THE UNIVERSITY OF ADELAIDE

Software-Centric and Interaction-Oriented System-on-Chip Verification

by

Xiao Xi Xu

B.E. (Automatic Control)

Shanghai Jiao Tong University, China, 1996

A thesis submitted for the
degree of Doctor of Philosophy

in

School of Electrical and Electronic Engineering

Faculty of Engineering, Computer and Mathematical Sciences

March 2009

Declaration of Authorship

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no mateiral previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being made available for loan and photocopying, subject to the provisions of the Copyright Act 1968.

Signature: _____

Date: _____

Abstract

As the complexity of very-large-scale-integrated-circuits (VLSI) soars, the complexity of verifying them increases even faster. *Design verification* becomes the biggest bottleneck in VLSI design, consuming around 70% of the effort and time in a typical design cycle. The problem is even more severe as the system-on-chip (SoC) design paradigm is gaining popularity.

Unfortunately, the development in verification techniques has not kept up with the growth of the design capability, and is being left further behind in the SoC era. In recent years, a new generation of hardware-modelling-languages alongside the best practices to use them have emerged and evolved in an attempt to productively build an intelligent stimulation-observation environment referred to as the *test-bench*. Ironically, as test-benches are becoming more powerful and sophisticated under these best practices known as *verification methodologies*, the overall verification approaches today are still officially described as *ad hoc and experimental* and are in great need of a methodological breakthrough.

Our research was carried out to seek the desirable methodological breakthrough, and this thesis presents the research outcome: a novel and holistic methodology that brings an opportunity to address the SoC verification problems. Furthermore, our methodology is a solution completely independent of the underlying simulation technologies; therefore, it could extend its applicability into future VLSI designs.

Our methodology presents two ideas. (a) We propose that system-level verification should resort to the *SoC-native languages* rather than the test-bench construction languages; the software native to the SoC should take more critical responsibilities than the test-benches. (b) We challenge the fundamental assumption that "objects-under-test" and "tests" are distinct entities; instead, they should be understood as *one* type of entities – the *interactions*; interactions, together with the *interference between interactions*, i.e., the parallelism and resource-competitions, should be treated as the focus in system-level verification.

The above two ideas, namely, *software-centric* verification and *interaction-oriented* verification have yielded practical techniques. This thesis elaborates on these techniques, including the *transfer-resource-graph* based test-generation method targeting the parallelism, the coverage measures of the concurrency completeness using Petri-nets, the automation of the test-programs which can execute smartly in an *event-driven* manner, and a software observation mechanism that gives insights into the system-level behaviours.

Acknowledgements

I thank my supervisors Prof. Cheng-Chew Lim and Prof. Michael Liebelt. They provided me with this research position, and they are the co-authors of my research publications. I am grateful to their advice and feedback during the development of this thesis. Cheng-Chew's help comes in all forms, including the resources he guarantees, the meetings he organises, the peer review he performs and the presentations he rehearses.

I would like to extend special thanks to Mr. Adriel Cheng, who has kindly opened his source codes in the SALVEM (Software Application Level Verification Methodology) approach to me. These codes have guided me to learn new programming languages, new tools and new technologies, and more importantly, I was able to understand the nature of software-based verification from them. It is these codes that have invited me to form my own idea. In addition, the Nios SoC used in my research was generated for the SALVEM project. I appreciate many scintillating talks with Adriel about SoC verification.

I would also like to thank Mr. Kiet To for interesting conversations on more general topics about typical computer structures.

This research work is supported by Australian Research Council Linkage Project (LP0454838). And the Australian Postgraduate Award (Industry) allows me to concentrate on the research.

Finally, I must thank my wife Hongqi Wu, who has given me energy and support throughout the research.

Contents

List of Figures

List of Tables

Abbreviations

ALU	**A**rithmatic **L**ogic **U**nit
BDD	**B**inary **D**ecision **D**iagram
BFM	**B**us **F**uctional **M**odel
CTL	**C**omputation **T**ree **L**ogic
DMA	**D**irect **M**emory **A**ccess
DUT	**D**esign **U**nder **T**est
EDA	**E**lectronics **D**esign **A**utomation
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSM	**F**inite **S**tate **M**achine
HDL	**H**ardware **D**escription **L**anguage
HVL	**H**ardware **V**erification **L**anguage
HW	**H**ard**w**are
IC	**I**ntegrated **C**ircuit
ISR	**I**nterrupt **S**ervice **R**outine
OOP	**O**bject **O**riented **P**rogramming
OS	**O**perating **S**ystem
RISC	**R**educed **I**nstruction **S**et **C**omputer
RTL, RT-Level	**R**egister **T**ransfer **L**evel
SoC	**S**ystem **o**n **C**hip
SW	**S**oft**w**are
TB	**T**est-**B**ench
TLM	**T**ransaction **L**evel **M**odel(ling)

TP	**T**est-**P**rogram
UART	**U**niversal **A**synchronous **R**eceiver and Transmitter
VLSI	**V**ery **L**arge **S**cale **I**ntegration

Dedicated to my girls: Hongqi, Jingyi and Grace.

Chapter 1

Introduction

1.1 Motivation and Contribution

1.1.1 Motivation

Owing to the advancing VLSI manufacturing technologies, the system-on-chip (SoC) solution – designing a whole system ready for application on a single chip – has become a popular design paradigm. Although there is no strict definition of SoC, a design could be counted as an SoC if it features *multiple components, including at least one processor, connected by on-chip interconnection*. The SoC paradigm has brought about rich benefits from various perspectives, including

- superior performances (higher frequency and lower power),
- wide applications (personal, wireless and military electronics) and
- overall lower design and manufacturing cost (fewer photomasks).

From the design complexity point of view, the SoC paradigm is also very beneficial – it has practically reduced designing a complex system to *integrating* pre-designed and reusable components. However, the *verification* of the SoC becomes the critical bottleneck in further improving SoC design productivity. Generally speaking, *verification* refers to the practice of detecting errors in designs. Designs that are not thoroughly verified are not worth manufacturing; and errors should be corrected as early as possible – correcting errors at a late stage could be forbiddingly costly.

Verification was regarded as the subservient issue compared with the *implementation* of a design. This view soon became invalid as designs became just moderately complex. The well known Moore's law suggests that the complexity of integrated circuits is growing at an exponential rate against time, whereas multiple sources claim the verification complexity is growing at a *double-exponential* rate [11, 74, 85], i.e., exponential with respect to Moores law. Figure 1.1 illustrates the growing "verification gap" between the integrated circuit (IC) verification capability and the IC design and manufacture capabilities. Nowadays about 50%-80% of the design time and efforts are spent in verification. It becomes well known that *verification is the biggest single bottleneck* [44] in integrated circuit design.

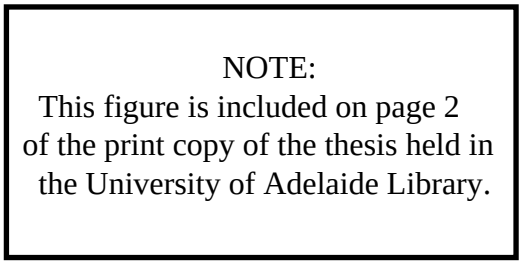


FIGURE 1.1: The verification gap from very-large-scale-integrated-circuit (VLSI) manufacturing point of view. While the manufacturing technology evolves, a state-of-the-art chip could have tens or even hundreds of millions of gates. However, the capability of designing such VLSI does not increase that fast. Worse, the capability of verifying such designs has been growing even slower. There is a huge and growing gaps between the verification capability and the design/manufacture capabilities. Adapted from: SIA Roadmap, 2001.

The features of a typical SoC impose great challenges on SoC verification in two respects.

- First, the large scale of hardware integration leads to sophisticated hardware-hardware interactions. Since an SoC has *multiple* components, the interactions between them could give rise to emerging properties that are not present in any single component.

- Second, the introduction of *software* into hardware behaviour leads to sophisticated hardware-software interactions. Since an SoC has at least one processor, software forms a new dimension of the SoC's behaviours and hence brings a new dimension in verification.

So far, there have been no SoC verification methodologies that address both the above challenges, which motivates our research on a holistic SoC verification methodology.

1.1.2 Contribution

The main approach to verifying a design, especially a very complex one, is by *simulation*. Simulation is so important to verification that the terms "simulation" and "verification" largely share the same meaning in practice. "Simulation" refers to the practice of running tests on *models* of a design before the design is actually manufactured. The term "model" refers to a presentation of the hardware under design in the form of software. The simulation approach inherently has the "simulation performance issue". That is, simulation is a very time-consuming process, while VLSI designers are constantly under the *time-to-market* pressure. Fast and accurate simulation is always desired; however, being fast and being accurate are always competing metrics for simulation-based approaches. The "verification gap" viewed from the simulation point of view is shown in Figure 1.2. As designs are becoming more complex, the requirement of thorough verification is soaring, whereas the performance of various simulation technologies is degrading. The simulation performance issue is more outstanding for SoC verification due to its high level of integration.

