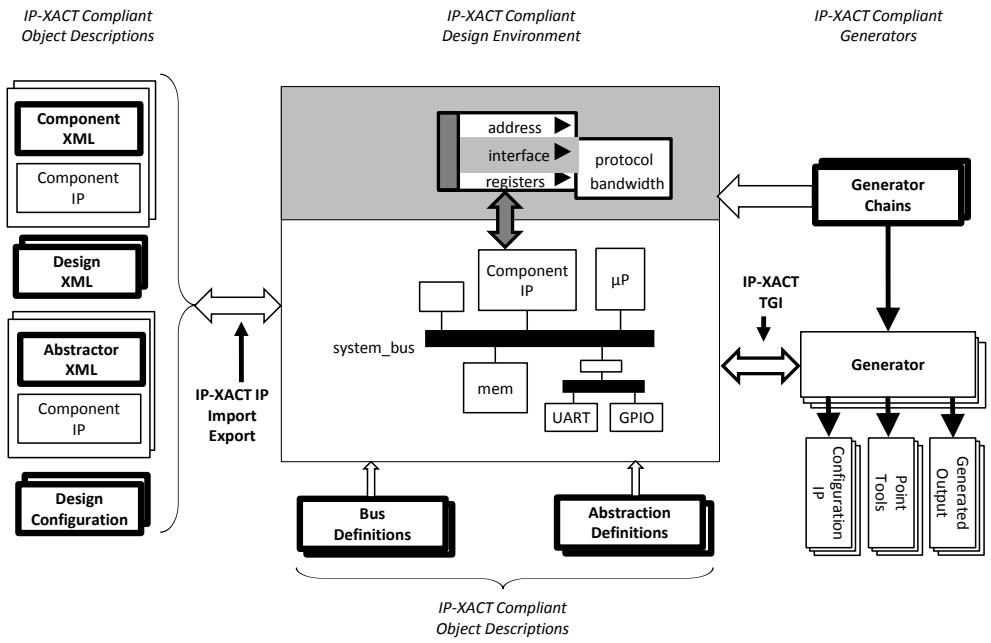


Figure 2.10 Overview of IP-XACT standard [29]**.**

Xilinx case – that IP-XACT tries to standardize. The idea is to have unified, vendor-neutral exchange format for both IP vendors and SoC integrators.

Figure 2.10 from IEEE 1685-2009 [29] shows a blueprint for IP-XACT based IP reuse and SoC integration. At its core, IP-XACT *defines an XML Schema as the standard electronic format for packaging reuse information of IPs, as well as for designing SoC systems by IP integration*. Several major XML schema elements are presented here, including ***component***, ***design***, ***abstractor***, ***design configuration***, ***busDefinition***, ***abstractionDefinition***, ***generator***, and ***generator chains***. Any top IP-XACT XML document belongs to one type of them.

Based on these elements, we have two main use scenarios with IP-XACT, as an IP provider or a SoC integrator. We give an explanatory listing of IP-XACT schema for these two scenarios, instead of a comprehensive standard repetition. For this, we also prefer an example based, graphical representation of the IP-XACT schema. Basics about XML Schema can be found in [46].

First, IP vendors use an IP-XACT ***component*** XML file to package all reuse-related information of an IP core, which accompanies this IP as its electronic data sheet. The information includes mainly how the IP can be configured and interconnected to other IPs via a memory-mapped bus connection, which is the main focus of IP-XACT other than more sophisticated on-chip architectures like Network-on-Chips. **Figure 2.11** shows several IP descriptions using IP-XACT *component*. Note that XML documents are depicted in graphics, as in the rest of this section.

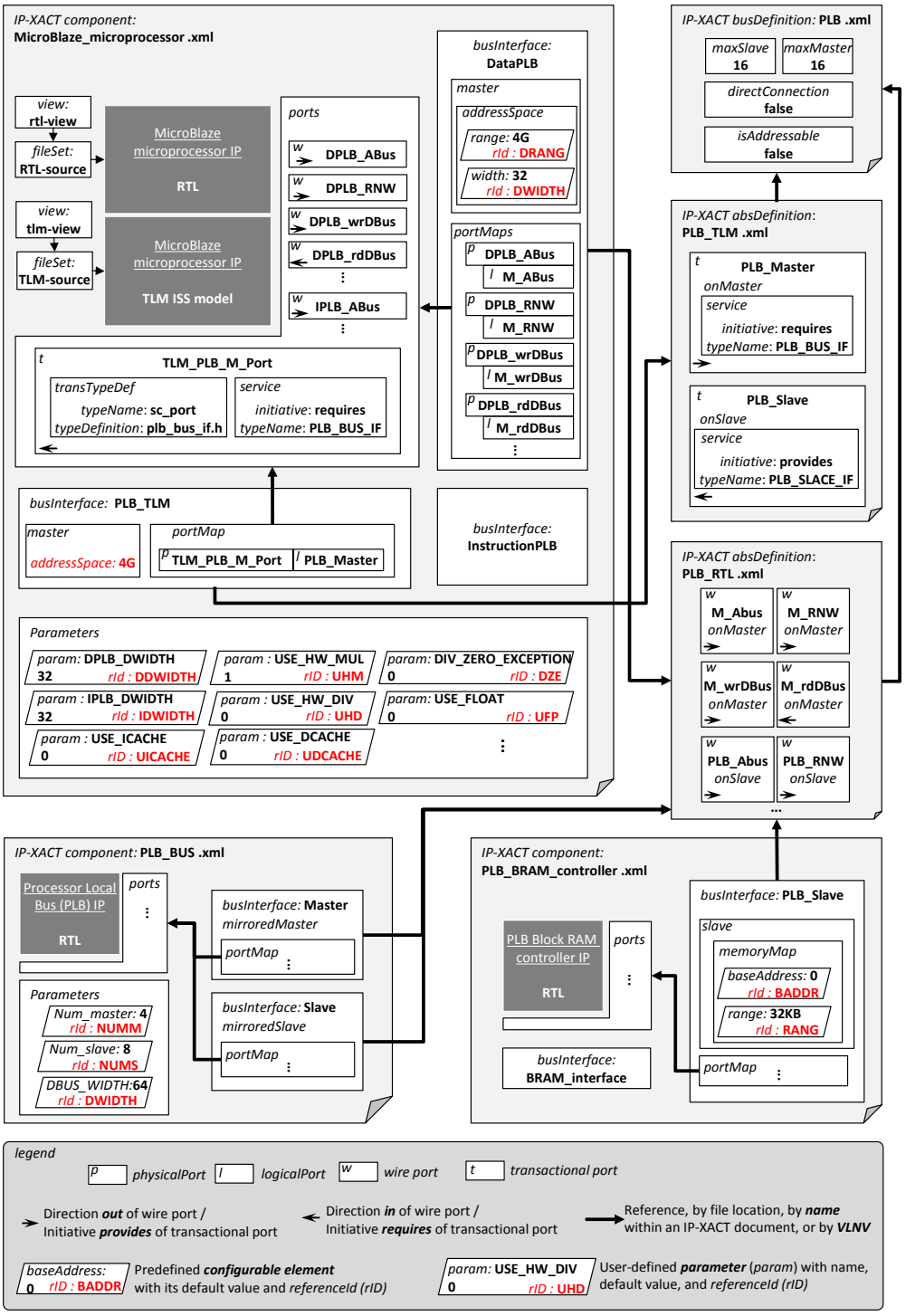


Figure 2.11 IP-XACT *componnet* XML schema for IP description.

- Multiple ***views*** of the IP could be described in its ***component*** XML document. If the IP are provided with both VHDL source files and an abstract C simulation model, we can then include two views, say, *VHDLSourceView* and *CModelView*

in its IP-XACT description. In each *view*, the used language could be described. A view references a set of files that implements this view.

- A file set to be referenced from a view is first documented with *fileSet*. Each *file* in a **fileSet** describes one of the real files or directories that comprise the IP. Particularly, it contains extensive information that can be leveraged when later the IP needs to be compiled or synthesized in design flows.

  This includes a *name* as the exact path to the file or directory and a *fileType* as the format of the file, which shall be selected from a pre-defined list with *systemCSource*, *vhdlSource*, *VerilogSource*, *swObject*, *swObjectLibrary*, etc. Other possible descriptions are *includeFile* as a Boolean tag to indicate whether the file is an include file, *logicalName* for the name of a library file, and *dependency* for a directory that this file depends on. Last, one can further specify with *buildCommand* explicitly commands and options that should be used in the file compilation.

- ***Ports*** of an IP to be exposed for connection can be declared as a collection of *port*, which should be of either a *wire* type or a *transactional* type. A *wire port* corresponds to a traditional scalar port or vectors of scalars in HDLs, such as *std_logic* and *std_logic_vector* by default for VHDL. *Direction* of the port shall be specified. If the port is a vector, its *left* and *right* bit should be also be specified.

  With *transactional* port, the latest Transaction Level Modeling (TLM) and TLM IPs are supported by IP-XACT. It is much tailored to SystemC transactional modeling that has become the de-facto TLM standard. First, the type of the port in SystemC can be expressed in *typeName*, such as the common *sc_port*, *sc_module*, *sc_export*, *sc_initiator*, *sc_target*, or *sc_socket*. Second, more importantly, the interface implemented by this port shall be detailed in a *service* structure, as TLM connection is essentially centered at SystemC interfaces. The *initiative* of a *service* is the direction of the interface implementation, having the value *requires* if it is a *sc_port*, *provides* if a *sc_export*, or *both* for a *sc_socket*. Another *typeName* included in *service* describes the exact SystemC type of the interface, along with *typeDefinition* indicating the real SystemC file that declares the interface. Either a wire port or a transactional port, it should have a ***name*** that is exactly how the port is named in the real IP model.

- Later in IP-XACT system integration, there are two alternatives of connecting two components. One is direct port-to-port connection and the other is based on pre-specified bus interfaces. For the latter, another two top elements of IP-XACT schema need to be explained first, namely ***busDefinition*** and

abstractionDefinition, which as a pair resemble a traditional signal specification for a bus protocol.

BusDefinition specifies general properties of a bus, such as its *maxMasters* and *maxSlaves*. Multiple *abstractionDefinitions* may belong to one *busDefinition*, as now both RTL and TLM are supported by IP-XACT.

Each *abstractionDefinition* is a collection of *port* descriptions, which provide quite similar information as those in a *component* element, but here for a specification purpose instead of declaration of implemented IP ports. Besides, a *port* in *abstractionDefinition* also specifies whether it is *required*, *optional*, or *illegal* to be present on the bus interface and, when present, whether it should be implemented as ***onMaster***, ***onSlave***, or ***onSystem***. Signals like system clock and reset should be grouped as *onSystem*.

- For the *bus/abstractionDefinition* based interconnection, the bus interfaces of an IP are declared by ***busInterfaces*** in its *component* description. Each ***busInterface*** possesses a unique reference to a pair of existing *busDefinition* and *abstractionDefinition*, so that a *busInterface* based interconnection between two components can be automatically verified by comparing the referenced *bus/abstractionDefinitions*.

  This also enables automated port connection during SoC integration, between two IP components, via a ***portMaps*** structure. Such a list of ***portMap*** is defined for each *busInterface*, which maps a ***physicalPort***, as reference to a *component port*, to a *logicalPort* that is reference to a *port* specified in *abstractionDefinition*. This defines actually how the bus protocol *abstractionDefinition* is implemented by this *busInterface*.

  Common bus features like *endianness*, *bitSteering*, and *bitsInLau* may further be described for *busInterface*. C*onnectionRequired* indicates whether the interface shall be connected when integrated.

- For memory-mapped system integration, it is essential that we describe the connection purpose of *busInterface* as one from seven types defined in IP-XACT - ***master***, ***slave***, ***system***, ***mirroredMaster***, ***mirroredSlave***, ***mirroredSystem***, and ***monitor***.

  Consider three typical IP components that are shown in **Figure 2.11**: a microprocessor core, an on-chip bus, and a memory controller. The microprocessor *component* description probably includes a *busInterface* in the mode of *master*, which mainly defines an ***addressSpace*** as the addressable range from this master. An executable image can also be referenced.

BusInterface of the memory controller *component* should have a mode *slave*, where a memory-map block such as a single ***addressBlock*** with specification of its ***baseAddress***, ***range***, and ***width*** can be defined. Specific registers within the *addressBlock* can also be described by their ***size*** and ***addressOffset***. The PLB bus *component* exposes two *busInterfaces*, one as *mirroredMaster* and another as *mirroredSlave*, to be connected to a matching *master* interface and a *slave*, respectively. All the components may further have *system* as well as *mirroredSystem* interfaces as in/outlets for system clock and reset signals.

We should note that in a *portMap* on *master*, *slave*, or *system busInterface*, the *physicalPort* shall implement the same direction as specified by the *logicalPort*. To the opposite, the *physicalPort* in a *portMap* on *mirroredMaster*, *mirroredSlave*, *or mirroredSystem busInterface* shall implement the inversed direction from the *logicalPort* specification. Mapping of component ports to bus specification enables their seamless interconnection later. Further, *monitor* is a special interface to be exposed by any component for verification purpose.

- In *component* and in IP-XACT generally, any element defined with an ***id*** attribute is configurable. For a *component* description, its configuration is done at a system *design* description that instantiates this component, by assigning the configurable elements new values under references to their *ids*, if the default values should not be effective. Slave ***baseAddress*** is a common use case of configurable element.

  In addition to an *id*, more attributes may be defined on a configurable element, to specify and constrain its value options. Input ***format*** of the element can be specified as one from ***bitString***, ***bool***, ***float***, ***long***, and ***string***. Attribute ***resolve*** defines how the element value should be configured, such as ***user*** indicating the value to be set by user input or ***dependent*** meaning that the value shall be calculated from other element values. Candidate values may also be specified in a ***choice*** structure as a list of enumerations. With ***minimum*** and ***maximum*** we further specify the lower and upper bound of the element value.

- Moreover, there are basically two categories of configurable elements in a *component* description. The first category is directly HDL derived, if the IP under description is in the form of a HDL model. These parameters do not have a pre-defined sematic, or meaning in IP-XACT schema, but they are immediate place holders for HDL model parameters. They describe, for example, constructor parameters of a SystemC module or generics of a VHDL entity. Such a parameter has a ***name*** associated that is directly taken from the model, besides all the above mentioned configuration attributes.

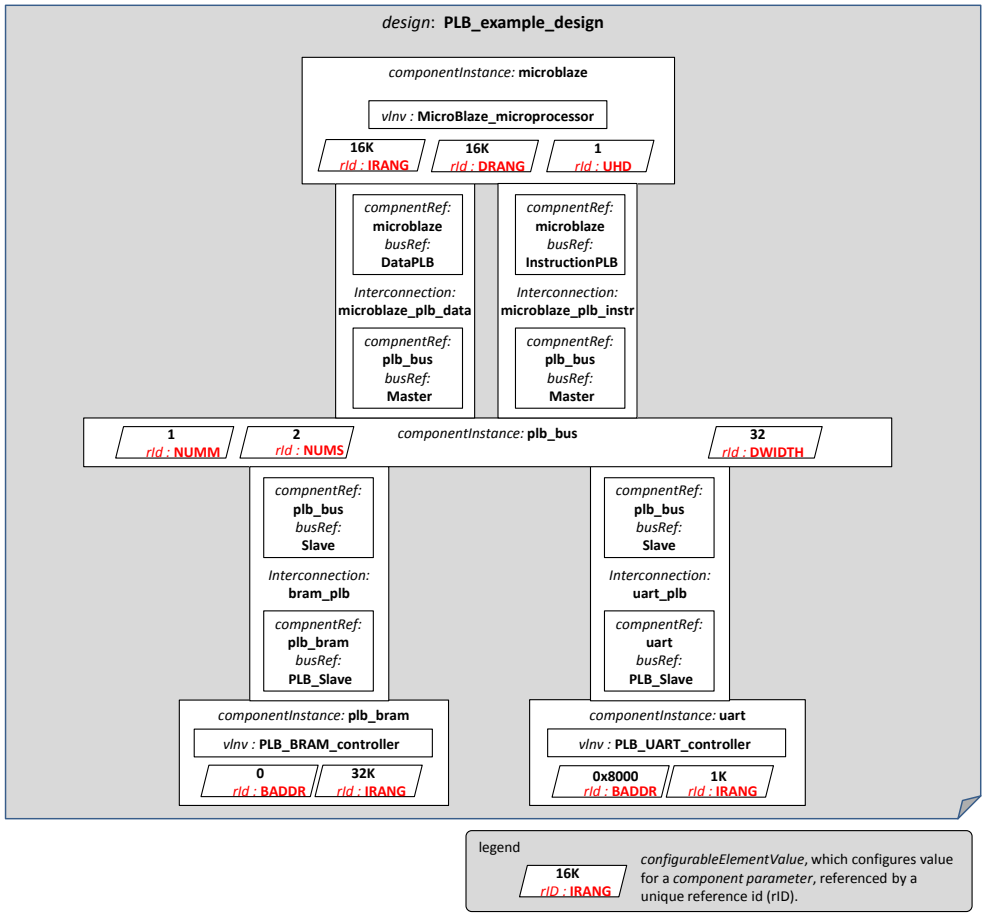


Figure 2.12 IP-XACT *design*. It describes a SoC integration design with *component* instantiation, configuration, and interconnection.

The second category includes those configuration elements with IP-XACT semantics. For example, IP-XACT defines a *baseAddress* for a *slave* bus interface, with this address specified as a configurable attribute. Being aware of this makes difference for us, as a configurable element with IP-XACT specified semantic means that we can take corresponding actions during the synthesis or manipulation on the element.

In the second scenario, SoC integrators use an IP-XACT *design* XML document to assemble an integrated system from existing *components*, as the example shows in **Figure 2.11**. This *design* mainly describes the instantiation of IP components, necessary interconnections between them, and their correspondingly derived configurations.

- A *design* instantiates all its components –processing elements and on-chip buses – by a list of ***componenInstances***. A ***componentInstance*** is assigned a unique ***instanceName*** within the *design* and has a reference to the concerned IP-XACT *component* description.

The *component* is identified by the ***vlnv*** unified cross-document referencing mechanism of IP-XACT, as also used by *abs/busDefinition* references. In this ***vlnv*** system, every IP-XACT top object/document shall possess a ***versionedIdentifier***, as a combination of ***vendor-library-name-version***, which uniquely identifies the document in all IP-XACT mentioned context. Then this top object can be referenced within another document by a ***libraryRefType*** element that consists of also *vendor-library-name-version* of that object. A single *instanceName* is enough for further identification of this component within this *design*.

- IP-XACT facilitates mainly memory-mapped bus interconnection for system level integration. Each connection between two components through a bus interface is defined by an ***interconnection*** element. Besides a name for the connection, an *interconnection* contains merely two references of component bus interfaces. Each such reference is a pair of names, one for the component instance name assigned within this design and the other for the name of the bus interface in the original *component* description. As both interfaces not only have references to the same *abs/busDefinition* that they intend to realize but also specify with *portMaps* how the *abs/busDefinition* are implemented by the component ports, we are able to resolve correct signal connections on this bus interface.
- In *design*, we also have the possibility of creating ***adHocConnections*** not via any bus specification but on a port-by-port basis. Each ***adHocConnection*** is defined as a list of two or more port references, to bundle multiple component ports together.
- We need to assign configurable elements of the instantiated and interconnected components with appropriate values according to this integration, such as the address offset of each slave interface, if they should vary from the default. For this, we can define in *componentInstance* a list of ***configurableElementValue***, each with a ***referenceId*** that is the **id** of the configurable element in the *component* description and its new value.
- Further, hierarchical design is also supported by IP-XACT, through the possibility of wrapping a *design* further as a *component*.

Besides these two use scenarios for IP integration, IP-XACT *generators* define standard integration interface between a main design environment and third-party tools: how the main design environment can launch a third party tool and how the latter can access the IP-XACT files in the former, through the interface called Tight Generator Interface. This tool integration is *not* a focus here and one can refer to IP-XACT standard for more information.

A general note here at the end of the section. A big challenge that we will address in this thesis is the provision of *systematic* verification for an IP-XACT SoC design. We

definitely find a gap between SoC design with IP-XACT and its functional verification, since an XML file in IP-XACT is *not* directly simulatable for verifying its behavior.

2.2. Simulation Based Functional Verification

Functional verification is the *process of verifying whether a design conforms to its specification*, as shown in **Figure 2.13**. Take a microprocessor design for example. The main specification to be verified regarding its functionality should be whether it can execute correctly sequences of instructions defined in the ISA that it intends to implement. Non-functional properties of a design, such as timing and power, are not the topic of our work.

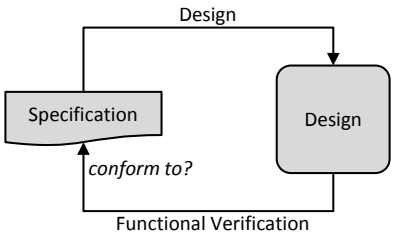


Figure 2.13 Functional verification: whether a design conforms to its specification.

We assume *simulation based verification*, with its principle shown by **Figure 2.14**, still the overwhelming technology employed for functional verification and therefore also taken as the basis of our entire work, though other ways of design verification do exist, such as model checking [47] or FPGA based prototyping. They are not discussed, since our verification methodology to be proposes is purely based on simulation, even eliminating *symbolic execution* that is be found in some literature on test generation. Therefore, this background section on verification is concentrated on simulation.

A simulation based function verification process is depicted here with five components: the *design under verification*, a *test generator* for generating input stimulus of design, a *monitor* for observing the design behavior during simulation and a *checker* for deciding the behavioral correctness, *metrics and measurement on the quality* of the simulation, a *simulator* for actually executing the whole. Each is explained in the following.

Through drawn as a one-direction process, the verification should be iterative. Mainly, once we find a bug after some simulation, the design has to be debugged and corrected. Then the simulation should be repeated as another iteration. The verification closure problem – the *done* question – will be governed quantitatively by quality metrics.

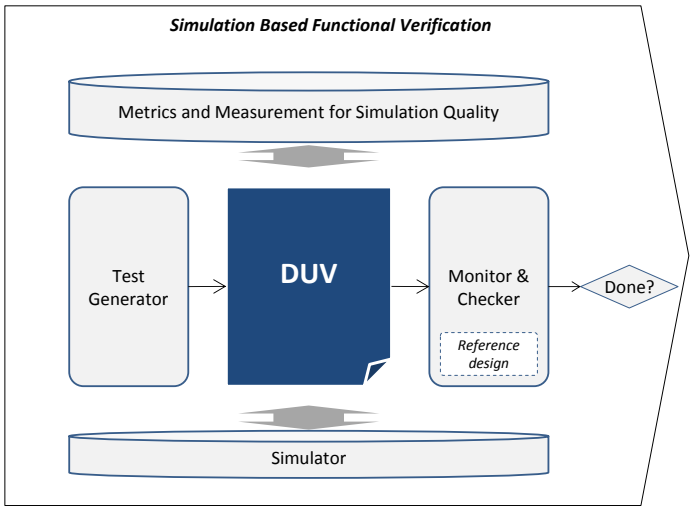


Figure 2.14 Simulation base functional design verification: common structure and elements. DUV: Design Under Verification.

Design Under Verification

Put in the context of our reference IP-based SoC design flow, a design under verification (DUV) at IP level should be a RTL or behavioral model described in common HDLs – VHDL, Verilog, C, or SystemC. At SoC system level, we assume IP-XACT the default language for SoC integration. The IP-XACT design many integrate IPs in RTL, TLM, or both. In all cases, we do not assume an IP design or SoC design to be synthesizable, with it possibly at early design stage, or mature, near-complete stage.

With IP design, it is also reasonable that we assume a *white-box* testing scheme, a general term understood in software testing, meaning that we are able to observe the internal execution of the design. The introduction of simulation quality metrics will also following this assumption.

IPs may become *black-box* in the SoC integration phase, meaning that their code, or the observation possibility on the code is not available anymore, though they can still be simulated together with each other. A case can be that an IP is provided as two pieces: one as a compiled simulation model compatible with some specific simulation tool, the other one as a synthesized or even hardened design only for further implementation. It is one of the reasons that, in a later chapter on SoC system design, we will consider defining a quality metric focused on IP-XACT as the design code.

Simulator

We assume the basic knowledge of HDL simulation with VHDL and Verilog, which are well established languages. If necessary, a short introduction to *discrete-event based simulation*, which is used in most HDL simulators, can be found in [28].

In the context of IP-based SoC design, a new requirements on the simulator is that it should support simulation of IPs in different forms [48], since the IPs can be developed in different environments. As **Figure 2.15** shows, ModelSim as a state-of-the-art simulator does provide a multi-language, mixed-level simulation engine.

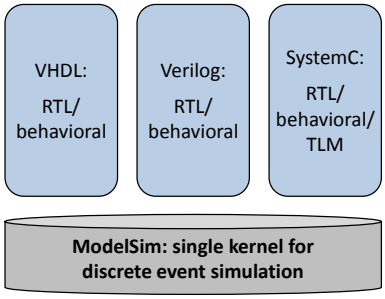


Figure 2.15 ModelSim(TM) simulator from MentorGraphics. It supports multi-language, mixed-level, singel-kernel simulation of hardware designs.

Multi-language simulation means that with ModelSim, several designs in various languages – VHDL, Verilog, and SystemC – can be integrated and compiled as a single object and simulated together, with all their original language semantics strictly reserved. Such simulation can be also be *mixed-level*, meaning the integrated design components be of different abstraction levels – RTL, behavioral, TLM.

This co-simulation is directly possible, since the original simulation engines behind these languages and modeling levels are all *discrete-event simulation*.

Still, as mentioned, we find that for SoC system integration, there is a gap of between IP-XACT design and simulation.

Test Generator

The test generator is responsible for the test generation task that selects a subset of design input to be applied as tests, considering the whole design input space as candidate set for selection. In general, as design input can be classified into different *types*, the input space can be divided into *regions*. For example, the instructions as input for a microprocessor design have strictly specified types.

For this test selection from a design input space, we can identify three fundamental approaches, as shown in **Figure 2.16**:

- *Directed test generation*: A test set is planned and selected from the input space before simulation, mainly manually by the tester. A fixed table listing this test set is constructed. Then the entries of the table are applied one by one in design simulation. Since all the test entries are constructed manually, test selection effort will be high, taking into consideration both specification and implementation.

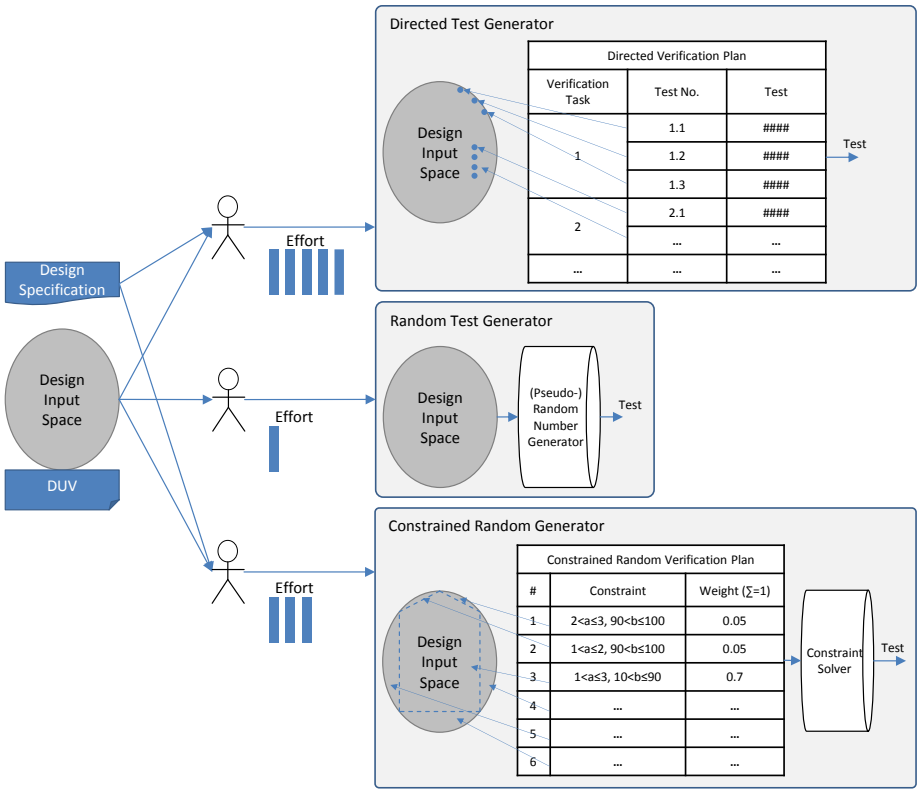
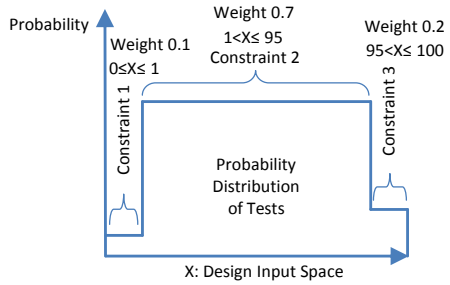


Figure 2.16 Test generation approaches compared.

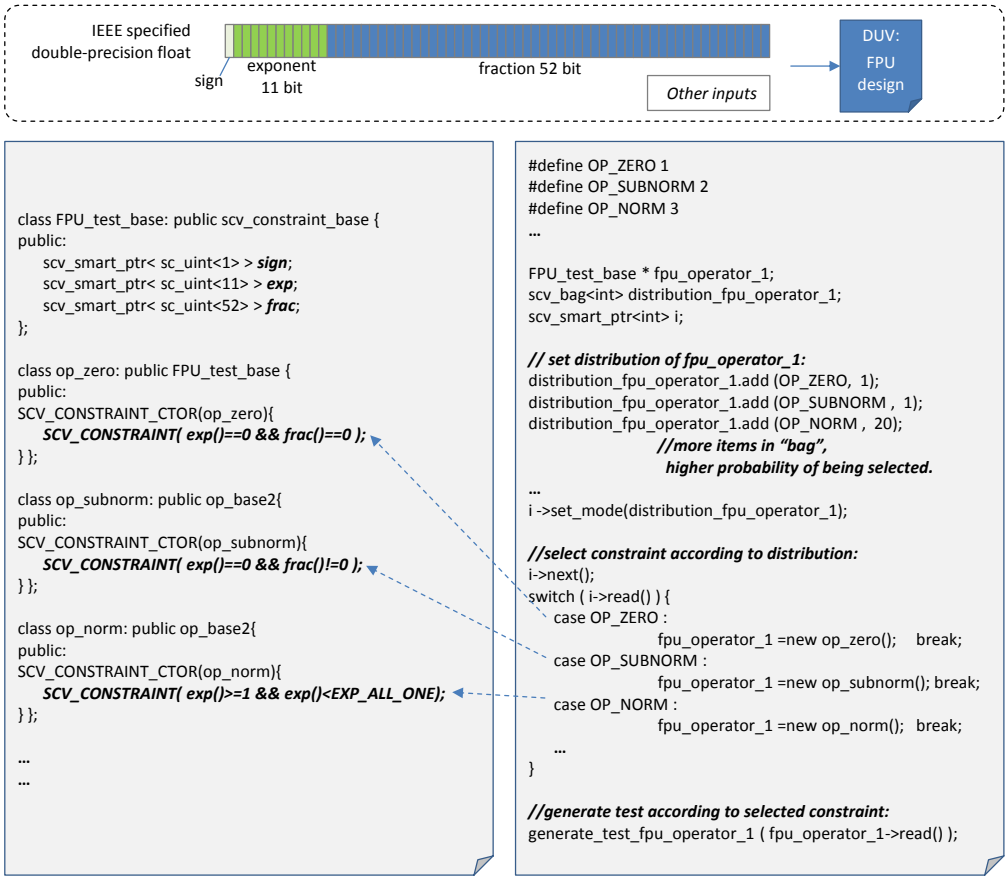
- *Random test generation*. At the opposite extreme, we have the pure random approach that lays a (pseudo) random number generator upon the entire input space, which then simply selects a test each time with an unbiased distribution on that space. The application effort of this approach should be minimal.
- *Constrained Random Test Generation*. This is the approach in-between and combines the advantages of both directed and random test generation. Instead of listing all individual tests that are considered interesting, *constraints* are used to divide design input space into regions. A constraint is selected each time and fed into a *constraint solver* that solves this constraint and generates a concrete test from this region. The selection of constraint can further be made with pre-defined weights that together add up to one.

We also use *Constrained Random Simulation* (CRS) to call the constrained random test generation based functional design simulation. Because of its employment in later chapters, we use **Figure 2.17** to explain more on the principle and advantage of CRS.

As shown by the upper part of the figure, with a set of constraints defined on the design input space and each associated with a weight for selection, we actually obtain a *probability distribution* of tests to be generated for simulation. The advantages are that we are able to: i) generate a significant amount of tests for exercising the design, as in



a) Weighted constraints imply a probability distribution of tests



b) Example weighted constraints definition, on an input field of a FPU design, using SystemC Verification Library

Figure 2.17 Constrained random test generation: principle and example.

random test, ii) at the same time, control the distribution of generated tests by assigning more weights to constraints of more interests, and iii) even adapt this test distribution during the simulation process, if our interest changes, for example the quality metrics to be presented later.

The second part of the figure gives a real example of CRS – how some weighted constraints are defined for test generation of a floating point unit (FPU) design, which is

expected to conform to an IEEE standard for double precision float arithmetic [49]. One float operator as one part of the entire design input is taken for example. The left part shows the constraints defined with constructs from *SystemC Verification Library (SCV)*, on field *exponent* and *fraction* of the input. The right part shows how the *scv_bag* is used to define the weights on each constraint, by throwing a corresponding amount of items into the "bag" for that constrains. The code shows also how the constrained test generation happens during simulation, with a constraint first selected according to the weighted bag and the solved to generate a test.

SCV constraints are solved by an integrated constraint solver in SystemC. One can find more discussion on this solver at, for example, [50]. The quality of constraint solving is *not* an issue considered in this work.

Monitor and Checker

Simulation produces *traces* that should be observed and checked for a decision whether the design had a correct behavior during this simulation. This observation and checking task is performed by the *monitor* and *checker*, respectively. We actually do not distinguish much between these two components of simulation.

Such a simulation trace records the history of value changes on each variable or signal included in the design under verification. As an example, **Figure 2.18** shows the trace from the simulation of a microprocessor design, using the ModelSim™ VHDL simulation tool. As a synchronous design, the values may change at each clock cycle. Two interfaces of the design, one to the instruction memory and the other one to the data memory, are recorded in this trace and shown in a wave form. We can then check the trace against, for example, another trace produced with a *reference design* – also called *golden model* in some cases meaning that it is assumed to have an absolute correct behavior – and the same tests, to see whether any deviation exists. The trace recording is usually a facility provided by the simulation tool, although it is the task of a user to define what should be recorded.

Regarding the format of such traces, Value Change Dump (VCD) has a ubiquitous appearance across various simulators. It produces a quite compact structure by adding each value change as a line of entry into the text-based trace file, after assign a symbol to each variable under recording. SystemC provides its own VCD support with facilities like *sc_create_vcd_trace_file* and *sc_trace*. The ModelSim tool uses a proprietary format called *Wave Log File (WLF)*, which is the data format behind **Figure 2.18**.

Simulation traces, such as that one in **Figure 2.18**, are an important input for our iterative, simulation based test generation, to be introduced in a later chapter.

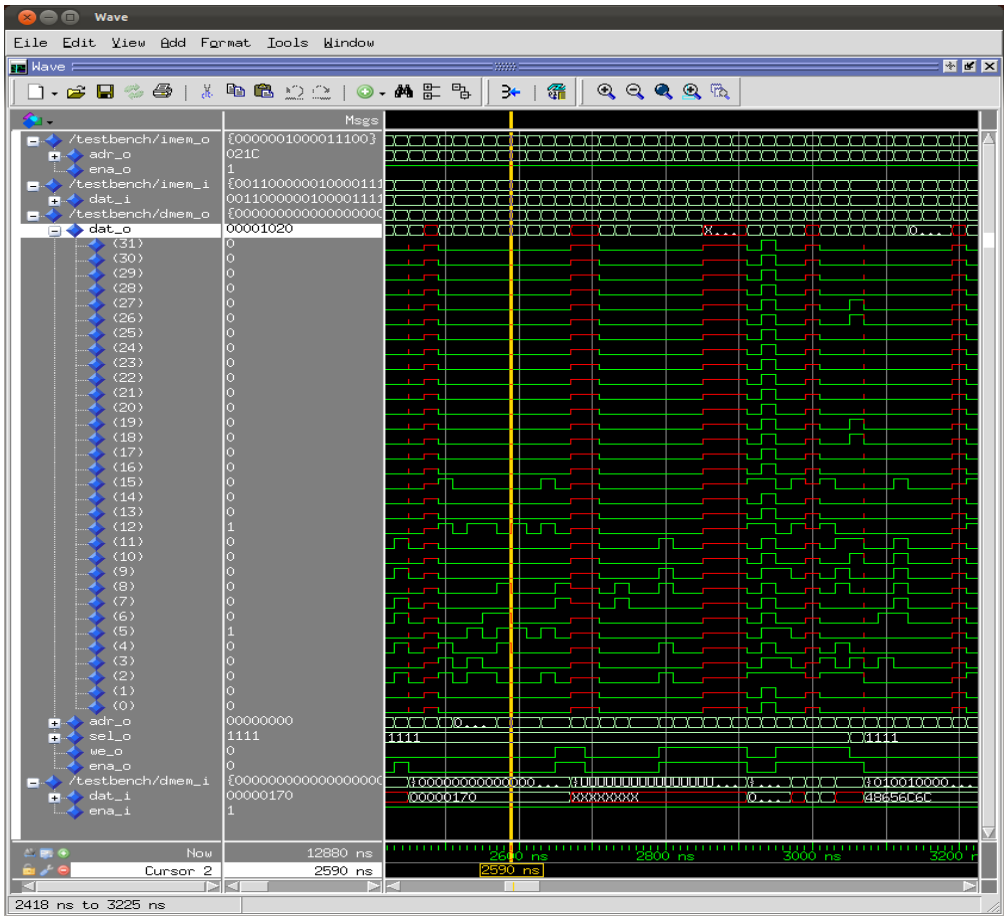


Figure 2.18 Example simulation traces in WLF format, monitored from a microprocessor design simulation with ModelSim. *imem_o*, *imem_i*, *imem_o*, and *dmem_i* are microprocessor ports and selected for monitoring.

Metrics for Simulation Quality

At some time, we have to answer the question: "are we *done* with the verification?" – which corresponds to the *verification closure* problem. We may recall *verification closure* as a point that we are sure that incompleteness and incorrectness no longer exist in the design under verification.

On one side, this *sureness* is indeed a *subjective* matter. On the other side, we are able to use quantitative metrics to gauge an object distance between the current verification status and the closure, and to use this gauge to decide a closure. This gauge is then also said to be measurement of the thoroughness, adequacy, or completeness of verification.

Statement coverage is one such metric in a relatively basic form. The introduction to a wide range of other metrics that can be used for hardware design simulation is made separately in Section 2.3. Before that, we define the approach that we call *quality metrics driven verification*.

2.2.1. Quality Metrics Driven Verification

Quality metrics driven functional verification is a simulation based design verification process that not only *employs one or a set of effective, quantitative metrics to systematically gauge a distance to verification closure* – the quality of verification, but also *integrates metrics-directed, preferably also automated test generation procedures for efficiently improving such quality measurement*.