There is distinction between a verification methodology and a simulation technology. Our research is about SoC verification methodology using the simulation approach; our research is *not* about addressing the *simulation performance issue*, which is technology issue in simulating all types of design including the SoC. Faster simulation platforms and technologies are always desirable but they are independent of the SoC-specific challenges, namely, the problems about hardware-hardware and hardware-software interactions, as mentioned in Section 1.1.1. Methodology and technology *both* contribute to a successful verification practice. Without methodology support, it is very likely to get a poor verification quality even using an advanced simulation technology; without proper technology support, a good methodology may not exert its full power. Technologies evolve rapidly while methodology is relatively stable.

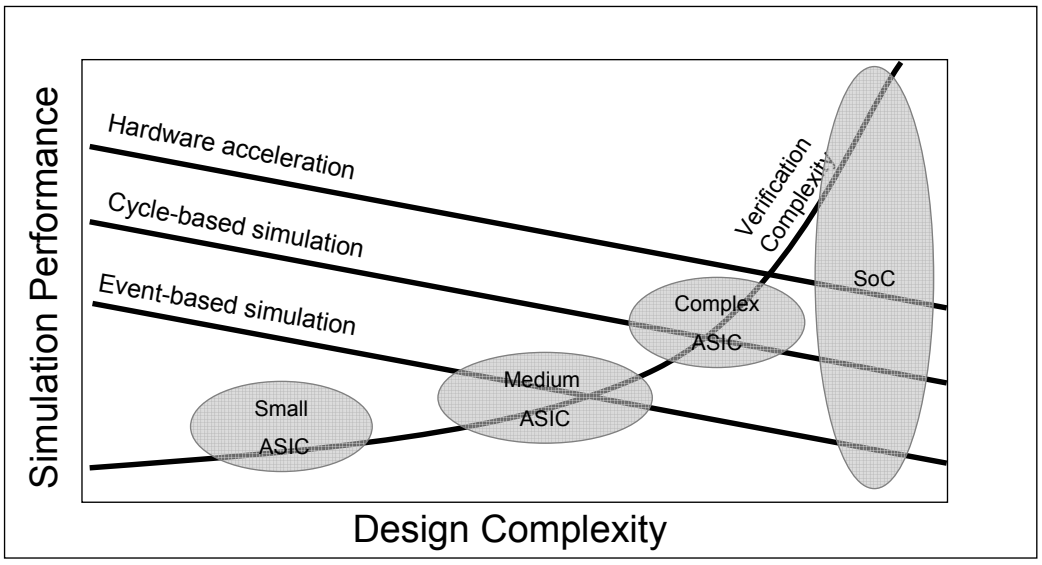


FIGURE 1.2: The verification gap from simulation point of view. Simulation is the main approach to design verification, and there are simulation platforms suitable for different abstraction levels. However, as integration level increases, simulation efficiency always decreases; while the requirement to thorough simulation increases. There is widening gap between the required and available simulation performance. Adapted from [70].

The contribution of this thesis is that it addresses the challenges specific to SoC and SoC-like designs and puts forward a methodology independent of the underlying simulation technologies.

Furthermore, the proposed methodology is very different from the current mainstream verification methodologies, which substantially centre on the construction of test-benches (TBs). A test-bench is a structure *external* to a design; it couples with a design during simulation. A common problem of these methodologies is that the software *native* to an SoC virtually does not have a proper position in the verification framework; and the TB structure tends to be very complicated. In contrast, our research proposes to systematically exploit software's capabilities in verifying an SoC, and to reduce the TB complexity. It should be noticed that using software for SoC verification is not new; our novelty is about partitioning software and test-benches into independent roles and organising them seamlessly in a well-defined verification framework. To our best knowledge, we do not think that there is any literature in which the relationship among the software, the test-bench and the SoC are so harmoniously arranged.

This thesis addresses the following problems that are not explicitly discussed or even dealt with by the current TB-centric methodologies.

(1) What is a *test* and what is an *object-under-test* in a *system* context?

(2) What are the *system-level specific* behaviours?

(3) How do we *automatically* generate tests that target the system-level specific behaviours?

(4) How do we implement tests on a system in the form of software? and

(5) What is the proper relationship among the test-bench, the software and the SoC under verification?

The quick answers to the above problems are:

(1) the term *tests* and *object-under-tests* should refer to one type of objects – the *interactions*;

(2) *concurrency* and the associated *resource-competition* form the *system-level specific* behaviours;

(3) using a model called transfer-resource-graph (TRG) to automate the test generation of concurrency and resource-competition;

(4) implementing tests in the form of an *event-driven* test-program, which is to be automatically generated by the TRG model;

(5) treating the test-bench, the software and the SoC under verification as one *continuum*, in which the software and the test-bench help each other in controlling and observing the SoC.

The full answers to these questions are elaborated in the rest of the thesis.

1.2 Thesis Overview

The main structure of the thesis is shown in Figure 1.3.

- Chapter 1 gives an introduction to the thesis, including the thesis motivation, contribution and structure.
- Chapter 2 lays a firm background for further development of the thesis. The main topics include:

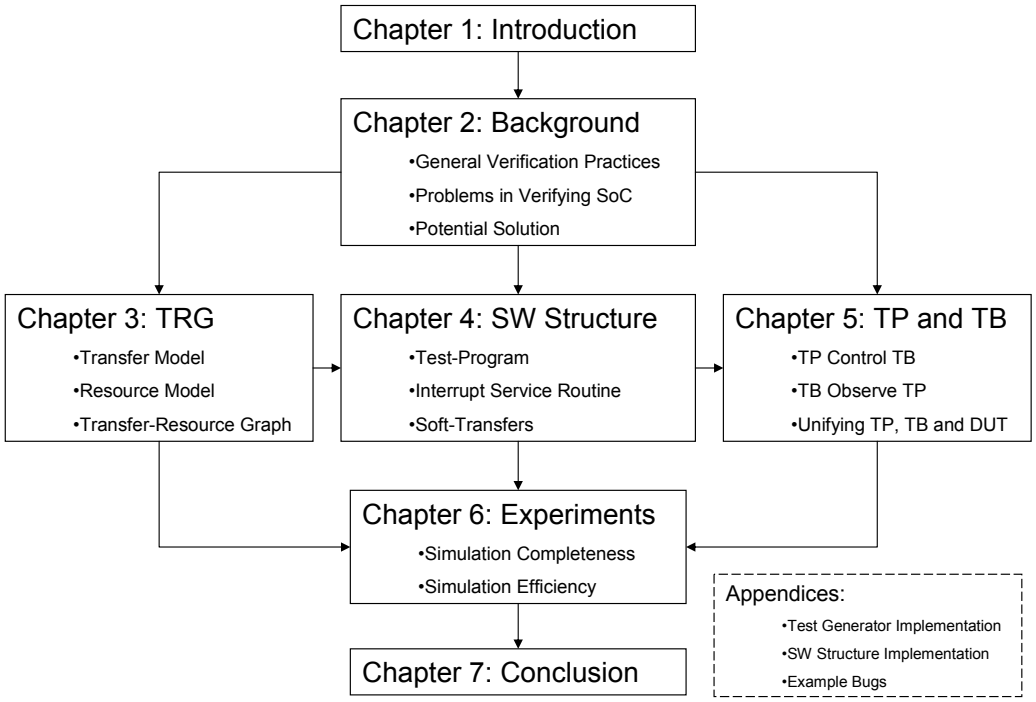


FIGURE 1.3: The structure of the thesis. The main contributions of the thesis are presented in three chapters (Chapter 3 to 5), followed by a chapter of experimental results. The logical link between the three main chapters is detailed in Chapter 2.

  - the general practices in the area of design verification;
  - the current problems in verifying a system-on-chip (SoC), and
  - the opportunities brought by the interaction-oriented thinking to address the SoC verification problems.

- Chapter 3 to Chapter 5 present the contributions of our research.
  - Chapter 3 introduces a formal model called transfer-resource graph (TRG) to solve the system level test-generation problem by discussing
    * the definition of an interaction model called *transfers* and how to model transfers as the building blocks for system-level verification;
    * the algorithm to generate test-cases of concurrency and resource-competition *automatically* using TRG;
    * the measurement of the test completeness by converting a TRG to a Petri-net.
  - Chapter 4 focuses on implementing the TRG-based tests in software. In this chapter, "software" is partitioned into three categories of components, each responsible for one type of roles. The respective requirements of these components together with their automation opportunities are elaborated.

  - Chapter 5 naturally moves to discuss the roles of the "hardware" verification infrastructures, i.e., the test-bench in the software-centric verification framework. This chapter focuses on the *relation between software and test-benches* in verifying an SoC, and finalises the theoretical contributions by proposing a novel framework of system-level verification.

- Chapter 6 demonstrates all experiments involved in the last three theoretical chapters. This chapter is also organised around two critical topics in simulation-based verification: (a) simulation completeness (i.e., coverages) and (b) simulation efficiency.

- Chapter 7 concludes the thesis.

Some miscellaneous topics including a few identified bugs in the SoC used in our research are arranged in the appendices.

1.3 Publications

The research has produced the following publications.

1. Justin Xu and Cheng-Chew Lim. *Modelling Heterogeneous Interactions in SoC Verification.* In IFIP International Conference on Very Large Scale Integration 2006, pages 98-103, October 2006.

2. Justin Xu and Cheng-Chew Lim. *Exploiting Concurrency in System-on-Chip verification.* In IEEE Asia Pacific Conference on Circuits and Systems (APCCAS 2006), pages 836-839, December 2006.

3. Xiaoxi Xu and Cheng-Chew Lim. *Using Transfer-Resource Graph for Software-Based Verification of System-on-Chip.* In IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 7, pages 1315-1328, July 2008.

4. Xiaoxi Xu, Cheng-Chew Lim and Michael Liebelt. *Positioning Test-Benches and Test-Programs in Interaction-Oriented System-on-Chip Verification.* In IEEE International Workshop on High Level Design Validation and Test (HLDVT 2008), pages 3-11, November 2008.

Chapter 2

Background

This chapter introduces background information regarding verification in three steps. First, we have a review of important concepts and techniques widely adopted in the general verification practices. Second, the problems associated with the current practices of system-level verification are identified. Third, we give an outline of our approach to solving these problems. The system-on-chip used in this research is also introduced in this chapter.

2.1 General Verification Practice

2.1.1 Overview

There are two categories of verification methods.

- Simulation-based methods or *dynamic verification*. In this category, the verification engineers develop a set of tests known as *testcases* to stress a given design. Hence, the design is often called design-under-test or DUT. A testcase could be a very abstract description of a scenario the DUT should be exercised in. In order to apply the abstract testcases to a DUT, a structure called *test-bench* (TB) needs to be constructed. The TB transforms and concretises the abstract testcases into "0/1" signals and directly interacts with the DUT using these signals.
- Formal methods or *static verification*. This category is called static since no tests are needed. Instead, the verification engineer should provide *design properties* (the properties a correct design should have) in the form of *temporal logic*. The design

is also required to be represented in the form of finite-state-machine (FSM). Then a binary-decision-diagram (BDD-) based model-checking tool computes whether the design abides by the properties. If one property is violated, the tool will produce at least one *counter example* – a sequence of input to the FSM that leads to the violation of that property.

Simulation can be regarded as the "experimental" approach, requiring a set of tests being applied to the DUT in a simulation environment; while formal methods attempt to "theoretically" prove that the design satisfies certain predefined properties. Each approach has its own strengths and limitations. Table 2.1 compares the two approaches. Generally speaking, the simulation-based approach, which is more conventional, imposes less restriction on the design than the formal approach does.

Category	User Input	Tool Required	Advantage	Drawback	Application
Simulation-based (dynamic)	Testcases; Test-benches.	Simulation-Platform	Flexible	Non-exhaustiveness; Simulation Performance issues.	Any executable specification
Formal (static)	Properties; FSM.	Model-checker	Exhaustive (for each property)	Property expressivity; Computation Complexity issues.	Control-intensive, FSM based designs

TABLE 2.1: Simulation-based verification and formal verification

2.1.2 Simulation-Based Verification

Simulation-based verification contains many inter-related and overlapping practices, concepts and terminologies. In order to describe them systematically, we organise them in four aspects:

- infrastructure – the basic elements involved in a simulation;
- building-block – the structure used to build a simulation environment;
- mechanism – the driving force of a simulation environment; and
- paradigm – the conventions in building a simulation environment.

2.1.2.1 Infrastructure: Abstraction Levels, Languages and Simulation Platforms

In simulation-based verification, a design should be modelled at various *abstraction levels* using some *modelling languages*, and then be simulated on some suitable *simulation platforms*.

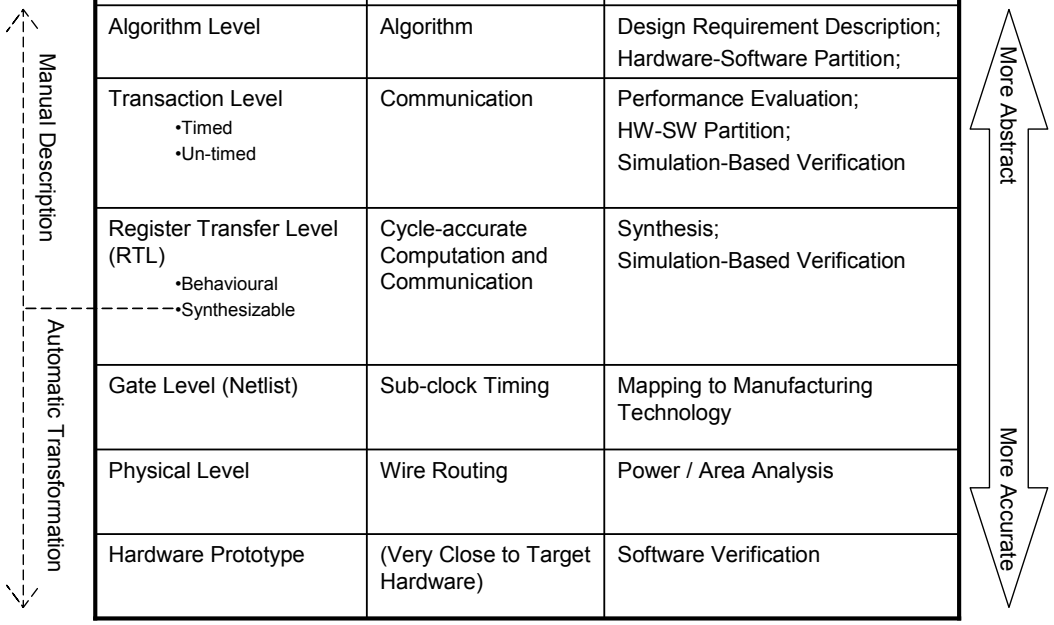
The term "simulation" suggests two worlds: (a) the world *in* which the model is simulated, or the *real world*; and (b) the world *for* which the model is simulated, or the simulated world, or the *virtual world*. The virtual world *abstracts away* some details of the real world phenomena. The term "abstraction level" refers to the degree of the omissions of details from the real world. The virtual world at each abstraction level captures certain parameters of the real world design. Each abstraction level has its suitable application. For instance,

- a register-transfer level (RTL) model is the level where "synthesisable" designs are described; and
- a transaction-level model (TLM) is rapidly gaining popularity in modelling test-bench components, whose behaviours are more abstract than a design-under-test (DUT).

Table 2.2 lists the main abstraction levels used in a design cycle with their applications, in a decreasing order of abstraction level.

Each level may have further divisions. For instance, an "RTL model" could either be behavioural or synthesisable. A synthesisable RTL model is very critical – it is accurate enough to be *automatically* transformed into models at lower levels (using a series of commercial tools), and also abstract enough to be composed by human in computer languages known as hardware-description-languages (HDLs).

Synthesisable models and more abstract ones need to be manually described in certain computer *languages*, such as Verilog and VHDL – the two HDLs dominating the RTL modelling. Different abstraction level models may require different languages; but there are no strict rules. For example, a synthesisable RTL model is described using a subset of HDL constructs called "the synthesisable subset"; it would be poorly productive to use this subset – sometimes even the full set of HDL constructs – to describe more abstract models for verification purpose. Hardware-Verification Languages (HVLs) such as *e* [55], OpenVera [82], SystemC [59] have been introduced in the past decade for verification purpose. More general-purposed languages such as C, C++ are also frequently used for early design validation, especially before a design is implemented in HDL. In recent years, SystemVerilog (SV) [62], a superset of



Abstraction Level	Phenomena Modelled	Application
Algorithm Level	Algorithm	Design Requirement Description; Hardware-Software Partition;
Transaction Level •Timed •Un-timed	Communication	Performance Evaluation; HW-SW Partition; Simulation-Based Verification
Register Transfer Level (RTL) •Behavioural •Synthesizable	Cycle-accurate Computation and Communication	Synthesis; Simulation-Based Verification
Gate Level (Netlist)	Sub-clock Timing	Mapping to Manufacturing Technology
Physical Level	Wire Routing	Power / Area Analysis
Hardware Prototype	(Very Close to Target Hardware)	Software Verification

TABLE 2.2: Abstraction levels. Synthesisable RTL is very critical since lower levels can be obtained automatically using commercial synthesis tools, while models at higher levels essentially require manual effort. Simulation is essentially performed at this level.

Verilog language constructs has recently been ratified as one of IEEE standards (IEEE Std 1800^{TM}-2005 [48]); it appears to be the ideal language since it combines the advantages of HDLs and HVLs.

The introduction of HVLs is an important step in the evolution of simulation-based verification. To simulate the behaviours of a DUT, a test-bench (TB) needs to be constructed. TB's responsibilities are much more abstract than the DUT's; HVLs ideally fit in the position of TB construction by providing the following constructs or capabilities:

(1) constrained randomisation mechanism – for automatic concretising abstract test-cases;

(2) property and assertion constructs – for error-capture and supporting formal verification;

(3) coverage constructs – for automatic coverage collection;

(4) messaging mechanism – for systematic logging as well as high-level control;

(5) support for transaction-level-modelling (TLM) – for convenient connection of TB components;

(6) object-oriented programming (OOP) capability – for increasing TB reusability.