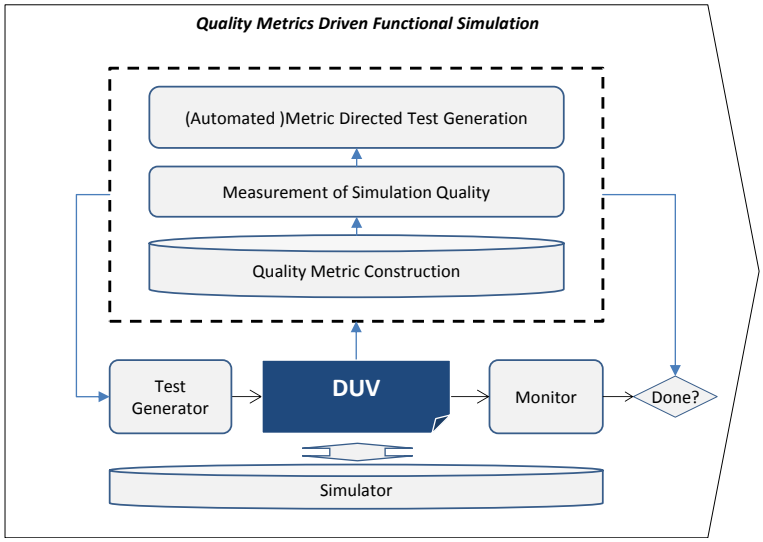


Figure 2.19 Quality metrics driven functional verification.

This idea is outlined in **Figure 2.19**, on top of an existing simulation flow. The quality metrics should first be defined on the design under verification. The metrics measurement is then used to guard a decision that "we are done with verification". It should further enhance an existing test generator, by an automated steering towards quality metrics.

From such a metrics driven simulation process, we can expect the following:

- Through strict governance of the simulation process by quality metrics, the verification should automatically achieve a high-quality status, when we decide verification closure according to these metrics. Certainly, this *high-quality* depends on the quality, or effectiveness of the metrics themself.
- Through efficient, automated test generation methods, the test selection effort should *not* be increased significantly and remain at the same levels.

These expected advantages are illustrated in **Figure 2.20**, on each of the three simulation approaches just presented.

Moreover, as stated, this systematic, stringent quality management by metrics driven simulation is a special necessity in an IP-based SoC design flow, since i) IP designs need to be verified as thoroughly as possible – as high-quality as possible – to ensure its

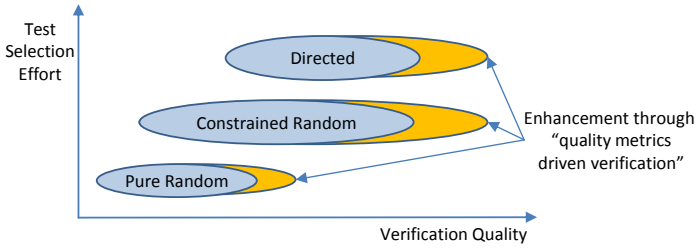


Figure 2.20 Expected enhancement from *quality metrics driven verification*.

successful integration in a possibly different place and ii) SoC system designs need also verification in a more systematic way because of the higher IP integration and other advanced techniques like TLM.

2.3. Quality Metrics for Functional Simulation

In the following, we introduce not only common basic metrics for HDL simulation, such as *statement* and *toggle coverage*, but also several advanced metrics that are active research topics, including *functional coverage*, the *observability based coverage*, and *mutation analysis*.

An emphasis is granted to *mutation analysis*, as it will become the focus our verification methods. Its problem of test generation is also compared to the *automatic test pattern generation (ATPG)* in circuit manufacturing test, because of their similarity in the use of fault modeling.

2.3.1. Statement Coverage

Statement coverage, also called *line coverage*, is defined as how many statements, or how much percentage of statements of a design have been executed during its simulation. It measures the degree of a design being exercised during simulation. In fact, because of its definition relying merely on statement execution, which is a common observation in software testing and hardware simulation, statement coverage has its wide application in them both.

Consider the decoding part of a microprocessor design, which may probably consists of several *case* or *if* branches, for the handling of individual instruction types. During simulation and test generation, if one type of instruction has been omitted and statements that belong to the corresponding branch of decoding then not been exercised, statement coverage will report this incompleteness of verification.

The rationale for statement coverage is straightforward. *Only* when a portion of design is executed, possible design errors residing in this portion may cause erroneous simulation

behavior and thereafter be observed.

Both the construction of the metric and its measurement should be of minimal cost. In particular, the cost of simulation, i.e. the decrease of simulation performance due to statement coverage measurement, should be negligible.

As one of the earliest metrics, and arguably the most basic one, statement coverage has integrated support in many HDL simulation tools, like the ModelSim simulator that we mentioned.

2.3.2. Toggle Coverage

Toggle coverage is another widely supported simulation metric by HDL simulators. When a signal bit has been once toggled from '0' to '1' *and also* from '1' to '0' during simulation, the bit has a 100% coverage. If only a one-way toggling has happened, it receive a 50% coverage. Then the toggle coverage for the whole design simulation is calculated by summing up results on all the bits.

This also measures the degree of design's exercise during simulation. The idea is that by enforcing a more intensive design activity in simulation – bit toggling, we should have a greater chance to incite as well as observe hidden design errors.

Measuring toggle coverage requires only some extra monitoring on simulation traces. No extra effort on metric construction is required and the original design simulation is also not affected by the measurement.

2.3.3. Functional Coverage

With *functional coverage*, the metric must first be defined by a user, by defining a set of functional *coverage points* that are interesting to the user. Each coverage point is defined on a design variable or one of its multiple fields, as a collection of so-called *coverage bins*, which represents specific ranges of that variable or field. For example, we want monitor the history of transaction *addresses* that happened on a PLB bus during simulation. A coverage point can be associated on the address of PLB transactions, with the bins gathered as the address ranges of all salves. These are the specific functionalities that should be exercised on PLB – therefore the name *functional* coverage.

The coverage bins records not only whether a variable range has been hit during simulation, but also the number of such hits. Moreover, the *product* of two coverage points can defined as a *cross coverage point*. To measure this cross coverage, values of both variables, on which the two coverage points are defined, should be observed at the same cycle of simulation.

We can further use a microprocessor design as another example. Assuming *instruction* is the variable for instruction input and *opcode* is a field of *instruction* representing its type – usually with a fixed bit-length in a RISC processor, we could easily define coverage bins on *opcode* according to the microprocessor ISA specification: *arithmetic*, *logical*, *shift*, *branch*, *load/store*, and so on. Then we are able to record the distribution of *opcode* in an entire simulation.

In fact, we may view toggle coverage as a very basic form of functional coverage. The toggling of one bit, back and force, is defined as a coverage point. This functional coverage metric is defined without advanced knowledge of the meaning of variables or signals, also without the user involvement.

Though the concept of functional coverage is quite natural, the native support from languages and tools just surfaced in recent years. The SystemVerilog language [51], as an effort to combine HDLs and Hardware Verification Languages and adopted as IEEE standard 1899-2005, provides direct constructs for functional coverage: *coverpoint*, *bins* that belong to a coverage point, and *cross* on a pair of coverage points. Recent research tries also to enhance SystemC with a functional coverage library [52] [53].

2.3.4. Observability Based Coverage

In [54], a so-called *observability-based coverage* is defined. It addresses a shortcoming of code coverage and functional coverage, both of which totally omit an important criterion for a testing or simulation process to be successful: any erroneous behavior of the design under testing must to be incited *and* propagated to specific design location, so that it can be observed.

For this, observability-based coverage introduces *symbolic tag* to model this error propagation. During design simulation, a symbolic tag Δ can be attached to a design variable, as a potential error that should be propagated through statements. The calculation with tags then follows a set of rules, called Δ-calculus. **Figure 2.21** shows two examples. Note that for Boolean operations, such as *AND*, a tag equals the *D*-calculus in gate-level test generation [55].

However, this error-modeling tag is made suitable for functional design simulation by defining also the calculus for other higher-level operations, such as *addition* or *multiplication*. For a statement $c = a + b$, the result is defined to receive a positive tag, if both operands are with a positive tag, or a positive tag versus a tag-free. If one operand has a positive tag and the other one a negative tag, the tags are defined to compensate each other and the result will have no tag. Also, propagation of tags through control statements are defined.

AND	0	1	0 + Δ	1 - Δ
0	0	0	0	0
1	0	1	0 + Δ	1 - Δ
0 + Δ	0	0 + Δ	0 + Δ	0
1 - Δ	0	1 - Δ	0	1 - Δ

c = a + b	b + Δ	b - Δ
a + Δ	c + Δ	c
a - Δ	c	c - Δ

Figure 2.21 Example Δ-calculus for *AND* and *addition* operation.

Such definition is necessary, since, in contrary to *D*-calculus, the observability-based coverage works with design simulation, without assuming the synthesizability of a design. This is a general difference between simulation metrics and gate-level fault models, which we will discuss more in a later section.

On the one hand, the observability-based coverage smartly addresses the problem with other metrics omitting error-propagation. On the other hand, its biggest disadvantage is the dependence on *symbolic calculation*, which i) requires an extra simulation engine and ii) is usually considered *not* practical for real designs. Though in [56] [57], advanced methods for calculating tags are proposed, meant to be more efficient compared to the original definition, practical adoption is still restrained. Further, the correlation between a tag and real design errors is neither straightforward nor investigated.

This homogeneous modeling of design errors is different from the heterogeneous error injection from *mutation analysis* in the next section.

2.3.5. Mutation Analysis

Mutation analysis, also called *mutation based testing* or just *mutation testing*, is a unique, fault-injection based simulation, or testing metric. It manages systematically the quality of functional simulation, by measuring the simulation's capability of revealing design errors – though artificially induced. This is similar to the observability-based coverage that we previously introduced, but different in the way of fault-injection.

The process of *mutation analysis* is summarized in **Figure 2.22**. As other metrics, it is supposed to be laid as an extra quality management layer upon a simulation based functional verification process.

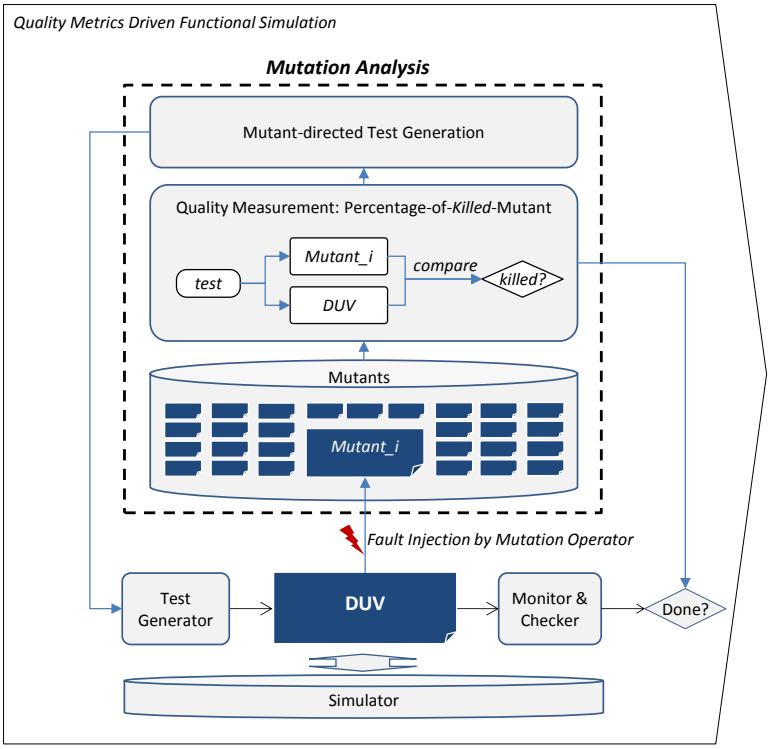


Figure 2.22 ***Mutation analysis*** **in the context of simulation based functional verification.**

First, a copy of design under verification is created and a so-called *mutation*, which is a single minor modification to the design code, is applied on that copy, such as:

a := b **and** c; $\xrightarrow{Muation}$ a := b **or** c;

Such mutation operations are defined by *mutation operators*. Each mutation operator defines a certain type of code modification, such as "replacing an *and* operator to an *or*" like above, "replacing a *plus* operator to a *minus*", "changing a '0' bit to '1' ". To enable mutation analysis for a specific language, a set of mutation operators should be firstly defined on the syntax of the language. This also brings the language-specific nature to a mutation analysis metric. Later in this section, we will see how an industrial HDL mutation analysis tool defines the mutation operators on, for example, VHDL.

The mutated copy is called a *mutant* of the design. A large amount of mutants can be generated to form a metric database, since theoretically we may apply each mutation operator to every possible location of the design code.

Each mutant is supposed to be simulated, separately, in addition to the simulation of the original DUV. A mutant is said to be *killed* by a test, if during simulation, it *produces*

a different output under this test compared to the output of the original design simulation. This can be decided by comparing the simulation traces of both at output ports.

The number, or *percentage of mutants that were killed* during a simulation process with a certain tests becomes the quality measure of this simulation, or this set of tests. Under this definition, a mutant can be removed from the metric database, as long as it was killed at some point.

As with other metrics, using such a quantitative measure on simulation progress, we are able to systematically handle the verification closure problem and answer that "are-we-done" question.

Nevertheless, there are two general problems that make this functional verification closure under mutation analysis difficult:

- Mutation analysis imposes a lot of *extra* simulation time upon the original verification process. As mentioned, the amount of created mutants can be huge for a design. If we apply each mutation operators to every possible operator or variable at every line of design code, we may obtain the number of mutants as (*Lines-of-code* × *#-of-mutation-operators* × *K*), assuming *K* a constant of average frequency that mutation operators can find their possible usage at a line, approximately.

  It means a design with, for example, a thousand lines may derive a mutation analysis metric with ten thousand mutants. Combined with the fact that all the not-yet-killed mutants need to be simulated separately with all tests generated, the metric measurement time can largely exceed that used for the actual simulation, and even become unmanageable without targeted, efficient test generation.

- The task of selecting a test that kills a mutant is itself a hard problem – as we will discuss later with more details. If we could have an automated, efficient procedure for generating mutant-killing tests, the first problem – too many mutants to be killed – would even become directly solved, or at least largely alleviated.

Therefore, this high computation requirement from mutation analysis has long been identified as the barrier of its adoption.

In the following, before going to define the test generation problem for mutation analysis and introduce further advanced techniques for easing the computation requirements on mutation analysis, we first try to explain the rationale behind mutation analysis as a verification quality metric. At the end, we will introduce as example a complex, industrial tool that implements HDL mutation analysis, by incorporating most of the advanced mutation techniques from research.

Rationale: Double Effectiveness

We discuss why mutation analysis can be used as a quality metric for functional verification, whose ultimate goal is to uncover any deviation between design and verification – incompleteness or incorrectness. Such a metric should be used to gauge a distance between the current verification status and the verification closure.

We discuss rationale behind using mutation analysis in two aspects:

- After the mutants are created, they are design errors and it is *an intrinsic and fundamental requirement for the simulation tests to be able to reveal these errors*.

  On the other side of this aspect, *if the simulation cannot reveal the mutation errors and killed the mutants, how can we be confident about the quality of the simulation*? It is similar to the other metrics in such consideration. If a statement has not been executed in a simulation, or a functional coverage bins been missed, we may have reasonable doubt about the thoroughness of the simulation – though we still cannot exclude the existence of possible design errors, if we have a 100% statement and functional coverage.

  To support this effectiveness argument on mutation analysis, the *mutation operators* should be defined to be representative of real design errors that a designer can possibly make. They should best be concluded from extensive, statistical study of such designer errors, for a specific language.

- A *Coupling Effect* is assumed, and partially proved by experimental studies, which states that *if a set of tests is able to kill more mutants created from a design under verification, they will also be able to expose the real exiting bugs in the design.*

  At the origin of mutation analysis for software program testing, the coupling-effect was proposed merely as premise [58]. Later, there has also been experimental studies [59] [60] to evaluate this premise, with positive results. The investigation on *coupling effect* is *not* included in this thesis, but with it used as a general assumption.

We call them the *double effectiveness* of mutation analysis.

Test Generation Problem

We describe the problem of test generation for killing a certain mutant. The problem is defined only to a necessary degree for this moment. A more accurate model for problem discussion and solution will be presented in Chapter 5.

A control flow graph (CFG) as shown in **Figure 2.23** should be enough for the moment. It extracts a structure from a software program – where mutation analysis originates – or a HDL design in VHDL, Verilog, or SystemC.

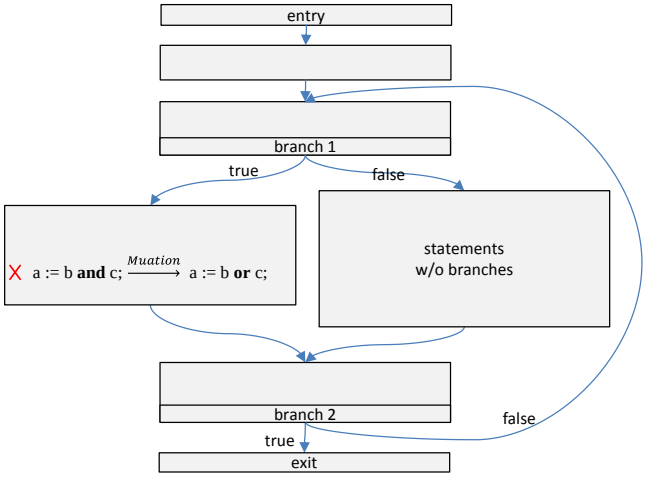


Figure 2.23 A control flow graph with mutation marked.

The CFG also represents a mutant by marking the mutation. Three tasks, or conditions must be fulfilled for killing this mutant.

- **Reachability**. The execution of simulation needs to first *reach* the location of mutation. Only when the mutated statement has been executed, we may observe a different mutant behavior from the original design.
- **Activation**. The mutant needs to be *activated* by the mutated statement being executed in such a way that a local deviation is created. It means that in the example mutation, the result of *(b or c)* is evaluated to a different value from that of the original code *(b and c)*, i.e. as a condition $(b \text{ or } c) \neq (b \text{ and } c)$.
- **Propagation**. Any local created deviation needs then to be *propagated* to the design output, so as to result in deviation also at the output and therefore, the mutant being killed according to definition.

Together, they form a *necessary and sufficient* condition for a test and the simulation under this test to kill the mutant.

The literature discussion on existing test generation methods will be left to the related work in later chapters. We only mention here than in general, the *propagation* problem has *not* been tackled. It is even not possible for an analysis with CFG, as propagation is naturally a *data* flow process.

At the opposite of mutant-killing test generation, there exists the problem of identifying so-called *equivalent mutants*, which are those mutants that intrinsically cannot

be killed by any mutants. Reasons for such impossibility for a mutant to be killed can be: i) The mutated statement is *not reachable* under all cases, for example as a redundant code; ii) The mutated statement, though syntactically different from the original one, will never compute a different result *in its context*. For example, we cannot differentiate *(a>0)* from mutated code *(a>=0)*, when *a* is always assigned a valued greater than 0 before this line. iii) Regarding propagation, the mutant is un-killable if, for example, the result of the mutation statement is not even used in further computation at all.

In any of these cases, the mutant is *equivalent* to the original design, under our observation at design output.

Automated identification of equivalent mutants is an un-tackled problem in mutation analysis research [61]. It is *not* a focus in our work.

Techniques for Mutation Analysis

Since long years of research on mutation analysis, in software testing and hardware design verification, advancing techniques have been proposed to reduce the cost of mutation analysis and to improve its adoptability. Some influencing ones are *selective mutation* [62], *mutation schemata* [63], and *weak mutation* [64] [65]. These have also be implemented by the industrial EDA tool for HDL mutation analysis, which will be introduced next and also used as a basis of our research.

- **Selective Mutation**. Recall the number of possible mutants to be generated by a set of mutation operators on a design approximately as (*Lines-of-code* × *#-of-mutation-operators* × *K*). With *selective mutation*, we make a simple trade-off and generate selectively a much smaller subset from all possible mutants, as **Figure 2.24** shows. We may exclude the application of some mutation operators. Or we may choose to apply a mutation operator less frequently, i.e. not applying it at every operator or variable where it can be applied. We may even just skip the mutation at specific lines of design code. The purpose is to compress the mutant database in a manageable size.

  The degree of compression is a trade-off between effectiveness of mutation analysis, in terms of its stringency of test qualification, and the required simulation time. The minimal set of tests required to kill all possible mutants is certainly a superset required in selective mutation. There are also early studies [66] to experimentally evaluate this relation.

- **Mutation Schemata**. Simulation time is not the only cost of mutation analysis, in fact. Time for *mutant compilation* is another, before we can simulate them. This compilation time is huge, if we assume thousands of mutants. The situation is even worse, if we consider the functional simulation as an iterative process and each

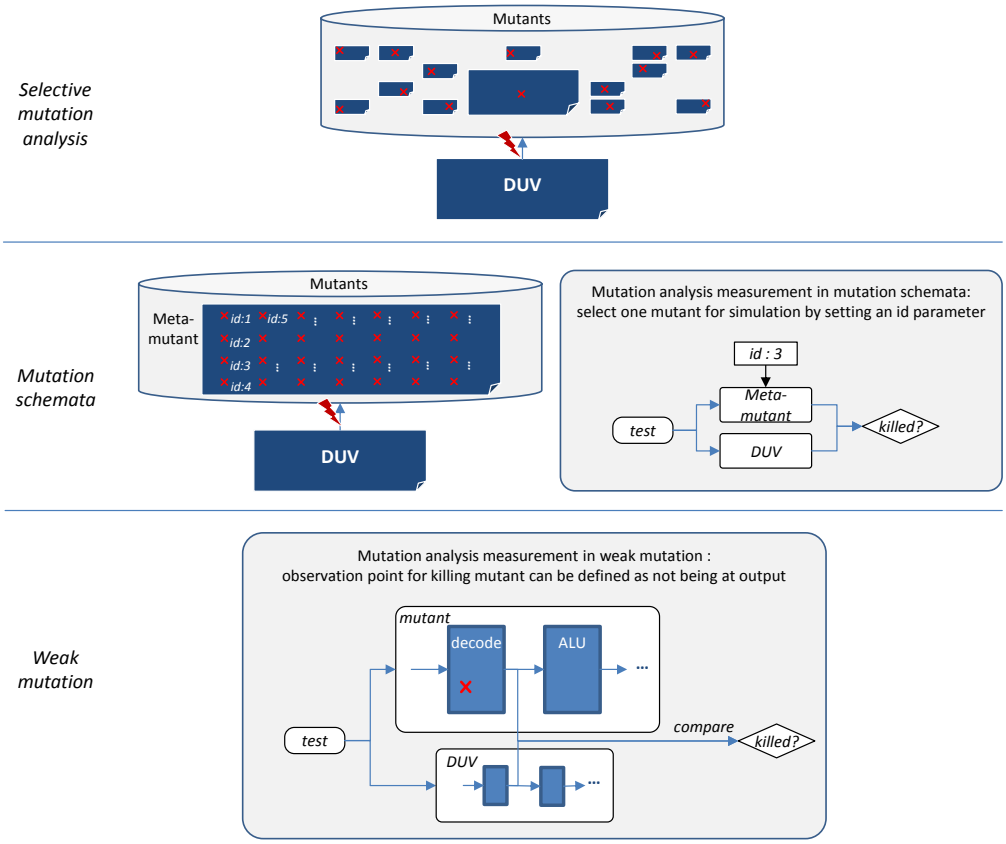


Figure 2.24 Mutation analysis techniques.

time the design is debugged and modified, all the mutants need to be created and compiled again, alongside the original design compilation.

With mutation schemata, mutants are created as *one* copy of DUV. As **Figure 2.24** shows, all the mutations are instrumented on that single copy, each coded with an *id* parameter. To simulate one mutant and see whether it can be killed by a test, the *id* parameter is set to select that corresponding mutant. The selection can be implemented by, for example, *if-then* statement that governs each mutation with a unique *id*.

Meta-mutant is used to call that single copy of DUV with all mutants coded in. We finish the compilation of all mutants by compiling only the meta-mutant once. Though the parameterization of meta-mutant introduces minor overhead, mutation schemata is almost a necessity for handling designs with practical size and thousands of mutants. Further, it has *not* any influence on the effectiveness of mutation analysis.

- **Weak Mutation.** Weak mutation is a further trade-off between mutation analysis effectiveness and the requirement on test generation. Instead of defining the *kill* of

mutants as deviation of simulation at *design output*, *kill* can be defined at a point anywhere along the path between mutated statement and output. As the example in **Figure 2.24** shows, the observation of whether a mutant being *killed* is defined right after the *decode* unit, on the result of decoding.

At one extreme, if *kill* is defined immediately after the mutation, i.e. on the result of the mutated statement, the requirement on *propagation* is eliminated and *kill* equals *activation*. At the other extreme, *kill* is defined on design output and we have the original mutation analysis, which is also called *strong mutation* for distinguishing.

As with selective mutation, the minimal set required for any specific weak mutation must be a subset of original, strong mutation.

Certitude: An Industrial EDA Tool for HDL Mutation Analysis

Mutation analysis has its origination in software testing [58] [67]. If we consider the task of *software testing* and that of *functional hardware design verification*, they have intrinsically *no* difference, to be finding any incompleteness and incorrectness of an implementation from its specification. In [68], [69], and [70], the application of mutation analysis to HDLs has been discussed for the first time.

From company Synopsys®, Certitude(TM) is an industrial EDA tool that implements mutation analysis for several HDLs, which include VHDL, Verilog, and SystemC. It can be used with most commercial simulation tools, such as ModelSim(TM) from MentorGraphics or VCS(TM) from Synopsys itself, thanks to a seamless integration.

Figure 2.25 shows two screenshot from the tool, which report the result of mutation analysis on the simulation of a VHDL floating point arithmetic design. As illustrated, Certitude implements *mutation schemata*, by instrumenting all mutants into one design copy. When we click on a colored mark, the mutation at this location and the induced mutant with a unique ID is shown, along with the mutant status after simulation: activated or non-activated, propagated or non-propagated when activated.

We list the names of several mutation operators – not complete – defied by Certitude, without going into their details, since many of them are self-explaining:

- **Operator-or-to-and**; **Operator-and-to-or**; **Operator-and-to-nand**;
- **SwapOperand**;
- **BitFlip-'0'-to-'1'**; **FlipFirst**; **FlipLast**;
- **DeadAssign**;
- **ConditionFalse**; **ConditionTrue**; **NegatedCondition**; **ElseDead**;

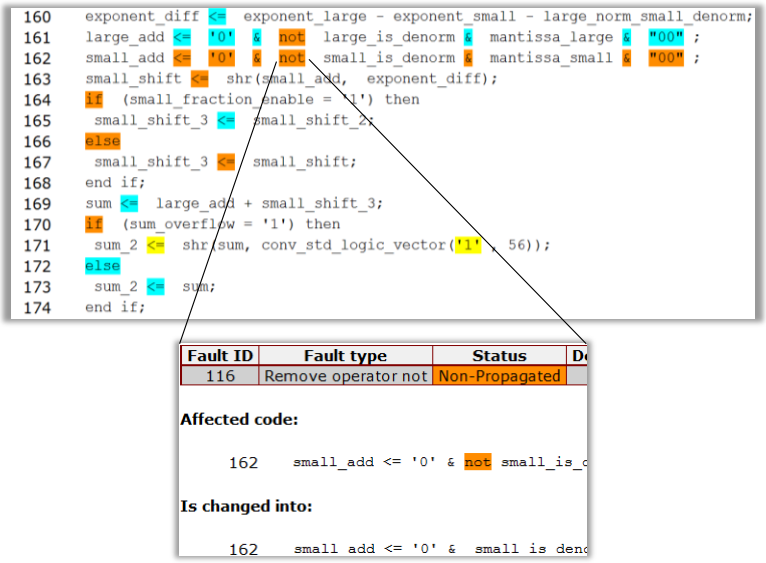


Figure 2.25 Certitude: an industrial EDA tool for HDL mutation analysis. The color marks annotate not only the locations of muation but also the mutant status during simulation. "Fault" here is used in equivalence with *mutant*.

Certitude also implements *selective mutation*, by allowing a user to set *MaxFaultPerLine* as a parameter for the tool during the creation of mutants. Moreover, *weak mutation* is implemented in a way that we can define any signal between units as one of the points for observing the kill of mutants.

We used this tool to construct examples and for evaluations. However, it does *not* imply any restriction of our methods, for mutation analysis enhancement, to this specific tool. More literatures on Certitude can be found in [22] [23].

2.3.6. Comparison of Metrics

With **Figure 2.26**, we further summarize a comparison between the simulation quality metrics that we introduced so far. In particular, we will compare the rest to mutation analysis, as it is our focused metric and will play a central role in all later chapters.

- **Metric construction**. Though the definition of mutation operators for a specific language, their implementation as code instrumentation, and the integration with simulation tools for mutant measurement are all complicated tasks, once they are finished as a tool, the mutation analysis becomes a fully automated process, except for test improvement. For a design under verification, the construction and compilation of its mutant database require little effort thanks to *mutation schemata*.

  The other metrics are also to be established by automation, except for functional coverage, which requires a user to define coverage points and bins for each design. This leads to another issue of functional coverage: the quality, or effectiveness of

Metric	Metric construction	Measurement	Test generation Problem
Statement coverage	+ automated	+ minimal cost	reach
Toggle coverage	+ automated	+ minimal cost	reach and toggle
Functional coverage	- manual	+ minimal cost	reach and hit
Observability-based coverage [54]	+ automated	symbolic	reach, symbolically activate and propagate
Mutation analysis	+ automated	- high cost	reach, activate, and propagate

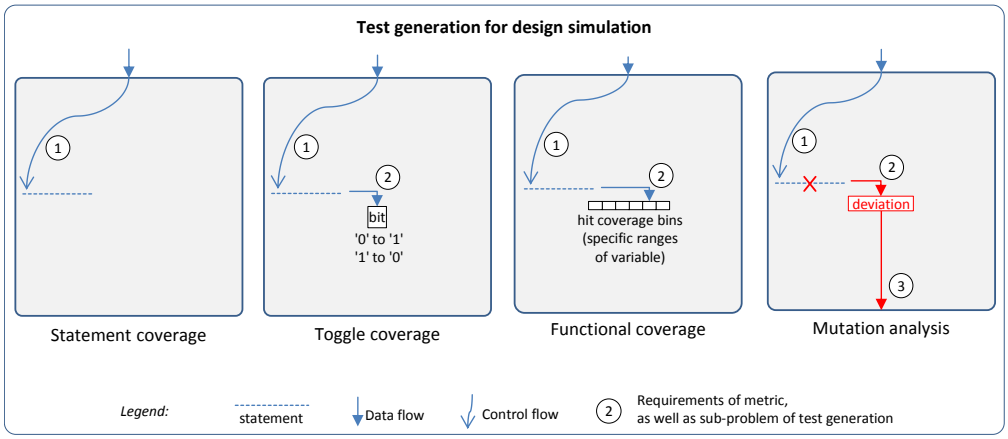


Figure 2.26 Comparison of metrics for simulation based functional verification.

the metric itself depends on the user capability and effort. Therefore, functional coverage is a *subjective* metric and mutation analysis, and the rest metric, are *objective*.

- **Measuring metrics**. As mentioned, the extra simulation time imposed by mutation analysis is the biggest challenge of its adoption, lagging behind other metrics. Still, we consider the symbolic simulation in observability-based coverage even more computation-expensive and *not* always practical.

- **Test generation**. Mutation analysis highlights an intrinsic requirement on simulation and its tests, namely their capability of stimulating potential design errors and propagating the erroneous behavior to pre-defined observation points. In this way, mutation analysis imposes a more stringent qualification on tests and, correspondingly, a more difficult job for automated test generation.

  This is not addressed by other metrics, except for the observability-based coverage. Again, in contrast to the symbolic-tag manipulation, mutation analysis relies totally on actual HDL simulation.

We may further observe that a hundred percent toggle or functional coverage does *not* necessarily lead to 100% statement. Also, full statement coverage does *not* imply 100% toggle, or 100% functional coverage, neither. They can be complementarily used. Further, it is clear that mutation analysis requires a 100% statement coverage, assuming mutants are distributed to every lines of code.

2.3.7. Circuit Manufacturing Test and ATPG

Although *circuit test* after IC manufacturing is another separate phase in the whole EDA flow and forms itself a big research area, we could immediately find its similarity to functional design verification, when mutation analysis is used, as both employ fault models to quality tests.

In the following, for a comparison we first introduce their differences in three aspects, as summarized **Figure 2.27**, and then conclude why the test generation algorithms – Automatic Test Pattern Generation (ATPG) – are not applied for mutation analysis.

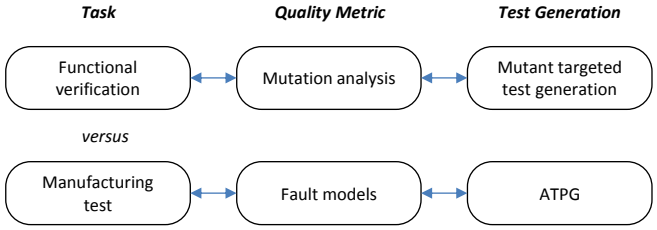


Figure 2.27 Aspects of comparison between circuit manufactring test and functional design verification.

First, the tasks of functional verification and manufacturing test are totally different in an EDA flow, as shown in **Figure 2.28**. Function verification intends to uncover errors that are introduced during HDL design, i.e. any deviation from specification. Manufacturing test is applied to *each* circuit device after their fabrication, to ensure no physical cell defects are introduced during this process. Presented only for a further comparison, in an FPGA based implementation flow, such circuit testing is no longer necessary, as there is simply no step of manufacturing, assuming the FPGA device is error-free.

Second, the rationale and mechanism behind defining a test qualification metric by fault modeling is different. This is shown by **Figure 2.29**, without going into the details of various gate-level fault models.

- In **mutation analysis**, we have discussed the rationale of mutants as *double effectiveness*: i) mutants model typical design errors and when they are created, simulation should be able to reveal them; ii) mutants are coupled with real design bugs, in a way that if simulation can kill mutants, it will also be able to find real

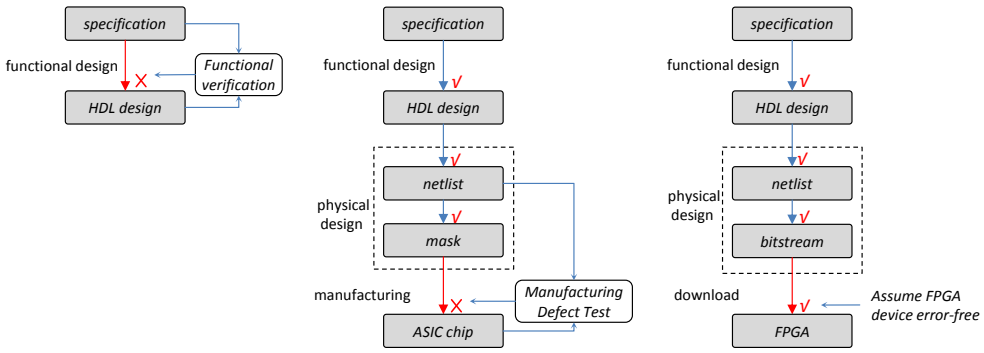


Figure 2.28 Task of functional verification and manufacturing test.

bug.

- In **gate-level fault models**, for example stuck-at fault, a logical gate fault abstracts a physical defect of certain type that may happen during chip manufacturing, as a direct mapping. Therefore, when tests are generated by ATPG that detect such a fault, they will guarantee the catching of that manufacturing defect.

Third, the test generation problem is usually on a different basis, for gate-level fault models in circuit test and HDL mutants in functional verification.

Usually, ATPG algorithms – consider the earliest D-algorithm and the follow-ons [55] [71] [72] on stuck-at faults for example – take only a combinational logic area as input. Even with the appearance of sequential ATPGs later [73], for large synchronous sequential logic, they still mostly follow a *structural testing* scheme and rely on *scan-chain* based techniques to restrict the problem to small logic areas and to apply the tests generated under such restriction, as illustrated by **Figure 2.30**. For the circuit under test in its scan mode, the registers as input for that specific design portion are set by scanning-in test input. The results to be checked are then scanned-out, be compared with expected results according to the original netlist.

For mutation analysis, tests are to be generated for the functional verification purpose and applied to design input. The entire design should be the target of any test generation

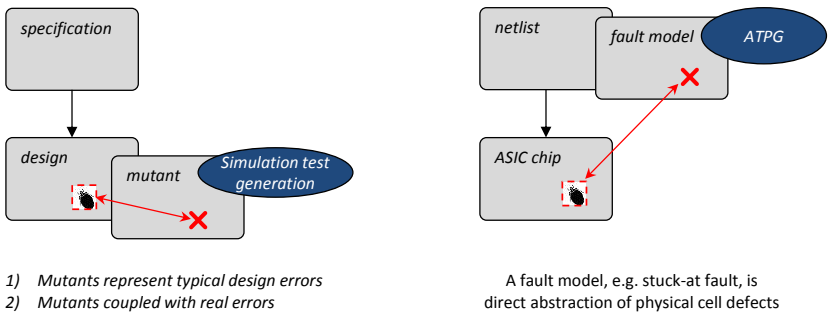


Figure 2.29 Rationale behind fault modeling in mutation analyis and gate-level fault models.

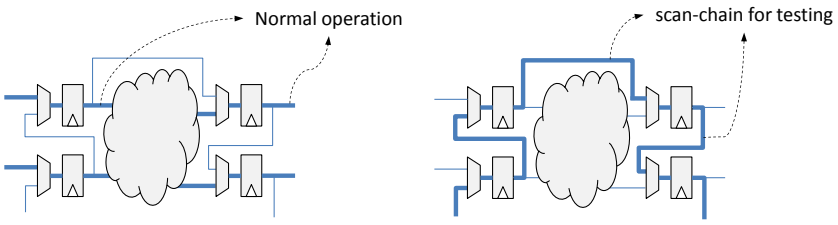


Figure 2.30 Scan-chain for structural testing, used by ATPGs.

procedure. Three sub-problems – reach, activate, and propagate – have to be considered, *from design input to output*.

To conclude, ATPGs are *not* used in test generation for mutation analysis, or any other simulation quality metrics, since i) ATPGs follow structural testing and usually do *not* take the entire design as algorithm input, for example a complete microprocessor design, and ii) ATPGs work on gate-level netlist and we use mutation analysis, or other simulation metrics, on any design that is simulatable, without assuming it to be synthesizable.

2.4. Summary

In this chapter, we have established the basis for our further discussion, by first presenting a reference flow for IP-based SoC design, and then introducing both fundamental and state-of-the-art methods and techniques that are employed at different locations of the flow.

We have identified one of the most important characteristics of IP-based SoC design to be the *division and separation of IP design and SoC system integration*. This has a key implication on our consideration of verification. At IP level, an IP design needs to be verified *systematically* and as *thoroughly* as possible. At system level, a SoC design also needs a *systematic* verification, which should further be *focused on the integration* of IPs. Our approach is to construct a series of metrics-driven verification methods that cover both IP and SoC system level.

The reference flow includes SystemC, TLM, and IP-XACT as state-of-the-art techniques for IP and SoC system design, which should be taken into account for verification. These are intensively studied topics in recent research on SoC design methodology. Further literature will be discussed in the related work section of each contribution chapter.

SystemC, with a discrete-event simulation core the same as most other HDLs, can be used for both behavioral and RTL IP design. Such IP designs can be wrapped into TLM components, where their interfaces for SoC on-chip communication are modeled by

function calls and bundled as *TLM interfaces*, which serve the central basis of TLM IP integration in SoC system design.

Therefore, at IP level, we will consider a design under verification to be RTL or behavioral, in traditional HDLs – VHDL and Verilog – or SystemC. At system level, a SoC design under verification can be integrated from RTL IPs, TLM IPs, or even mixed.

IP-XACT is *the* standard language for describing IP metadata – its design files, exposed on-chip bus interfaces, and configurable parameters – and SoC integration based on these metadata. By assuming IP-XACT as the default SoC design language, we should be able to concentrate on the verification of system integration.

We further assume *simulation* as our way of functional design verification. We have outlined the components in a simulation process: the DUV, a simulator that possibly supports multi-language, RTL/TLM mixed-level simulation, a test generator, a monitor and checker, and the quality metrics that stands at the center of our solution to the verification closure challenge. In particular, we have introduced three different approaches for test generation: *directed*, *random*, and *constrained-random* that combines the advantages of the previous both and will be highly exploited in Chapter 4. Actually, *metaheuristic search based test generation* may be classified as another alternative, which will be considered in Chapter 5.

Then, we have defined what a *quality metrics driven verification* is and introduced various metrics that are currently in use. In particular, we have compared mutation analysis to other metrics and identified its unique requirement for test to reveal the typical, purposely injected design errors. The rationale behind such stringent test qualification have further been summarized by us as *double effectiveness*.