From the academic point of view, HDLs are powerful enough to construct any test-benches. However, compared with HVLs, HDLs are not convenient to compose complex test-benches in the sense that the user needs to do a lot of programming at low abstraction-level; and the outcome of the programming could hardly be reused *across abstraction levels*. Nevertheless, HVLs also have limitations; they are proprietary languages, having reusability issues *across simulation tools*. Recently, the trend is that traditional HDLs are extended to include features commonly found in HVLs to become the so-called "hardware-description-and-verification-languages" (HDVLs), which all EDA vendors agree to support. SystemVerilog, extending Verilog, is such a language. It is very likely that vendor-dependent HVLs will be replaced by HDVLs as they mature.

While new languages provide convenient constructs, and each language has its own strength [13], it should be noted that no language provides answers to the verification problem in itself. Cohen [27] observed that the term "verification" refers to a generic concept rather than language-oriented technologies, and that languages are just "tools" but not the "methodologies".

The choice of languages happens in the *modelling stage*; for the *simulation stage*, a suitable "simulation platform" needs to be selected. Table 2.3 lists some simulation platforms and their simulation speeds.

Commercially available simulation platforms are powerful tools that allow models at different abstraction levels and/or in different modelling languages to run simultaneously. This flexibility permits the trade-off between simulation speed and simulation accuracy to suit different applications. Some of these trade-off options are as following.

(1) Model the DUT in HDL and model the TB in HVL. This is widely used in component-level verification.

(2) Model DUT components that were already extensively verified (e.g., commercially available ones) as abstract TLMs, and model those insufficiently verified as RT-Level models.

(3) Similar to the above, but load the extensively verified portion onto the field-programmable-gate-array (FPGA) and leave insufficiently verified portion as RTL models. This technique is called *hardware acceleration*, the focus is on the RTL side.

(4) Load the entire design onto the FPGA to get fastest simulation speed. This type of simulation is called *emulation*. Emulation allows for the verification of the application-software.

Simulation Platform	Description	Typical Speed	Application
Event Driven	Simulation driven by signal changes	10^2 ~10^5 Hz	RTL verification
Cycle Accurate	Events synchronized at each DUT cycle	10^4 ~10^6 Hz	Synchronous DUT only
Hardware Acceleration	Part of the DUT realized as true hardware (FPGA)	10^5 ~10^6 Hz	Accelerating RTL simulation
Hardware Prototyping	Entire DUT realized as true hardware (FPGA)	10^6 ~10^8 Hz	DUT SW verification
Co-simulation	DUT parts simulated on separate platforms	10^4 ~10^6 Hz	DUT SW verification

TABLE 2.3: Comparison of simulation platforms.

(5) When full design emulation is not possible due to FPGA capacity, load the insufficiently verified components onto FPGA but leave the others, usually including the CPU, as abstract soft models. Like (3), this practice features a partial FPGA loading; but the motivation is more like that in (4). With proper tool support, this practice is often called "HW-SW co-simulation" or "HW-SW co-verification".

Abstraction levels, hardware modelling languages and simulation platforms are neither absolutely dependent nor absolutely independent. They together form the *infrastructure* for simulation-based verification. Table 2.4 lists some typical combinations.

2.1.2.2 Building Blocks: Test-bench Components and Reusability

To apply tests to a design, a test-bench (TB) must be constructed for the design-under-test (DUT); it is the TB that directly applies tests to the DUT. A test-bench is not a monolithic block, but a collection of organised components. Therefore, a TB and a SoC DUT are similar in the sense that both have internal structures.

Walker [87] states that a *canonical* test-bench should provide the following components:

(1) A test reader: to read the tests and convert them into commands to the driver.

Application	Abstraction Level	Modelling Languages	Simulation Platforms
Bus Protocol Validation	Transaction-level (TL) modelling	Timed HVL	Event-driven Simulation
Component-level Functional Verification	RTL	DUT: Synthesizable HDL TB: Behavioural HDL	Event-driven Simulation
Timing Checking	Gate level	Synthesizable HDL transformed Netlist	Gate-level Simulation
SoC HW-SW Co-verification	Processor: ISS; Rest of SoC: RTL	Processor: C++; Rest of SoC: HDL	Cycle Accurate Simulation
System boot-up test	Hardware Prototype	Synthesizable HDL transformed FPGA implemetation	FPGA Emulation

TABLE 2.4: Common combinations of abstraction-levels, modelling-languages and simulation-platforms.

(2) A driver: to accept the logical commands and drive signals to the DUT and the emulator.

(3) A watcher (or monitor): to observe the DUT physical response and convert it to logical events. A watcher usually incorporates functionalities of logging and measuring.

(4) An emulator: to work as a reference model of the DUT, often called the "golden model" or the "reference model".

(5) A checker: to compare the behaviours of the DUT with that of the emulator and report any mismatches.

Their relations are shown in Figure 2.1.

Just like components in a DUT, components in a TB can be represented at various abstraction levels as well as in various languages, and it is the simulation platform that handles the variety seamlessly. TB components which directly interact with the DUT are usually called bus-functional-models or BFMs; they could be composed at lower abstraction levels in HDLs. Other TB components need to be coded as behavioural models in HDL or transaction level models (TLMs) in HVL, so that they could conveniently handle the abstract test-cases defined by users.

HVLs are developed to facilitate the common requirements in typical test-benches. For instances,

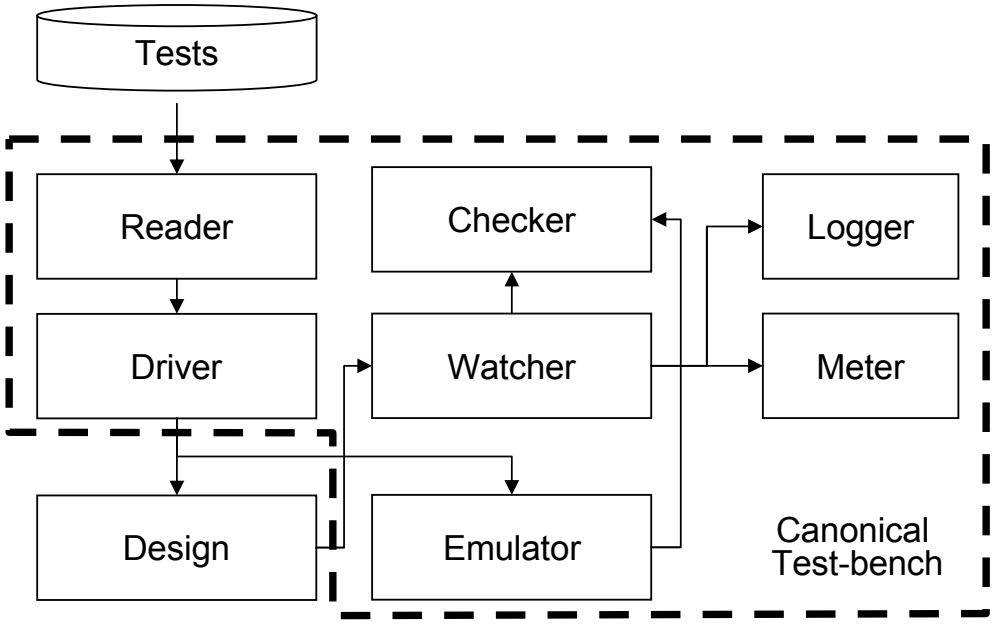


FIGURE 2.1: The canonical test-bench includes various functionalities in simulation, making test-bench construction a time-consuming and error-prone process. Also, *tests* are not a part of test-bench. Additional effort is required for test-generation.

- the constrained-randomisation mechanism in HVLs fits well for the driver;
- the assertion mechanism fits well for the checker;
- the coverage mechanism fits well for the monitor; and
- the messaging mechanism fits well for the requirements of communication between TB components;

The term "test-bench reusability" is a concept with many aspects. In other words, there are many forms of reusability. A trivial one is to reuse TB components from an old project simply because the identical DUT components are used in the new project. Non-trivial forms of reusability include the following:

- reusability across the integration-levels of the DUT. TB components having been used in component-level verification could be *modified and reused* in system-level verification;
- reusability across abstraction levels / modelling languages / simulation platforms. For instance, a DUT component written as a transaction-level-model (TLM) in HVL, which

was created and used in the stage of protocol validation, could be reused as the *reference model* for its RTL implementation;

- reusability of common facilities required by various TB components. Information logging and messaging are such facilities. This kind of reusability is in the form of class inheritance.

The HVLs' contributions to TB reusability include (a) their support to transaction-level-modelling (TLM) and (b) their support to the object-oriented-programming (OOP) paradigm, which respectively contribute to the second and the third type of reusability. In [16], the author describes in details many useful techniques to build reusable test-benches.