From now on, mutation analysis becomes a real focus of our research on *quality metrics driven verification*, though in general, we do not see our methods restricted to mutation analysis, meaning that their adaptability to other metrics should be straightforward. Identification of *equivalent mutants* is a problem *not* tackled in this work.

The basic problem of mutant-aiming test generation has been defined as three sub-problems: *reachability*, *activation*, *propagation*. Advanced techniques for alleviating the problem of high computation requirement from mutation analysis have been introduced, including *selective mutation*, *mutation schemata*, and *weak mutation*. Certitude, a sophisticated HDL mutation analysis tool from the EDA industry, has been presented, which will also be used in our evaluation.

Moreover, we have presented a brief but essential comparison between APTG in manufacturing test and mutation analysis in functional verification. ATPGs are not used in functional verification because of its structural working scheme at gate-level and the assumption that our design under verification is *not* necessarily synthesizable.

In next chapter, we will present an overview of our methodology to systematically enhance the quality of functional verification for IP-based SoC design, using the metrics driven approach.

CHAPTER 3: Methodology Overview

In this short chapter, we give an outlook on the overall contribution of this thesis: *a systematic, simulation based, quality metrics driven functional verification methodology for IP-based SoC design*, as shown in **Figure 3.1.**

The bottom part of the figure refers to the IP and SoC design flow, languages, and methods that we have discussed in the previous chapter. In particular, we have motivated the need for *metrics driven verification* (MDV), as well as advanced metrics such as *mutation analysis*, which is identified as the focus of this thesis. Recall that with MDV, a verification process should not only be guarded by a quality metric, but also use metric-targeted test generation to efficiently improve such quality.

Based on the discussions, we may generally identify the following gaps between state-of-the-art techniques and our desire for an efficient yet quality-enhancing verification flow. Concrete motivation for each chapter will be expanded later.

- At IP design phase, with the emerging of EDA tools for HDL mutation analysis recently – such as Certitude, which leverage a long history of mutation analysis research, we still *lack efficient, practical test generation methods for this HDL mutation analysis*.
- At SoC system design phase, with the recent establishment of standard system-level design languages and techniques, for example TLM and IP-XACT, we still *lack a systematic verification way for SoC system-level, in general, and any quality metric for such verification, in particular*.

For this, our verification methodology consists of three main components:

- For the functional verification of an IP design with HDL mutation analysis, we first consider using random simulation to achieve *a primary level of killed mutants*. We propose to integrate *a feedback directed adaptation loop* into constrained random simulation (CRS). The goal is that by consistently adjusting a test model in CRS, we will be able to obtain a more efficient process of killing mutants. This will be discussed by Chapter 4.

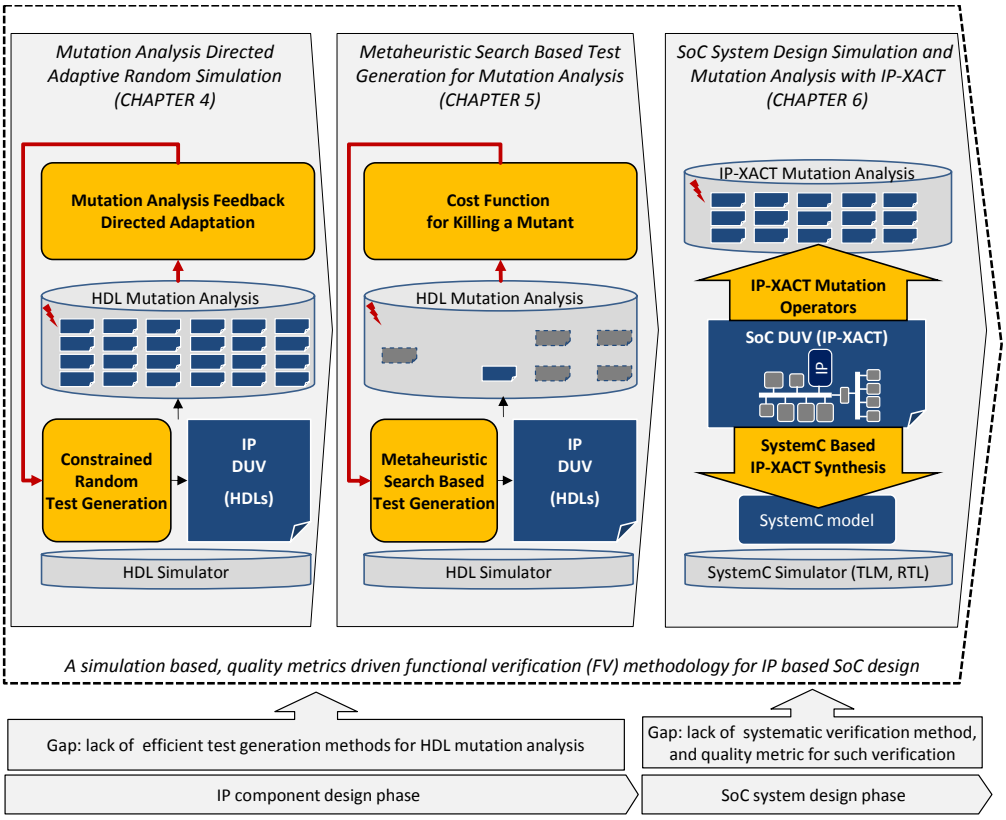


Figure 3.1 Overview of our methodology. Main contributions are highlighted.

- After random simulation, we expect some "hard" mutants left un-killed. We further consider applying a *metaheuristic search based test generation* to *each* of them. It means that a metaheuristic – for example a *local search* – is employed to search the design input space, to iteratively move towards a target test that can kill the mutant. To steer such search, we need to define a cost function that measures the progress of a HDL mutant being killed. We will present a *graph based definition of such cost function* in Chapter 5. With these first two components, we expect an extensive, high-quality IP verification.

- Moving to system level, we first assume IP-XACT as the default language for SoC integration. For a systematic SoC verification framework, we propose i) *SystemC based IP-XACT synthesis* to enable SoC system designs simulation and ii) *a set of mutation operators on IP-XACT schema* to enable mutation analysis for such simulation. They will be detailed in Chapter 6.

Last, we want to emphasize the coherence of these chapters as an integrated verification flow, which should find its scenarios of application by i) a SoC integrator, who is usually required to build one or several of its own special, product-differentiating IPs, which are then assembled together with third-party IPs – in such a case, it can benefit

from all three components, for both IP and SoC verification activities, and ii) an IP provider/licenser, who only develops IP level designs and should find the first two components as systematic enhancement to IP verification quality.

CHAPTER 4: Mutation Analysis-Directed Adaptive Random Simulation

4.1. Introduction

This chapter presents the first component of our verification methodology. For the first phase of an IP verification, which is meant to be as comprehensive as possible, it is yet reasonable for us to rely on a random-simulation based, light-weight method to reach a primary quality level under the mutation analysis metric.

We have explained the advantage of *constrained random simulation (CRS)* over *pure random*. With a probability model defined by weighted constraints on design input, we are not only able to generate a significant amount of tests for exercising the design, as in random testing, but also able to control the distribution of generated tests by assigning more weights to constraints of more interests, which we can*not* do with pure-random simulation.

Motivation for Metrics Directed Adaptive Random Simulation

However, when CRS being employed as the basis for our *metrics driven verification* approach, there are several problems appearing, which can be viewed as the general motivation of this chapter. Based on a microprocessor design example, **Figure 4.1** gives an illustration of these problems.

- First, initially, the probability model for random test generation is *not* defined with the quality metric – the mutants – in mind. The tests to be generated are *totally not* aimed at the target of our verification: killing the mutants. Therefore, we may expect that the test generation is *inefficient* with regard to mutation analysis.
- Second, the metric changes over simulation time, as killed mutants are consistently removed from the mutant database and the remaining mutants become the reduced target. It is almost impossible to assume that the test probability model will just

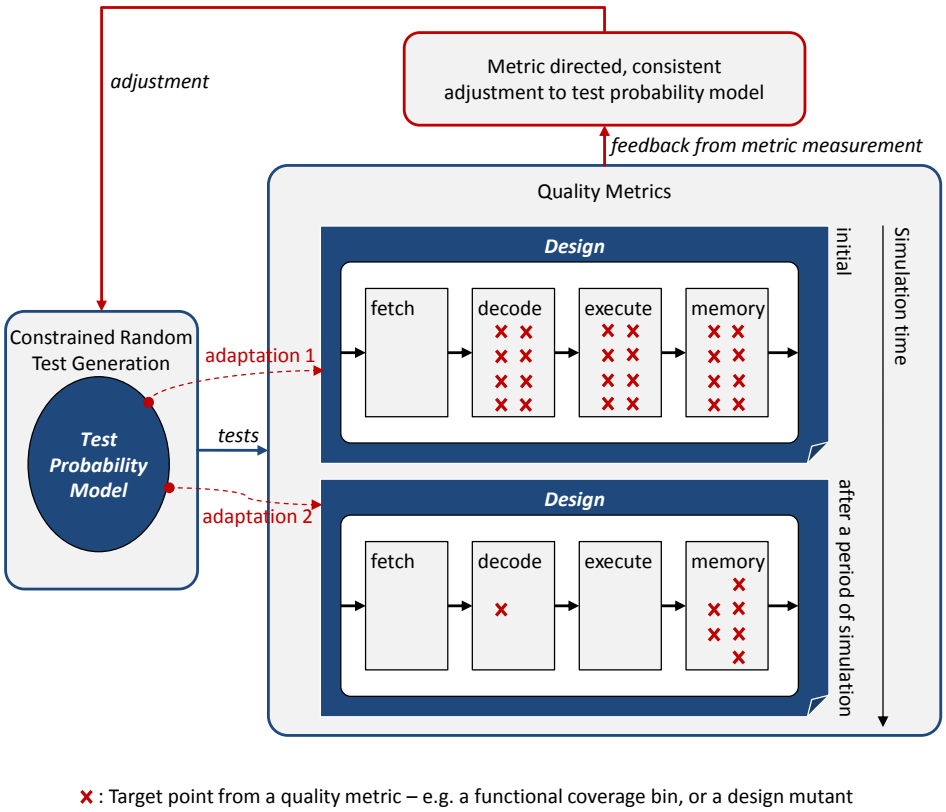


Figure 4.1 Motivation of metric feedback directed random simulation. Adaptation reason 1: test model originally *not* constructed as specific for the metric; adaptation reason 2: test model needs adaptation to the continuously changing metric.

match the changing metric. Therefore, again, inefficient tests are expected that are *not* aimed at killing the remaining mutants.

- Further, inefficient test generation is a more severe problem in particular to mutation analysis, since i) we have a stringent qualification on tests, with the mutant-killing problem already difficult to satisfy, and ii) if tests are generated aimlessly, mutation analysis requires high cost of simulation time to examine whether *each* mutant can be killed, compared to other metrics, for example functional coverage, where only *one* simulation is necessary for checking *all* coverage bins.

In fact, the problems apply *not* only to the combination of random simulation and mutation analysis, *but also to other quality metrics like functional coverage*. Only, they will be exaggerated with mutation analysis, because of the high simulation requirement, and become a more urgent motivation.

Therefore, to mitigate these problems, we consider an *adaptive* method for random test

generation, which should *continuously steer* the test model towards the mutation analysis metric, so as to obtain a more *efficient* test generation process, i.e. having more mutants killed with less tests.

There are three components in such an adaptation loop, as outlined by the figure. The first is a constrained random test generation process, containing a test probability model that should provide us the opportunity to tune and steer the test generation. The second is the quality metric measurement process, with the metric consistently changing under the randomly generated tests. The third is the adaptation block, which correlates observation, or feedback from metric measurement to any desired adjustment on test model.

Contribution of the Chapter

This chapter, as the first component of our mutation analysis driven functional verification methodology for IP-based SoC design, *contributes by proposing a mutation analysis-directed adaptive random simulation method, which is aimed at improving HDL mutation analysis efficiency. For this, we propose i) first, a combined use of Markov chain and weighted constraints for random test modeling, which enables dynamic adjustment to a probability model, ii) second, dynamic mutation schemata that not only reduces the cost of HDL mutation analysis but also enables detailed feedback collection, and iii) third, an efficiency-improving heuristic that calculates and applies consistently adjustment to test generation, with the expectation that more mutants will be killed with less tests.*

Organization

After the general motivation, we unfold the rest of the chapter with an overview of our proposal on this adaptive method, at the beginning of Section 4.2. The three constituent parts of it – the random test modeling, the dynamic mutation schemata, and the adaptation heuristic – are elaborated from Section 4.2.1 to 4.2.3, with the overall procedure again summarized in Section 4.2.4. Related literature is comprehensively discussed in Section 4.3. And the chapter is concluded by Section 4.4.

4.2. Mutation Analysis-Directed Adaptive Random Simulation

As shown in **Figure 4.2**, we propose an adaptive random test generation method for HDL design simulation, which is directed by mutation analysis as the simulation quality metric as well as adaptation basis. The simulation framework consists of several innovative components.

- **Markov-chain and weighted constraints modeled random test generation.** A prerequisite for any adaptive random simulation is a probability model for test generation, with parameters that can be adjusted dynamically at simulation time.

We employ a Markov chain augmented with weighted constraints for this random test modeling purpose. Combined, they provide us the chance to steer test generation towards particular types and sequences of tests. Further advantages and definitions of this modeling will be explained.

- **HDL mutation analysis with *dynamic mutation schemata*.** For mutation analysis, first, *mutation schemata* – using a meta-mutant to instrument all mutants into a single design copy – should be leveraged to create the mutant database. Since we are verifying IP level HDL designs, e.g. a microprocessor design, that usually have thousands of lines of code, thousands of mutants can be generated. With mutation schemata, we need only a single compilation with *meta-mutant*.

  Second, we consider *strong mutation analysis* as the final measurement of simulation quality: the *killing* of mutants is defined as whether there is any deviation at design *output*, instead of at any of its internal intermediate signals in the case of *weak mutation analysis*. As mentioned, the definition of a *kill*-point is mainly a trade-off: if we use strong mutation for *more stringent requirements* on simulation tests, *more simulation time* should also be expected. Since our goal is indeed an as-thorough-as-possible verification for an IP design, it is reasonable for us to choose the more strict quality metric.

  Further, we propose an extension to mutation schemata as *dynamic mutation schemata*. The *dynamic* means that the simulation of individual mutants is dynamically created, or *forked* from the meta-mutant simulation. The mechanism is specific for HDL mutation analysis and, by such, we not only obtain the necessary information for test mode adaptation but also improve the efficiency of mutation analysis

  Killed mutants will be marked and kept out from further mutation analysis, as we finally measure the overall quality of an entire simulation process, instead of any subset of test data. It is *not* strictly specified by our framework which percentage of killed mutants is the adequate level for the random simulation phase and raises a signal for moving to the heavier-weight search based test generation phase. One reasonable way of such decision may be that we exit the random simulation as soon as the number fo killed mutants stops to increase for a certain period of time.

- **Mutation analysis-directed adaptation to test generation.** *During* simulation, we apply a continuous adaptation to the test model based on Markov chain and weighted constraints, by adjusting their parameters. The purpose is to enhance the *efficiency* of the simulation process under this test model, based on knowledge that

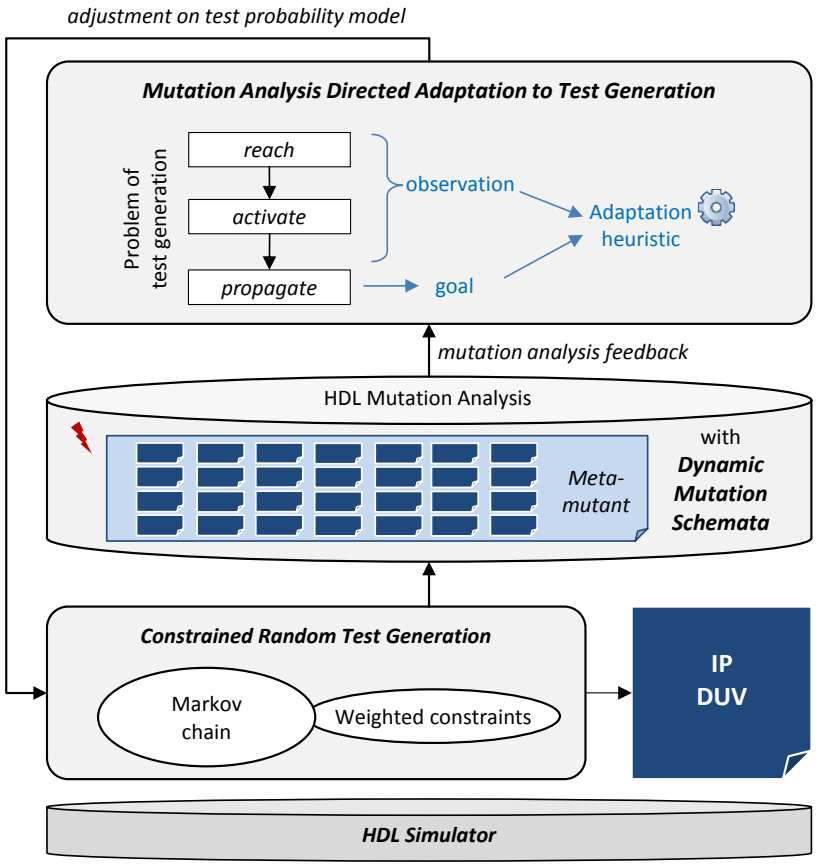


Figure 4.2 Mutation analysis directed adaptive random simulation.

we can collect on-the-fly during simulation. Here, a higher *efficiency* can be achieved with regard to our quality metric – mutation analysis, meaning more killed mutants by less tests.

We define a *heuristic* for this adaptation. The goal of the heuristic is specified by the fundamental problems of test generation in mutation analysis: *reach*, *activate*, and *propagate*. They also specify what information we should observe and collect from mutation analysis, i.e. the *feedback*. The heuristic then tries to correlate the feedback to the goal, which will be explained and formulated in Section 4.2.2.

This adaptive simulation is necessarily executed in a *closed-loop* style, since we should not only close the gap between the initial test modeling and mutant database – the quality metric, but also steer the model towards the dynamically changing metric, whenever dead mutants are removed.

- **Design Under Verification (DUV).** This mutation analysis directed adaptive random simulation framework applies mainly to IP-level designs, which usually

have a strict interface specification on input-output behavior, for example the ISA for a microprocessor design.

We consider the designs under verification to be RTL or behavioral, described in HDLs including traditional VHDL and Verilog, SystemC, and even C. The design can be in any development stage, early or near-complete. Therefore, there is *no* assumption of its synthesizability.

For verification, we simply assume the existence of a golden model. This model conforms fully to the design specification, for example an ISA. Randomly generated tests are applied directly as design stimulation. Comparison of simulation behavior between a golden model and DUV decides the design's correctness.

Also note that *each time* the design is modified – either through design refinement or debugging, we need to restart the whole simulation procedure for another round of verification.

- **HDL simulator.** Any HDL simulator capable of constrained random simulation, such as the ModelSim tool that is employed in our evaluation later, should be able to support this adaptive simulation. ModelSim supports also all the IP design languages that we consider: VHDL, Verilog, and SystemC.

4.2.1. Random Test Generation with Constrained Markov Chain

We first introduce the some basics of Markov chain and how it can be mapped to a random test generation model. Then, we present an extension to this modeling technique by attaching weighted constraint. The resulting test generation iteration is summarized at the end.

Markov Chain

In its basic form, a *Markov chain* with finite states can be described as a directed graph $M = (V, E, P)$:

- V is a set of states, or nodes that form the Markov chain,
- $E \subseteq V \times V$ is a set of directed edges, in which there exists one edge from each node to every node, including itself,
- P is a labeling function from E to non-negative real numbers, which represents the *probability* of each edge being selected for next transition from the present node.
- With $E_{out}(v)$ as all the edges out from $v \in V$, we have the probabilities $\sum_{edge_i \in E_{out}(v)} P(edge_i) = 1$.

Figure 4.3-a) shows a simple Markov chain model with two states: s_0 and s_1. At state s_0, we have a significantly higher probability $P((s_0, s_0)) = 0.9$ of taking a transition back to this current state, compared to the chance of moving to the other state s_1: $P((s_0, s_1)) = 0.1$. In contrast, after entering state s_1, the model has an equal chance between staying at s_1 or going back to s_0. After a long sequence of transitions, we can image that a pattern of consecutive s_0 will frequently occur.

The probabilities on edges can also be *tuned* dynamically. This means that, if we become more interested in pattern $s_0 s_1$, we can simply adjust the model by: $P((s_0, s_1)) = 0.9$ and $P((s_0, s_0)) = 0.1$.

The transition process of a Markov chain has the characteristic of being *memoryless*, meaning that the next transition depends only on the current state, *not* on the earlier transition history.

Test Modeling with Markov Chains

Figure 4.3-b) illustrates how a Markov chain can be used to model a random test generation process, by an example with microprocessor test instructions.

This test modeling is intuitive. First, each node of the Markov chain represents one type of tests that we consider to be specific for the design, such as an ISA (Instruction Set Architecture) category that we model in the example. Then, a sequence of tests can be generated by transitioning through the chain. Following each transition, a test is randomly selected from the type that the transition destination represents. The starting point for a transition sequence is *not* important.

Therefore, at each intermediate node, probabilities on the edges out from this node model the chance of each destination node, a type of tests, being selected for next test generation. In the microprocessor example, all the edges between ISA nodes are assigned equal probabilities: $P((Arith, Arith)) = P((Arith, Multiply)) = P((Arith, Shift)) = P((Arith, Branch)) = P((Arith, Store/Load)) = 1/|E_{out}(Arith)| = 0.2$. By such, we can expect equally distributed tests for all nodes. This all-equal-probability further means that initially, we model *no* biasing on the test generation process, but relying only on the basic ISA information.

The Markov-chain based random test modeling provides us the following possibilities:

- First, a Markov chain allows us to steer the distribution of a single test input towards particular areas which we regard as more interesting.

  Consider that we start test generation with an all-equal-probability Markov chain. Assume that after some period, we see most of the un-killed mutants remaining in the *barrel shift unit* of the design, because, somehow, they are a

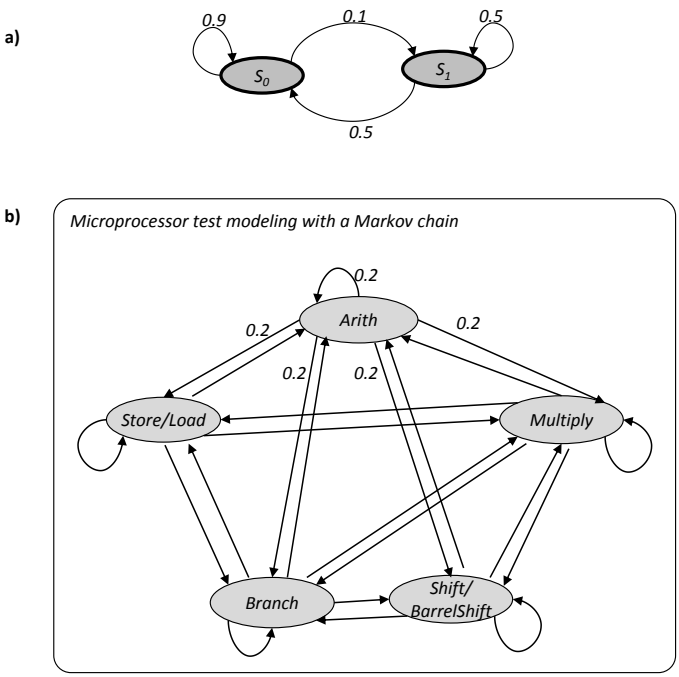


Figure 4.3 a) a simple Markove chain exmple; b) Example test modeling with Markove chain for a microprocessor .

difficult job. Then, we may expect an acceleration of the mutation analysis process, if we generate more *barrel-shift* tests, by adjusting all five incoming edges on the corresponding node to a relatively high level.

- Second, if the interaction between two adjacent test types/nodes is considered to have a particular impact on design simulation, we can also steer the test generation to encourage such a pattern. Note that in a Markov chain, any two nodes are connected and therefore adjacent.

  It is *not* possible to model the impact of a test pattern of longer sequence, since only the immediate dependence between two nodes can be reflected in Markov chain – its *memoryless* characteristic.

Weighted Constraints to Extend a Markov Chain Model

As an extension to the basic mechanism above, we further integrate *constraint-based random test generation* into the Markov chain-based test modeling. The principle of constrained random test generation and its advantage have been introduced in the background chapter.

- To each node $v \in V$ in a Markov chain M, we may extend M by attaching to v a set of *weighted constraints* that are defined on design input, or sub-fields of the input.

- These constraints at v are classified into groups. Constraints from different groups are defined on non-overlapping input fields.
- With $W(c)$ representing the weight on a constraint c and $Group(c)$ as all constraints in the same group with c, their weights should sum up to 1: $\sum_{c_i \in Group(c)} W(c_i) = 1$.

While each node v should already represent a particular area from design input space, the constraints further divide that area. As previously explained, these constraints again specify a probability distribution on v.

By such, we have a two-level modeling of random test generation. The advantage is that a finer adjustment to the test model is made possible, by adjusting the weights on constraints.

Figure 4.4 shows an example of such extension: how a Markov chain is augmented with weighted constraints for a finer modeling of floating point tests. The original Markov chain contains four nodes to represent four valid operations specified for a floating point unit (FPU) design. The FPU design can be a stand-alone IP, or an auxiliary unit in a microprocessor IP, which then makes this Markov chain also part of a larger ISA model.

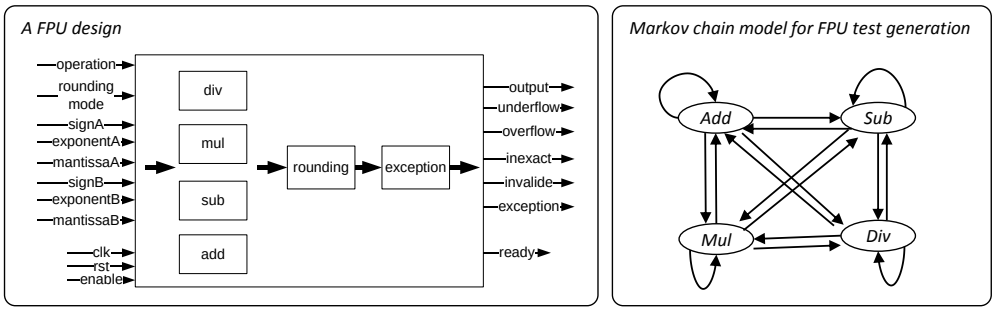
The table lists the constraints defined and attached to node *multiply*. They are specified with constraint structures from *SystemC Verification Library* (SCV). Further, they are *grouped* by the input fields that they constrain, without overlapping: on the rounding mode, the first operand, and the second operand. The classification of operand values and rounding modes from the constraints definition is according to IEEE floating point standard [49], which should be the specification for the FPU design. Initially, all constraints in the same group share an equal weight for random selection.

The *constraint satisfaction problem* [74] imposed by this constraint extension for test modeling is *not* the focus of our method. Verification languages, such as the SCV mentioned here, commonly integrate constraint solving facility and can be seamlessly leveraged to complete our test generation.

Test Generation Iteration

To summarize, the overall test generation process modeled as a *constraint-extended Markov chain* follows the following steps:

1) From a current node $v_{current}$ of the Markov chain model, we select the edge for next transition $v_{next} \in E_{out}(v_{current})$, following probabilities on the edges.
2) At node v_{next}, for ***each*** constraint group associated with v_{next}, we select one constraint according to the weights in the group.



Weighted constraints to extend node ***Mul***

constraint_group: Operand_A		
constraint	**SCV representation of constraint**	**weight: W(constraint)**
Zero	SCV_CONSTRAINT(exp_a()==EXP_ALL_ONE && mantissa_a()!=0);	0.1
Min_Subnorm	SCV_CONSTRAINT(exp_a()==0 && mantissa_a()==1);	0.1
Subnorm	SCV_CONSTRAINT(exp_a()==0 && mantissa_a()==1);	0.1
Max_Subnorm	SCV_CONSTRAINT(exp_a()==0 && mantissa_a()==FRAC_ALL_ONE);	0.1
Min_Norm	SCV_CONSTRAINT(exp_a()==1 && mantissa_a()==0);	0.1
Norm	SCV_CONSTRAINT(exp_a()>=1 && exp_a()<EXP_ALL_ONE);	0.1
Max_Norm	SCV_CONSTRAINT(exp_a()==(EXP_ALL_ONE -1) && mantissa_a()==FRAC_ALL_ONE);	0.1
Infinity	SCV_CONSTRAINT(exp_a()==EXP_ALL_ONE && mantissa_a()==0);	0.1
NaN	SCV_CONSTRAINT(exp_a()==EXP_ALL_ONE && mantissa_a()!=0);	0.1
One	SCV_CONSTRAINT(exp_a()==EXP_ONE && mantissa_a()==0);	0.1
constraint_group: Operand_B		
…	…	…
constraint_group: Rounding		
…	…	…

Figure 4.4 Weighted constraints to extend Markov-chain basd modeling of random test.

3) Solve the constraint to generate the corresponding test input that are specified by this constraint. For any input field that does *not* receive a value from constraints, generate it randomly.

4) Take the transition to v_{next} and start next iteration, by setting it as the new $v_{current}$.

Initially, the test model is defined mainly with information on the design interface, with little consideration on the design's internal architecture. Therefore, at the beginning of a simulation process, we assign equal probabilities to Markov-chain edges and equal weights to node-attached constraints.

For later adjustment, we assume that each time a $test$ is generated, a record $(test, edge_{test} = (v_{start}, v_{end}), constraint_{test})$ is saved as further reference to the origin of $test$, where $edge_{test}$ and $constraint_{test}$ are the edge transitioned and *constraint* solved for the generation of $test$, respectively.

Note that actually, there can be not only $constraint_{test}$ but *multiple* constraints used from different groups for generating $test$. Only for the simplicity of presentation, we formulate the adjustment of constraint weights for *one* constraint group. The same adjustment should be applied to each group.

4.2.2. Heuristic Closed-loop Adaptation to Test Generation

With **Figure 4.5**, we first show a motivation of how we formulate this *mutation analysis-directed, closed-loop test adaptation*, as a heuristic approach. The final adaptation heuristic, with the goal to improve mutation analysis efficiency, is devised by considering i) the ultimate problem of test generation in mutation analysis, ii) the feedback from mutation analysis as input for adaptation, and iii) hypotheses that we consider being reasonable for correlating the mutation analysis feedback to the test generation problem.

- **Test Generation Problem**: we may recall that the test generation problem for killing a HDL design mutant requires the mutant simulation to first *reach* the mutation statement, then *activate* this mutant by executing the mutated statement in such a manner that a local deviation is created, and *propagate* this deviation to any output of the design.
- **Adaptation Input**: we use mainly the statistic of how many mutants were totally activated by each test during mutation analysis, as input for calculating the adjustment. Summarized as $(test, N_{activation})$, this information comes from the *dynamic mutation schemata* process that will be introduced in next section. Besides, from last section, we record an entry $(test, edge_{test}, constraint_{test})$ from the Markov chain-based test generation process, for each test generated, with $edge_{test}$ and $constraint_{test}$ as the edge and contraint that were used for generating this test, respectively.
- **Hypotheses**: two simple hypotheses are proposed in order to correlate mutation analysis feedback to the test generation goal. They also become the direct rationale behind how we formulate the heuristic.
  - *Activation-propagation hypothesis*: if a test activates a lot of mutants in simulation, it also leads to simulation that kills many mutants in the end. In other words, we assume that the *mutant-activation capability of a test is coupled with its final mutant-killing effect*. This is reasonable in a

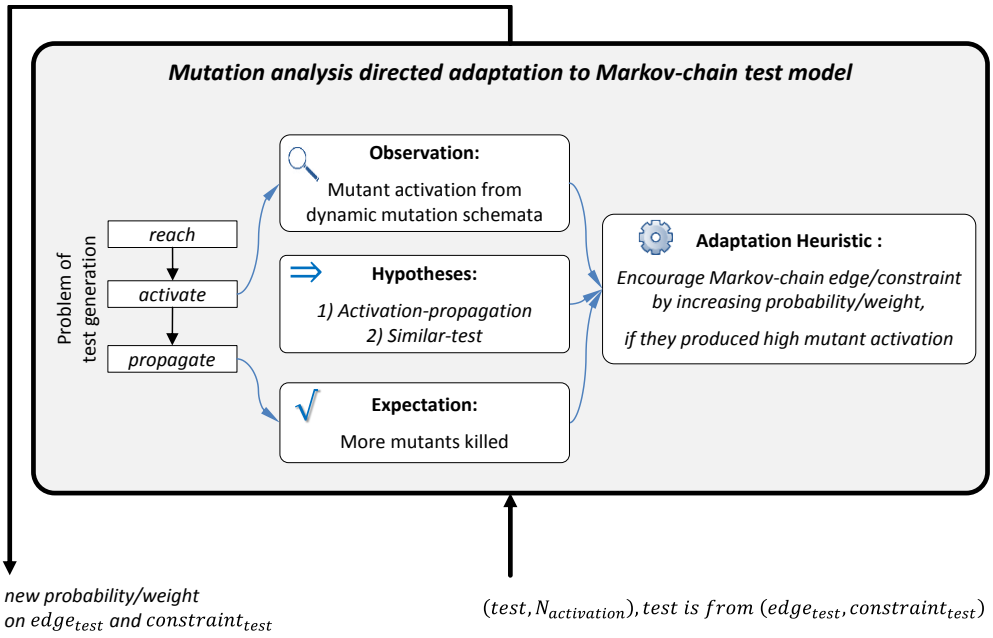


Figure 4.5 How we devise the adaptation heuristic.

straightforward manner: activation precedes propagation and is a necessary condition for killing a mutant.

- *Similar-activation hypothesis: if a test activates a lot of mutants, the Markov-chain edge and constraint that were used for generating this test should further generate tests that similarly activates many mutants.* Basically, a pair of Markov-chain edge/constraint represents tests of a same type. We expect them possessing *similar* mutant-activation capabilities.

Based on the considerations above, the adaptation heuristic works by adjusting the probability/weight of the corresponding Markov-chain edge/constraint according the test's activation efficiency. We formulate the calculation of this adjustment in the following.

Adaptation Heuristic

Each time the adaption is triggered, with $(test, N_{activation}, edge_{test} = (v_{start}, v_{end}), constraint_{test})$ as input, we first calculate a test *efficiency* value as:

$$efficiency = \frac{N_{activated}}{N_{mutants-unkilled}}$$

where $N_{mutants-unkilled}$ is the number of *un-killed* mutants, which are constantly reducing during mutation analysis. This *efficiency* becomes an estimation of the test's potential to kill mutants. Based on our second hypothesis, this estimation applies also to future tests to be generated from $(edge_{test}, constraint_{test})$.

A certain amount of incremental adjustment to probability/weight of test generation is then calculated as

$$\begin{cases} P_{incr} = \dfrac{efficiency}{|E_{out}(v_{start})|} \\ W_{incr} = \dfrac{efficiency}{|Group(constraint_{test})|} \end{cases}$$

where with $edge_{test} = (v_{start}, v_{end})$, $E_{out}(v_{start})$ is the set of all edges that come out from v_{start} and $Group(constraint_{test})$ represents all constraints that belong to the same constraint group from which $constraint_{test}$ is selected.

By this, we try to manage an appropriate magnitude of adjustment each time, by taking into account the total number of candidates for each random selection.

Then, the new probability/weight on $edge_{test}$ and $constraint_{test}$ are increased by P_{incr} and W_{incr} respectively, as

$$\begin{cases} P'(edge_{test}) = \min\{P_{old}(edge_{test}) + P_{incr}, P_{MAX}\} \\ W'(constraint_{test}) = \min\{W_{old}(constraint_{test}) + W_{incr}, W_{MAX}\} \end{cases}$$

where P' and W' represent the probability and weight after this adjustment and P_{old} and W_{old} are the old values . P_{MAX} and W_{MAX} play the role of *two maximum bounds, so as to prevent other edges as well as constraints from starving*. In our evaluation with microprocessor design, we have set both of them to be 0.9.

For each edge $e_i \in E_{out}(v_{start})$ and each constraint $c_i \in Group(constraint_{test})$ except $(edge_{test}, constraint_{test})$, we distribute the remaining probability/weight by

$$\begin{cases} P'(e_i) = \left(1 - P'(edge_{test})\right) * \dfrac{P_{old}(e_i)}{1 - P_{old}(edge_{test})} \\ W'(c_i) = \left(1 - W'(constraint_{test})\right) * \dfrac{1 - W_{old}(c_i)}{1 - W_{old}(constraint_{test})} \end{cases}$$

such that i) their previously gained bonuses are proportionally preserved and ii) $\sum_{e_i \in E_{out}(v_{start})} P'(e_i) = 1$ and $\sum_{c_i \in Group(constraint_{test})} W'(c_i) = 1$.

By such a gradual but consistent adaptation, we encourage those Markov chain edges and constraints from which tests activating more mutants are generated. Based on the activation-kill and similar-activation hypotheses, we expect that an improved mutant-activation rate and therefore mutant-killing rate can be observed, i.e. a higher mutation analysis result with less simulation effort with this adaptive random test generation.

4.2.3. Dynamic Mutation Schemata

We propose an extension to the original *mutation schemata* [63] which has been introduced in the background chapter. The resulting process is called *dynamic mutation schemata*.

For the convenience of presentation, we first introduce several notations for basic HDL simulation and mutation schemata, as elementary constructs. We then define the dynamic mutation schemata process based on these notations.

Notations for Original *Mutation Schemata*

For a HDL *design under verification* D to be simulated, we first use:

- $D_0 \Leftarrow Sim_{INIT}(D)$ to denote the execution of a HDL simulation initialization phase for design D, with the result notated as D_0, i.e. the whole state of D after initialization.
- $D_{t+1} \Leftarrow Sim(D_t)$, where $t = 0,1,2,3,\ldots$ is used to represent the execution of one HDL simulation cycle at $t+1$, which changes the state of D fron D_t to D_{t+1}.

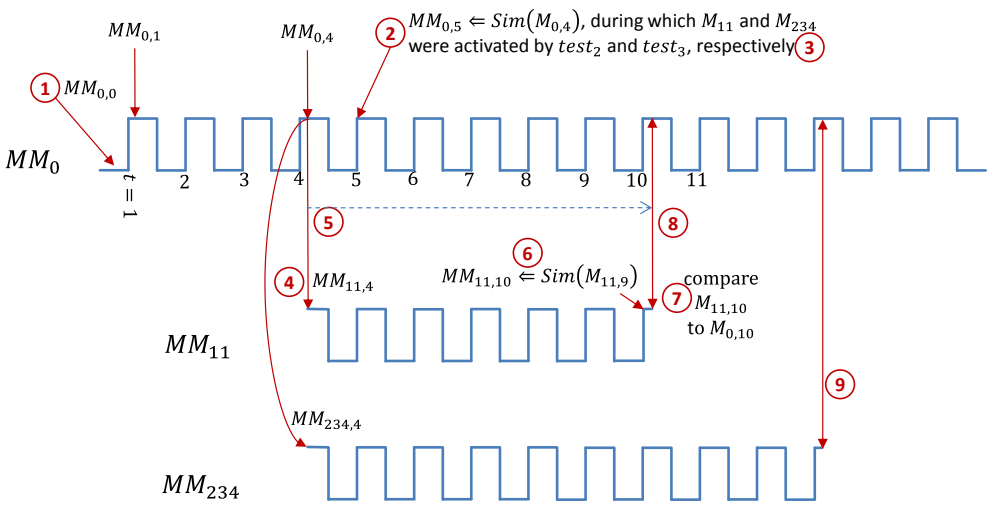
Note that Sim_{INIT} and Sim represent a simulation *with specific tests*, without the tests attached to the notation.

Recall that in *mutation schemata*, all mutants are encoded into one *meta-mutant*, each with a unique *mutant ID*. This ID should be designated to select the corresponding mutant for simulation. We use:

- MM to denote the *meta-mutant* and MM_k to represent mutant k by assigning the mutant ID for MM to be $k \in [0, \#\_of\_mutants]$. During simulation of MM_k, only the mutated statement with ID k is used, with all the other mutations unmasked and the original statements executed.
- When mutant ID k is assigned 0, MM_0 represents the meta-mutant with all mutation masked. Simulation of MM_0 has the same trace as the original design D.
- $MM_{k,0} \Leftarrow Sim_{INIT}(MM_k)$ and $MM_{k,t+1} \Leftarrow Sim(MM_{k,t})$ according to the notation above for HDL simulation, with MM_k as the design.

Extension as *Dynamic Mutation Schemata*

We propose an extension to this original mutation schemata, as shown by **Figure 4.6**. We call it *dynamic mutation schemata*, since in the mutation analysis process, the meta-mutant is continuously simulated as a main thread and the simulations of individual mutants are *dynamically forked and ended*, if they are *activated* during meta-mutant simulation.



Dynamic Mutation Schemata:

①$MM_{0,0} \Leftarrow Sim_{INIT}(M_0)$
At each cycle $t+1, t = 0,1,2,3,\ldots$:
②$MM_{0,t+1} \Leftarrow Sim(M_{0,t})$,
during which we obtain $List_{activated} = (\cdots, (MM_i, test_{MM_i}), \cdots)$

For each activated MM_i in $List_{activated}$
③update$(test_{MM_i}, N_{activation})$
④$MM_{i,t} \Leftarrow Fork(M_{0,t}, i)$
⑤mask MM_i in MM_0 as separate-running and no-check-for-activation

For each running forked mutant j simulation:
⑥$MM_{j,t+1} \Leftarrow Sim(M_{j,t})$
⑦compare $MM_{0,t+1} \leftrightarrow MM_{j,t+1}$,
⑧if no deviation found, delete MM_j and unmask MM_j in MM_0
⑨if deviation at design output, delete MM_j as it is killed

(⑩ adapt test generation with $\langle test, N_{activation} \rangle$)

Figure 4.6 Dynamic mutation schemata for HDL mutation analysis.

At the beginning, we have only MM_0 launched for simulation, with tests consistently generated as input from our random test generator. After initialization, MM_0 is simulated at each cycle. In the illustration of **Figure 4.6**, we assume that the design is synchronized at each clock rising edge.

Note that during $Sim(MM_{0,t})$, we are able to determine for each mutant whether it was activated, and by which test it was activated. A list $List_{activated}$ can be recorded as

$$List_{activated} = (\cdots, (MM_i, test_{MM_i}), \cdots)$$

where $test_{MM_i}$ represents the test that activated mutant MM_i. This should be possible, if

during the meta-mutant simulation, i) we calculate a mutated statement in parallel to the original one, for a comparison to see whether the mutant is activated ii) for each design sub-unit, we maintain a record which test is currently resident at this unit. For example for a microprocessor design simulation, in each pipeline unit there should be a corresponding instruction that is currently executed by this unit. We assume that such a record can be maintained during simulation, for any pipelined design. Then, when a mutant is activated during meta-mutant simulation, we know the test that is responsible for the design unit containing this mutant.

For each activated MM_i from $List_{activated}$, we first update the record entry of that mutant-activating test: $N_{activation}$ in $(test_{MM_i}, N_{activation})$ is increased by 1.

Then, we try to *fork* a continuing simulation for MM_i, from the current meta-mutant simulation. For this, we further assume the availability of a *fork* functionality: $MM_{i,t} \Leftarrow Fork(MM_{0,t}, i)$, which first creates a copy of $MM_{0,t}$ and then change the mutant ID of this copy from 0 to i. Such a *fork* is possible, since mutant i has never been activated until t and, therefore, $MM_{0,t}$ and $MM_{i,t}$ should represent the same design state in such a case.

The activated and forked mutants are *masked* in the meta-mutant simulation, since they are now simulated in separate threads and no longer required to be checked for activation.

Each such forked mutant simulation thread, say MM_j, is simulated at every clock cycle: $MM_{j,t+1} \Leftarrow Sim(M_{j,t})$. The result $MM_{j,t+1}$ is compared to $MM_{0,t+1}$ from meta-mutant simulation. There are just two outcomes from this comparison:

- If no deviation is found between them, it means that the simulation of mutant MM_j has *converged back* to the meta-mutant simulation. That thread for simulating MM_j can be aborted. We unmask it in MM_0 to resume the activation-checking for MM_j.
- If any deviation appears at design output, it means the activation has been successfully *propagated* and, by definition of mutation analysis, mutant MM_j is killed.

The main advantage of this dynamic mutation schemata is the saving of simulation time for HDL mutation analysis. Individual mutants are simulated in a dynamic, just-in-time manner, based on meta-mutant simulation.

At the end, we are able to compile the input for the adaptation heuristic: $(test, N_{activation})$ for each $test$, which are consistently updated during our dynamic mutation schemata. Note that we should trigger the adaptation heuristic only when a $test$ will no longer receive any activation update.

4.2.4. Summarized Procedure

In **Figure 4.7**, the summarized procedure is presented for the proposed mutation analysis directed adaptive random simulation. The purpose is to give the reader a clearer overview on this simulation process.

Test model preparation

From design specification, e.g. an ISA, construct a Markov chain model $M = (V, E, P)$ and extend it by attaching weighted constraints to V;

Initially, all edges and constrains are assigned equal probability/weight;

Create meta-mutation MM from design under verification;

Start simulation

WHILE still within simulation budget DO

Generate a $test$ from M and record $(test, edge_{test} = (v_{start}, v_{end}), constraint_{test})$, as described by Section 4.2.1;

Simulate MM_0 and each activated mutants MM_i for a cycle with $test$ as input, and update activation statistics of tests, as described by Section 4.2.3;

FOR each test has not been updated for a certain time, if any, DO

Calculate and apply an adjustment (P', W') on constrained Markov chain model, as described by Section 4.2.2.

End

END WHILE;

End

Figure 4.7 Summarized procedure for adaptive random test generation directed by mutation analysis.

We do not repeat the explanation of the steps. For the evaluation chapter, we have implemented i) the constraint augmented Markov chain with the *SystemC Verification Library*, ii) the dynamic mutation schemata by utilizing the Tcl interfaces of tool Certitude and ModelSim, and iii) the adaptation heuristic also in Tcl.

4.3. Related Work

We review literature that has a focus as we have: random simulation methods that are made adaptive and dynamically steered under a specific simulation quality metric.

First, in **Figure 4.8**, we give a tabular view of the literature and, in particular, which metrics are targeted by the adaptive simulation. Note that in literature, term *coverage metric* is used for the same meaning as *quality metric* in this work. *Coverage-directed* and *metrics-directed* also refer to the same.

Metric as adaptation target	Literature
Functional (verification-plan) coverage	[15]: *Coverage directed test generation for functional verification using Bayesian networks* (2003)
Functional (signal-switching) coverage	[75]: *Microprocessor verification via feedback-adjusted Markov models* (2007)
Observability-based Coverage	[16]: *A Functional Validation Technique: Biased Random Simulation Guided By Observability-Based Coverage* (2001)
Assertion based coverage	[78] Simulation knowledge extraction and reuse in constrained random processor verification (2013)

Figure 4.8 Related work: metrics-directed adaptive random simulation.

- The method in [15] begins with a test planning and the coverage is defined as the amount of pre-planned verification tasks that have been simulated, e.g. specific transactions from a CPU unit. It can be viewed as a functional coverage. Then, an evolving Bayesian Network is constructed to model the correlation between test generation directives and the verification-plan coverage.
- In [75], the adaption of random simulation is aimed at *exciting more signal-switching activities at specific locations*. Based on the assumption that the increase of such signal activities in simulation will also lead to higher chances of inciting real design bugs in that portion, the final goal is to improve the efficiency of bug detection, i.e. number of discovered design bugs by a certain number of simulation effort.

  Similar to our approach, the random test generation is modeled using a Markov chain. However, *no* further constraints-based modeling is used as we do. Extra monitors are necessary to be attached to those signals under consideration, so as to collect a weighted *score* of switch activities. This score is then taken as input to the calculation of adjustment to probabilities on Markov-chain edges.

  Besides, the signal switching monitor is extended to also include signals that precede a target signal under observation. They are assigned less weights when summed up into the *score*, according to their distances to the target signal. This is called *depth-driven activity monitoring*.
- [16] ([76] and [77] similarly) builds adaptive constrained simulation based on the so-called *observability-based coverage*, which we have discussed in the background chapter. Recall that in *observability-based coverage*, *tags* are

introduced as symbolic disturbance to variable values. Their propagation during simulation is defined on logic, arithmetic, and control operators.

It forms a semi-formal method. First, the circuit design itself is modeled as a Markov chain at steady state. The controllability and observability of the nodes, with regard to tags, are estimated using a *limited depth re-convergence*. Targeting this estimation, an optimization algorithm tries to iteratively perturb the probability distribution on random input generation, each time when the tag coverage stops to increase.

- [78] is one of the most recent effort on coverage-directed constrained random simulation, which targets the *assertion-based coverage*. An *assertions* [79] in simulation based verification is simply a statement embedded and co-executed with the "actual" design, asserting whether a specific condition on design state, or a sequence of states is satisfied at that point. Observing shortage of covered assertions, in the verification of a microprocessor, the authors propose a knowledge-learning methodology that tries to extract knowledge during simulation and reuse them to i) further exercise the already covered assertions and ii) generate tests that should hit those un-covered assertions.

  A feature based rule learning approach is applied. First, an instruction sequence as test input is converted into multiple *snippets*, each as a block with equal length. These snippets are classified into two classes ($S_{covered}$ and $S_{not-covered}$, as positive and negative samples), by whether they covered assertions in simulation or *not*, according to the simulation trace. Then, ISA dependent *features* are extracted from the positive samples, such as the data dependences in a sequence of instructions (this is also considered in our approach). *Rules* are mined, as the knowledge, each representing a hypothetical proposition from a specific collection of features to $S_{covered}$, i.e. assertions been covered. Concerning the techniques and procedures used for rule mining, one can directly look into the literature.

Comparison of Literature to Our Work

Contribution from our method compared to other literature on adaptive random simulation can be concluded as follows:

- First, our method uniquely takes the mutation analysis metric as the target of adaptive simulation. Further, based on our consideration that mutation analysis is an advanced emerging quality metric for HDL design simulation, we view our method a step beyond state-of-the-art techniques.
- We employ a combination of constraints and Markov chain to model test generation and enable adaptation, which is *not* to be found in other methods and

provides us a finer adaptation basis.

- Our adaptation heuristic is based on a unique, more complex test generation problem in mutation analysis: reach, activation, and propagation of mutants. These are aspects that are *not* covered by other adaptation methods.

To the best of our knowledge, it is the first effort on such mutation-analysis directed adaptive random simulation, to improve the efficiency of HDL mutation analysis. Our evaluation on the method efficiency, in a later chapter, is also based on a state-of-the-art HDL mutation analysis tool: Certitude [21].

4.4. Summary

We have proposed a novel method to improve the efficiency of HDL mutation analysis within *constrain random simulation*, being aware of the problem that i) initially, the random test model is defined *not* specifically for a set of mutants and ii) along with the advancing of simulation, un-killed mutants as the remaining target also change. They become the motivation for adaptive random test generation.

The simulation method consists of three parts, for random test modeling and generation, for HDL mutation analysis, and for a consistent adaptation to test generation.

- The Markov chain and constraints based test modeling enables us not only to steer the distribution of a single test input towards our interest, but also to encourage the generation of a certain pattern of two consecutive tests.
- The *dynamic mutation schemata* leverages the advantage of original mutation schemata by creating and compiling only one meta-mutant. It extends this efficiency by dynamically *forking* necessary executions of individual mutants and merging them back when the executions succeeded or converged.
- The adaptation heuristic is devised based on the intrinsic problem, or conditions of mutation analysis test generation: *reach*, *activate*, and *propagate*. Basically, test patterns that activated more mutants are encouraged, with the expectation that they will continue to activate many mutant and, therefore, also kill mutants. This encouragement is realized through the adjustment of probabilities/weights on Markov chain edges/constraints.

By this, we expect a derived simulation process that is *not only* measured under the mutation analysis metric, but also *self-steering* towards this metric by adaptive, automatic test generation – thus a *metrics driven verification* method. It severs the first component of our methodology, and the first phase of an IP verification.

In the evaluation chapter, we will mainly investigate whether the adaptation heuristic equipped simulation process is indeed able to improve the efficiency of HDL mutation analysis, i.e. it killing more mutants with less random tests generated.

In general, we see the method *not* limited mutation analysis, with *no* restriction of its application to other metrics.

This contribution has been first proposed in [7] and further elaborated in [1].

CHAPTER 5: Metaheuristic Search-Based Test Generation for Mutation Analysis

5.1. Introduction

The *feedback directed random simulation* presented in last section is an advanced, yet light-weight method to obtain a primary level of verification quality under mutation analysis. Nevertheless, we expect that in most cases, the random simulation in general can*not* reach an adequately high percentage of killed mutants. For example, this adequacy level can be a best-effort within the time budget for IP verification.

This chapter presents the second component of our verification methodology, a heavier-weight, more complex method to *handle each of the remaining mutants from random simulation*. It becomes also the second phase for a thorough IP verification.

The general problem is that there still lacks an efficient, practical test generation method for HDL mutation analysis, i.e. to generate simulation tests that kill a HDL design mutant, in particular, when we consider a complete microprocessor IP design, for example. One reason is that professional EDA tool for HDL mutation analysis recently just emerged. Related literature will be investigated after the presentation of our novel method.

Motivation for Metaheuristic Search Based Test Generation

Metaheuristic search, or simply metaheuristic, is a search algorithm on a discrete search space that aims at finding an optimal solution on that space under a certain given objective by trying to iteratively improve a current candidate solution. They are called *meta*heuristics as the algorithms propose little constraints on the concrete problem that they can solve, i.e. the search space and the search objective.

Figure 5.1 describes the basic principle of using such metaheuristic search for test generation. The goal of the search is to find a target test and the search space is just design

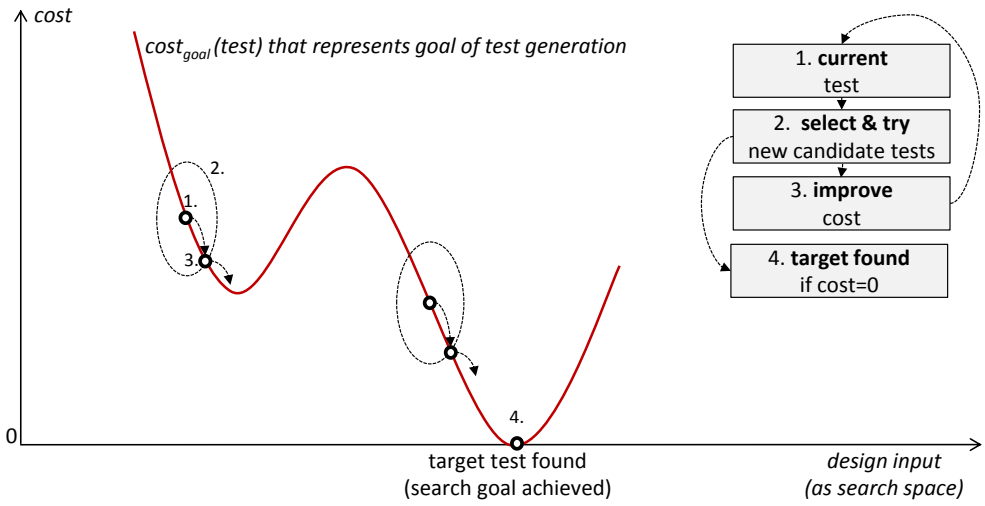


Figure 5.1 Principle of metaheuristic search based test generation. It can also be called *simulation-based* test generation, since *only simulation* is relied on to "try-and-improve" tests.

input. Since we have a specific goal of test generation in the context of quality metrics driven verification, for example killing a mutant, and the input space for a HDL design is indeed discrete, it is fundamentally possible for us to apply metaheuristic search.

The key to enable a search algorithm for test generation is the definition of an *objective cost function*, or simply *cost function*, which represents the goal of test generation, since a metaheuristic already defines the basic iteration framework for improving such cost. As shown in the figure, a metaheuristic tries to move iteratively to another test that has a *reduced* cost from the current test. If the cost is reduced to zero, we automatically reach a target that satisfies the test generation goal.

We can find a wide range of metaheuristics with different candidate-selection and moving mechanisms, from simple to complex. One basic example is *local search*, which selects the *neighbors* of a current solution as candidates for examination. A cost-improving neighbor is then identified as the new coordinate for search. Some more advanced examples that have been applied to test generation include *simulated annealing*, as a variant of local search to escape so-called local-optima, and *genetic algorithm* [80].

Though in a metaheuristic search based approach as shown in **Figure 5.1**, a solution – a target test that kills a design mutant – is *not* guaranteed for test generation, our method has the significant advantage that it *relies only on actual design simulation to evaluate tests* and therefore, avoids completely *symbolic simulation* or *constraint solving*, as we will discuss and compare in the related work. Therefore, it can be just integrated into a simulation process to iteratively optimize tests, which makes it a practical solution to mutation analysis test generation for even complex IP designs, assuming simulation is practical.

Contribution of the Chapter

The contribution of this chapter, as the second component of our mutation analysis driven functional verification methodology for IP-based SoC design, is the proposal of a simulation test generation method which is based on metaheuristic search and aimed each time at finding some functional test that kills a HDL design mutant. As the key of such search, we propose an objective cost function that is able to perform effectively the search steering towards mutant-killing tests.

Organization

The rest of the chapter is organized as follows. First, we introduce the overview of our method that applies metaheuristic search to the test generation in HDL mutation analysis. Here, a local search procedure is also outlined as a basic but concrete metaheuristic example. Then, the major space of the chapter is devoted to Section 5.3, which defines a cost function that should be able to effectively steer a metaheuristic search towards a mutant-killing target test. Section 5.4 further discusses related work from literature and why they do *not* qualify an appropriate solution. The chapter is summarized and concluded by Section 5.5.

5.2. Applying Metaheuristic Search to Mutation Analysis

We propose a metaheuristic based test generation method for HDL mutation analysis, as shown by **Figure 5.2**. Mutation analysis is employed as the consistently focused, representative quality metric for IP design simulation. We do *not* restrict the method to a specific search algorithm. Instead, we focus on the definition of a meaningful, effective cost function that could be integrated into any metaheuristic to make a test generation procedure.

The search targets every time *one* mutant left un-killed from the random simulation phase. Its objective is to find a test that kills this mutant. The input of the cost function is the simulation traces from mutation analysis, which makes the test generation pure simulation based. The key is the definition of an objective cost function that measures the progress of this mutant being killed, when it is still not the case.

Recall that to kill a specific mutant, a test is required to generate a simulation that (i) reaches the mutation statement, (ii) executes the fault-injected expression with certain values such that the expression evaluates to a different result from the original expression and (iii) propagates this difference to the design output boundary. They are called *reachability*, *activation* and *propagation* conditions, or sub-problems of mutation analysis test generation.

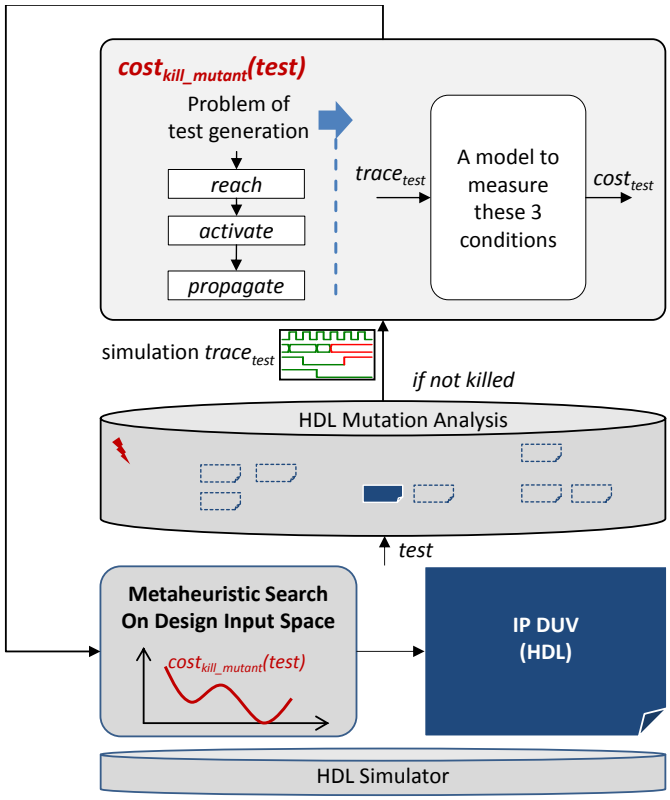


Figure 5.2 Metaheuristic based test generation for HDL mutation analysis. It targets each

Therefore, the core the cost function, to be presented as the main content of this chapter, is a model that *measures the degree of these three conditions from fully satisfied*.

Before the elaboration of the cost function, we present a local search on HDL designs, as an example metaheuristic that may be applied for our test generation method, assuming the availability of a cost function. The simple local search is chose, since, as mentioned, our focus if *not* the search algorithm but the cost function definition for HDL mutation analysis. With a basic metaheuristic, we *should already be able to evaluate the effectiveness of a cost function as a search steering guidance*.

A Local Search Example

The procedure in **Figure 5.3** begins with a random selection of test and then iteratively tries to move to a better local neighborhood test, so as to land hopefully on a target test.

With an initial test randomly selected, its cost is calculated and we enter the loop for reducing the cost iteratively. First, a list of so-called *neighbor_test* based on the current test are identified.

We may consider this neighborhood function in a general way based on HDL types. A straightforward scheme is that we adjust one input variable each time. For an integer

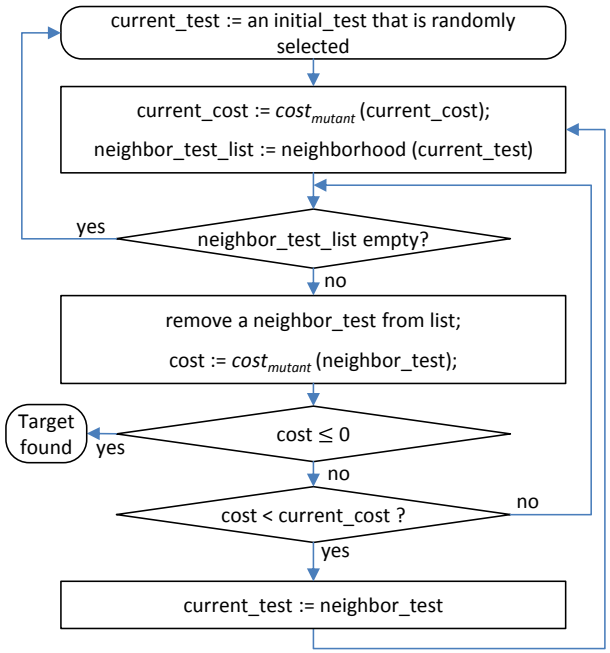


Figure 5.3 Local search example.

variable, we can have two neighborhood moves, one by increasing and another by decreasing *half* from its current value. For a bit or bit-vector variable, its neighbor values should be those with one single Hamming distance from the current bits. For an enumeration type, the candidates should be all the other possible values.

We may also consider a neighborhood function more concretely, for example for a microprocessor design, by defining it as adjusting one instruction field each time, such as toggling the carry bit of an *add* instruction, or increasing/reducing slightly the *immediate* field of an immediate-instruction.

In another inner loop, we examine the cost of neighbor tests one by one. When the cost is reduced to zero, we find a target test that kills the mutant. When a smaller cost appears, we assume that additional useful information for killing the mutant has been included into test by the neighborhood move. It should be an improved test and therefore set as *current_test* for further iterations.

If we unfortunately could not find any neighbor test that reduce the current cost, we encounter a so-called *local-optima*. One basic solution can be that we just restart from another initially picked point. Certainly, the total restart needs to be limited with some *Maximum_Iteration*.

There can be more sophisticated variants to local search, as introduced. Still, the optimal setting of a search algorithm, for example this neighborhood function in local search, is *not* the focus of this work. With this simple local search, the main purpose is to evaluate the *steering effectiveness* of the cost function, on real designs in later experiments.

5.3. A Cost Function for Search Based Test Generation of HDL Mutation Analysis

We propose a cost function that is able to estimate the progress of a test killing a HDL design mutant, so as to steer effectively a metaheuristic search. We define a Control and Data Flow Graph (CDFG) as the underlying data structure, since i) similar structures have been commonly used for analysis and synthesis of HDL designs, both RTL and behavioral, and even designs in C/SystemC, and ii) it just enables us to handle the problems of mutation analysis test generation – *reachability*, *activation*, and *propagation* – by the inclusion of both control and data flow.

After the definition of this CDFG structure, we present the cost function by first explaining its general idea and, then, formulating its calculation in details.

5.3.1. A Control and Data Flow Graph (CDFG)

Graph representations with both data and control dependencies have been used in *HDL synthesis* as well as verification [81] [82] [83] [84] [85]. For the purpose of mutation analysis, we propose a variation with explicit data nodes on both control and data flow. Extracted from a HDL design under verification and taking into account one of its mutation:

Definition 5.1: A *Control and Data Flow Graph* (CDFG) is a graph $CDFG_{DUV,mutant}$, or simply $CDFG = (V, S, E, \delta, O, s_{mutant})$ where

- $V \cup S$ is the nodes of the graph and E is the edges.
- S is the set of statement nodes that each represents either an assignment statement or a branch statement in the design and $V = \{v_1, \dots, v_n\}$ are the data nodes each for a signal variable.
    - For each branch statement the branch evaluation is treated as a separate statement generating an extra Boolean-valued data node, i.e., the branch result. Only *if* statements are discussed in the following, as generally other branches like a *case* statement can be transformed to *if* branches.
    - We further distinguish $V_{bran} \subset V$ as data nodes from branch statements.
- $E \subset (V \times S \cup S \times V)$ is a set of directed edges, each representing either a control dependence or a data flow dependence.
    - For each statement in the design, the corresponding node has inflow edges from data nodes of its operand signals, and a single outflow edge to the data node of its assigned signal.

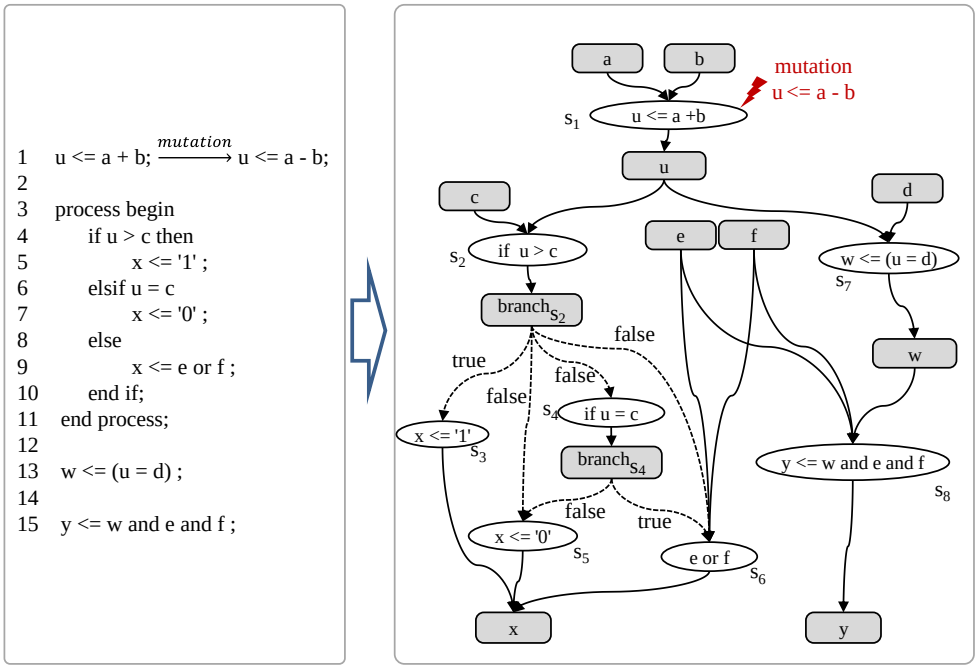


Figure 5.4 Example control and data flow graph extracted from a piece of HDL design.

- Extra *control edges* connect a branch result node to *all* the statement nodes that are contained in this branch, which represents control dependencies. Such nodes comprise $S_{bran-controlled}$. Each $s \in S_{bran-controlled}$ may have one or multiple control edges $branch(s) \subset V_{bran} \times \{s\}$. Every $e \in branch(s)$ is labeled by δ with a Boolean value to indicate in which case it should be executed in simulation according to the branch result. $\delta : E \rightarrow \{true, false\}$.
- $O \subset V$ are the output ports of our design under verification, where simulation results are compared to determine whether the mutant is killed.
- $s_{mutant} \in S$ is the statement where the mutation is injected.
- We use s_{mutant} to not only represent the node but also the original statement and s'_{mutant} to represent the mutation injected statement.

We have further the following notations:

- We use $out(s) \in V$ to represent the single out-flow data node of a statement, i. e. the operation result, and $in(s) \subset V$ as in-flow data nodes, i.e. the operands, of s for any $s \in S$. If s
- We use $out(v) \subset S$ to represent out-flow and $in(v) \subset S$ as assignment nodes for any $v \in V$. We can assume that if v is not assigned in any branch, $in(v)$ should have a single statement that assigns it. If v is indeed contained in some

branch, $in(v)$ may have multiple statements. This $V_{bran-controlled} \subset V$ can be simply identified by $\{v|\ in(v) \subset S_{bran-controlled}\}$.

Example 5.1-1:

Figure 5.4 shows an example design – declaration of signal and ports are left out – that leads to a $CDFG$ with $V = \{a, b, c, d, e, f, u, v, branch_{s_2}, branch_{s_4}, x, y\}$, $V_{bran} = \{branch_{s_2}, branch_{s_4}\}$, $O = \{x, y\}$, $s_{mutant} = s_1$, and S, E, δ to be identified straightforward in the figure. ■

This definition of CDFG should lead to an easy implementation of data structure and algorithm. Since $s_{mutation}$ is the only difference for individual mutants, a CDFG structure basically requires a single construction. Loop dependences are further included in such a CDFG without extra effort. Moreover, the mapping of simulation traces onto a CDFG is straightforward, by mapping values to variable nodes.

5.3.2. CDFG Based Cost Function Definition: Outline

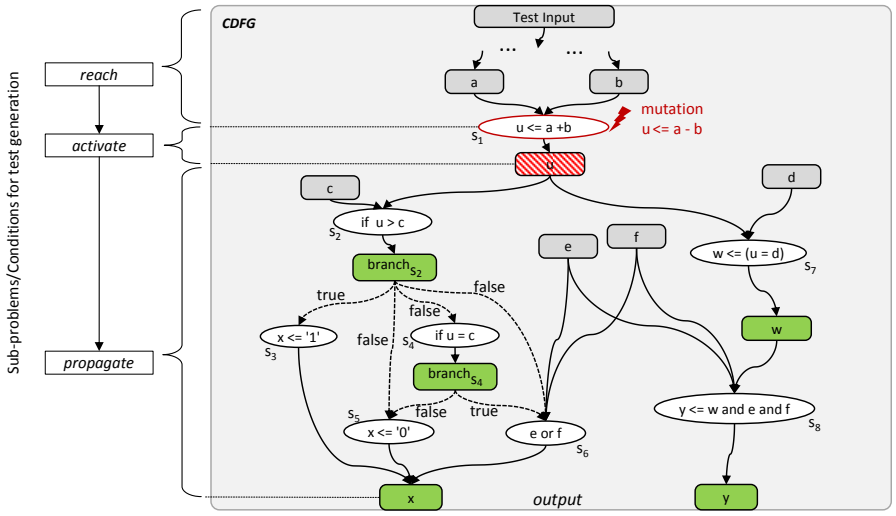
With **Figure 5.5**, we outline the idea of our definition of a cost function for HDL mutation analysis, which should measure the progress of a test killing a HDL mutant. It is based on the CDFG example above.

Assume that we have mapped a pair of simulation traces, from the original design simulation and mutant simulation, at one specific cycle *both* onto *data nodes* of the CDFG. With this mapping, our ultimate task is to calculate a *cost* value that measures or estimates whether the three sub-conditions of mutation-analysis test generation – *reach*, *activate*, and *propagate* – are satisfied during this simulation and, if not, *how far they are from satisfied*.

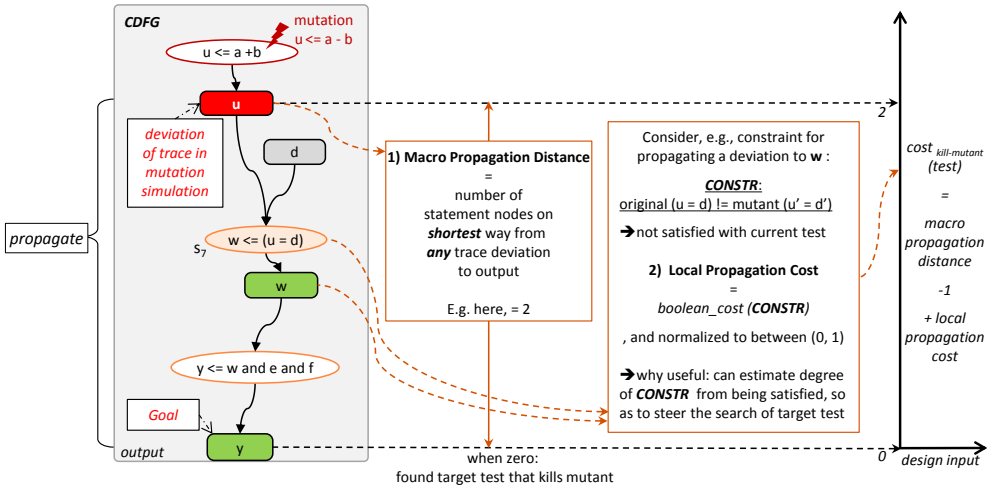
- **Propagation**. We first discuss the *propagation* after activation, which means that some data nodes already receive a deviate value in mutant simulation compared to original design simulation, such as node u in the figure.

  1) With one or multiple such mutant deviations, we first measure the *number of statement nodes on the shortest path from any deviation to design output*, to be a ***macro propagation distance***. In the example figure, this is 2.

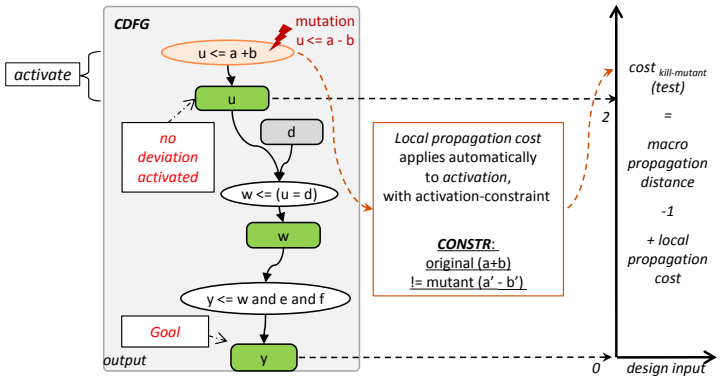
  Used in a search algorithm, this distance becomes a quantitative, *macro* estimation of the progress of the HDL mutant being killed in one simulation, since our final goal of search is, since our final goal is exactly to make such deviation to appear on design output. When it is reduced to zero, after some search iterations, we automatically obtain a target test that generates a mutant simulation trace with deviation on design output, and mutant killed by definition.



a) How CDFG handles problem of mutation analysis test generation



b) How *propagation* is handled.



c) How *activation* is also taken into account.

Figure 5.5 Idea and outline of CDFG based cost function.. Red data nodes mean a deviation in mutant simulation trace.

The reason of us counting only statement nodes is that they are exactly the obstacles of mutation effects propagating further, though they are the propagation medium at the same time. On a CDFG, a mutation effect may spread as the input of multiple statement nodes further to multiple data nodes. Such mutation effects will also be blocked at a statement, if the statement despite one or multiple mutation effect as its operands computes a same result as in the original design simulation.

2) Another ***local propagation cost*** is added to supplements *macro propagation distance*. As a value between 0 and 1, it intends to bring *a finer scale* to our cost function.

As illustrated in the figure with node u as example, it is calculated on a statement node that follows a mutant deviation. A closer look at how the propagation is blocked by s_7 is possible, if we consider a constraint for propagation is exactly: the result of s_7 in original design simulation is different from that in mutant simulation, as ***CONSTR*** defined in the figure. That this constraint was not satisfied with the current test and simulation is the exact reason why deviation at u is blocked by s_7.

Then, leveraging the table in **Figure 5.6**, we are able to estimate the closeness of such a propagation constraint from being fully satisfied, by transforming the constraint into a Boolean expression.

Consider another basic example why this *boolean_cost* from the table is just useful. Since $boolean\_cost(a > b) = |a - b|$, for $a = 5$ and $b = 1$ that do not satisfy $(a > b)$, we have a cost as 4. Assuming that we made a change, for example in a search algorithm, by $a = 4$ and b remaining the same, we may conclude that it was a *good search direction*, because the *cost is reduced* to 3.

In the end, this *boolean_cost*(***CONSTR***) should be normalized to a value between 0 and 1, to be *added* to the macro propagation distanced, just like the *centimeter* scale to *meter* on a ruler.

- **Activation**. Without further effort, the *local propagation cost* handles the activation problem, since an activation-constraint can be derived similar to the propagation constraint: result of mutant simulation deviated from that of original design simulation.

The only difference here is that, at the right side of the not-equal constraint, the mutated expression should be used, as the example shows in the bottom part of **Figure 5.5**.

In such a case, with regard to *macro propagation distance*, to which the *local propagation cost* should be added, since we do *not* have any deviation, we may

Boolean Expression *e*	***boolean_cost(e) as* Cost Function Value**
Boolean	0 if true, 1 otherwise
a<b, a≤b, a=b, a>b, a≥b	0 if true, abs(a-b)+K[a] otherwise
a≠b	0 if true, K otherwise
$B_1 \wedge B_2$	*boolean_cost*(B_1)+ *boolean_cost*(B_2)
$B_1 \vee B_2$	0 if either is true, *boolean_cost*(B_1)×*boolean_cost*(B_2) / (*boolean_cost*(B_1)+*boolean_cost*(B_2)) otherwise

•a. K is a small constant

Figure 5.6 *boolean_cost()* that estimates the degree of a Boolean expression from being satisfied [90]. It can be applied to a propagation/activation constraint $B_{original} \neq B_{mutant}$, if we transfrom the constraint into $B_{original} \wedge \overline{B_{mutant}} \vee \overline{B_{original}} \wedge B_{mutant}$.

simply use that *distance of u plus 1*, as a hypothetical distance even one step farther than the very first possible deviation.

- **Reachability**. We assume *reachability* easy to be satisfied in any simulation and, therefore, that it does not require particular guidance in a search algorithm. This is in fact what we can observe in most HDL simulations.

5.3.3. Macro Propagation Distance

Following the idea from last section, in this section we detail the definition of *macro propagation distance*, such that an implementation can also be easily derived.

With regard to a specific $mutant$, the mutation analysis process with $test \in T$ produces a pair of simulation traces. We denote with $W_{mutant,test}$ the original design simulation trace and with $W'_{mutant,test}$ the trace with the mutant. $W_{mutant,test}$ and $W'_{mutant,test}$ contain values from the real simulations. For the actual format of such simulation traces, as mentioned in the background chapter, VCD (Value Change Dump) and WLF (Wave Log File) are commonly used examples.

We define the cost function first for each cycle of the simulation trace pair $W_{mutant,test}$ and $W'_{mutant,test}$. The final cost is then the minimal from all cycles. We denote with (ω, ω') such a cycle snapshot from $W_{mutant,test}$ and $W'_{mutant,test}$, with

$$\begin{cases} \omega = \{\, \omega(v_1), \omega(v_2), \cdots, \omega(v_n) \,\} \\ \omega' = \{\, \omega'(v_1), \omega'(v_2), \cdots, \omega'(v_n) \,\} \end{cases}$$

where $\omega(v_i)$ represents the value of v_i from $W_{mutant,test}$ at the cycle i and $\omega'(v_i)$ the value of v_i from $W'_{mutant,test}$ at that cycle. This maps (ω, ω') directly onto the CDFG. $W_{mutant,test}$ can be represented as $\{\, \omega_1, \omega_2, \cdots \}$ considering all simulated cycles and $W'_{mutant,test}$ as $\{\, \omega'_1, \omega'_2, \cdots \}$.