Reusable components can become commodities. Reusable *design* components are called "intellectual proprieties" (IPs), while reusable *verification* components are called "verification IPs" (VIPs). Commercialisation of IPs and VIPs respectively contributes to the SoC design paradigm and system-level verification. Silver [78] observed: "Verification IP is becoming much more than a BFM or a passive checker designed to monitor a set of assertions at the interface. Commercial verification IP offers pre-defined compliance tests, inject errors, and drive directed-random traffic through the interface for system-level verification."

However, reusability *alone* is not the solution to the construction of sophisticated test-bench. Reusable TB components simply make it relatively easy to build such a TB. According to [64, 87], reusability is mostly efficient for similar projects, similar components, or in a closely organised design team. It is impossible to implement an entirely reusable verification system [92].

The reason why reusing is never sufficient is rather philosophical. A test-bench tends to treat its DUT as a whole. However, the *whole is greater than the sum of parts*, and a "supra-system taken as a whole displays greater behavioural variety and options than its component-systems" [86]. When we apply this general principle to simulation-based verification, we can predict that at every higher integration-level, even if we could reuse all TB components, we still have to enhance the test-bench with *new* capabilities in order to stimulate and observe the new behaviours/properties. This explains why the test-bench complexity has to grow faster than the design complexity in terms of gate-count. Thus the phenomenon that the test-bench is even larger than the DUT itself, reported in several sources [65, 80], is now understandable and predictable.

Reusable TB components serve as the building-blocks for the control-observation mechanisms of verification.

2.1.2.3 Mechanism and Methodology: Test Generation and Coverage

Simulation-based verification is essentially a *control-and-observe* process; and a test-bench serves as the *vehicle* of control and observation. The test reader and the test driver (in Figure 2.1) are in the control part; while the monitor, the emulator and the checker form the observation part. Human intelligence is indispensable to construct a control-observe mechanism that drives the verification process, which includes (1) test-generation (the control part) and (2) coverage measures (the observation part).

- *Test Generation*: The TB components stimulating a DUT do *not* really address the test generation problem. They simply *convert* test-cases, which are more abstract, to stimuli, which are more detailed. The constrained-randomisation facility provided by state-of-the-art modelling languages only addresses the converting/concretising issues of the transformation. The generation of abstract test-cases is a manual process requiring very specific knowledge about the DUT.
- *Coverage Measures*: Simulation based methods suffer an inherent problem – the test completeness problem. There is no absolute criterion upon which to claim that the DUT has been sufficiently stressed in simulation. What kind of information to be included in coverage measurement and how to interpret the coverage information require human intelligence.

In practice, to maximise the probability to find design bugs in complex designs, *multiple* test-generation methods and *multiple* coverage metrics are needed.

Common coverage metrics include:

- Statement-based metrics, also known as code coverage. They derive from the field of software verification. In this category of coverages, the DUT in HDL is treated as software components. These metrics include (a) line, (b) toggle and (c) conditional coverage (See 6.2.) Statement-based coverages are supported by simulation platforms.
- Functional coverages. They refer to user-defined behaviour space which the DUT should cover. HVLs provide convenient constructs to define functional coverage spaces. Defining a coverage space requires subtle balance between granularity and feasibility. That is to say, the coverage space should be large enough to include important corner cases; meanwhile, the space should be small enough to be practically covered during the verification stage.

Human intelligence is also needed to feed the coverage information (observation) back into test-generation (control), so that the coverage spaces can be traversed quickly. Although coverage information may not perfectly measure verification completeness, traversal on *multiple* coverage spaces can greatly increase the "confidence level" established upon the DUT. The feedback mechanism is important to achieve high confidence level in short time.

The feedback mechanisms come in two flavours:

- online feedback. In this category, stimulus-generation is connected with the coverage information collected *in the same simulation run*, so that a coverage space could be quickly traversed in that simulation run;
- offline feedback. In this category, tests are already generated *before* simulation, and the coverage information is analysed *after* the current simulation run and is used to guide the next round of test generation. The coverage space may require many simulation-runs to be covered, which is the case for a complex DUT.

In short, the mechanism that drives the simulation-based verification is test-generation (control) and coverage (observation), which depend on human creativity and insight.

Given a category of DUT, the implementation of the control-observation mechanism, i.e, the test-generation and the coverage schemes, are referred to as the "methodology" *for that category of DUT*. However there already exist the so-called "verification methodologies" [60, 61, 81, 83] proposed by major EDA vendors. The next section introduces these vendors' methodologies.

2.1.2.4 Paradigm: Test-bench Centric Verification Methodologies

The control-observation mechanism requires a heavy involvement of human creativity; thus, like in many other engineering practices, there comes the need to *standardise* (but not restrict) human's creativity. This is a common requirement from EDA tool users as well as from EDA vendors. The tool users prefer to stick to verification practices that have succeeded before, while the tool vendors need to support their customers through consistent terminologies.

Therefore, EDA tool vendors propose their test-bench construction *conventions*, or *paradigms*, together with pre-programmed TB components and constructs, as open sources to users, under the name of "verification methodologies". The term "methodology" here subtly

differs from the "methodology" we are proposing in this thesis. The former is not bound up with any specific design category, so it should be better understood as "conventions" or "paradigms" to build test-benches; the latter is tightly associated with a certain design category – SoC, and does not focus on TB construction.

Such a vendor's methodology is implemented in certain OOP-capable modelling languages (e.g., SystemVerilog) well adopted by users; such a methodology provides

- prototypes of commonly used TB components in the form of *classes*;
- programming conventions to (a) interconnect TB components, (b) connect TB components with DUT components;
- general principles to build TB at transaction level.

Methodology	**Full Name**	**Modelling Language**	**Affiliation**
RVM	Reference Verification Methodology	OpenVera	Synopsys
VMM	Verification Methodology Manual	SystemVerilog	ARM; Synopsys
AVM	Advanced Verification Methodology	SystemVerilog; SystemC	Mentor
OVM	Open Verification Methodology	SystemVerilog; SystemC	Mentor; Cadence
*e*RM	*e* Reuse Methodology	*e*	Verisity

TABLE 2.5: Some *verification methodologies* have recently emerged. Many of them have rapidly become obsolete. These methodologies are actually conventions and guidelines for test-bench construction endorsed by various electronics-design-automation (EDA) vendors.

The available methodologies are listed in Table 2.5; they are endorsed by different EDA vendors. Armed with the competing simulation tools from the vendors, these methodologies are

involved in the so-called "methodology wars" [61]. There is no consensus on which methodology is superior, and comparing these competing methodologies does not have much academic value. Nevertheless, these methodologies altogether are fostering important concepts in the verification practice, including

- for infrastructure:
  - abstraction-level: adopting transaction-level modelling (TLM);
  - modelling languages: mixing the OOP paradigm with the traditional hardware description paradigm.
- for building blocks:
  - structure: organising TB components into *layered* structures. Figure 2.2 (source from [8]) demonstrates a typical layered TB.
  - reusability: (a) reusing TB components across RTL, TLM and even higher abstraction levels, and (b) reusing common facilities by class inheritance.
- for control-observe mechanism: generating stimuli using the constrained-randomisation facility, which could be fed-back with the online information from functional coverage.

Among these concepts, the idea to increase TB reusability by *elevating* abstraction level is one hallmark of these methodologies. However, as mentioned in Section 2.1.2.2, "reusability" comes in different forms, not just limited to elevating abstraction level. In fact, increasing TB abstraction levels destroys TB's synthesisability – another form of reusability. Section 5.3.2 discusses why TB synthesisability is important. Our methodology supports this form of reusability.

Layered TB structure is another important aspect of mainstream verification methodologies. Figure 2.2 (sourced from [17]) is the layered TB recommended in VMM. (Same concept can be found in [16, 35, 60])

- Signal Layer: including *interfaces* that provides signal-level connectivity to the DUT;
- Command Layer: including BFMs that drives/observes DUT (via interfaces) for *atomic* operations such as read and write.
- Function Layer: generating functional *transactions* to the command layer; providing certain degree of abstraction level for higher layers;

- Scenario Layer: generating *transaction sequences*; managing the function layer (and its sub-layers).
- Test layer: including user-defined applications of the services provided by lower levels.