We use further $\omega(s)$ for an $s \in S$ to represent the evaluation of statement s with variable values in ω and $\omega'(s)$ the evaluation of s with values in ω'.

With this mapping we define the *macro propagation distance* as the first component of the cost function as

$$macroPropagationDistance(\omega, \omega') = \min\big(dist(v)\big)\ for\ all\ v \in V \text{ with } \omega(v) \neq \omega'(v) \quad (5.1)$$

where, and in the following, $dist(v)$ is defined as the *number of statement nodes on the shorted path from v to any output node in O*.

We call each pair of $\omega(v) \neq \omega'(v)$ a *mutation effect* on v. By this simple formula, we basically measure how far the *mutation effects* in the simulation traces, if any exists, are still away from reaching the output nodes, by the definition of a mutant being killed.

Example 5.1-2:

With an input $(a = 4, b = 1,\ c = 0, d = 2)$ for Example 5.1-1, we can calculate a $macroPropagationDistance = dist(u) = 2$, with u receiving the only mutation effect $\omega(u) = 5, \omega'(u) = 3$.

Assume that in a next search iteration, we adjust the input a little and consider another candidate with $a = 3$. The new test will propagate the mutation effect through $s_7$and lead to a $macroPropagationDistance = dist(w) = 1$. This implies then a guidance on the right search direction. The new test can be designated as the coordinate for further search iterations that follow. ■

In the case that no mutation effect exits, i.e. the mutant simulation trace matches totally the original simulation trace, we define

$$macroPropagationDistance(\omega, \omega') = dist\big(out(s_{mutant})\big) + 1$$

i.e. the propagation distance of its outflow variable node.

With this inclusion, we are able to take into account the *activation* condition, since later a local cost can be analyzed on s_{mutant}. This is also based on our assumption that statement reachability is usually satisfied.

We notice that $dist(v)$ for each variable node in a CDFG is a static value. They can be computed directly after the construction of CDFG and attached to the nodes, for inquiry when necessary.

Last, CDFGs with looped flows will encounter *no* special problem with regard to the calculation of $dist(v)$, since the distance is calculated with regard the *shorted* path and therefore *not* following a loop.

For real-world designs, for example the microprocessor design or the floating point design that are used in our evaluations we expect that their control and data flows have much more stages and, therefore, $macroPropagationDistance$ measurement can serve a reasonably fine-grained search directive. Moreover, the introduction of extra data nodes for control flows enhances the measurement. With regard to implementation, an extra Boolean signal needs to be inserted for each branch to record its value during simulation.

As another result during the calculation of $macroPropagationDistance(\omega, \omega')$, We can collect a set of nodes with farthest propagated mutation effects as $Frontier(\omega, \omega') \subset V$:

$$Frontier(\omega, \omega') = \{v | \omega(v) \neq \omega'(v) \text{ and } dist(v) = macroPropagationDistance(\omega, \omega')\} \quad (5.2)$$

They will be used as the basis for calculating a *local propagation cost*, since, as the name mentions, these are the frontier of propagation.

5.3.4. Local Propagation Cost

In this section, we discuss the detail of *local propagation cost*. We first present an example for its essential idea. Then we formulate the procedure of its calculation in different situations and, in particular, how this can be implemented on a CDFG.

For each farthest propagated mutation effect, for example at node u in **Figure 5.5**-b), we take a closer look at why it is blocked by the *statement nodes that have it as an operand* – there s_2 and s_7. At each such statement, there is a straightforward condition for the mutation effect to propagate through:

$$\omega(s_7) \neq \omega'(s_7) \text{ for } s_7$$

Where $\omega(s_7)$ is used to denote the computation of s_3 with the values in ω, as previously defined. This corresponds to $\big(\omega(u) = \omega(d)\big) \neq \big(\omega'(u) = \omega'(d)\big)$.

The *key here is that we can transform any condition, or constraint* $\omega(s) \neq \omega'(s)$ *for a* $s \in S$ *with regard to mutation-analysis result* (ω, ω') *equally to a Boolean expression* $\omega(s) \wedge \omega'(\overline{s}) \vee \omega(\overline{s}) \wedge \omega'(s)$, on which a satisfaction degree can then be calculated, by leveraging the *boolean_cost* table from **Figure 5.6**.

We also note that such calculation relays purely on actual simulation values (ω, ω') that are *not* symbolic.

Example 5.1-3: local propagation cost

On the CDFG from Example 5.1-1, with input $(a = 4, b = 1, d = 0)$ the mutation effect at u will be blocked at s_7, as condition $\omega(s_7) \neq \omega'(s_7)$ is not satisfied with $\omega(u) = 5$, $\omega'(u) = 3$, and $\omega(d) = \omega'(d) = 0$. Nevertheless, its satisfaction degree can be

estimated, as the *local propagation cost* that we call, by

$$boolean\_cost\big(\omega(s_7)\big) = boolean\_cost\big(\omega(u) = \omega(d)\big) = boolean\_cost(5 = 0) = 5$$

$$boolean\_cost\big(\omega(\overline{s_7})\big) = boolean\_cost\big(\omega(u) \neq \omega(d)\big) = boolean\_cost\ (5 \neq 0) = 0$$

$$boolean\_cost\big(\omega'(s_7)\big) = boolean\_cost\big(\omega'(u) = \omega'(d)\big) = boolean\_cost(3 = 0) = 3$$

$$boolean\_cost\big(\omega'(\overline{s_7})\big) = boolean\_cost\big(\omega'(u) \neq \omega'(d)\big) = boolean\_cost\ (3 \neq 0) = 0$$

$$\begin{aligned} &localPropagationCost_{s_7}(\omega, \omega') \\ &= boolean\_cost\big(\omega(s_7) \neq \omega'(s_7)\big) \\ &= boolean\_cost\big(\omega(s_7) \wedge \omega'(\overline{s_7}) \vee \omega(\overline{s_7}) \wedge \omega'(s_7)\big) \\ &= 5 \times 3 \,/\, (5 + 3) \\ &= 1.875 \end{aligned}$$

Consider that we are in some search procedure and another candidate test is selected by a slight increase of a to 5. This leads to a new $localPropagationCost_{s_7}$ as $6 \times 4/(6 + 4) = 2.4$, which should be seen as a hint of wrong search direction as it increases the cost. Going the opposite direction we could try $a = 3$, which reduces the cost to $4 \times 2/(4 + 2) = 1.33$. The reduction gives a sign of test improvement and the search should be encouraged to follow this direction.

If we follow this way and further decrease a to 1, we land on a test that satisfies the local propagation condition at s_3. The mutation effect spread further through s_3 and automatically $macroPropagationDistance$ is also reduced, by 1 at least. ■

To conclude the essential ideas of *local propagation cost*, for a mutation effect to propagate through a HDL design statement $s \in S$ on CDFG, we calculate

$$\begin{aligned} &localPropagationCost\ _s(\omega, \omega') \\ &= boolean\_cost\big(\omega(s) \neq \omega'(s)\big)\ = boolean\_cost\big(\omega(s) \wedge \omega'(\overline{s}) \vee \omega(\overline{s}) \wedge \omega'(s)\big) \end{aligned} \quad (5.3)$$

In the following, we formulate a procedure for calculating such local cost for every $v \in Frontier(\omega, \omega')$, as an extension to $CDFG$.

1) We extend a CDFG by attaching a Boolean cost function to each variable node, specifically for HDL mutation analysis.

Commonly, we can identify three types of HDL operations: *arithmetic operations* such as addition or multiplication, *bit manipulation operations* such as concatenation or shift, and *Boolean operations* that include logical operations as well as relational operations. Our first observation is that it should be relatively easy for a mutation effect to propagate through arithmetic operations and bit operations and, therefore, a finer scale with local cost is *not* necessary at the corresponding nodes.

For example, a node with $h + i$ will not be examined for local cost, since, if one of its operands h or i has a mutation effect, it should be highly probable for the node also to of

compute a deviated value in mutant simulation. In contrast, Boolean operations may expose particularly low probability for a mutation effect to get through. When a receives a mutation effect, it is easy for (a *and* b *and* c *and* d) to *mask* this deviation in mutant simulation and block the propagation. For another example Boolean operation a > b, mutation effect on a propagates through, only when deviation is of a big enough magnitude.

Therefore, we consider calculating the local propagation cost only at variable nodes from Boolean operations. Importantly, this includes all the *branch nodes* that we build into CDFG, by which all the design control flows are taken into account.

For this, we first assume that $S_{Boolean} \subset S$ in a $CDFG$ are statement nodes with Boolean evaluation and $V_{Boolean} \subset V$ is $\{v|\ in(v) \subset S_{Boolean}\}$. For each branch controlled statement node $s \in S_{bran-controlled}$ and $s \in S_{Boolean}$, we aggregate all its incoming control edges $branch(s) = \{e_1 = (v_1, s), e_2 = (v_2, s), \cdots\}$ and extend its evaluation ω_{ext} as

$$\omega_{ext}(s) = \big(\omega(v_1) = \delta(e_1)\big) \wedge \big(\omega(v_2) = \delta(e_2)\big) \wedge \cdots \wedge \omega(s) \quad (5.4)$$

We have $\omega'_{ext}(s)$ in the same way. Then, for each branch controlled Boolean variable $v \in V_{bran-controlled}$ and $v \in V_{Boolean}$, we consider all $in(v) = \{s_1, s_2, \cdots\}$ and attach to it a local cost function

$$localPropagationCost_v(\omega, \omega') = boolean\_cost\big((\omega_{ext}(s_1) \vee \omega_{ext}(s_2) \vee \cdots) \neq (\omega'_{ext}(s_1) \vee \omega'_{ext}(s_2) \vee \cdots)\big) \quad (5.5)$$

The function is calculated by expansion with (5.3) and further with (5.4). In short, all the control dependences of this variable node are taken into account when calculating the local Boolean cost. This is no repetition of the design simulation, but only an analysis of the simulation results, with already happened values from the simulation.

For $v \notin V_{bran-controlled}$ and $v \in V_{Boolean}$, its $in(v)$ should be a single Boolean statement $\{s_1\}$ that $s \notin S_{bran-controlled}$ and $s \in S_{Boolean}$. We simply attach to v the local cost function

$$localPropagationCost_v(\omega, \omega') = localPropagationCost_s(\omega, \omega') \quad (5.6)$$

where $localPropagationCost_s(\omega, \omega')$ is defined and explained with (5.3).

There is one small exception for $v \in V_{Boolean}$ when $in(v)$ just contains the mutation statement s_{mutant}. In such a case, if $v \in V_{bran-controlled}$ and $in(v) = \{s_1, \cdots, s_{mutant}, \cdots\}$, we adjust (5.5) and attach to v:

$$localPropagationCost_v(\omega, \omega') = boolean\_cost\big((\omega_{ext}(s_1) \vee \cdots \vee \omega_{ext}(s_{mutant}) \vee \cdots) \neq (\omega'_{ext}(s_1) \vee \cdots \vee \omega'_{ext}(s'_{mutant}) \vee \cdots)\big) \quad (5.5')$$

If $v \notin V_{bran-controlled}$ and $in(v) = \{s_{mutant}\}$, we adjust (5.6) as

$$localPropagationCost_v(\omega, \omega')$$
$$= boolean\_cost\big(\omega(s_{mutant}) \neq \omega'(s'_{mutant})\big)$$
$$= boolean\_cost\left(\omega(s_{mutant}) \wedge \omega'\left(\overline{s'_{mutant}}\right) \vee \omega(\overline{s_{mutant}}) \wedge \omega'(s'_{mutant})\right) \qquad (5.6')$$

Taking into account this exception *enables us to take into account the activation condition.* When there are no mutation effect in the mutant simulation, (5.5′) or (5.6′) calculates exactly the degree of the activation condition being satisfied.

If $v \notin V_{Boolean}$ not from a Boolean statement, we attach to it a small constant number K as its local cost, which reflect the assumption that it may very easily receive a mutation effect.

We notice that this extension is also static to $CDFG$. A mapping from this definition to an implementation should be relatively straightforward, which we will have experiments in our evaluation chapter.

Example 5.1-4: CDFG extension of Example 5.1-1 for calculation of *localPropagationCost*.

As an example of the above defined extension, we attach local cost functions to the $CDFG$ in **Figure 5.4**. Inputs $\{a, b, c, d, e, f\}$ are ignored as they always receive the same values in mutant simulation. It is also *not* necessary for node u, as it is *not* result from a Boolean operation and we expect that it *will probably* have a deviation in mutant simulation, i.e. the *activation* is probable, in this example.

For variable nodes$\{v, y, branch_{s_2}, branch_{s_4}, x\}$:

- $localPropagationCost_v = boolean\_cost\big(\omega(s_7) \neq \omega'(s_7)\big) = boolean\_cost\left(\omega(g = d) \wedge \omega'\big(not\ (g = d)\big) \vee \omega\big(not\ (g = d)\big) \wedge \omega'(g = d)\right)$, by (5.3)
- $localPropagationCost_y = boolean\_cost\big(\omega(s_8) \neq \omega'(s_8)\big) =$ $boolean\_cost\left(\omega(v\ and\ e\ and\ f) \wedge \omega'\big(not\ (v\ and\ e\ and\ f)\big) \vee \omega\big(not\ (v\ and\ e\ and\ f)\big) \wedge \omega'(v\ and\ e\ and\ f)\right)$, by (5.3)
- $localPropagationCost_{branch_{s_2}} = boolean\_cost\big(\omega(s_2) \neq \omega'(s_2)\big) =$ $boolean\_cost\left(\omega(u > c) \wedge \omega'\big(not\ (u > c)\big) \vee \omega\big(not\ (u > c)\big) \wedge \omega'(u > c)\right)$, by (5.3)
- $localPropagationCost_{branch_{s_4}} = boolean\_cost\big(\omega_{ext}(s_4) \neq \omega'_{ext}(s_4)\big) =$ $boolean\_cost\big(\omega_{ext}(s_4) \wedge \omega'_{ext}(\overline{s_4}) \vee \omega_{ext}(\overline{s_4}) \wedge \omega'_{ext}(s_4)\big)$ by (5.5), where by (5.4):
  - $\omega_{ext}(s_4) = \big(\omega\big(branch_{s_2}\big) = false\ \big) \wedge \omega(u = c)$
  - $\omega'_{ext}(s_4) = \big(\omega'\big(branch_{s_2}\big) = false\ \big) \wedge \omega'(u = c)$

- $\omega_{ext}(\overline{s_4}) = not\ \left((\omega(branch_{s_2}) = false\ )\ \wedge\ \omega(u = c)\right)$
- $\omega'_{ext}(\overline{s_4}) = not\ \left((\omega'(branch_{s_2}) = false\ )\ \wedge\ \omega'(u = c)\right)$

- $localPropagationCost_x = boolean\_cost\left((\omega_{ext}(s_3) \vee \omega_{ext}(s_5) \vee \omega_{ext}(s_6)) \neq (\omega'_{ext}(s_3) \vee \omega'_{ext}(s_5) \vee \omega'_{ext}(s_6))\right)$ by (5.5), where by (5.4):
  - $\omega_{ext}(s_3) = (\omega(branch_{s_2}) = true\ )\ \wedge\ \omega(s_3)$
  - $\omega_{ext}(s_5) = (\omega(branch_{s_2}) = false\ ) \wedge (\omega(branch_{s_4}) = false\ )\ \wedge\ \omega(s_5)$
  - $\omega_{ext}(s_6) = (\omega(branch_{s_2}) = false\ ) \wedge (\omega(branch_{s_4}) = true\ )\ \wedge\ \omega(e\ or\ f)$
  - $\omega'_{ext}(s_3)$, $\omega'_{ext}(s_5)$, $\omega'_{ext}(s_6)$, $\omega_{ext}(\overline{s_3})$, $\omega_{ext}(\overline{s_5})$, $\omega_{ext}(\overline{s_6})$, $\omega'_{ext}(\overline{s_3})$, $\omega'_{ext}(\overline{s_5})$ and $\omega'_{ext}(\overline{s_6})$ follow the same way and we leave out the description.

Note that ∧ and HDL native *and* are used interchangeably and single bits are also treated as Boolean, which are minor implementation issues.

Consider that our current test as search coordinate is simply $\{a = 1, b = 1, c = 0, e = 1, f = 1\}$. Output x does not receive a mutation effect but we can calculate

- $boolean\_cost(\omega_{ext}(s_3) \vee \omega_{ext}(s_5) \vee \omega_{ext}(s_6)) = 0$
- $boolean\_cost\left(\overline{\omega_{ext}(s_3) \vee \omega_{ext}(s_5) \vee \omega_{ext}(s_6)}\right) = boolean\_cost(\omega_{ext}(\overline{s_3}) \wedge \omega_{ext}(\overline{s_5}) \wedge \omega_{ext}(\overline{s_6})) = 1 \times 1/(1+1) + 0 + 0 = 0.5$
- $boolean\_cost(\omega'_{ext}(s_3) \vee \omega'_{ext}(s_5) \vee \omega'_{ext}(s_6)) = 0$
- $boolean\_cost\left(\overline{\omega'_{ext}(s_3) \vee \omega'_{ext}(s_5) \vee \omega'_{ext}(s_6)}\right) = boolean\_cost(\omega'_{ext}(\overline{s_3}) \wedge \omega'_{ext}(\overline{s_5}) \wedge \omega'_{ext}(\overline{s_6})) = 0 + 0 + 1 \times 1 \times 2/(1+1+2) = 0.5$
- $localPropagationCost_x = 0.5 \times 0.5/(0.5 + 0.5) = \mathbf{0.25}$

Suppose that we are in a local search and have a neighborhood test $\{a = 1, b = 1, c = 0, e = 0, f = 1\}$ by adjusting only e. The new local cost at x is then

- $boolean\_cost\left(\overline{\omega'_{ext}(s_3) \vee \omega'_{ext}(s_5) \vee \omega'_{ext}(s_6)}\right) = boolean\_cost(\omega'_{ext}(\overline{s_3}) \wedge \omega'_{ext}(\overline{s_5}) \wedge \omega'_{ext}(\overline{s_6})) = 0 + 0 + 1 \times 1 \times 1/(1+1+1) = 0.33$
- $localPropagationCost_x = 0.5 \times 0.33/(0.5 + 0.33) = \mathbf{0.2}$

This shows how $localPropagationCost$ reflects the gradual improvement of test, with regard to propagation of HDL mutation effect. ■

---

2) We calculate local cost at each variable node that are the potential propagation points from $Frontier(\omega, \omega')$ and select a minimal among all.

Recall that $Frontier(\omega, \omega') \subset V$ are the farthest propagated mutation effects that we have collected during the calculation of *macro propagation distance*.

But first, we can define for each $v \in V$, $Propagation(v) \subset V$ $\{v' | in(v') \bigcap out(v) \neq \emptyset, \text{and } dist(v') = dist(v) - 1\}$. They are variable nodes that are connected to v by two edges over one statement node and, at the same time, one-step nearer to design output than v. They represent potential destination of propagation of any mutation effect on v. As long as they receive a propagated mutation effect, the macro propagation distance will be reduced too.

Note that $Propagation(v)$ is also a static properties of v, which can be computed directly on $CDFG$.

Then, $Propagation(\omega, \omega') \subset V$ is collected as

$$Propagation(\omega, \omega') = \bigcup_{v \in Frontier(\omega, \omega')} Propagation(v) \quad (5.7)$$

And an overall minimal local cost can be calculated as:

$$localPropagationCost(\omega, \omega') = \min_{v \in Propagation(\omega, \omega')} \left(localPropagationCost_v(\omega, \omega') \right)$$

In a simple example with **Figure 5.7**, only the $localPropagationCost_e$ function attached to node e will be used for cost calculation, since i) x does *not* belong to $Frontier(\omega, \omega')$, ii)) g is *not* from a Boolean operation, and iii) f does *not* belong to $Propagation(a)$.

3) If $Frontier(\omega, \omega') = \emptyset$, we calculate the local propagation cost at the result variable from the mutation statement, by

$$localPropagationCost(\omega, \omega') = localPropagationCost_{out(s_{mutant})}(\omega, \omega')$$

This measures a degree for the *activation* condition to be satisfied, in order to generate the mutation effect.

4) The overall cost on trace (ω, ω') is summed up as:

$$cost(\omega, \omega') = macroPropagationDistance(\omega, \omega') - 1 + localPropagationCost(\omega, \omega')/H$$

where H is a big constant that intends to always reduce the impact of local cost under 1.

Since we consider only the simulation and verification of synchronous HDL designs, $cost(\omega, \omega')$ is calculated for all cycles and the smallest one is selected as $cost(test)$.

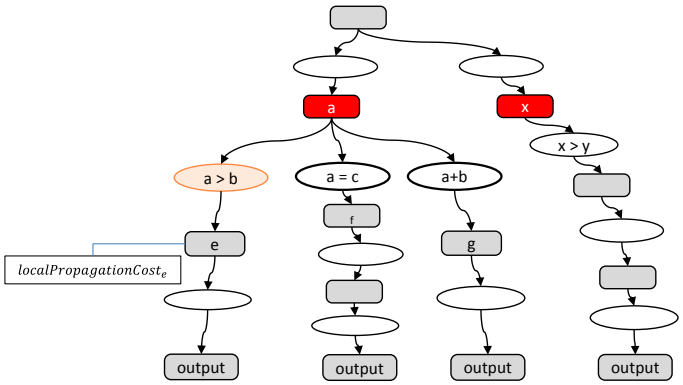


Figure 5.7 This example shows that only $localPropagationCost_e$ will be used for cost calculation.

5.3.5. Algorithmic Summary and Complexity

In **Figure 5.8**, we give an algorithmic summary for the calculation of $cost_{mutant}(test)$, which measures the progress of killing a specific HDL $mutant$, in particular, with regard to *activation* and *propagation*.

We briefly discuss the complexity of this algorithm:

- $dist(v)$ as a static value, $Propagation(v)$ as a static list of nodes, and a function $localPropagationCost_v$ are all statically prepared on CDFG and, therefore, impose *no* impact on simulation time.
- With $dist(v)$ available, the calculation of (5.1) and (5.2) requires only value comparison for each variable which should require little effort. Also, (5.7) requires little effort as a simple aggregation of the static $Propagation(v)$.
- For each node collected by (5.7), the effort of calculation with $localPropagationCost_v$ should be at the same level as the original statement that computes v. Assuming that the set of candidate nodes for propagation will be a small fraction of all variables, the calculation of such local cost should be minor compared to the original simulation time.

Therefore, we conclude that the overall time for calculating this $cost_{mutant}(test)$ for each test, *based on the exiting simulation trace values from this test*, should be minor to the original simulation time, which is important for its integration into a search algorithm. And we should be able to observe this property in our evaluation experiment with microprocessor and floating point unit design.

Algorithm of $cost_{mutant}(test)$

// To be used as the cost function in **Figure 5.2**, *for each specific mutant*
H is a big constant and K is a small constant, which reduces the impact of local cost to be between $(0,1)$;

Data Structure Preparation

Construct from design a $CDFG = (V, S, E, \delta, O, s_{mutant})$, by Definition 5.3.1;

Compute $dist(v)$ and $Propagation(v)$ for each v in V;

Extend $CDFG$ and attach local cost functions to variable nodes, based on (5.5) (5.6) (5.5') and (5.6');

Input

Input is $test \in T$, which makes $cost_{mutant}$ a function: T→real value, T is …

$W_{mutant,test}$ and $W'_{mutant,test}$ are a pair of simulation traces from $test$, consisting of $\{\omega_1, \omega_2, \cdots\}$ and $\{\omega'_1, \omega'_2, \cdots\}$ for simulation cycles

Start

Set $macro\_cost = dist\big(out(s_{mutant})\big) + 1$;
FOR each cycle i in $W_{mutant,test}$ and $W'_{mutant,test}$ DO
    // (ω_i, ω'_i) is from $W_{mutant,test}$ and $W'_{mutant,test}$, as described;
    Set $macro\_cost_i = macroPropagationDistance(\omega_i, \omega'_i)$, by (5.1);
    Set $macro\_cost = \min\{macro\_cost_i, macro\_cost\}$;
END FOR;
Set $local\_cost = H$;
FOR each value changed cycle i in $W_{mutant,test}$ and $W'_{mutant,test}$ again DO
    IF $macro\_cost_i$ equals $macro\_cost$ THEN
        Identify a list $F_i = Frontier(\omega_i, \omega'_i)$, by (5.2);
        IF F is not empty THEN
            Identify a list of candidate nodes for propagation $P_i = Propagation(F_i)$ by (5.7);
            FOR each v *in* P DO
                Set $local\_cost_i = localPropagationCost_v(\omega, \omega')$ as attached to v;
                Set $local\_cost = \min\{local\_cost_i, local\_cost\}$
            END FOR;
        ELSE
            Set $local\_cost_i = localPropagationCost_{out(s_{mutant})}(\omega_i, \omega'_i)$ as attached;
            Set $local\_cost = \min\{local\_cost_i, local\_cost\}$;
        END IF;
    END IF;
END FOR;
Set $cost = macro\_cost - 1 + local\_cost/H$

End

Figure 5.8 Algorithmic summary of cost function.

5.4. Related Work

We have proposed metaheuristic search based test generation for killing HDL mutants. In this section, related literature is discussed. On one hand, we review several *fault oriented test generation* methods and discuss why they do *not* suite our problem of killing HDL mutants. On the other hand, we also discuss *search based test generation methods* that targets *other* metrics and why they do *not* apply to HDL mutation analysis.

- In the background chapter, we have compared the HDL mutation and gate-level fault models, as well as mutation-based simulation test generation to traditional Automatic Test Pattern Generation (ATPG) algorithms [55] [71] [72] [73]. ATPGs does *not* directly apply to HDL mutation analysis in functional verification, as i) relying on scan-chain techniques, they are basically based on structural testing scheme and do not take a whole design as input and ii) with HDL mutation, we do *not* assume the synthesizability of a design under verification, which can be VHDL, Verilog, or C/SystemC. In our search based approach, we define the objective cost function based on a control and data flow data structure, which can be extracted from both RTL/behavioral designs. The final test generation applies to any HDL designs that are simulatable.

- The *observability-based coverage* [54], also discussed in the background chapter, has a similar test generation problem to mutation analysis, since a *tag* also models an error to be propagated. In [86], it is transformed to a *Hybrid Boolean Satisfiability* (HSAT) problem. Based on a structural graph compiled from the HDL design description, a mixed set of Boolean and linear constraints is generated for both the tagged and untagged versions. Then, for each output data node, another constraint is added to guarantee the tag detection. At last, the collected HSAT problem is solved to obtain the target test data.

- The original mutation analysis based test generation [87] relies on *linear constraint solving*. First, the program is *symbolically executed* to establish the path from input to a fault location. At each branch predicate, a constraint is collected for the intended path. When the fault statement is reached, another constraint is added to handle the *activation* condition. Then tests are generated by solving the entire set of constraints. The *propagation* problem is *not* considered.

- In [68], VHDL designs are translated to SW programs and fed into the software mutation analysis tool in [87] , so as to generate mutation-oriented test data for both design verification and manufacturing testing.

We can see that these fault-oriented test generation methods rely mostly on *symbolic execution* and *constraint solving* to obtain a definitive target test. As symbolic execution

may encounter the *path explosion problem* [88] and constraint satisfaction problems also face high complexity [89], they are regarded *not* scalable to large designs, in general.

In contrary, search-based test generation methods intend to find target tests based only on *actual* design or program execution, in an iterative manner. Therefore, they are expected to scale well in line with HDL simulation, when applied for functional design verification. The trade-off is that a search success is *not* guaranteed and the search performance may vary. A survey on search based software test generation can be further found in [80].

- [90] systematically discusses how to apply search based test generation to a specific coverage metric: *path coverage*, i.e. to achieve complete execution of a specific program path. Sub-goals are defined as satisfaction of intended branches. For each such branch, a cost function is defined to steer the branch satisfaction, which is evaluated with the variable values during actual program execution, to be minimized to zero.

  Actually, path coverage subsumes the *reachability* problem in mutation analysis and could be complementary to our method. However, as mentioned, we assume that in HDL simulation, reachability (line coverage) is easy to satisfy. Therefore, we focus our cost function definition as well as search on *activation* and *propagation*.

- This search-for-path-coverage principle is applied in [91] to mutation analysis. A similar cost function is defined on the test input space and reflects the progress of path-following. *Ant Colony* search is employed to minimize the cost and find the target test. Again, only mutation *reachability* is taken into account by the cost function.

- Further, we can find *hybrid techniques combining simulation based search and formal methods* for test generation, such as the abstraction-guided simulation presented in [92] [93] and [94]. Coverage of a specific set of design states is their search goal. A Finite State Machine (FSM) abstracted from the design is used to guide the search of test inputs that reach a target state. [92] builds the abstraction by selecting the design module containing the verification property and the modules that interacts closely with it, under some complexity constraint with regard to the final product FSM. With data-mining techniques, this abstraction can be also done as in [93] and [94] by partitioning state variables that are high correlated to the target state.

  Based on the abstract FSM model, pre-images of the targets state are iteratively computed via a Satisfiability (SAT) engine. Then, a simulation trace can be mapped to the abstract model to obtain the current state. The distance from the current state

to the target state becomes the cost function of search, guiding the search towards a target test input.

Equipped with such guidance, the search algorithms employed include a simple random walk in [92], more sophisticatedly a cultural algorithm in [93] and a genetic algorithm in [94]. The SAT engine also intervenes during search to bridge the current state to a closer state, when the search heuristics get stuck at a dead-end state.

Comparison of Literature to Our Work

We conclude the distinction between our search-based test generation method for mutation analysis and those found in the literature as follows.

- The most significant difference is that our method is purely based on actual HDL design simulation. It can be integrated into any simulation-based verification process. No design synthesizability needs to be assumed. Also, no symbolic manipulation or simulation is required. Moreover, compared to the abstraction-based hybrid approaches, the graph structure that we extract from a design to define the search cost function represents the *static structure* of the design instead of its state transitions. No symbolic methods or SAT is needed for the computation on this graph and we resort only to actual simulation values for the cost calculation.
- Compared to other metric-oriented, search-based test generation methods, only our cost function definition handles all three problems in mutation analysis: reaching, activating, and propagating a mutant.

To the best of our knowledge, it is the first such effort to develop a search-based, non-symbolic test generation method for HDL mutation analysis.

There is also discussion in [83] related to *automated extraction* of similar CDFG structures as used in our cost function. We view this as reasonable future work to complement the automation flow of our method.

5.5. Summary

We have considered the problem of test generation for killing a specific design mutant, for HDL mutation analysis. This corresponds to the problem of handling each of the un-killed IP design mutant after the adaptive random simulation phase, in the context of our metrics driven functional verification flow for IP-based SoC design.

We have proposed a novel, metaheuristic search based method for such test generation. The idea is that we apply a search algorithm on the design input space. In iterations, the

search evaluates and improve the candidate tests, towards some final target that kills the mutant. This approach has the advantage of relying only on actual design simulation, in contrast to symbolic execution or constraint solving that we have seen in the related work.

As the key of enabling such a search with the goal of killing a mutant, an objective cost function has been proposed. It is devised exactly with the three conditions for killing a HDL design mutations in mind: *reach* the mutation statement, *activate* the mutant with a local deviation, and *propagate* such deviation to output.

Therefore, we have modeled and analyzed these conditions on a Control and Data Flow Graph, since it enables a *direct mapping of the conditions onto that graph and then a quantitative measurement of them from being satisfied* – in particular, the activation and propagation.

This quantitative measurement, after we mapping a mutation-analysis simulation trace onto the CDFG, consists of a *macro propagation distance* as a general distance of mutation effects to design output and a *local propagation cost*, which transforms local propagation conditions to Boolean expressions and then leverages a *boolean_cost* to estimate the satisfaction degree of such conditions. Together, they provides a complete search guidance with regard to HDL mutant activation and propagation. Also, the cost function takes existing simulation traces as input and impose minor calculation effort to the actual simulation.

In the evaluation chapter, we will mainly investigate the effectiveness of the cost function as the steering of a local search algorithm, i.e. whether it can consistently lead the search to a target mutant-killing test, for a real IP design and simulation. The evaluation or comparison with more complex metaheuristics is seen as reasonable future work.

Although the method is established with mutation analysis as the quality metric, we see *no* restriction on its application to other metrics.

This contribution has been first published in [6] and further elaborated in [1].

CHAPTER 6: SoC System Design Simulation and Mutation Analysis with IP-XACT

6.1. Introduction

In this chapter, we present a *systematic* verification method for SoC system design. The method is simulation-based and with mutation analysis integrated as the quality metric for such simulation. In the IP-based SoC design paradigm, this is where we assemble pre-verified IP components into an integrated SoC system.

Motivation for System Simulation and Mutation Analysis with IP-XACT

First, we assume *IP-XACT* [29] as the default language that we use for SoC system design, since:

- IP-XACT is *the* standard for IP re-use and SoC integration, therefore just suit our overall methodology. It should be more reasonable for us to establish this system verification method with IP-XACT, as opposed to a proprietary language, such as MHS (Microprocessor Hardware Specification) that we mentioned for SoC design on Xilinx FPGA. Also, SoC design in IP-XACT is more evident, if we assume that IPs are provided with IP-XACT as metadata
- Creating a system verification method based on IP-XACT should enable the verification to focus on IP integration – their instantiation, interconnection, and parameter configuration.

This focus of verification on IP integration through IP-XACT is even necessary, since i) we can*not* expect the availability of IP code and a white-box system test and ii) we need to handle the increasing complexity of IP and IP integration by assuming the correctness of delivered IPs. This has been elaborated in our background chapter requirement as *division and separation of IP design and SoC system integration*. The previous two methods for mutation analysis driven verification – *adaptive random simulation* and

metaheuristic based test generation– are exactly our effort towards a thorough IP verification and, thus, its correctness.

Further, we assume *system simulation* as a necessary verification step for SoC system design, before its final implementation to ASIC or FPGA. Even for FPGA based implementation with relatively low cost, system simulation provides a *far better observability* compared to a final testing on FPGA. Nevertheless, system simulation does *not* intend to replace emulation or FPGA prototyping.

Therefore, the first problem for establishing an IP-XACT based, systematic verification methods is that *SoC system designs in IP-XACT are not directly simulatable*. Since they are in the form of XML files and XML is *not* executable, we need at first a simulation engine for IP-XACT. Our approach is to transform an IP-XACT design to another system model that is simulatable. The destination language that we choose for this transformation is SystemC.

The second question is, how can we systematically manage the quality of such system verification based on IP-XACT simulation? Following our consistent focus on *metrics driven verification* for IP-based SoC design, and assuming mutation analysis the advanced, effective metric that we employ, we consider the problem of enabling *mutation analysis with IP-XACT*. Here, a key should be the definition of *mutation operators* on IP-XACT – how XML errors are to be injected into IP-XACT system designs.

Contribution of the Chapter

With this chapter, we *contribute by proposing a SoC system design simulation and mutation analysis framework based on IP-XACT, to be the third, system-level component of our mutation analysis driven functional verification methodology for IP-based SoC design. The framework consists further of two contributions. The first is a SystemC based IP-XACT design synthesis and simulation flow that enables the functional verification of SoC designs. The second is the definition of a set of mutation operators on IP-XACT schema, which enables IP-XAXT mutation analysis as an advanced quality metric for system simulation.*

Organization

In Section 6.2, we first give an overview of our proposal for an IP-XACT based SoC design simulation and mutation analysis framework and, in particular, why SystemC is chosen as the target platform. Then, Section 6.3 details the IP-XACT-to-SystemC synthesis flow and rules. Section 6.4 introduces a list of IP-XACT mutation operators. In Section 6.5, we present an Eclipsed-based tool that we have implemented for our proposal. Related work in literature is discussed in Section 6.6 and the chapter is concluded by Section 6.7.

6.2. An IP-XACT Design Simulation and Mutation Analysis Framework

Figure 6.1 shows an overview of our proposal: an IP-XACT design simulation and mutation analysis framework, for systematic functional verification of SoC system designs. The framework consists further of two components, or interacting flows.

The first is a *SystemC based IP-XACT synthesis flow*, which takes an IP-XACT XML design file as input and generates a SystemC model as output. It is proposed as a well-defined transformation process by a set of checking and mapping rules, to be introduced in next section. The generated simulation should match the original functionality of the IP-XACT design. This functionality is how we should interpret the execution behavior of an IP-XACT design, i.e. its semantics.

Although there is no formal definition on the behavioral semantics of IP-XACT as a structural and HDL-neutral format, the *execution behavior* of an IP-XACT system design, to be either simulation or real circuit operation, is already implied by a combination of individual behaviors from the included IP components and their integration described by IP-XACT. For this, we also assume that IPs are always packaged being accompanied by a simulation model. For example, in the Xilinx FPGA design environment, although the *MicroBlaze* microprocessor IP comes only as a hard IP without source code, another model is provided for system integrated simulation. If an IP is presented as a soft core, RTL or TLM, it is directly simulatable.

The reason for us making SystemC the synthesis destination is that *only* it provides a single platform for multi-language, mixed-level simulation – RTL, behavioral, or TLM, as we have introduced in Chapter 2. We assume that TLM is a state-of-the-art method necessary for inclusion and our SoC system design may contain TLM IPs. IP-XACT indeed handles both RTL and TLM.

With a modern simulation tool such as ModelSim, SystemC and other HDLs – VHDL and Verilog – can be simulated above a single kernel with all their original semantics retained.

The second component of the framework is an *IP-XACT mutation analysis flow*, referring to the creation of IP-XACT *design* mutants and the measurement of whether they can be killed under simulation, by seeing whether they produce deviated simulation traces. For this, our main effort is devoted to the definition of a set of mutation operators on IP-XACT.

Mutation analysis is language specific. The rationale behind applying the principle of mutation analysis to any new design language is that i) each mutation operator models a small syntactic error that may commonly be made by a designer and *should* be uncovered

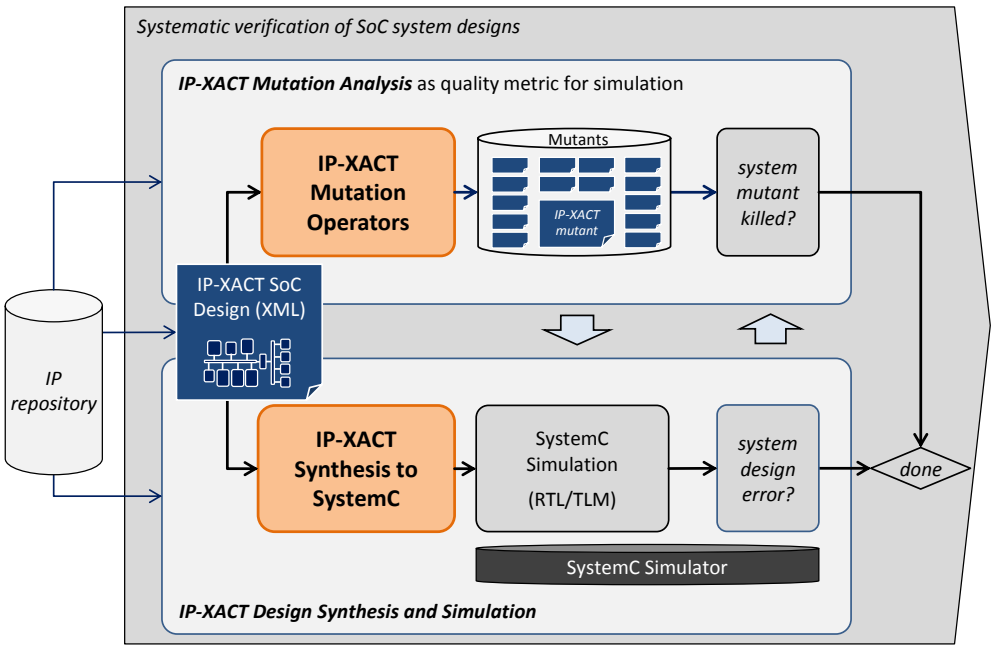


Figure 6.1 IP-XACT SoC system design simulation and mutation analysis framework.

by simulation and ii) these single small errors are supposed to be coupled with more complex potential bugs, in the sense that if a set of tests can kill those artificially generated mutants, they should also be able to reveal the real bugs in the design. We call it the *double effectiveness* of mutation analysis as a quality metric for functional verification, which is expected to be also applicable to SoC system design with IP-XACT.

Therefore, IP-XACT mutation operators are defined on IP-XACT XML schema as the target language. They represent errors that we can implant into an IP-XACT XML *design* document, to mimic representative errors that one can make with IP-XACT *design*.

The derived mutation analysis flow is then intended to qualify the simulation based IP-XACT design verification. As interaction between these two flows, each mutant should be fed into the synthesis and simulation flow, with the traces retrieved for measuring the *kill* of this mutant.

In the end, we have this systematic verification framework for IP-XACT based SoC system design, as one important step towards solving the verification closure problem at system-level – *are we done with system verification*.

At the moment, we see the software running on a SoC to be the system tests. We leave the automated improvement of system tests as part of further work.

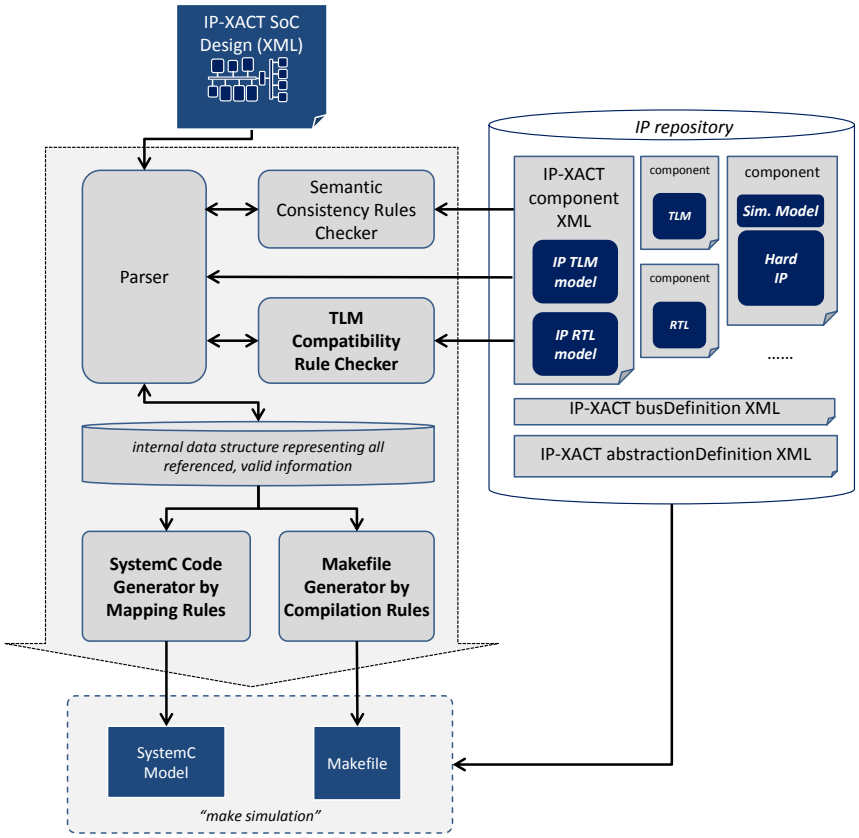


Figure 6.2 SystemC based IP-XACT synthesis flow.

6.3. SystemC Based IP-XACT Design Synthesis and Simulation

As shown in **Figure 6.2**, the *SystemC based IP-XACT synthesis* is defined as a straight one-pass flow, which goes through a series of processors that are derived from a set of pre-defined rules. The processors require and retrieve also information from an IP repository that contains IP-XACT described IPs and bus/abstraction definitions. An IP-XACT-to-SystemC model generator is implied from this flow, which we have implemented as an Eclipse tool for further experiments.

Parser

The parser, as detailed in **Figure 6.3**, parses not only the IP-XACT *design* but also all the IP-XACT *components* instantiated in the design and all the *bus/abstractionDefinitions* that are referenced by the design and components.

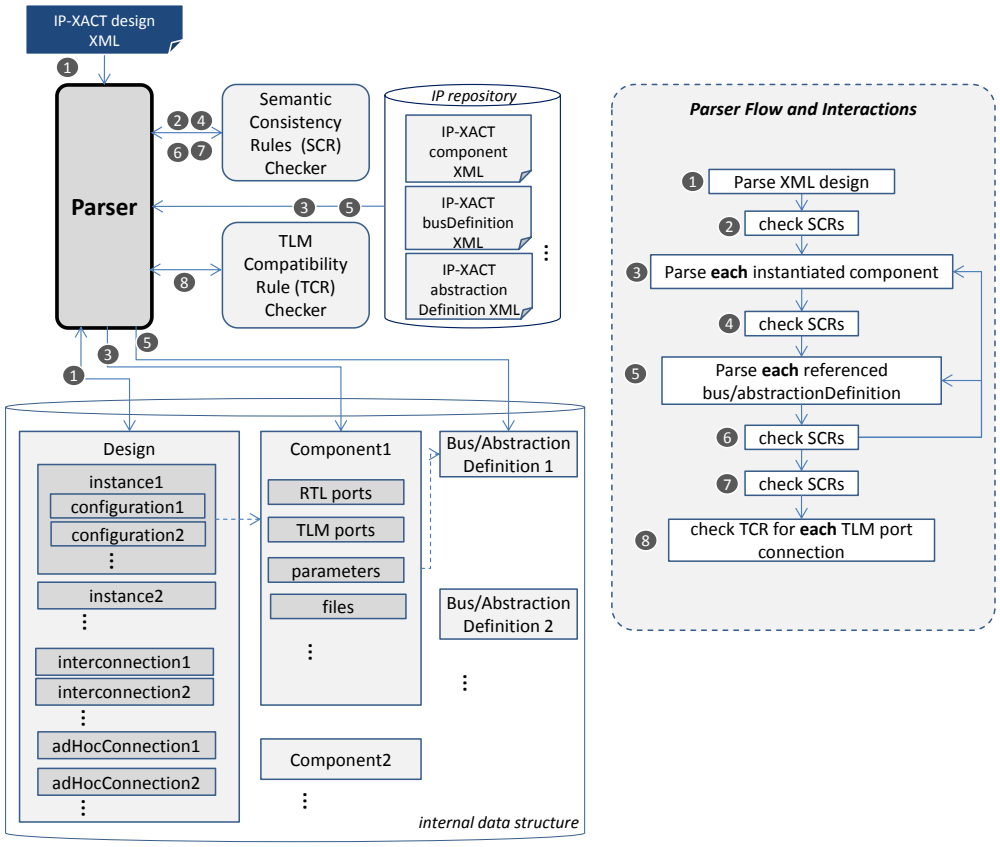


Figure 6.3 Flow of parser and its interactions with other parts.

The parser itself should be derived directly from XSD schema definitions in IP-XACT standard, so as to make sure that a design is both well-formed and valid according IP-XACT schema. For this purpose, it needs interaction to the *Semantic Consistency Rules Checker* and TLM Compatibility Rule Checker, as shown in the figure.

After the parsing, we have a one-to-one internal representation of the design, which consists of the component instances, their configurations and their connections. Although Java classes are used for this purpose in our implementation, we propose no definition or restriction on this internal representation. It should be straightforward since XML is a total structured representation. No intermediate code is generated before the final SystemC and Makefile generation.

The final output of the parser is a one-to-one, both syntactically and semantically correct internal representation of the IP-XACT design and all the components instantiated.

Semantic Consistency Rules Checker

The *Semantic Consistency Rules* (SCRs) are a set of rules defined in the IP-XACT standard that IP-XACT documents should conform to in addition to the IP-XACT schema.

Rule number	Rule description	Single document check
SCR 1.1	Every IP-XACT document visible to a tool shall have a unique VLNV.	No
SCR 2.4	An interconnection element shall only connect a master interface to a slave interface or a mirroredmaster interface.	No
SCR 5.7	configurableElement elements within componentInstance elements shall only reference configurable elements that exist in the component referenced by the enclosing componentInstance element; the value of the referenceId attribute of the configurableElement element shall match the value of the id attribute of some configurable element of the component.	No
SCR 6.26	A wire port with a direction of out shall not have adriver element.	Yes
SCR 8.1	The width of an address block included in a memory map shall be a multiple of the memory map's addressUnitBits.	Yes

Figure 6.4 Example *Semantic Consistency Rules* from IP-XACT standard [29]**. They need to be implemented in IP-XACT synthesis.**

They define the required *consistency among the IP-XACT elements* in one document or across several documents.

There are a total of 185 such rules listed in IP-XACT [29] Annex B. They will all be examined by the *Semantic Consistency Rules Checker*, when they are concerned. The table in **Figure 6.4** gives several examples of these rules.

For example with SCR 2.4, for each bus interconnection, our checker must retrieve first both components and then both bus interfaces that are referenced by the interconnection and check their types in the scope of those seven possibilities. Here *single document check* means that the elements consistency cannot be determined in a single file but only by checking multiple documents. For example, the uniqueness of a VLNV required under SCR 1.1 can be only claimed after seeing all the documents that we maintain.

TLM Compatibility Rule Checker

We propose a TLM compatibility rule to ensure the semantically correct integration of TLM components. This rule specifies how SystemC TLM ports should be described in IP-

XACT, such that our SoC synthesizer can unambiguously, automatically determine whether and how two TLM components can be connected.

This is necessary as IP-XACT does *not* provide enough specification on TLM port semantics. We do assume that traditional RTL compatibility between signals is well resolved by IP-XACT standard.

SystemC TLM semantics is established on an interface-port binding mechanism, as we have discussed in the background of TLM. Based on this, we notice that that SystemC interface classes for TLM communication can be considered as a non-private inheritance tree starting from *sc_interface*.

For IP-XACT based description and integration of TLM ports, our *TLM compatibility rule* states:

- If an IP-XACT transactional port describes a TLM port that implements a SystemC interface to *provide* for binding, its IP-XACT service types description should include names of *all the inherited* interfaces, or interface implementation classes, from this interface, besides the name of itself.
- If an IP-XACT transactional port describes a TLM port that *requires* a SystemC interface for binding, its service types should include name the interface that it expects.
- The compatibility of two IP-XACT transactional ports are determined by seeing *whether the provided interface names contain the required interface name*.

Figure 6.5 shows one such example.

- The IP-XACT description for *TLM_port_1* should include *TLM_IF_1*, *TLM_IF_2*, *TLM_IF_3*, *TLM_IF_4*, and *TLM_IF_6*. It is indeed capable of providing all these communication services.
- The IP-XACT description for *TLM_port_2* is only required to include *TLM_IF_2*.
- Then, the compatibility of *TLM_port_1* and *TLM_port_2* during IP-XACT based system integration can be directly decided as positive.

For SystemC code generation later, two TLM ports can be bound safely by casting the type of the providing port to that of the requiring port, after checking their TLM compatibility.

SystemC Code Generator

After all the parsing and consistency/compatibility checking procedures, the mapping from IP-XACT to SystemC is relatively straightforward, as IP-XACT design has a concise

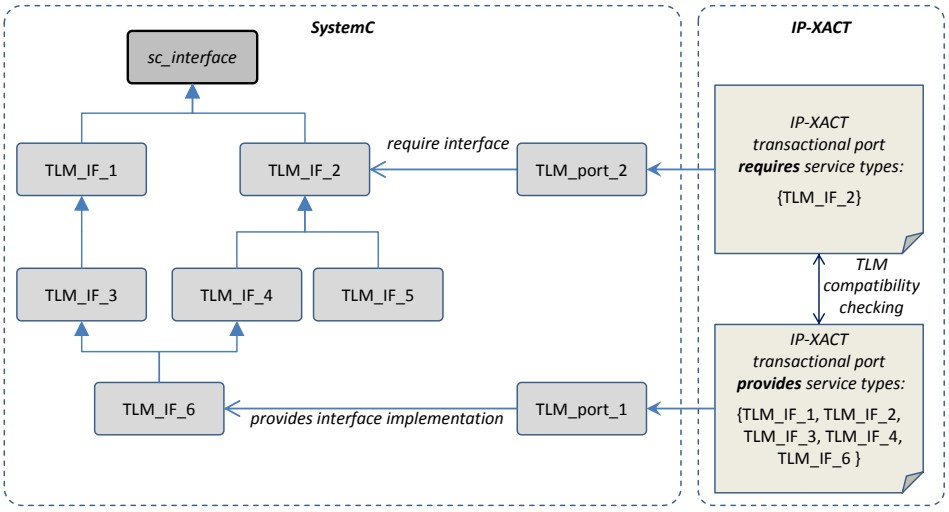


Figure 6.5 Example for TLM compatibility rule.

structure for instantiation, parameterization, and interconnection of components, which correspond directly SystemC.

- A component is instantiated as a SystemC module object. It is required that the referenced *component* has a *name* reflecting its actual module class name. The *instanceName* is used as the name of the object as well as the SystemC module name string.
- We require the component to have a uniform parameterization interface as *setParameter*, which expects the *name* of the parameter and its *configurableElementValue* in design.
- With *interconnection*, ports of two components are connected via bus interfaces. We can conclude two essential cases for TLM port binding, as we have discussed in TLM introduction: a TLM *sc_port* to *sc_export* binding or a TLM *sc_port* to TLM module direct binding. The case is selected by seeing whether the *require* port is a *sc_export* type in its IP-XACT description. More importantly, the compatibility between the two TLM ports is determined by the TLM compatibility rule beforehand.
- For an *adHocConnection*, two ports are connected directed for a particular purpose, such as a reset signal. We consider such RTL signal binding straightforward if not trivial.

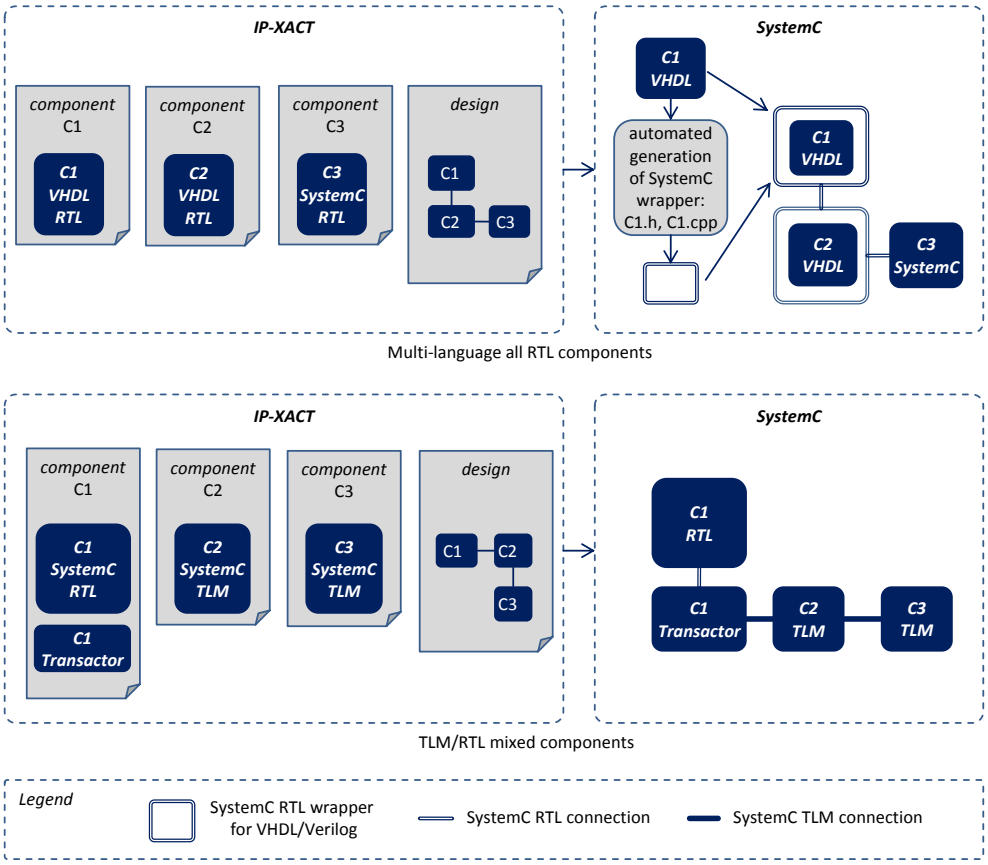


Figure 6.6 SystemC enabled multi-language, mixed-level IP-XACT simulation.

More interestedly, we consider how multi-language, mixed-level integration of IP components can be enabled in SystemC generation, as shown in **Figure 6.6**:

- **Multi-language all RTL components**. Firstly, a SystemC wrapper can generated for each component that is not in SystemC. Since at its core SystemC has an event-based simulation engine and it provides comprehensive hardware specific data types, a SystemC wrapper for a VHDL/Verilog module is straightforward. Modern simulators, for example *ModelSim* [95], even integrate such automated SystemC wrapper generation function for other HDLs.
- **All TLM components**. We need only a SystemC top design to instantiate and bind them in.
- **TLM-RTL mixed components**. In this case, we further assume that for a RTL component to be integrated, no matter in which HDL, it is packaged with an accompanying RTL-TLM transactor, which we have introduced in SystemC and TLM background. We view it as a natural assumption, since if a designer wants

to do a TLM based system design integration, the components to be integrated should necessarily expose TLM interfaces.

Further, as stated for the parser, this SystemC generation does *not* impose any restriction on the internal data representation, though in our implementation Java classes are used.

Makefile Generator

The objective of generating a *Makefile* is to have a *fully automated* compilation and simulation process. Together with SystemC code generation, we are then able to launch immediately a system simulation with an IP-XACT design as input, if the design is correctly integrated. As we have stated, this automated process not only becomes itself a simulation based verification tool for SoC system-level design but also satisfies prerequisite for mutation analysis.

Until now we have *not* designated any SystemC simulator as the target environment of our SystemC based IP-XACT synthesis and simulation. However, this last step for a *Makefile* generation is meant to be bound to a specific SystemC simulator, since the *compilation and simulation commands need to be specific*.

We may have two candidates. Either we use the reference SystemC simulator that comes from the SystemC standard working group, or we take another commercial HDL simulation tool that implements the SystemC standard.

In this section, we assume *ModelSim* [95] as our destination simulator, because, on the one hand, it is one of the leading industrial simulation tool and, on the other and more significantly, it is capable of co-simulating all the other major HDLs with SystemC, such as VHDL, Verilog, and SystemVerilog.

Such *compilation rules* are illustrated in **Figure 6.7**. They define how the compilation related information, mainly the *fileSets* description, from the design-referenced IP components can be combined to valid ModelSim commands in a Makefile script. The commands are composed according to the type of each *file* declared in IP-XACT *fileSets* and for the generated SystemC design file:

- For each *systemCSource* typed file, a compilation command is created using *sccom* and takes into account all the include files declared for this component. The other SystemC typed files – *systemCSource-2.0*, *systemCSource-2.0.1*, *systemCSource-2.1*, and *systemCSource-2.2* – are treated the same way.
- For each *vhdlSource* typed file, a *vcom* compilation command is created. Other files typed as HDL source files receive the same handling, including *vhdlSource-87*, *vhdlSource-93*, *VerilogSource*, *systemVerilogSource*, etc.

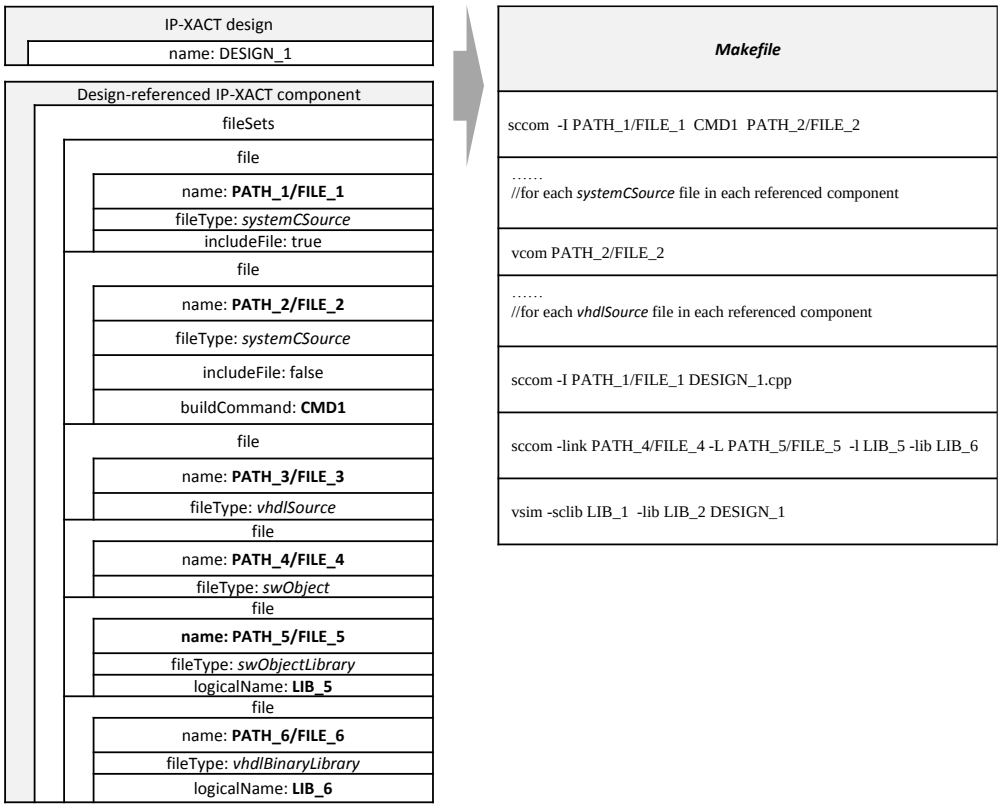


Figure 6.7 Makefile generation that targets *ModelSim*.

- The generated top SystemC design file is then compiled by *sccom*.
- A linking command with all SystemC objects, SystemC libraries, and other HDL libraries is generated with *sccom -link*.
- A simulation command is added such that the Makefile becomes a complete script for compilation and simulation of the generated SystemC design.

As mentioned, with advanced facilities from *ModelSim*, IP components in other HDLs can be easily wrapped in SystemC. Such wrapper source files should be similarly compiled and linked, using the compilation rules.

6.4. Mutation Operators on IP-XACT

We define a set of mutation operators on IP-XACT *design* schema. When applied on an IP-XACT *design*, each such operator introduces a small modification to that IP-XACT XML document. The result is another valid IP-XACT *design* document.

An example can be a perturbation to a parameter configuration, such as changing the design configured transmission rate of a UART component. This rate modification, as a

bug injection, is supposed to be discovered by the system verification, i.e. the SystemC simulation derived from the design mutant produces a different trace compared to that from the original design.

We take a *define-and-evaluate* approach to obtain a set of mutation operators. In this section, we first try to consider and formulate several possible and reasonable mutation operators for IP-XACT *design* schema. They must be valid, meaning that the modification to be introduced by an operator should *not* break the syntax and consistency/compatibility rules for IP-XACT.

Later in the evaluation chapter, the *effectiveness* of these mutation operators will be investigated with real SoC design examples, by whether they can somehow reveal the quality weakness of system simulation, i.e. any generated mutant can*not* be distinguished. This evaluation is also viewed as a *selection* process to sieve out ineffective operators for IP-XACT mutation analysis.

Our first effort to define such a set of mutation operators for IP-XACT, as listed in **Figure 6.8**. They are explained in three groups:

- **Parameter modification operators:** The mutation operators in this class perturb a parameter configuration. The parameters can be, for example, the model generics of a UART component, the type of an Ethernet controller, the address/data-width of a bus, or its arbitration scheme. Mutation of these parameters introduces small errors into the system integration and may result in erroneous data flows among components.

  The first operator *ParRep* uses another valid value, for example a pre-defined *choice* in IP-XACT, to replace an existing parameter configuration. Operator *ParIns* inserts into the design a configuration for some parameter. The replacement or insertion value can be chosen randomly. A third operator called *ParDel* deletes a configuration, so that the default value of this parameter takes into effect.

- **Connection deletion operators:** Designers can omit some connections between components. The mutation operators in this class model such errors and delete completely a connection description. In IP-XACT design, we have two kinds of component interconnections. One is the connections through pre-defined bus interfaces and another one is ad-hoc connections, i.e. not through any bus protocol. Operator *BusDel* operates on the former and *AdhocDel* operates on the latter.

- **Memory-maps modification operators:** This class of operators introduces deviations on the address spaces of slave components from their original configurations, which makes the testing software have a wrong view of the hardware system. With erroneous interaction between software and hardware, it

Mutation Operator Name	Description	Example
ParRep	Replace a parameter configuration with another valid value	*ABus_width = 64 ←128*
ParIns	Insert a parameter configuration with a valid value	*↓ DBus_width = 128*
ParDel	Delete a configuration	~~*Arbitration_policy = priority*~~
InterConnDel	Delete a bus interconnection	~~*Bus_Interconnection: Comp_1 - Bus_1*~~
AdhocConnDel	Delete an ad-hoc connection	~~*AdHoc_Connection: Comp_1 - Comp_2*~~
BAddrIncr	Increase the base address of a slave component	*Base_address= 0x10000 ←0x10040*
HAddrDecr	Lower the high address of a slave component	*High_address= 0x10000 ←0x0FFF0*
AddrExch	Exchange the address spaces of two memory-mapped slave components	*Component_1_Addr_block =0x00000~0x01FFF* ↑↓ *Component_2_Addr_block =0x08000~0x09FFF*

Figure 6.8 IP-XACT mutation operators.

may further lead to a wrong behavior of the system and a negative test verdict, if the testing software is comprehensive enough.

Operator *BAddrIncr* increases the base address of a slave component by a small value, with the caution that it should not exceed the upper address boundary of the component. Respectively, operator *HAddrDecr* decreases a slave high address to a level not less than the base address. Another *AddrExch* operator chooses two slave components and makes an exchange of their address spaces.

Another contribution from our side is an experimental implementation of IP-XACT mutant generation, based on such mutation operators.

6.5. A Tool Implementation

We have implemented an Eclipse-based tool for the whole proposal on SystemC based IP-XACT design synthesis, simulation, and mutation analysis. It provides also basic editing functionality of IP-XACT documents. This implementation, on the one hand, investigates the feasibility of our proposal and, on the other hand, provides the prerequisite to further experiment based evaluation.

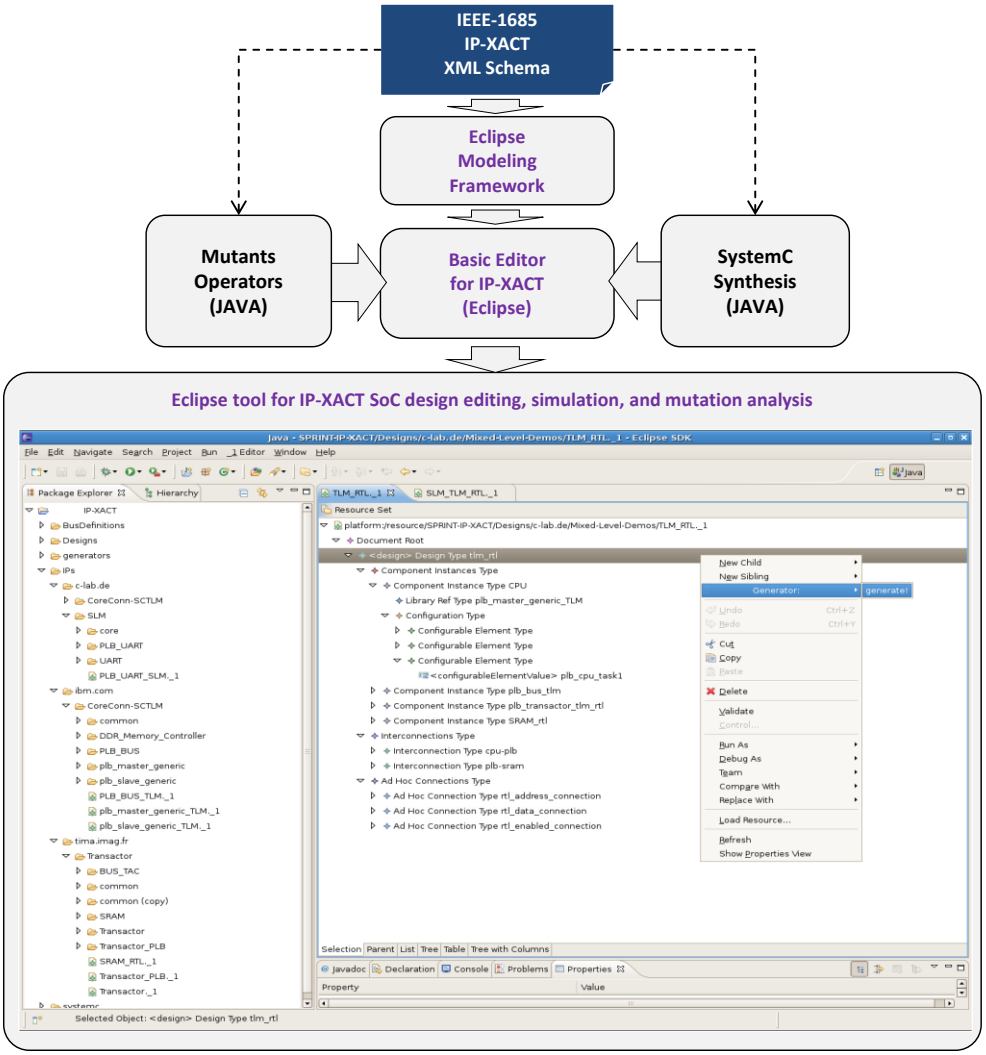


Figure 6.9 An Eclipse based tool implementation, for the proposal of IP-XACT system design synthesis, simulation, and mutation analysis.

Figure 6.9 shows an overview of how the tool was constructed, as well as a screen shot. First, we leveraged *Eclipse Modeling Framework* (EMF) [96] to obtain a basic IP-XACT editor. *EMF* is a Java framework that facilitates the building of Eclipse based modeling tools, by automatically generating a set of Java classes from a structured meta-model, such as XML Schema or UML, among others. The generation is based on a one-to-one mapping from the types and elements of the meta-model. The mapped Java classes are then able to create, parse, manipulate, and output documents that are instances of the meta-model. They are further integrated by EMF into Eclipse as a fully functioning Eclipse editor.

We made IP-XACT schema the input of EMF. The output was a basic editor for all kinds of IP-XACT documents. We used the standard schema version in IEEE-1685 [29].

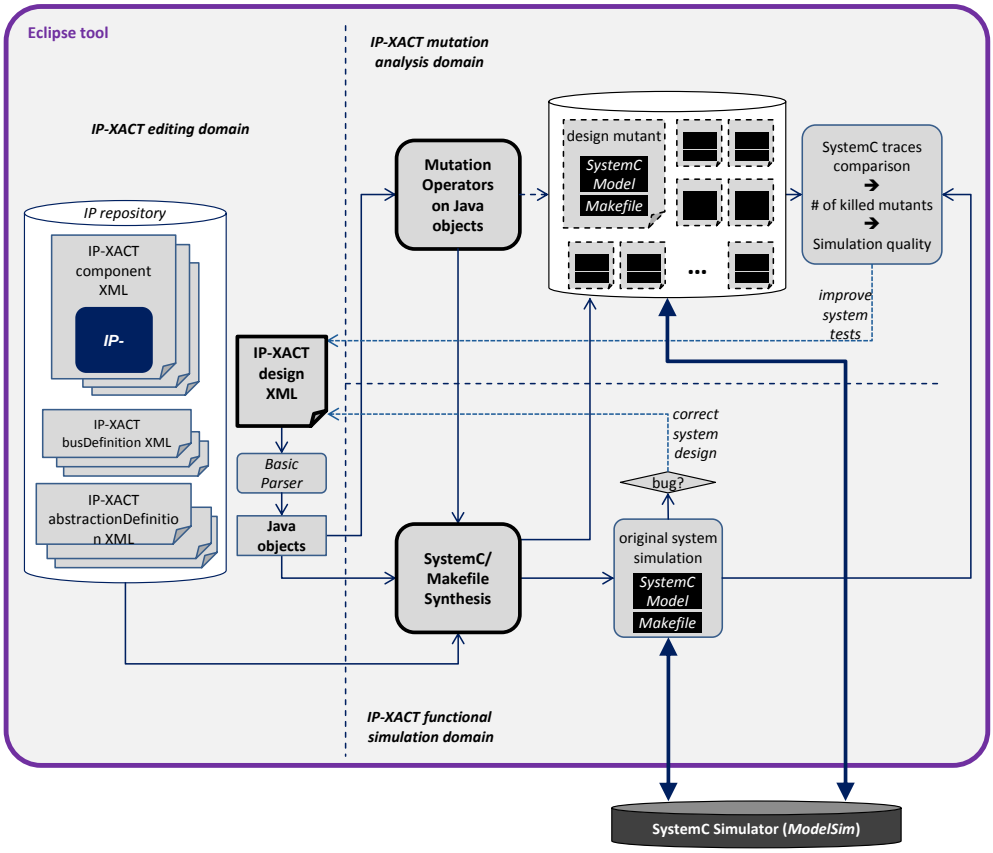


Figure 6.10 Detailed working flow of our IP-XACT tool.

Then, following the proposal and definitions from previous sections, both mutation operators and SystemC synthesis were coded into the basic editor, on top of the Java classes mapped from IP-XACT schema. When an IP-XACT design is read in by the editor, corresponding Java objects are created that reflects exactly the same structure as in the IP-XACT document. With these objects, the extended editor can, for example, change a parameter value according to mutation operator as well as write a code line of component instantiation to a SystemC file.

Figure 6.10 sketches the detailed working flow of this Eclipse tool at runtime. Essentially, it is implemented as an instance of the systematic system verification framework that has been presented in Section 6.2. The runtime functionality for IP-XACT can be seen as divided into three interacting domains: the elementary *editing domain*, the *mutation analysis* domain, and the *functional simulation domain*, which further rely on an external SystemC simulator – ModelSim in this case.

In the flow, it is only one implementation decision that system design mutants are created in the form of synthesized SystemC models, instead of fault injected IP-XACT documents. We chose this implementation way since, on the one hand, it is insignificant

for the mutation analysis to eliminate the intermediate step of having mutants as IP-XACT documents and, on the other hand, it saves the tool effort at runtime. Still, we view the IP-XACT mutation analysis remaining *not* restricted to SystemC but relatively independent, if we suppose another simulation engine for IP-XACT.

6.6. Related Work

We have presented a SystemC based IP-XACT design simulation flow and an IP-XACT mutation analysis layer upon this simulation. On the one hand, we can find the following literature that proposes other system simulation and verification methods based on IP-XACT:

- In [97], a small IP-XACT extension, called *IP-XACT++* is proposed to support Model Driven Engineering (MDE) in SoC design. The authors consider that in MDE, various abstraction levels as *meta-models* and the transformations between them should be clearly defined. In this work, a specific level called *Transaction Accurate* (TA) is focused. A TA meta-model is defined in XML schema that represents an extension to IP-XACT. In the schema, TA elements such as "*TAComponents*" are defined.

  Further, they mention that through the definition of a SystemC meta-model (not detailed) and the transformation between it and the TA meta-model by an *ATL* (ATLAS Transformation Language) transformation language, an extended IP-XACT-to-SystemC generation can be obtained, for this particular TA level.

- In [98], IP-XACT is combined with another computation model UNIVERCM (UNIversal VERsatile Computational Model) [99], to support system integration with not only digital IP components, but also analog IPs as well as hardware-dependent software. First, UNIVERCM is capable of generating homogeneous representation and simulation of heterogeneous components. Then, IP-XACT (extended) descriptions are extract from UNIVERCM components. Last, an IP-XACT system *design* can be built and, with the help of UNIVERCM, a system simulation model with all types of components can be generation and simulation.

  The benefit is that IP-XACT is now used as unified platform for all components and system description, with automated round-trip between UNIVERCM and IP-XACT. IP-XACT standard components can be directly integrated.

- In [100] [101], the authors try to integrate IP-XACT and also benefit from its capability of component description and integration, into a UML/MARTE [102] based design framework, called COMPLEX. In this context, IP-XACT is also

extended to be able to describe i) performance-related semantic information and ii) embedded software such as drivers as well as operating system.

The reason for such component extension is that the COMPLEX framework has an emphasis on performance evaluation. In the end, a specific performance model is generated from IP-XACT system design, to be simulated by a proprietary engine.

- [103] [104] are novel application of IP-XACT to partially reconfigurable system design with FPGA. UML/MARTE is similarly employed as the design frontend. IP-XACT is used to describe both static components and partially reconfigurable components. Interestedly, in the evaluation chapter, we will also present an IP-XACT tool experiment with reconfigurable system. We will have a focus to show the simulation capability of the tool.

On the other hand, we can find the following work that also proposes applying mutation analysis to other high-level languages, especially to SystemC/TLM, since they are widely employed in the research area of SoC system modeling.

- In [105], a SystemC error and mutation injection tool is presented. Four types of error injection are defined: OPR (Operator Replacement),VCR (VAR=>Constant Replacement), CCR (Constant Replacement), and ROR (Relational Operator Replacement). A unique feature of this tool is that, instead of creating source code mutants directly, the error injection is implemented as a plugin for the GCC compiler.
- In [106] [107], mutation analysis is also considered for SystemC. However, the author concentrated on the *concurrency* aspect of SystemC designs, for example, how to stir a deadlock situation by error injection. Such concurrency mutation operators include:
  - Modify Function Timeout, e.g. by changing *wait(time)* to *wait(time/2)*, or to *wait(time*2)*,
  - Modify Concurrency Construct Count, e.g. changing *sc_semaphore(num)* to *sc_semaphore(num-1)*, or to *sc_semaphore(num+1)*,
  - Remove Concurrency Construct: e.g. by removing a wait, or notify statement,

  and so on. The mutation operators are evaluated with several standard TLM examples.
- In [108], a mutation model is proposed specifically for TLM communication interface. First, primitives defined in SystemC TLM 2.0 are modeled by EFSMs

(Extended Finite State Machines). For example, a *nb_get(data)* is modeled as a state transition with a *true* trigger, meaning the transition will immediately happen when called, without waiting for any event.

Then, a total of 19 faults, or mutation operators are defined on these EFSM models for TLM 2.0 communication primitives. The 19 operators belong to three categories. The first is modification on destination states of an EFSM model, such that, for example, the misuse of a blocking/non-blocking communication is modeled. The second is modification on the transition triggering functions. The third is directly replacement of a TLM communication primitive with another one from the library. These operators are then evaluated with standard TLM 2.0 examples.

Comparing Literature to Our Work

Compared to the literature on SoC system simulation and metrics that are mentioned above, the work in this chapter has its own unique contribution, since:

- Our work is among the first to propose this systematic simulation of IP-XACT SoC designs by SystemC generation, incorporating both RTL and TLM. As SystemC and TLM prevail and become required elements for system modeling, the IP-XACT-to-SystemC generator serves a non-replaceable bridge between IP-based SoC assembly and its functional verification with the underlying IPs.
- We define a mutation analysis-based simulation metric directly on IP-XACT schema. This contrasts with other emerging metrics that are mostly built for SystemC. Assuming IP-XACT the starting language for SoC system integration, and following the principle of mutation analysis that errors should be modeled on language syntax, IP-XACT mutation analysis should make a unique, effective quality metric for SoC design verification.

As we do not have the availability of other SystemC related tools mentioned in the literature, a direct comparison of the metrics have not been conducted in our evaluation.

6.7. Summary

We have considered the problem of providing a systematic verification method for SoC system design. In particular, we assume IP-XACT the target design language to be used.