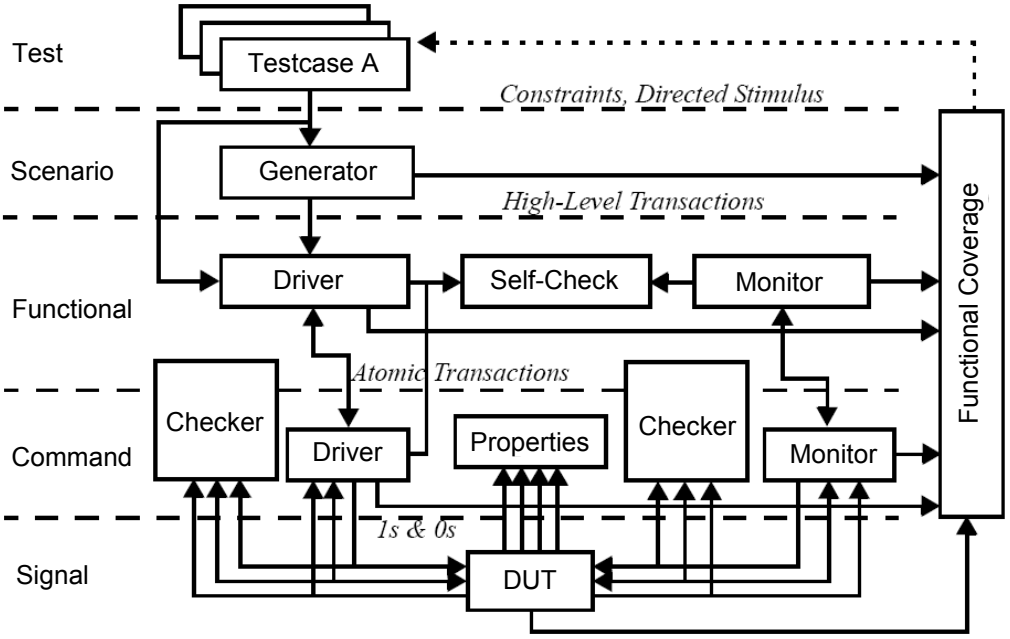


FIGURE 2.2: The layered test-bench structure recommended by Synopsys' verification methodology manual (VMM). Various facilities are roughly organised in layers. Timing is not taken into account at functional and higher levels. However, such a sophisticated structure is still the *vehicle* to apply abstract "tests", which are assumed already available. Adapted from [17].

According to [60], these layers form (1) the *operational domain* (the lower layers), where TB components are timed models working closely with the DUT HW, and (2) the *analytical domain* (the higher layers), where TB components work as ordinary objects in general-purposed software programming, facilitating the control-observation mechanism.

Organising TB in layers is an empirical practice – the rule to separate layers is not clear, especially for the higher layers.

2.1.3 Formal Verification

While simulation-based methods continue to work as the main verification approach, formal methods prove to be a useful supplement. However, the equivalence-checking (EC) technique

[36] finds its application mostly in proving the *equivalence between RTL model and gate-level model* and does not help function verification at RT-Level, since it assumes that the RTL model is correct. Formal verification mostly refers to the model-checking (MC) technique [10], which can *exhaustively* check if a control-intensive design satisfies or violates user-defined properties.

More specifically, model-checking is a category of methods which automatically prove finite-state-machine (FSM) against *temporal properties* by the means of state-space traversing. Figure 2.3 shows how model-checking works. A user provides both (a) the FSM model of the design and (b) the temporal properties to be proved or disproved. The model checker takes the inputs and gives result for each property. (a) If the property holds, it gives a simple yes answer; (b) otherwise, it gives counter-examples that violate the property.

The desired properties need to be formally expressed in *temporal logic* by users. One temporal logic is *enhanced computation-tree-logic* or CTL* [10]. CTL and linear-temporal-logic (LTL) as two subsets of CTL*. A temporal property in CTL* is a hierarchy of sub-properties, which can eventually be decomposed into the following elements.

- Atomic Propositions. They are *non-temporal properties associated with states*, and cannot be further decomposed.
- Boolean Combinators. They are classical logic operators: conjunction $\wedge$, disjunction $\vee$ and negation $\neg$, applicable to both temporal and non-temporal properties. They can derive other operations such as logical implication $\Rightarrow$.
- Temporal Combinators, including `X` and `U`.
  - `X` means "next"; expression `X`P specifies that the next state satisfies the property P;
  - `U` means "until"; expression P`U`Q represents that property P holds from the current state until a future state satisfies Q.

  Other temporal combinators such as "future" `F`, "always" `G` and "weak until" `W` are derived from `X` and `U`. A property immediately constructed with temporal combinators is *associated with a state-path* rather than with a single state.
- Path Quantifiers `E` and `A`. A property immediately constructed with path quantifiers is associated with one state.
  - `E` means "there exists a path"; expression `E`P represents that there exists a path (from the current state) where property P holds;

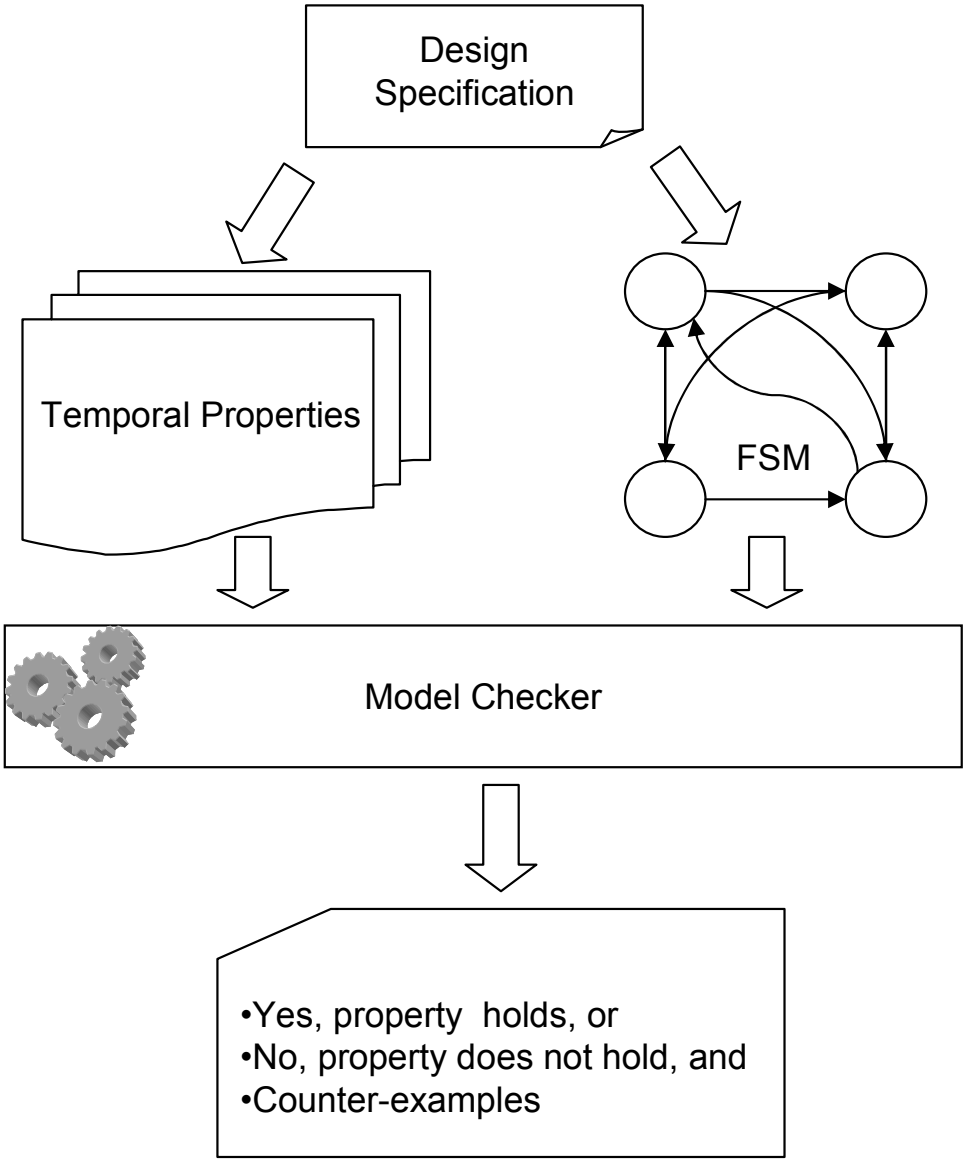


FIGURE 2.3: Using the model-checking technique for property proving/disproving. Users should prepare the design as a finite-state-machine (FSM) model and also provide the desired properties in *temporal logic*. Given an FSM and a property of that FSM, the model-checker gives a definite yes/no answer. Counter-examples will be provided if the answer is no.

– A means "for all paths"; expression AP represents that for all paths (from the current state) the property P holds.

A user combines these elements to specify desired properties for a design; the commonly desired properties for a design include the following.

- Reachability, typically in the form of EFP;
- Safety, typically in the form of A($\neg$*Response* W*Request*);
- Liveness, typically in the form of AG(*Request* $\Rightarrow$ AF*Response*);
- Deadlock-freeness, in the form of AGEXtrue, i.e., always having successor state;
- Fairness, typically in the form of A(GF$User1 \wedge$ GF$User2$).

Set-operations, including *intersection*, *union*, *complement* and *FSM state-space-traversing*, form the heart of model-checking. Suppose that an atomic proposition Q is nested in a temporal property P, then the state-set where P holds, denoted as $Sat(P)$, could be computed iteratively using set-operations, from the set $Sat(Q)$ (which is trivial to get since Q is atomic). Then, if the underlying FSM M's initial states are in $Sat(P)$, then we could claim that property P is satisfied on the machine M.