Also assuming *simulation* a necessary and significant step for any systematic system verification, we have considered the problem of enabling simulation for IP-XACT designs. For this, we have proposed an *IP-XACT-to-SystemC synthesis* flow, by a set of semantics, compatibility, and mapping rules. With an IP-XACT XML design as input, the flow is able

to generate a SystemC model that is directly simulatable. A *Makefile* is generated by this flow too, which incorporates compilation of the generated system model and all the involved IP components, so as to provide a fully automated process from IP-XACT to simulation.

SystemC is selected as the synthesis target, since it provides the *only* platform for RTL/TLM, VHDL/Verilog/SystemC multi-language, mixed-level simulation. And we view such inclusion of TLM and SystemC a necessity.

Further, following our consistent employment of mutation analysis driven verification for IP-based SoC design, we have considered the problem of enabling mutation analysis on IP-XACT. Based on the principle of mutation analysis, we have defined a set of mutation operators on IP-XACT XML schema, as representative error that can be made. The derived *IP-XACT mutation analysis* interacts with the SystemC based simulation and lays a quality metric layer upon this simulation.

Together, they form an integrated framework that enables a systematic verification for SoC system designs with IP-XACT.

As an experimental implementation of this framework, an Eclipsed-base prototype tool has also been presented. The tool is a prerequisite for further experiment-based evaluation of our proposals.

In the evaluation chapter, by exercising the tool with several real SoC designs, we will mainly investigate i) the feasibility of the SystemC-based IP-XACT synthesis and simulation and ii) the effectiveness of the defined IP-XACT mutation operators.

The contribution in this chapter has been summarized in [5], with the SystemC-based IP-XACT synthesis and simulation further presented in several other occasions: [12] [10] [9] and [2].

CHAPTER 7: Evaluation

This chapter provides an experimental evaluation of the proposed methods from Chapter 4 to Chapter 6, based on real IP and SoC designs.

7.1. Objectives

First, by **Figure 7.1** we give an overview of the evaluation. The methods and flows from the previous chapters were applied to what we see as an instance of IP based SoC design. The three arrows of application reflect the following main evaluation objectives that we have identified:

- **Evaluation objective 1:** To validate that the constrained Marko chain-based, feedback-directed adaptive random simulation from Chapter 4 is *able to improve the efficiency of mutation analysis.* The *efficiency* should be measured as the number of tests required to kill a certain number of mutant. It should be compared with random simulation without feedback adaption.
- **Evaluation objective 2**: To validate that the CDFG-based cost function defined in Chapter 5 is able to *serve as an effective search directive*, so that it consistently steers a metaheuristic search to some target mutant-killing test. The success rate and performance of such a metaheuristic search will be measured on those difficult mutants that are left un-killed in random simulation, as the search is meant to succeed the random simulation phase.
- **Evaluation objective 3**: To validate the *general feasibility* of the concepts on SystemC based synthesis, simulation, and mutation analysis of IP-XACT SoC designs. The concept validation should be based on our prototype tool implementation in Eclipse. Further, as a secondary goal, the effects of the defined IP-XACT mutation operators should be investigated – i.e. how the mutants are generated and killed under these operators.

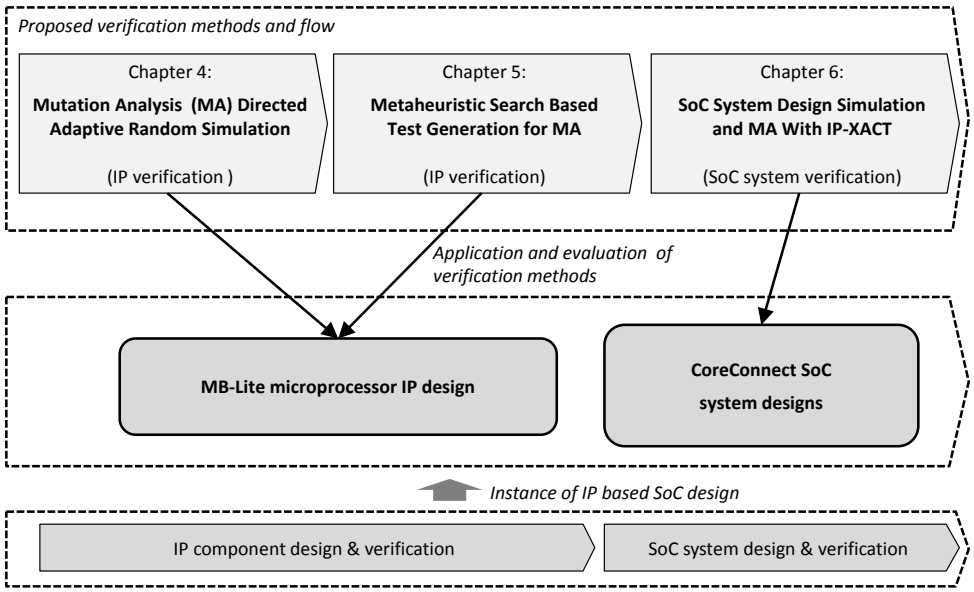


Figure 7.1 Objectives of evaluation.

As **Figure 7.1** shows, for IP level, we took a microprocessor IP as the design under verification, which is called *MB-Lite* [109] implementing the MicroBlaze ISA from Xilinx. It served evaluating objective 1 and 2, by exercising the first two components of our methodology – the adaptive random simulation and metaheuristic-based test generation. For system level, we exercised the IP-XACT tool with several designs based on *CoreConnect* SoC architecture. Here, the evaluation objective 3 was the target. The experiments further comprise an integrated evaluation of the mutation-analysis-driven verification methodology.

For the selection of these study objects, we took two aspects into account. First, we intended to evaluate the methods on real working designs. The microprocessor core and its associated FPU are both synthesizable and able to execute standard-specified instructions. The SoC system designs host software, too. Second, we considered that MicroBlaze microprocessors, FPU, and CoreConnect are all popular employment in SoC research [110] [111] [112] [113].

As mentioned, Certitude from Synopsys, as a state-of-the-art EDA tool for HDL mutation analysis, was used for IP level mutation analysis.

7.2. MB-Lite Microprocessor IP Verification

This section presents the microprocessor IP verification that goes through the proposed *adaptive random simulation* and *metaheuristic search-based test generation*, targeting the mutation analysis metric provided by Certitude.

7.2.1. Design Under Verification and Mutants

Microprocessor is considered an essential component in most SoCs and *MicroBlaze* is a popular ISA from Xilinx. Various IPs that implement this architecture have been used in literature for SoC and embedded systems research [110] [111] [112] [113]. The specification of MicroBlaze ISA can be found in [114].

MB-Lite is a VHDL IP core that implements MicroBlaze ISA. It has been first presented at *Design Automation and Test in Europe* 2010 [109], Further, there is an open source description to be found at [115], for others to review the verification.

Nevertheless, it lacks the support for floating point instructions. Therefore, we extended this *MB-Lite* by integrating into it another IEEE-754 compatible floating point unit (FPU) – IEEE-754 [49] is the specified format by MicroBlaze ISA.

Figure 7.2 shows the outlined microarchitecture for the *MB-Lite IP design with FPU.* The main microprocessor consists of a five stage pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write-back (WB). The FPU supports pipelined as well as *non*-pipelined operations.

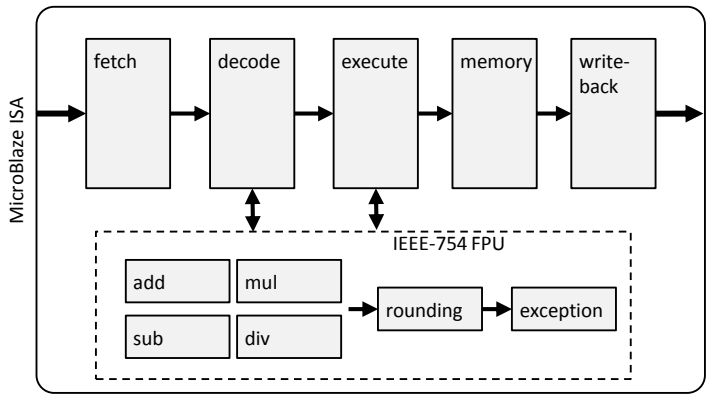


Figure 7.2 Design Under Verification: MB-LITE microprocessor design with FPU.

By such, the IP core is ready to execute binary code compiled by the standard MicroBlaze compiler *mb-gcc*, included in XILINX FPGA tools. All together, the IP has about 4K lines of code.

This is a *near-mature* IP design. Again, the goal of the experiments is *not finding any real bug* in the design, but to show the efficiency and effectiveness of the simulation methods with regard to mutation analysis metric.

Figure 7.3 lists a summarized report from the Certitude tool, after it creating the initial mutant database as verification quality metric. In total, ***1662*** valid mutants were generated, scattered on all the VHDL files. Another 85 mutants were generated but then

File name	Mutants Total Valid	Disabled By Certitude (Equivalent)	Mutants Total (incl. Equivalent)	Killed	Non-Killed
[mblite]/core/std_Pkg.vhd	167	2	169	0	167
[mblite]/core/decode.vhd	445	37	482	0	445
[mblite]/core/execute.vhd	216	0	216	0	216
[mblite]/core/fetch.vhd	31	2	33	0	31
[mblite]/core/mem.vhd	47	2	49	0	47
[mblite]/core/core_Pkg.vhd	45	0	45	0	45
[mblite]/FPU/fpupack.vhd	12	5	17	0	12
[mblite]/FPU/fpu_add.vhd	52	1	53	0	52
[mblite]/FPU/fpu_div.vhd	113	0	113	0	113
[mblite]/FPU/fpu_mul.vhd	84	0	84	0	84
[mblite]/FPU/fpu_sub.vhd	65	2	67	0	65
[mblite]/FPU/fpu_round.vhd	40	2	42	0	40
[mblite]/FPU/fpu_exception.vhd	209	14	223	0	209
[mblite]/FPU/fpu.vhd	136	18	154	0	136
All Source Files (6)	**1662**	85	1747	0	**1662**

Figure 7.3 Initially generated mutants (report sumary from Ceritude).

identified as *equivalent* mutants by Certitude. We will have a short discussion on both equivalent and non-equivalent mutants that could not be killed at the end of verification.

7.2.2. Adaptive Random Simulation

For the implementation of the adaptive random simulation:

- We modeled the Markov chain and constraints for random test generation with the *SystemC Verification Library (SCV)*. MicroBlaze instructions [114] as well as the contained IEEE-754 FPU operations are modeled with 12 Markov-chain nodes and 17 constraints. Similar instructions are not distinguished and grouped into one node, such as *add*, *addc*, *addk* and *addkc*. Example constraints have been previously discussed. SystemC and VHDL co-simulation is supported by the simulation tool *ModelSim*.
- We realized the *dynamic mutation schemata* by utilizing the ***Tcl*** interfaces of the tools Certitude and ModelSim.
- We also implemented the adaptation heuristic in ***Tcl***, both the calculation and the adjustment to the SCV model. At initialization, all the edges and constraints are assigned equal probabilities/weights for being selected.

To investigate the efficiency of our method – *evaluation objective 1*, we compared three simulation processes: i) the adaptive random simulation, ii) a random simulation process with test generation under the same Markov chain model, but *without* the in-loop

adaptation heuristic, and iii) the *dhrystone* benchmark as a software program that is compiled with the Xilinx compiler *mb-gcc* for MicroBlaze ISA, with another 150 directed FPU tests planned in.

Figure 7.4-a) shows as the main result this efficiency comparison: the total number of killed mutants until a certain number of tests being simulated.

The adaptive random simulation managed to kill 1579 (95.0%) out of the total 1662 mutants after 1000K tests (MicroBlaze instructions). This compares to the *non*-adaptive version that was only able to kill 1308 (78.7%) with this amount of tests. Both random simulations were repeated three times to obtain these average values, each time with a different random seed.

We see this as the first evidence that the adaptation heuristic, based on mutation analysis feedback, is indeed able to improve the efficiency of a HDL mutation analysis process.

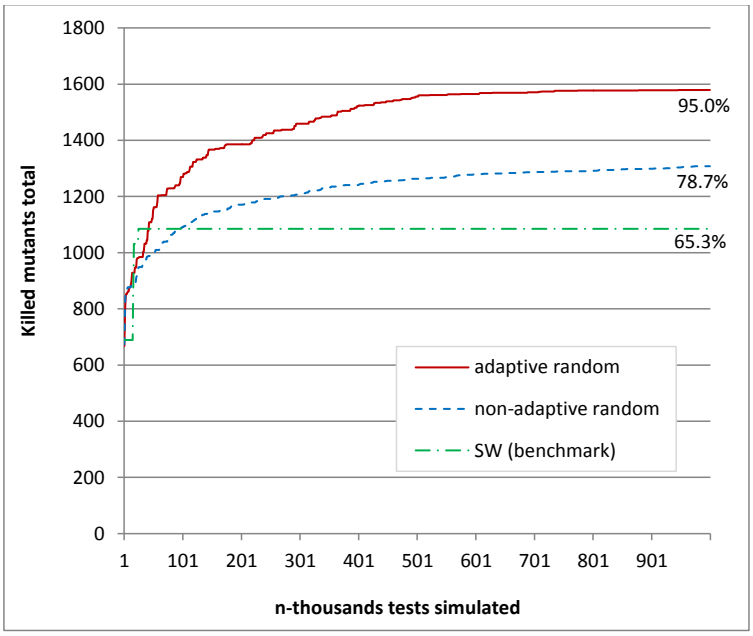
The bottom part of **Figure 7.4** just provides another view of the result data. The motivation is from an easy observation that there is a certain set of mutants that were trivially easy to be killed. In fact, around 800 mutants – about half of the total – could be eliminated by the first thousands tests, in all simulation experiments. Therefore, to limit the impact of these trivial mutants and amplify the significance of those non-trivial mutants, we devised a *quality index* (QI) as an adjusted result of mutation analysis, simply by

$$Quality\ Index = (N_{killed-mutants}/N_{total-mutants})^2 \times 100$$

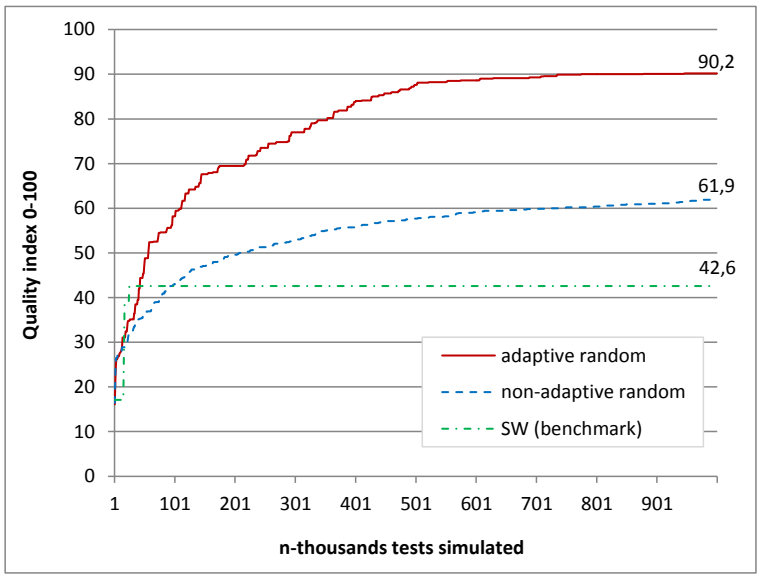
By this, we are able to highlight the progress of killing hard-to-kill mutants. These are exactly the *verification holes* that we need to cover. The improvement by our method is made more prominent.

The software binary was outperformed by both random simulations. The *dhrystone* benchmark program together with the planed FPU tests was only able to kill 1085 mutants, or 65.3% of the total. After an initial period, it delivered only waste of cycles without increasing the killed mutants any more, since it was a benchmark and *not* built for exercising this specific design. It was inferior to the continuous progress in random simulations. We used it merely as a reference, though it has some competence by exploiting the knowledge from the compiler.

Figure 7.5 is an attempt to explain the efficiency improvement from the adaptive random simulation, compared to the non-adaptive one. It shows the record on the *number of activated and killed mutants by each thousand test* – in one experiment from the three repetitions. We can see that as the remaining, un-killed mutants decreased, the adaptive test generation managed to maintain a relative high rate of activation, by adjusting the Markov chain model. In contrast, the non-adaptive simulation lost the percentage of



a) Efficiency as number of killed mutants



b) Efficiency as adjusted *quality index*

Figure 7.4 Mutation analysis efficiency compared (average from 3 repetitions, each with a different random seed).

activated mutants a lower level, when the initial easy-to-kill mutants were removed from the metric and it was *not* able to adjust itself to this change.

In average, the adaptive random simulation needed about 12.5 hours to finish the 1000K tests and the non-adaptive random took 7.4. Indeed, more mutant activation will lead to increase of HDL simulation time. However, this increase is limited thanks to the

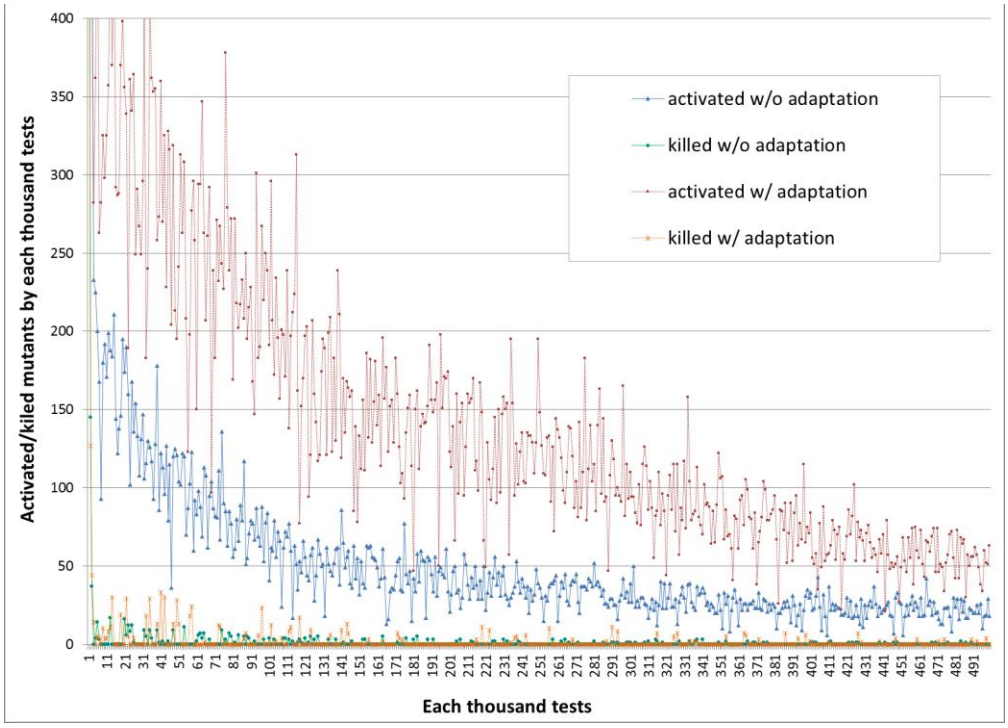


Figure 7.5 Explanation to the efficiency improvement. Adaptive rando simulation saw more activated/killed mutants by *each* thounsand tests.

use of dynamic mutation schemata with Certitude, since activated mutants require only temporarily forked simulation.

Therefore, even considering simulation time for efficiency instead of number-of-test, **Figure 7.6** shows the advantage from adaptive random simulation. Within the same period of 10 hours simulation, the adaptive simulation reached a quality index of 89.6 compared

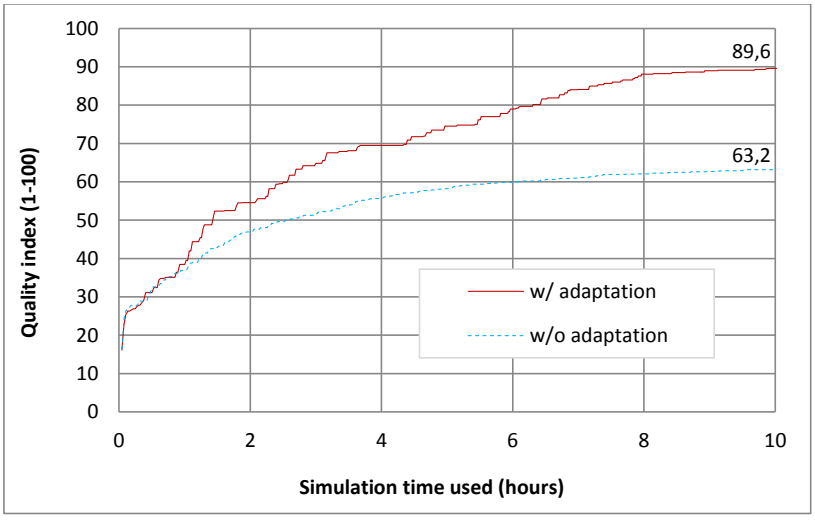


Figure 7.6 Mutation analysis efficiency in simulation time used.

to 63.2 from non-adaptive. The simulations time was measured on a Linux PC with 2.4 GHz processor.

As mentioned, the transition point – until which time the random simulation should be stopped and we move to the search based test generation for each un-killed mutant – was *not* optimized.

7.2.3. Metaheuristic Search-based Test Generation

As the starting point for experimenting the metaheuristic search based test generation, the table in **Figure 7.7** gives a summarized report from Certitude after one adaptive random simulation process (one of the three repetitions). There were 83 mutants that could *not* be killed by the preceding random simulation, which becomes exactly the objects of our experiments in this section.

Basically, we implemented the example *local search* presented in Section 5.2, integrating a CDFG-based cost function following the principle of Section 5.3. Some more implementation details:

- We extracted the control and data flow graph manually from the design VHDL code, which contains five main microprocessor pipeline stages and another 6 FPU data flow units. Local cost functions are also manually programmed and attached to the CDFG structure.
- Some input fields are considered type integer for neighborhood selection, for example the exponent field of a FPU operand. Recall that for an integer input, we have two neighborhood candidates, one by increasing and another by decreasing *half* from its current value. Others are treated as simple bit or bit-vector.
- In each search iteration, we simulated a test sequence with MicroBlaze 100 instructions. The neighbor candidates were limited to 100. It means a simulation effort of 10,000 instructions in each iteration.
- We allow moving to a non-improving (but best-in-the-iteration) neighbor in case of local optimum. Each search experiment was terminated after 200 local search iterations.

To investigate the effectiveness of the CDFG-based cost function – *evaluation objection 2*, we compared two search processes: i) the local search steered by CDFG cost function and ii) the same local search but only with a dummy cost function that always delivers the same value.

Figure 7.8 shows the results after applying the local search implementation on *each* of the 83 remaining hard-to-kill mutants. The top part shows separate experiments on each

File name	Mutants Total Valid	Disabled By Certitude (Equivalent)	Mutants Total (incl. Equivalent)	Killed	Non-Killed
[mblite]/core/std_Pkg.vhd	167	2	169	156	11
[mblite]/core/decode.vhd	445	37	482	421	24
[mblite]/core/execute.vhd	216	0	216	210	6
[mblite]/core/fetch.vhd	31	2	33	31	0
[mblite]/core/mem.vhd	47	2	49	47	0
[mblite]/core/core_Pkg.vhd	45	0	45	42	3
[mblite]/FPU/fpupack.vhd	12	5	17	12	0
[mblite]/FPU/fpu_add.vhd	52	1	53	46	6
[mblite]/FPU/fpu_div.vhd	113	0	113	110	3
[mblite]/FPU/fpu_mul.vhd	84	0	84	77	7
[mblite]/FPU/fpu_sub.vhd	65	2	67	63	2
[mblite]/FPU/fpu_round.vhd	40	2	42	36	4
[mblite]/FPU/fpu_exception.vhd	209	14	223	192	17
[mblite]/FPU/fpu.vhd	136	18	154	136	0
All Source Files (6)	**1662**	85	1747	1579	**83**

Figure 7.7 Un-killed mutants after adaptive random simulation (report sumary from Ceritude). *Each* became the target of a search experiment .

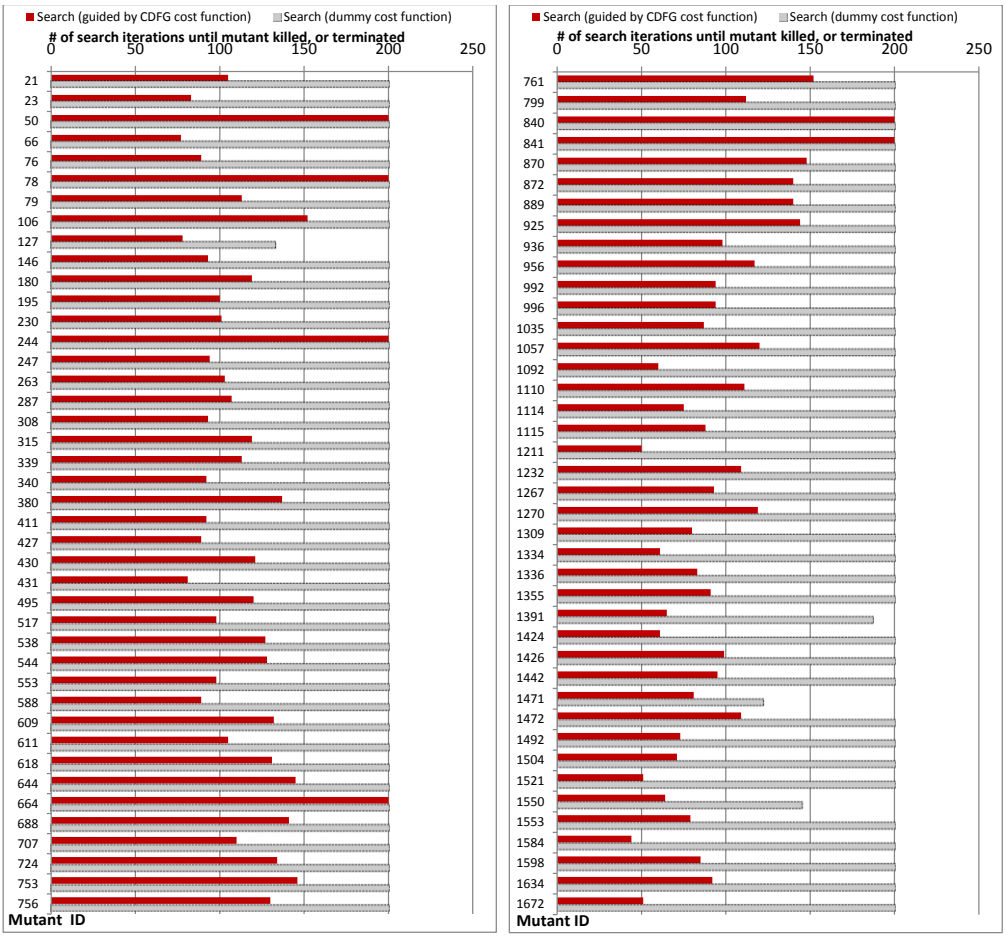
mutant, with search performance by the required iterations until a success, or a fail after the maximally allowed 200 iterations. The bottom part provides a summary of the search results.

We can observe that *in most cases* – 77 search instances, the search steered by our CDFG-based cost function was able to reach a target mutant-killing test, before the maximally allowed number of iteration. In average, it required 108 iterations until the target was found. This effectiveness of steering becomes obvious, when it is compared to the performance of the dummy function, which only succeeded in 4 cases by chance.

There were indeed failed cases, but only a few. It has *not* been further investigated whether these in-the-end un-killed mutants are actually equivalent mutants, or just tricky enough to avoid all our effort.

Overhead of the local search, mainly from calculating the cost function, was measured to be always minor in comparison to the time of design simulation itself, which conforms to our previous analysis in the arithmetic summary. Note that although a large number of neighbor tests must be examined in the search, they was no wasted time, since this simulation-based examination is a direct part of the verification.

Further, a specific search instance is discussed in the following, to provide a closer observation on the search steering under the CDFG cost function.



Search applied	Mutants left un-killed after adaptive random simulation	Search instances that succeeded /failed-after-200-iterations	Average iterations until a success
Local search guided by CDFG cost function	83	77 /6	108
Local search with dummy cost function	83	4 /79	N/A

Figure 7.8 **Performance of local search**. CDFG based cost function compared with a dummy cost function, to demonstrate the steering effectiveness.

Example Search Instance with Mutant-76

We discuss one example search instance, with the mutant that has an ID 76. The purpose is to provide a closer observation on how the cost function was able to steer a local search towards a target test.

Figure 7.9 first shows the mutant. It is created by Certitude at the FPU *add* unit, as changing the VHDL signal assignment at line 148 from one to zero. A small portion of the CDFG, which contains those variable and statement nodes that are close to the mutation

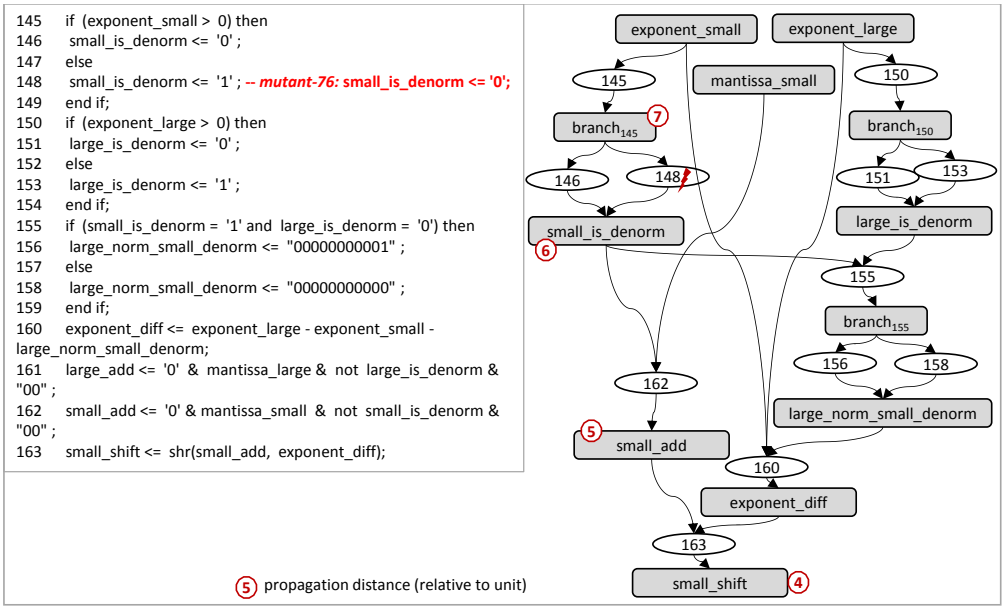


Figure 7.9 Mutant with ID 76 and a portion of design CDFG.

statement, is also shown in the figure. Propagation distances of those nodes relevant to the discussion are also annotated.

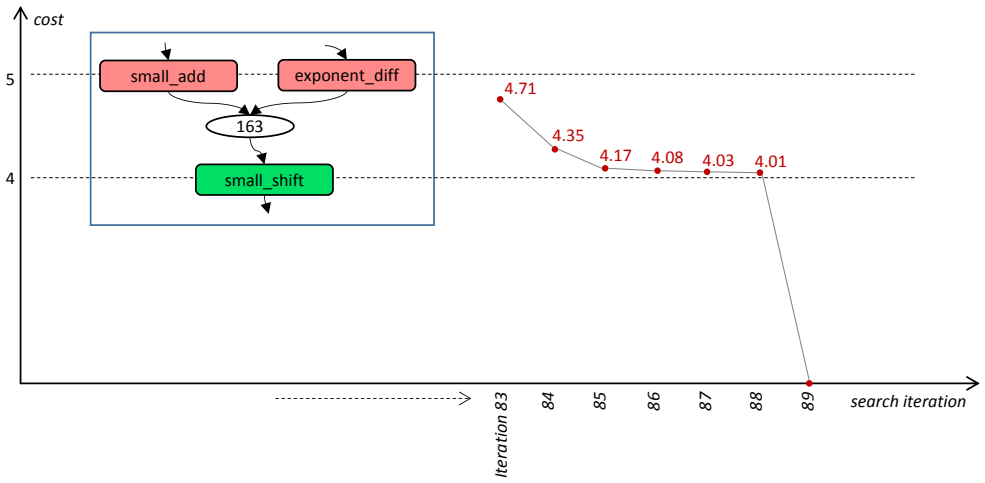
Figure 7.10 draws the reappearance of a search instance with mutant-76. In particular, it details the iterations that are executed just before a target test was found, by listing the cost function calculation at each step.

In iteration *83* in the search, we found a good test that was able to activate mutant-76 and propagate deviation in mutant simulation to as far as node *small_add* and *exponent_diff* (marked as red), but did not manage to propagate this to node *small_shift* through statement 163: *small_shift <= shr (small_add, exponent_diff)*. The local cost function that we attached to node *small_shift* when creating the CDFG is

$$localPropagationCost_{small\_shift} = left\_most\_one(small\_add\ xor\ small\_add') - \max(exponent\_diff, exponent\_diff')$$

where *left_most_one* returns the index of the left most bit that is *'1'*. Recall that such a local cost function is defined to exactly reflex the condition that a mutation deviation can be generated at this node. Also note that a local cost should be normalized to a value between (0, 1) and then added to the propagation distance. By such, the cost was calculated as 4.71 under this test, which is also the best for this iteration.

In iteration *84*, we could find another better test by decreasing test input *exponent_small* – recall that in our local search, we adjust a single test input field to get a neighbor test. The improved cost was 4.35.



Iteration 83: best test found was
operation: 00, *rounding mode*: 10,
operand 1: 0 00001001001(73) 00100000000000000000000000000000000000000000000000000010 (*sign1, exponent1, mantissa1*),
operand 2: 0 00000000000 00000000000000000000000000000000000000000000000000000000 (sign2, exponent2, mantissa2)
➔propagation_distance = 5, local_cost=71, cost= 5-1+0.71=4.71

Iteration 84:
neighbor test 1 : only increase *exponent1* to 00001101101 (109 as integer, increased by 36 from 73)
➔propagation_distance = 5, local_cost=72, cost= 5-1+0.72=4.72 ➔ worsened, wrong search direction ×
neighbor test 2 : only decrease *exponent1* to 00000100101 (37 as integer, decreased by 36 from 73)
➔propagation_distance = 5, local_cost=35, cost= 5-1+0.35=4.35 ➔ improved from 4.71, use for next iteration √
(other neighbor tests also led to worsened cost)

Iteration 85:
neighbor test 1 : *exponent1* =00000110111 (55) ➔propagation_distance = 5, local_cost=53, cost= 5-1+0.53=4.53 ➔worsened ×
neighbor test 2 : *exponent1* =00000010011 (19) ➔propagation_distance = 5, local_cost=17, cost= 5-1+0.17=4.17 ➔improved √
(other neighbor tests also led to worsened cost)

Iteration 86:
neighbor test 1 : *exponent1* =00000011100 (28) ➔propagation_distance = 5, local_cost=26, cost= 5-1+0.26=4.26 ➔worsened ×
neighbor test 2 : *exponent1* =00000001010 (10) ➔propagation_distance = 5, local_cost=8, cost= 5-1+0.08=4.08 ➔improved √
(other neighbor tests also led to worsened cost)

Iteration 87:
neighbor test 1 : *exponent1* =00000001111 (15) ➔propagation_distance = 5, local_cost=13, cost= 5-1+0.13=4.13 ➔worsened ×
neighbor test 2 : *exponent1* =00000000101 (5) ➔propagation_distance = 5, local_cost=3, cost= 5-1+0.03=4.03 ➔improved √
(other neighbor tests also led to worsened cost)

Iteration 88:
neighbor test 1 : *exponent1* =00000000111 (7) ➔propagation_distance = 5, local_cost=5, cost= 5-1+0.13=4.05 ➔ worsened ×
neighbor test 2 : *exponent1* =00000000011 (3) ➔propagation_distance = 5, local_cost=1, cost= 5-1+0.01=4.01 ➔ improved √
(other neighbor tests also led to worsened cost)

Iteration 89:
neighbor test 1 : *exponent1* =00000000100 (4) ➔propagation_distance = 5, local_cost=2, cost= 5-1+0.02=4.02 ➔ worsened ×
neighbor test 2 : *exponent1* =00000000011 (2) ➔local_cost=0, propagation_distance as well as cost was automatically reduced too – in fact also reduced to zero ➔ mutant killed ■

Figure 7.10 A search instance with mutant 76. *Local cost* guided the propagation through node 163 and, consequently, to a target test.

In the following iterations, from iteration *85* to *88*, the search could consecutively find cost-improving tests, which reduced the cost to 4.01. Thanks to that guidance from the local cost, the search could continuously move nearer to the full satisfaction of the propagation condition at statement node 163. Note that, without this guidance, the probability of *small_shift* receiving a mutant deviation is extremely low, which is why it could not be killed during the random simulation phase.

And at iteration *89*, we finally reached a test that reduced the local cost to zero, leading to a successful propagation at node 163. Therefore, the propagation distance should automatically be reduced, by one at least. In this instance, the mutant deviation created at *small_shift* was luckily able to propagate all the way directly to the FPU output and further microprocessor output, which made mutant-76 be killed by definition.

7.3. CoreConnect SoC Design Verification

We implemented several case studies to exercise our Eclipse-based IP-XACT tool and the concepts behind: IP-XACT based SoC system simulation and mutation analysis. It was our main objective to *demonstrate the general feasibility of these concepts*. A secondary objective was to *evaluate the effect of the IP-XACT mutation operators*.

The case studies were constructed using a TLM design library from IBM, which is provided for the TLM based modeling and evaluation of CoreConnect/PowerPC SoC. In the following, we will briefly introduce this library and, in particular, the relevant IP cores to be used in the case study designs, so that we can easily understand the design scenarios. Then, two case studies are detailed, one based on reference designs from the library, and the other one as a TLM based verification scenario for an existing FPGA design.

7.3.1. Introduction to PEK: A TLM IP Libaray for SoC Design

IBM provides this PEK [41]– *PowerPC Evaluation Kit* – as a library to facilitate the TLM enabled system-level modeling, exploration, and evaluation of CoreConnect/ PowerPC based SoCs. For this, it consists mainly of an extensive collection of IP components in TLM for the CoreConnect architecture, as well as several reference designs. Not only functional but also other aspects can be modelled with this library, such as the timing and power consumption of a SoC. We focused only on the functional integration.

Figure 7.11 shows how PEK models the CoreConnect architecture as a TLM framework. We list several IP models to be used later:

- ***PLB***, ***OPB***, ***PLBOPBBridge***, and ***DCR***: These are the TLM models for the SoC on-chip communication defined by the CoreConnect architecture, which includes the PLB, OPB, and DCR bus specifications. As mentioned, the communication realized by these bus models is *cycle-accurate*, with regard to the original timing specification. It means if we model and integrate the computation components, for example a CPU, also in a cycle-accurate way, we should have the possibility to obtain a fully cycle-accurate system model. The data and hand-shake protocols are transmitted through particular data structures: *PLB_REQUEST*, *OPB_REQUEST*, and *DCR_REQUEST*.

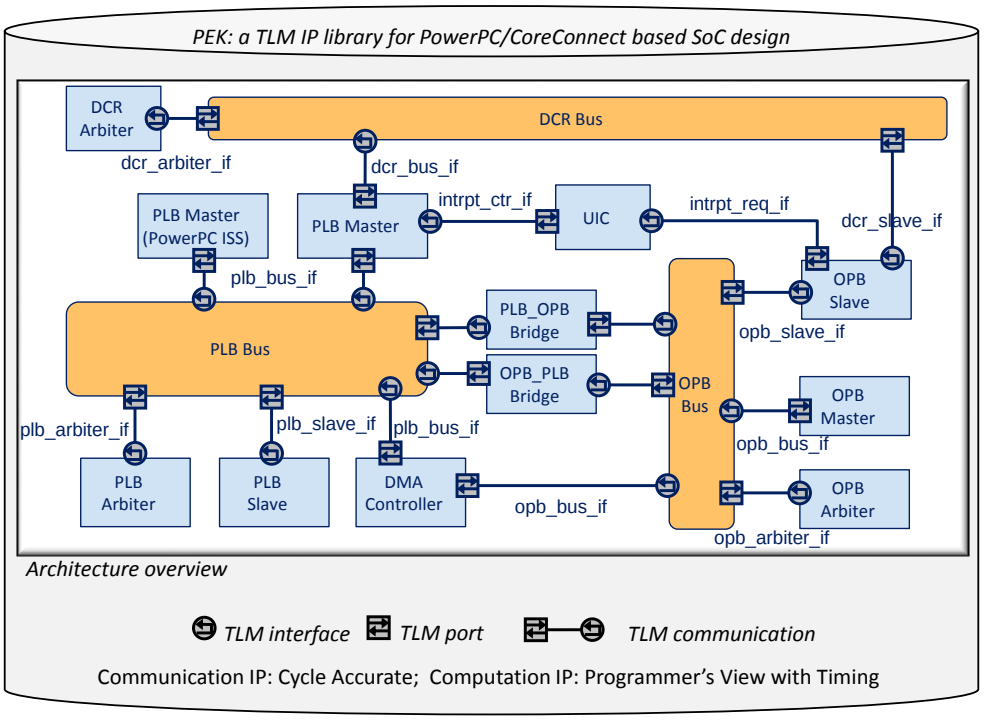


Figure 7.11 PEK (PowerPC Evaluation Kit) SoC library [41].