Binary-decision-diagram (BDD) based model-checkers use ordered-reduced-binary-decision-diagram (ORBDD, usually further shortened to BDD) [22] to represent a state-space *implicitly* and *canonically*. The implicitness means that the BDD can represent a huge state space without memorising any specific state. Then the set-operations become the operations on BDDs. The canonicity means that there is only one possible ORBDD to represent a space; this fact makes BDD operations very efficient. BDD-based model-checking is also known as symbolic model-checking.

However, BDD may still suffer from the state-explosion problems due to the fact that its efficiency heavily depends on the *order* of state-variables. Searching the optimal order is computationally expensive. Another type of model-checking is boolean satisfiability (SAT) based methods, in which, the temporal properties are limited within k time frames. Therefore, this category is called bounded model-checking (BMC) [26]. In BMC, the input, state and output of the FSM in k consecutive time-frames are unfolded and then coupled with the property logic to form a SAT instance, i.e., a combinational logic with one output. Figure 2.4 demonstrates this technique. The unfolded input, output and state are fed to the property logic P, whose final output is negated. If this SAT instance is not satisfiable, then the

property P is proved to be true; otherwise, any assignment to the initial state s_0 and the unfolded inputs is a counter-example to the property.

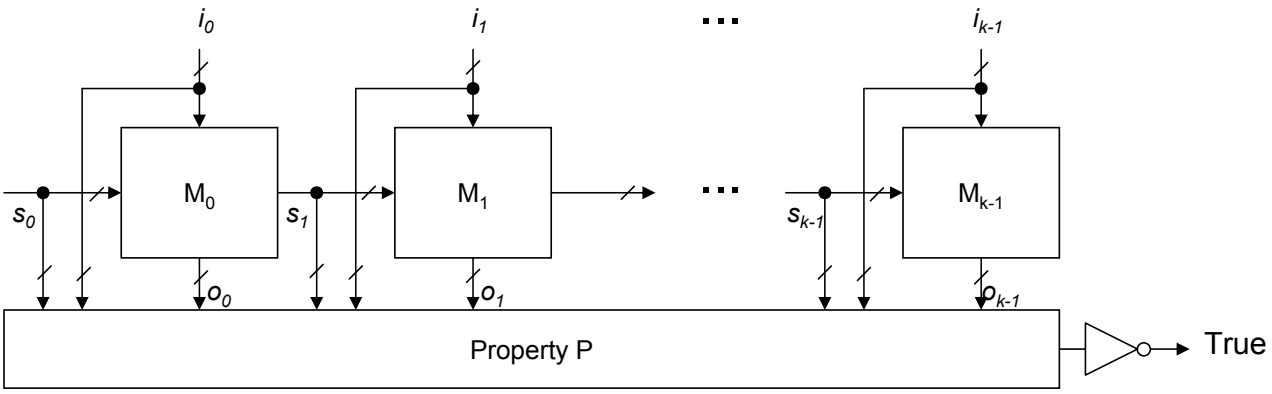


FIGURE 2.4: Satisfiability-based bounded model checking (BMC). Temporal properties are assumed not exceeding k cycles; then, the finite-state-machine is unfolded k times to cascade into a wide combinational logic, whose outputs are used to specify the k-bounded temporal property. The output of the property logic is negated and assumed to be `True`; if the model-checker fails to find any assignment of $s_0, i_0, ...i_{k-1}$ to satisfy the negated property, the property is proved; otherwise, the assignment is the counter-example to the desired property.

Given a property P, the advantage of the formal method over simulating testcases for P is that the former attempts to *exhaustively* check the paths leading to P. However, this kind of exhaustiveness only applies to P; it should not be mis-interpreted as the verification completeness. The verification completeness problem for formal methods is to decide whether enough properties have been specified by users [52]. The limitations of formal methods also include:

- The dilemma to provide a FSM model. An very abstract FSM may not accurately describe a design; a very detailed one may exceed the model-checker's capability.
- The difficulties to provide temporal properties. A user may provide incorrect or insufficient properties; and there can be gaps between the desired properties and the expressivity of the temporal logic accepted by the model-checker.

In addition to proving or disproving temporal properties formally, model-checkers could also serve as test-generators for the simulation-based method [15, 42], due to its capability to generate counter-examples. A counter-example is an input sequence that violates a temporal property. If we feed a model-checker with the negation of a property P for which we want to

generate tests, a counter-example to the negation $\neg P$ is actually an input sequence leading to property P. This technique is usually applied to the complex control-intensive logic (e.g., a RISC processor's pipeline control), where direct property-proving may not be practical. This technique is used to generate the elusive input sequence that would result in very rare corner-cases. In practice, however, the user need to perform non-trivial job to transform the counter-examples into executable stimuli.

2.2 System-Level Verification Problem

2.2.1 Overview: System-Level Bugs

At system-level, the *concurrency* or *parallelism*, among multiple components is the defining characteristics of a hardware system. Concurrency forms a new verification dimension. System-level bugs are usually discovered in *corner cases* where parallel processes interfere with each other in an un-expected way. *Resource-competition* is an inevitable consequence of concurrency. HW components could show functional problems when competing with each other for resources, even if they have already passed component-level verification. Listed in [64], unique bugs at system-level include:

- Interactions between blocks that are assumed verified;
- Conflicts in accessing shared resources;
- Arbitration problems and dead locks;
- Priority conflicts in exception handling;
- Unexpected hardware/software sequences.

All these bugs are related to component-to-component interactions, especially to concurrent interactions with resource competitions. Therefore the key to system-level verification is to construct concurrency.

Currently, neither formal methods nor general simulation-based approaches are dealing with the *system-level* behaviours such as concurrency/resource-competition satisfactorily.

2.2.2 Formal Methods: Not in the Position

Formal methods are simply not in the position to discover system-level bugs due to the nature of these bugs, so the industry is depending less on formal methods [11, 45].

- Bugs caused by implementation details: system-level bugs could arise from an inaccurate or a mis-interpreted design specification, *as well as* from the detailed implementation of that specification. Formal methods may suit well for the former, in which implementation details could be abstracted away. However, if the design is represented as an FSM with implementation details, it can easily choke the model-checkers.
- Control- and data-intensive failures: system-level bugs often arise in scenarios in which data-intensive and control-intensive behaviours are loosely intertwined; whereas formal methods work best with control-intensive applications.
- Failures across components: it is often impossible to attribute a system-level bug to a *particular* hardware component; instead, the bug may be caused by the ill-matched behaviours of multiple components [64]. It will be very difficult and unscalable for formal methods to deal with combined or communicating FSMs.

More importantly, the fact that the user is responsible to provide properties to formal tools is the fundamental barrier to applying formal methods on *system-level* verification. A hardware system, which is made of multiple components and may be represented with implementation details, *do not have fixed failure modes*. Therefore, verification engineers are constantly in a dilemma of "expecting something unexpected". As a consequence, they cannot postulate those properties that they are yet to know.

2.2.3 Simulation: DUT-TB Dualism

System level verification substantially relies on the simulation approach, in which tests are applied to the design-under-test (DUT) via a structure called *test-bench* (TB). However, the current practices centring on test-bench (TB) construction has introduced and even enforced the "DUT-TB dualism".

Figure 2.5 illustrates this dualism – the TB *stimulates* the DUT and *observes* the response from the exterior of the DUT. The dualism also manifests itself in the form of the divergence between the techniques to develop DUTs and those to develop TBs. A DUT and its TB are rapidly becoming two distinct entities.

- The languages used to develop a DUT continues to be HDLs. Moreover, for accurate simulation, the DUT should be described in the *synthesisable subset* of HDL constructs. A DUT is largely understood as a *hardware structure in the simulated world*.
- The languages used to develop a TB migrate to HVLs and other object-oriented programming (OOP) capable languages. These languages provide *dynamic* constructs to facilitate *dynamic* connections. However, being dynamic also means the loss of synthesisability. The state-of-the-art TBs require OOP paradigm or even beyond [16]. In this sense, a TB is more a *software phenomenon in the real world* than a hardware structure in the simulated world [60].

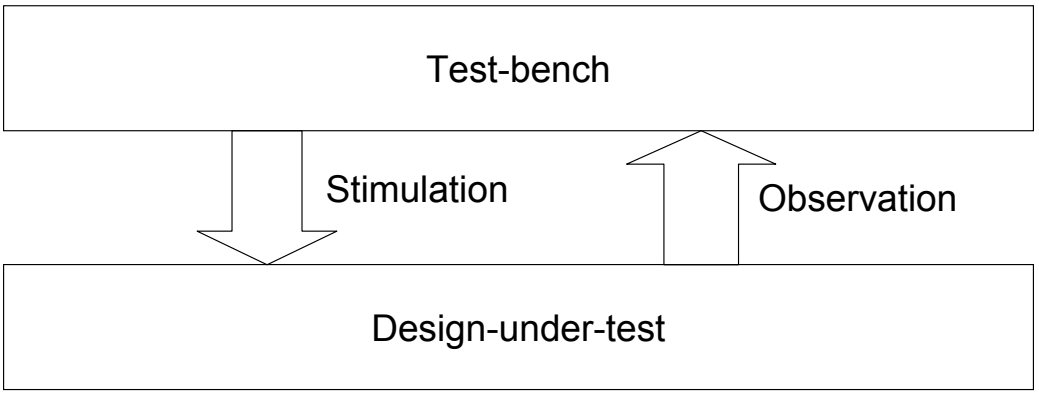


FIGURE 2.5: Test-bench (TB) stimulates and observes design-under-test (DUT) from the exterior of DUT, forming the "TB-DUT dualism". The TB and the DUT are becoming very different entities; the technique to build the DUT and the technique to build the TB are diverging rapidly.