- ***PPC_ISS***: a PowerPC 405/440 Instruction Set Simulator (ISS) wrapped as a TLM component. It models a PowerPC microprocessor in a CoreConnect system, with three main TLM ports to be connected: a PLB master port for instruction, another one for data, and a DCR master port. It implements further the interrupt interface in TLM, accepting control from an interrupt controller. Parameters that can be configured during instantiation of this component include, for example, size of instruction/data cache, width of each PLB connection, master IDs on PLB as well as DCR, path to the executable file, and other ISS related options. The ISS is synchronized with PLB and DCR through SystemC clocks.
- ***DDR_MC2PLB4_MODULE***: a Double Data Rate (DDR) memory controller, which contains also a cycle-approximate memory model that mimics the industry standard DDR SDRAM interface. The controller is supposed to be connected to a PLB bus through the PLB slave interface. It can be used together with the ISS component and loaded with a binary cross-compiled for the ISS. Then all the bus protocol, DDR controller, and the memory model will be exercised during ISS execution. Possible configurations of this controller are PLB data/address width, high/low address on PLB, mode of cycle accuracy, other timing as well as row/bank number for the memory model, and so on.

- ***UIC***: it models a *Universal Interrupt Controller* that handle interrupts for a CPU. Up to *32* inputs can be connected and configured. Further, two types of interrupts – *critical* and *non-critical* – are supported. All such communications are carried out through an *INTRP_REQUEST* data structure, as with the buses.
- ***UART16750***: a Universal Asynchronous Receiver/Transmitter (UART) device that can be attached to OPB. It receives data from, or transmits data to its serial port, during which it also initiates interrupt to a CPU. The serial port can be connected to a component called *file_reader_writer*, that reads a file as the UART input or record the UART output. The FIFO size can be configured for this component.
- ***Console***: this models an input/output terminal external to a SoC model. When connected to a UART, it facilitates an interaction with the SoC, for example for testing purpose.
- ***IoDevice***: an IO model that mimics several file accessing interfaces. When it is attached to PLB through its PLB slave interface, a program running on the ISS can use these interfaces – *close*, *fstat*, *isatty*, *lseek*, *open*, *read*, *stat*, and *write* – to access this model and perform file operations, as real files are available . The buffer size of an *IoDevice* can be configured.
- ***EMAC***, ***GMII***, ***GmiiDevice***, and ***MAL_CONTROLLER***: together, these cores provide modeling facility for SoC design with Ethernet interfacing. *EMAC* models an Ethernet media access controller that complies with the IEEE standard 802.3 for Ethernet Media Access Control protocol. In 1000-Mbps mode, it operates in connection with a *GMII* (Gigabit Media Independent Interface), which in turn connects to a *GmiiDevice* that models a standard Ethernet PHY. On the other side, an *EMAC* connects to a *MAL_CONTROLLER* core, which transfers packet directly between memory and EMAC, by behaving as a master on PLB. Then an Ethernet software stack maintains merely the memory descriptor from this
- *MAL_CONTROLLER*: it provides mainly a data transfer facility between memory and a packet-oriented core, such as the EMAC core just mentioned. It minimizes the involvement of a CPU in such Ethernet traffic.

7.3.2. Two SoC Case Studies on IP-XACT Tool

In the first case study, we excised our Eclipse-based IP-XACT tool with two reference designs from PEK. We show only the first exercise with **Figure 7.12**. Basically, the design is a SoC scenario that exercises two Ethernet 1-GB high speed serial (HSS) link cores.

There are three main traffic flows being generated, which at the same time serves the test of this SoC system integration: i) the Ethernet traffic on top of Ethernet controllers, the MAL controller, the cycle-accurate DDR memory model, and SW stack, ii) the UART traffic, and ii) the additional file operation traffic, through the mimic of files by *IoDevice*.

All the related TLM IPs in PEK were first carefully documented as IP-XACT *components*, with necessary *bus/abstractionDefinitions* for PLB, OPB, and DCR. Then, a corresponding IP-XACT *design* for the Ethernet SoC is modeled.

The working process of our IP-XACT tool has been introduced before. Here we do not go to the details again. Some statistics from the tool exercise will be presented later together with the second case study.

In the second case study, we reused the *hybrid-task SoC design* that has been presented in the background chapter for discussing the reference flow. As mentioned, we designed and implemented this *hybrid-task SoC* and required IPs as a demonstration of the *CPU-FPGA task migration* idea [13] [11].

To exercise the IP-XACT tool, we considered this experiment: TLM based simulation and verification of this hybrid-task SoC design, as presented by **Figure 7.13**. A corresponding TLM system was created to model, simulate and, based on such simulation, verify the functional correctness of the hybrid-task SoC that is originally described as RTL.

On top the existing IP-XACT components and bus/abstractionDefinitions for PEK, IP-XACT descriptions for the hybrid-task TripleDES and hybrid-task manger IPs were first created. To integrate them into TLM simulation, TLM wrappers are also created for RTL, on their OPB interfaces. Then the hybrid-task SoC described in format *MHS* – Microprocessor Hardware Specification – was transformed into an IP-XACT design.

Tests for the original system were written as scripts running on a PC console that is connected to the FPGA board through a serial interface. Such a script consists of operation commands for the hybrid-task system: r*estart_task [sw|hw]*, *suspend_task*, *migrate_task*, *resume_task*, *step_task*, etc. Data streams for the TripleDES task were also fed through these commands.

We constructed tests for the TLM based hybrid-task SoC by imitating this mechanism. It is possible, since all the required components, including a UART and console model, are provided by PEK. In this way, all the TripleDES encryption/decryption and CPU-FPGA task migration scenarios were be tested with TLM.

The first result that we can report is that, in both case studies – on two PEK reference designs and a SoC design of our own, the IP-XACT tool was able to complete the generation and simulation of mutants, SystemC and Makefiles for all three SoC designs in IP-XACT.

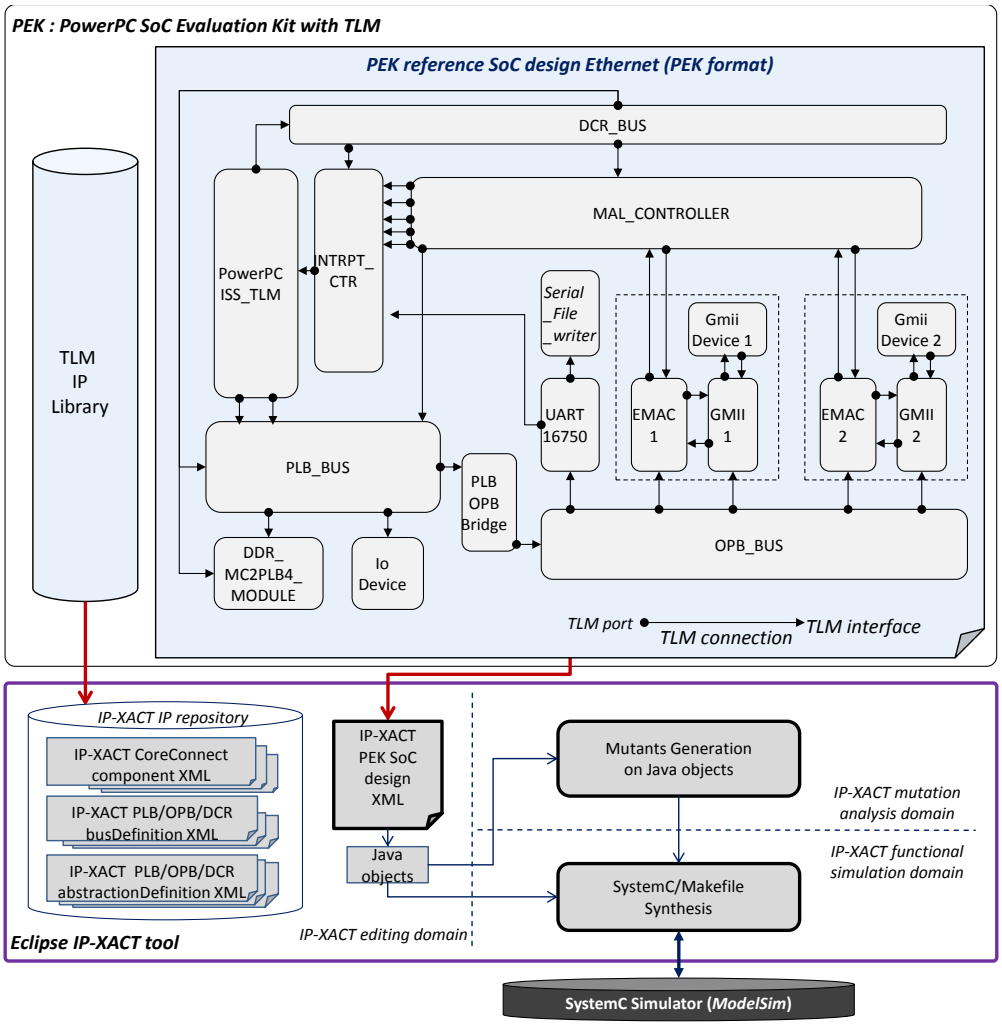


Figure 7.12 IP-XACT tool exercise with PEK reference SoC design (two Ethernet 1-GB high speed serial (HSS) link cores).

Figure 7.14 shows more statistics from the IP-XACT tool exercises, in particular, with regard to IP-XACT mutation analysis: the number of generated and killed mutants. At the end of the TLM system simulations, *79%*, *81%*, and *71%* mutants were killed respectively, out of the *151*, *134*, and *68* mutants that were generated in total under the seven basic mutation operators on the IP-XACT schema. The tests were improved manually by, for example, generating more Ethernet traffic and simply repeating more commands for hybrid-task operations, to make a reference and compare the mutation analysis results from the original tests. For the simulation of each mutant, we measured it as *killed*, if a different system simulation trace was recorded.

A large part of the total mutants were generated by the parameter group of mutation operators, such as reducing or increasing the FIFO size in the UART component, trying different cache sizes in PowerPC configuration, setting another priority scheme for the

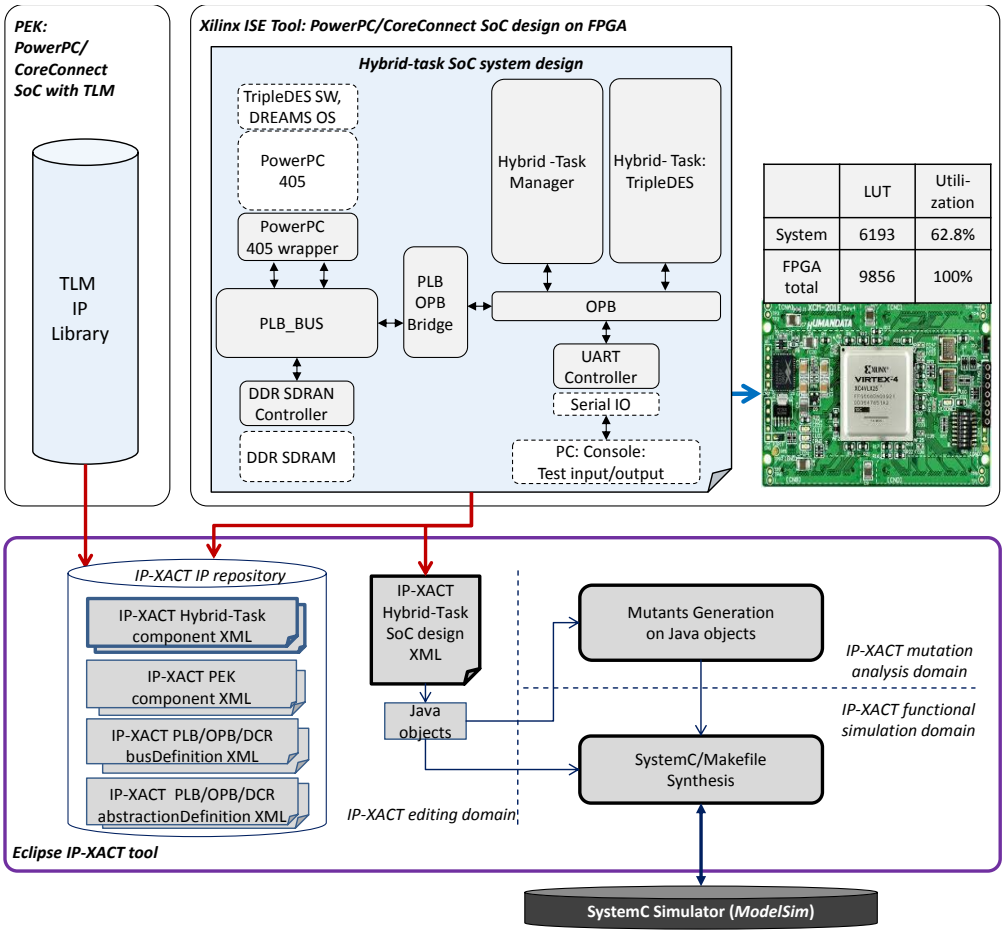


Figure 7.13 IP-XACT tool case study 2: TLM based simulation and verification of a *hybrid-task SoC*.

PLB arbitration, configuring TX/RX FIFO sizes for the Ethernet controller, etc. The *InterConnDel* operator was observed to be trivial (generating mutants that are too easy to be killed), if not totally unnecessary.

Indeed, at this moment, the selection and completion of IP-XACT mutation operators have not been optimized. Still, this represents our first effort, and the first published effort towards i) definition of mutation operators for IP-XACT, which we assume as the standard SoC system-level design format, and ii) a tool implementation for such mutation analysis. We have argued its necessity as a systematic metric for SoC system design. Our experimental tool and the case studies demonstrated this effort.

IP-XACT SoC exercise	IP-XACT IP included	Mutants	By Original system tests								By improved tests (Total)
			ParRep	*ParIns*	*ParDel*	*Baddr-Incr*	*Haddr-Decr*	*Addr-Exch*	*Inter-Conn-Del*	Total	
PEK Reference Design 1	17	Generated	69	18	20	16	19	4	5	**151**	151
		Killed	54	15	14	15	14	3	4	**119**	137
		Percentage	78%	83%	70%	94%	74%	75%	80%	**79%**	91%
PEK Reference Design 2	14	Generated	71	12	11	17	15	4	4	**134**	134
		Killed	59	10	7	14	13	2	4	**109**	126
		Percentage	83%	83%	64%	82%	87%	50%	100%	**81%**	90%
Hybrid-Task Design	8	Generated	28	11	8	7	7	3	4	**68**	68
		Killed	17	7	6	6	5	3	4	**48**	64
		Percentage	61%	64%	75%	86%	71%	100%	100%	**71%**	94%

Figure 7.14 More information on IP-XACT tool case studies: statistics of mutation analysis.

CHAPTER 8: Conclusion

In this thesis, we have proposed a simulation-based functional verification methodology for IP-based SoC design, which is driven by mutation analysis as a consistent metric for verification quality.

The background for our methodology includes mainly i) the increasing prevalence of SoCs, with IP-reuse and integration as the central design paradigm, ii) the emerging EDA tools and application of HDL mutation analysis, and iii) the emerging new languages and standards for SoC design, such as TLM and IP-XACT.

In this context, we have been able to identify the general problems as: i) at IP design phase, we lack efficient, practical test generation methods for HDL mutation analysis and ii) at SoC system design phase, we lack a systematic verification way as well as a quality metric for such verification. Therefore, our proposed verification enhancement flow consists of three components to address these problems.

First, considering the verification of an IP design, it is reasonable for us to employ light-weight constrained random simulation (CRS) to obtain a primary level of killed mutants. However, CRS can be inefficient, as it is defined neither for the original nor for the changing metric. The problem can be particularly amplified, since mutation analysis is a time-consuming metric. This has motivated us to *integrate a continuous, heuristic adaptation loop into CRS*. We have proposed using a *constraint-extended Markov chain* to model random test and provide the basis for such adaptation. We have also presented *dynamic mutation schemata* to efficiently carry out HDL mutation analysis and provide feedback. Then, the adaptation heuristic works by encouraging Markov-chain edges/constraints that could activate more mutants. In the evaluation experiment with the MB-Lite/FPU IP, we were able to observe the adaptation indeed leading to more activated as well as killed mutants. We achieved our goal of enhancing the mutation analysis efficiency in CRS.

Second, there are “hard” mutants expected to be left un-killed after this adaptive random simulation. Avoidance of any symbolic simulation has motivated us to apply

metaheuristic search based test generation to handle each of the remaining mutants. Such a metaheuristic, for example a simple local search, searches the design input space and tries to move gradually to a mutant-killing target test, relying only on guidance from real design simulation, though a solution is not guaranteed. As the key contribution here, we have defined an *objective cost function to effectively steer such search for HDL mutation analysis*. The basis of the cost function is a *Control and Data Flow Graph* (CDFG), which is exactly capable of modeling the test generation problem: *reach-activate-propagate*. The cost function is then comprised of a *macro propagation distance* and a *local propagation cost*, which measures the degree of activation and propagation conditions being fully satisfied. The MB-Lite/FPU IP evaluation showed that this cost function was consistently able to steer a local search procedure successfully towards mutant-killing tests.

Third, moving to SoC system design, the consideration of TLM and IP-XACT as well as the need to provide a consistent quality metric by mutation analysis has motivated us to propose a *SystemC based framework for IP-XACT design simulation and IP-XACT mutation analysis*. An *IP-XACT-to-SystemC synthesis flow* is defined to enable IP-XACT simulation. It provides a single platform for multi-language, mixed-level simulation, including RTL, behavioral, or TLM, at SoC system level. For this synthesis, we have also considered important *TLM compatibility rules* for IP-XACT-based compatibility check and safe binding of TLM components. Generation has been defined not only for IP-XACT-to-SystemC, but also for a Makefile composition, so as to provide a fully automated simulation process. Upon this simulation facility, IP-XACT mutation analysis has been added by the *definition of a set of mutation operators on IP-XACT schema*, which represent possible errors within an IP-XACT system design. We have also implemented an Eclipse-based prototype tool realizing all these functionalities. In the evaluation with several CoreConnect/PowerPC SoC integrations, we were able to confirm the tool's capability of completing the generation as well as simulation of mutants, SystemC and Makefiles and, therefore, also prove the general feasibility of the concepts behind the tool. We showed also the capability of the defined IP-XACT mutation operations of qualifying system tests.

Together, our methods provide a systematic, novel enhancement to functional design verification, based on HDL mutation analysis, TLM, and IP-XACT that are state-of-the-art techniques. In particular, they accommodates IP-based SoC design paradigm, by increasing the thoroughness of IP verification and focusing on IP integration at SoC system level. We view the thesis as a meaningful step towards closing the verification gap in the context of SoC design.

8.1. Outlook

The following aspects have *not* been fully explored by this thesis, at the moment, but are considered as reasonable future work.

- *Parameters of the methods have not been optimized* or strictly evaluated, such as the optimal transition point from random simulation to search based test generation, the best manner of modeling a Markov chain, the best move mechanism in local search, and so on. The thesis has focused on firstly establishing the methods as valid and effective.
- The CDFG serves the basis data structure in our cost function definition for HDL mutation analysis. One limitation is that we still *lack an automation tool for extracting such CDFG*. In the evaluation experiment, we built the CDFG manually from the MB-Lite and FPU design. It limits us from evaluating the metaheuristic test generation on further examples. This can be a practical step for further work.
- It is reasonable for us also to investigate and compare the performance of other metaheuristics when applied for HDL mutation analysis and test generation, under steering from the CDFG cost function. For example, advance Ant Colony algorithm has been employed in related work for test generation [91]. If we have an automatic CDFG extraction, such investigation would be with less burden.
- As mentioned, *more comprehensive evaluation of IP-XACT mutation operators* will be future work.
- In this work, the *functional verification at SoC system level has been limited to hardware IP integration, without considering a software-integrated system testing*. This comes from the thesis's focus on hardware design verification, without touching the area of hardware-software co-design. Indeed, embedded software is becoming an increasingly significant part of the whole SoC development effort. Systematic, metrics-managed testing of SoC system together with embedded software should be investigated. In fact, together with our colleagues, we have made the first step towards a unified covering of all hardware, embedded software, and system aspects with mutation analysis. In [4] and [3], we have proposed using a dynamic translation based emulator – called QEMU [116] – to *enable mutation analysis of embedded software binaries*, for scenarios where they are provided in a hard-IP-like manner without source code.

Bibliography[1]

[1] T. Xie, W. Mueller and F. Letombe, "Mutation-analysis driven functional verification of a soft microprocessor," in *Proc. of 25th IEEE System On Chip Conference (SOCC)*, Niagara Falls, NY, USA, 2012.

[2] T. Xie and W. Mueller, "An IP-XACT-to-SystemC Model Generator for Mutation Analysis," in *Proc. of International Workshop on Metamodelling and Code Generation for Embedded Systems (at ESWeek)*, Tampere, Finland, 2012.

[3] M. Becker, C. Kuznik, M. M. Joy, T. Xie and W. Mueller, "Binary mutation testing through dynamic translation," in *Proc. of 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Boston, USA, 2012.

[4] M. Becker, D. Baldin, C. Kuznik, M. M. Joy, T. Xie and W. Mueller, "XEMU: an efficient QEMU based binary mutation testing framework for embedded software," in *Proceedings of the 10th ACM international conference on Embedded software*, Tampere, Finland , 2012.

[5] T. Xie, W. Mueller and F. Letombe, "IP-XACT based system level mutation testing," in *Proc. of 16th IEEE International High-Level Design Validation and Test Workshop (HLDVT)*, Napa Valley, USA, 2011.

[6] T. Xie, W. Mueller and F. Letombe, "HDL-Mutation Based Simulation Data Generation by Propagation Guided Search," in *Proc. of 14th Euromicro Conference on Digital System Design (DSD)*, Oulu, Finland, 2011.

[7] T. Xie, W. Mueller and F. Letombe, "Efficient mutation-analysis coverage for constrained random verification," in *Distributed, Parallel and Biologically Inspired Systems*, Springer, 2010, pp. 114-124.

[8] M. Becker, G. Di Guglielmo, F. Fummi, W. Mueller, G. Pravadelli and T. Xie, "RTOS-aware refinement for TLM2.0-based HW/SW designs," in *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, Dresden, Germany, 2010.

[9] T. Xie, G. B. Defo and W. Mueller, "An Eclipse-based Framework for the IP-XACT-enabled Assembly of Mixed-Level IPs," in *Proc. of Intl. Workshop on Hands-on Platforms and Tools for Model-based Engineering of Embedded Systems (HoPES)*, Paris, France, 2010.

[10] T. Schattkowsky, T. Xie and W. Mueller, "A uml frontend for ip-xact-based ip management," in *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, Nice, France, 2009.

[11] M. Goetz, F. Dittmann and T. Xie, "Dynamic relocation of hybrid tasks: Strategies and methodologies," *Microprocessors and Microsystems*, vol. 33, no. 1, pp. 81-90, February 2009.

[12] T. Schattkowsky and T. Xie, "UML and IP-XACT for Integrated SPRINT IP Management," in *Proc. of 5th International UML-SoC Workshop (at DAC)*, Anaheim, USA, 2008.

[13] M. Goetz, T. Xie and F. Dittmann, "Dynamic Relocation of Hybrid Tasks: A Complete Design Flow," in *Proc. of Intl. Workshop on Reconfigurable Communication-centric System-on-Chip (ReCoSoC)*, Montpellier, France, 2007.

[14] P. Rashinkar, P. Paterson and L. Singh, System-on-a-chip verification: methodology and techniques, Springer, 2001.

[15] S. Fine and A. Ziv, "Coverage Directed Test Generation for Functional," in *Proc. of the 40th Design Automation Conference*, Anaheim, CA, USA, 2003.

[16] 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

[1] [1]-[13] are cited publications as author or co-author

Proc. of the IEEE Intl Conf. on Computer Design: VLSI in Computers & Processors (ICCD), Austin, TX, 2001.

[17] F. Faggin, M. E. Hoff Jr, S. Mazor and M. Shima, "The History of the 4004," *IEEE Micro*, vol. 16, no. 6, pp. 10-20, 1996.

[18] ITRS, "International Technology Roadmap for Semiconductors 2011 Edition - Design," Available at http://www.itrs.net/Links/2011ITRS/Home2011.htm, 2011.

[19] J. Bergeron, Writing testbenches: functional verification of HDL models, vol. 2, Kluwer Academic Publishers Dordrecht, 2003.

[20] P. Wilcox, Professional Verification: A Guide to Advanced Functional Verification, Springer, 2004.

[21] Synopsys, "Certitude," 2013. [Online]. Available: http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/certitude-ds.aspx.

[22] M. Hampton and S. Petithomme, "Leveraging a commercial mutation analysis tool for research," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, 2007.

[23] N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton and F. Letombe, "Functional qualification of TLM verification," in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, 2009.

[24] IBM, "CoreConnect Bus Architecture," Available at https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture, 1999.

[25] D. Lattard, E. Beigne, C. Bernard, C. Bour, F. Clermidy, Y. Durand, J. Durupt, D. Varreau, P. Vivet, P. Penard and others, "A telecom baseband circuit based on an asynchronous network-on-chip," in *IEEE International Solid-State Circuits Conference (ISSCC 2007), Digest of Technical Papers*, 2007.

[26] P. Vivet, D. Lattard, F. Clermidy, E. Beigne, C. Bernard, Y. Durand, J. Durupt and D. Varreau, "FAUST, an Asynchronous Network-on-Chip based Architecture for Telecom Applications," in *Proc. of Design, Automation and Test in Europe (DATE'07)*, 2007.

[27] C-LAB, "COCONUT project: A Correct-by-Construction Workbench for Design and Verification of Embedded Systems," Available at http://www.c-lab.de/en/rd_projects/completed_research_projects/2010/coconut/, 2010.

[28] S. Liao, G. Martin, S. Swan and T. Grötker, System design with SystemC, Kluwer Academic Pub, 2002.

[29] IEEE, 1685-2009 - IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows, 2009.

[30] W. Kruijtzer, P. van der Wolf, E. de Kock, J. Stuyt, W. Ecker, A. Mayer, S. Hustin, C. Amerijckx, S. de Paoli and E. Vaumorin, "Industrial IP Integration Flows based on IP-XACT™ Standards," in *Proc. of DATE'08*, Munich, Germany, 2008.

[31] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proc. of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2003.

[32] S. Pasricha, "Transaction level modeling of SoC with SystemC 2.0," in *Synopsys User Group Conference (SNUG)*, 2002.

[33] P. J. Ashenden, The Designer's Guide to VHDL, Third Edition, Morgan Kaufmann, 2010.

[34] D. E. Thomas and P. R. Moorby, The Verilog® Hardware Description Language, vol. 2, Springer, 2002.

[35] S. M. Rubin, Computer aids for VLSI design, Massachusetts, USA: Addison-Wesley Reading, 1987.

[36] A. Bruce, M. Kamal Hashmi, A. Nightingale, S. Beavis, N. Romdhane and C. Lennard, "Maintaining consistency between SystemC and RTL system designs," in *Proceedings of the 43rd annual Design Automation Conference (DAC)*, 2006.

[37] N. Bombieri and F. Fummi, "On the Automatic Transactor Generation for TLM-based Design Flows," in *Proc. of 11th IEEE International High-Level Design Validation and Test Workshop (HLDVT)*, 2006.

[38] T. Zhang, L. Benini and G. De Micheli, "Component selection and matching for IP-based design," in *Proc. of Conference and Exhibition on Design, Automation and Test in Europe (DATE)*, 2001.

[39] OpenCore, "Triple DES encryption core," Available at http://opencores.org/.

[40] IBM, "Processor Local Bus 128-bit Specification," Available at https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture, 2007.

[41] IBM, "PowerPC 405 Evaluation Kit with CoreConnect SystemC TLMs," Available at http://www.ibm.com/developerworks/power/pek/index.html.

[42] N. Bombieri, F. Fummi and G. Pravadelli, "On the evaluation of transactor-based verification for reusing TLM assertions and testbenches at RTL," in *Proc. of the conference on Design, automation and test in Europe (DATE)*, 2006.

[43] M. Dales, "SWARM 0.44 Documentation," Department of Computing Science, University of Glasgow, 2000.

[44] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli and M. Olivieri, "Mparm: Exploring the multi-processor soc design space with systemc," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 41, no. 2, pp. 169-182, 2005.

[45] N. Romdhane, "ARM RealView ESL APIs," in *5th North American SystemC User's Group (NASCUG) Meeting*, 2006.

[46] E. Van der Vlist, XML Schema: The W3C's Object-Oriented Descriptions for XML, O'Reilly Media, Inc., 2011.

[47] E. M. Clarke, O. Grumberg and D. A. Peled, Model checking, MIT press, 1999.

[48] M. Bombana and F. Bruschi, "SystemC-VHDL co-simulation and synthesis in the HW domain," in *Proc. of the conference on Design, Automation and Test in Europe: Designers' Forum-Volume 2*, 2003.

[49] IEEE Computer Society, "IEEE Std 754-2008: IEEE Standard for Floating-Point Arithmetic," Available at http://dx.doi.org/10.1109%2FIEEESTD.2008.4610935, 2008.

[50] D. Große, R. Ebendt and R. Drechsler, "Improvements for Constraint Solving in the SystemC Verification Library," in *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, 2007.

[51] IEEE, 1800-2009 Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language, 2009.

[52] C. Kuznik and W. Müller, "Aspect enhanced functional coverage driven verification in the SystemC HDVL," in *Proc. of the 8th International SoC Design Conference*, 2011.

[53] M. F. S. Oliveira, C. Kuznik, H. M. Le, D. Große, F. Haedicke, W. Müller, R. Drechsler, W. Ecker and V. Esen, "The System Verification Methodology for Advanced TLM Verification," in *Proc. of Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS'12)* , 2012.

[54] S. Devadas, A. Ghosh and K. Keutzer, "An Observability-Based Code Coverage Metric for Functional Simulation," in *Proc. of the IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, USA, 1996.

[55] J. P. Roth, "Diagnosis of automata failures: A calculus and a method," *IBM journal of Research and Development*, vol. 10, no. 4, pp. 278-291, 1966.

[56] F. Fallah, S. Devadas and K. Keutzer, "OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification," in *Proc. of the 35th annual Design Automation Conference (DAC)*, San Francisco, CA, USA, 1998.

[57] F. Fallah, S. Devadas and K. Keutzer, "OCCOM-efficient computation of observability-based code coverage metrics for functional verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 1003-1015, 2001.

[58] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34-41, April 1978.

[59] A. Offutt, "The coupling effect: fact or fiction," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131-140, 1989.

[60] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 1, pp. 5-20, 1992.

[61] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649-678, 2011.

[62] A. P. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *Proc. of the Fifteenth Annual International Computer Software and Applications Conference (COMPSAC'91)*, 1991.

[63] R. H. Untch, A. J. Offutt and M. J. Harrold, "Mutation analysis using mutant schemata," *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 3, pp. 139-148, 1993.

[64] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, no. 4, pp. 371-379, 1982.

[65] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE Transactions onSoftware Engineering*, vol. 20, no. 5, pp. 337-344, 1994.

[66] A. J. Offutt, G. Rothermel and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th international conference on Software Engineering*, 1993.

[67] R. A. DeMillo, D. S. Guindi, W. McCracken, A. Offutt and K. King, "An extended overview of the Mothra software testing environment," in *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, 1988.

[68] G. Al Hayek and C. Robach, "From specification validation to hardware testing: A unified method," in *Proceedings of International Test Conference 1996*, 1996.

[69] C. Aktouf, G. Al-Hayek and C. Robach, "Concurrent testing of VLSI digital signal processors using mutation based testing," in *Proceedings of 1997 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 1997.

[70] G. Al-Hayek and C. Robach, "From design validation to hardware testing: A unified approach," *Journal of Electronic Testing*, vol. 14, no. 1-2, pp. 133-140, 1999.

[71] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Transactions on Computers*, vol. 100, no. 3, pp. 215-222, 1981.

[72] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Transactions on Computers*, vol. 100, no. 12, pp. 1137-1144, 1983.

[73] T. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," in *Proc. of the European Conference on Design Automation*, 1991.

[74] S. C. Brailsford, C. N. Potts and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," *European Journal of Operational Research*, vol. 119, no. 3, pp. 557-581, 1999.

[75] I. Wagner, V. Bertacco and T. Austin, "Microprocessor Verification via Feedback-Adjusted Markov Models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 26, no. 6, pp. 1126-1138, June 2007.

[76] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber and K. Keutzer, "Coverage-Directed Generation of Biased Random Inputs for Functional Validation of Sequential Circuits," in *Proc. of the 10th IEEE International Workshop on Logic and Synthesis*, 2001.

[77] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design & Test of Computers,* vol. 18, no. 4, pp. 36-45, 2001.

[78] W. Chen, L.-C. Wang, J. Bhadra and M. Abadir, "Simulation knowledge extraction and reuse in constrained random processor verification," in *Proceedings of the 50th Annual Design Automation Conference (DAC)*, 2013.

[79] M. F. S. Oliveira, H. Zabel and W. Müller, "Assertion-Based Verification of RTOS Properties," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'10)*, 2010.

[80] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability,* vol. 14, no. 2, pp. 105-156, 2004.

[81] G. G. De Jong, "Data flow graphs: system specification with the most unrestricted semantics," in *Proceedings of the conference on European design automation*, 1991.

[82] R. Camposano, "Path-based scheduling for synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 10, no. 1, pp. 85-93, 1991.

[83] R. Namballa, N. Ranganathan and A. Ejnioui, "Control and data flow graph extraction for high-level synthesis," in *Proceedings. IEEE Computer society Annual Symposium on VLSI*, 2004.

[84] R. A. Bergamaschi, "Behavioral network graph: unifying the domains of high-level and logic synthesis," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference (DAC'99)*, 1999.

[85] Q. Zhang and I. G. Harris, "A data flow fault coverage metric for validation of behavioral hdl descriptions," in *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design (ICCAD)*, 2000.

[86] F. Fallah, P. Ashar and S. Devadas, "Simulation vector generation from HDL descriptions for observability-enhanced statement coverage," in *Proc. of the 36th annual ACM/IEEE Design Automation Conference (DAC'99)*, 1999.

[87] R. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions onSoftware Engineering,* vol. 17, no. 9, pp. 900-910, 1991.

[88] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International journal on software tools for technology transfer,* vol. 11, no. 4, pp. 339-353, 2009.

[89] E. Tsang, Foundations of constraint satisfaction, Academic press London, 1993.

[90] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering,* vol. 16, no. 8, pp. 870-879, 1990.

[91] K. Ayari, S. Bouktif and G. Antoniol, "Automatic mutation test input data generation via ant colony," in *Proc. of the 9th annual conference on Genetic and evolutionary computation*, 2007.

[92] S. Shyam and V. Bertacco, "Distance-guided hybrid verification with GUIDO," in *Proc. of the conference on Design, automation and test in Europe (DATE'06)*, 2006.

[93] W. Wu and M. S. Hsiao, "Efficient design validation based on cultural algorithms," in *Proc. of Design, Automation and Test in Europe (DATE'08)*, 2008.

[94] A. Parikh, W. Wu and M. S. Hsiao, "Mining-guided state justification with partitioned navigation tracks," in *Proc. of IEEE International Test Conference (ITC'07)*, 2007.

[95] Mentor Graphics, "ModelSim," Available at http://model.com/.

[96] D. Steinberg, F. Budinsky, M. Paternostro and E. Merks, EMF: Eclipse Modeling Framework, 2nd Edition ed., Addison-Wesley, 2009.

[97] A. El Mrabti, F. Pétrot and A. Bouchhima, "Extending IP-XACT to support an MDE based approach for SoC design," in *Proc. of DATE'09.*, 2009.

[98] D. Braga, F. Fummi, G. Pravadelli and S. Vinco, "The strange pair: IP-XACT and univerCM to integrate heterogeneous embedded systems," in *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2012.

[99] L. D. Guglielmo, F. Fummi, G. Pravadelli, F. Stefanni and S. Vinco, "UNIVERCM: The UNIversal VERsatile Computational Model for Heterogeneous System Integration," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 225-241, 2013.

[100] F. Herrera and E. Villar, "A framework for the generation from UML/MARTE models of IPXACT HW platform descriptions for multi-level performance estimation," in *Proc. of IEEE 2011 Forum on Specification and Design Languages (FDL)*, 2011.

[101] F. Herrera, H. Posadas, E. Villar and D. Calvo, "Enhanced ip-xact platform descriptions for automatic generation from UML/MARTE of fast performance models for DSE," in *Proc. of 15th Euromicro Conference on Digital System Design (DSD)*, 2012.

[102] Object Management Group, "UML Profile for MARTE v1.0," 2009.

[103] G. Ochoa, E.-B. Bourennane, O. Labbani and K. Messaoudi, "IP-XACT and marte based approach for partially reconfigurable systems-on-chip," in *Proc. of IEEE 2011 Forum on Specification and Design Languages (FDL)*, 2011.

[104] G. Ochoa-Ruiz, O. Labbani, E.-B. Bourennane, P. Soulard and S. Cherif, "A high-level methodology for automatically generating dynamic partially reconfigurable systems using IP-XACT and the UML MARTE profile," *Design Automation for Embedded Systems*, pp. 1-36, 2012.

[105] P. Lisherness and K.-T. (. Cheng, "SCEMIT: a SystemC error and mutation injection tool," in *47th ACM/IEEE Design Automation Conference (DAC'10)*, 2010.

[106] A. Sen, "Mutation operators for concurrent SystemC designs," in *10th International Workshop on Microprocessor Test and Verification (MTV'09)*, 2009.

[107] A. Sen and M. S. Abadir, "Coverage metrics for verification of concurrent SystemC designs using mutation testing," 2010.

[108] N. Bombieri, F. Fummi and G. Pravadelli, "A mutation model for the SystemC TLM 2.0 communication interfaces," in *Proc. of Design, Automation and Test in Europe (DATE'08)*, 2008.

[109] T. Kranenburg and R. van Leuken, "MB-LITE: A robust, light-weight soft-core implementation of the MicroBlaze architecture," in *Proc. of the Conference on Design, Automation and Test in Europe (DATE'10)*, 2010.

[110] S. Xu and H. Pollitt-Smith, "A multi-microblaze based SOC system: from SystemC modeling to FPGA prototyping," in *Proc. of 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP'08)*, 2008.

[111] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose and E. Deprettere, "Daedalus: toward composable multimedia MP-SoC design," in *Proceedings of the 45th annual Design Automation Conference*, 2008.

[112] X. Guo, Z. Chen and P. Schaumont, "Energy and Performance Evaluation of an FPGA-Based SoC Platform with AES and PRESENT Coprocessors," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Springer Berlin Heidelberg, 2008, pp. 106-115.

[113] G. Kornaros, "A soft multi-core architecture for edge detection and data analysis of microarray images," *Journal of Systems Architecture*, vol. 56, no. 1, pp. 48-62, 2010.

[114] Xilinx, "MicroBlaze Processor Reference Guide v10.1i," http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf, 2010.

[115] T. Kranenburg, "MB-Lite," Available at http://opencores.org/project,mblite, 2012.

[116] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005.