Although this dualism appears reasonable and proves fruitful in component-level verification, relying on the TB *alone* is inherently flawed in controlling and observing system-level behaviours. The following complications arise from the dualism.

The distinction between external and internal behaviours. As suggested in Figure 2.5, the TB tends to treat the DUT as a black box. It applies stimulation and observation from the *exterior* of a DUT, so it is inherently problematic to force the TB to control and observe the DUT's internal behaviours. There are *white-box* approaches, i.e., adding control and observation points around components inside a DUT, to supplement the black-box approach. However, we could argue that this approach is still black-box natured in the sense that the similar controllability/observability issues still exist at component level.

The soaring TB complexity. As mentioned in 2.1.2.2, when components are integrated into a system, *new* capabilities has to be added in the TB to test the emerging properties caused by the integration. That is why TB complexity could grow faster than DUT complexity. Two TB construction principles, i.e., (a) layered TB structure and (b) reusing TB components [17, 60] could only mitigate but not solve the problem. This complexity issue will eventually prohibit us from relying on TB *alone* to verify a more complex DUT. We further argue that the construction of sophisticated TBs has actually *defocused* the attention on the DUT itself. Verification engineers are easily trapped in the painstaking process of *TB development*, and distracted away from the more creative task of *testcase generation*.

The test generation problem. A test-bench is essentially the *vehicle* to transform abstract testcases into physical stimuli to the DUT. The TB does not really handle testcase generation; it always assumes that abstract testcases are already available. In a layered TB, each layer is simply responsible for *transforming* tests from a more abstract form to a more detailed one. The best stimuli-generation mechanism provided by state-of-the-art HVLs is merely "constrained randomisation", which is largely independent of the central characteristics of a system, namely, the concurrency and the associated resource-competition; and it is mostly used to concretise abstract testcases. It is still human who is responsible for providing abstract testcases in the form that the TB understands. In this sense, TB-centric verification methodologies help to increase the users' *productivity* in TB construction; but by no means could they replace the users' *creativity*.

In a word, the DUT-TB dualism creates serious complications for system-level verification. The TB tends to be very complex to take control and observation responsibilities; but in the end, it still does not touch the central question of test generation.

Then how to evaluate the EDA vendors' TB-centric methodologies? Many regard that the emergence of these methodologies *allow* users to construct complex TBs for complex DUTs. However, this is in fact a typical "chicken-and-egg" paradox and could be interpreted in the opposite way – we could argue that it is the complex DUTs that *require* sophisticated TBs, which is the reason why TB-centric methodologies *are required to* emerge. We favour the second interpretation and regard these methodologies as the efforts to simplify *the process* to build sophisticated test-benches, not the effort to simplify *test-benches themselves*. In contrast, our approach attempts to shift the centre of SoC verification away from test-bench construction to software construction. Software native to the DUT has actually *overturned* the traditional "DUT-TB dualism".

2.2.4 Software: the Third Entity

Software (SW) could be responsible for the majority of the SoC functionalities, but software does not have a proper position in TB-centric verification methodologies. Although software is involved in VLSI design/manufacturing in several ways as described below, none of them is significantly contributing to SoC integration verification.

- SW-based verification is naturally used in processor verification [14, 30, 31, 32, 40, 47, 50, 56, 57, 75]. In this application, SW is organised at instruction-level and usually targets at *micro-architecture* of the processor-under-test. Therefore, this category of verification methodologies does not apply to system-level verification.
- SW-based tests are also found in the area of SoC manufacturing-testing [6, 68]. Since the driving force of design verification and manufacturing test are substantially different, those methods shed limited light on the area of SoC verification. The former targets at RTL design bugs, while the latter targets at the gate *stuck-at* fault model.
- The idea of "HW/SW co-verification" is practiced as running an operating system (OS) and application software on a SoC model for the purpose of software verification. Therefore, running these software components is the "liability" rather than the "asset" to the hardware team. The "HW/SW co-verification" concept stays at infrastructure level, i.e., simulation performance level [23, 66] and lacks methodological support.
- SW in the form of hardware diagnostics programs could be interpreted as the "asset" to SoC verification. However, these diagnostics (a) are either too simplistic or too specific, and (b) are poorly automated and require manual development. So using this form of software cannot serve as a major verification approach.

Software is the valid *third entity* alongside the DUT and the TB. For an SoC DUT, it is common practice for a verification engineer to write tests in the form of software snippets. The languages (C and assembly) used in these snippets are native to the DUT, not native to the simulation environment as the test-bench is. These software snippets typically write configurations to control registers, and read the status from the status-registers. This common practice actually demonstrates software's capabilities in controlling and observing a DUT. Although writing testcases in software is often treated as an *ad-hoc* verification technique, we should realise that the introduction of software in hardware verification has *overturned* the traditional TB-DUT dualism.

Software, as the third entity, seems to cause further complications in addition to those already caused by the TB-DUT dualism. For instance, it is unclear how to position the software and the test-bench properly in a verification framework, as suggested in Figure 2.6.

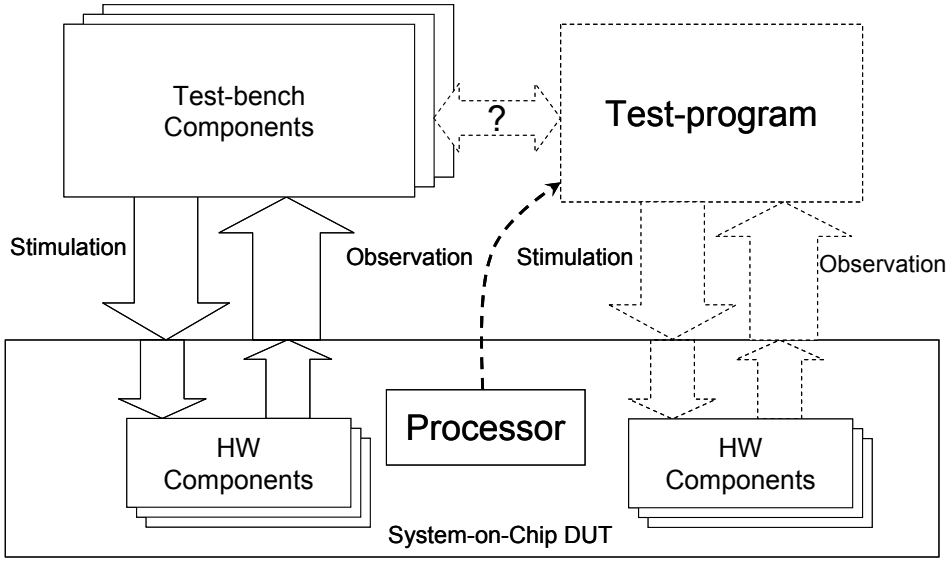


FIGURE 2.6: For a system-on-chip (SoC), stimulation and observation could be provided either by the test-bench (TB) or by the test-program (TP). TB controls and observes the DUT through signals; The TP controls and observes the DUT through WRTIE/READ operations and interrupts. The relation between the TB and the TP is not clear, causing conceptual confusions in SoC verification.

This problem cannot be addressed by TB-centric verification methodologies. In fact, by treating TB-construction as the *software phenomena in the real world*, (for example, adopting OOP paradigm in TB construction), these TB-centric methodologies neglect the *software phenomena in the virtual world*. In the sequel, we will refer the software in the virtual world also as "test-program" (TP) to contrast with the term "test-bench" (TB).

In addition to the unclear TB-TP relationship, how to generate testcases and organise them in TP is another open question. This problem is not answered even by some SW-centric verification approaches [33, 39, 41]. For example, in [39], Hunsinger *et al.* briefly compared the test-bench approach with the software approach. Regarding the latter, he proposed to introduce custom operating systems on top of hardware. This idea attempts to resolve the difficulties at physical-level programming. But it does not give answers to how to produce high-level TPs; instead it still suggests that the testcases should be based on *pre-defined* test rules.

There are also common pitfalls in utilising TPs for system-level verification. The essentially *sequential* execution of software could obscure the inherent *parallelism* in a system. It is a common mistake to view system behaviours as the *sequential* execution of instructions, rather than a collection of parallel activities. Also, existing TP generation methods [21, 24, 25] tend to overly focus on non-system-level issues such as statement-based coverages and program size.

2.3 Our Solution: Software-Centric and Interaction-Oriented Verification

The problematic relationship between (1) an SoC, (2) its test-bench and (3) the test-program causes complication and confusion in system-level verification. On the other hand, software, as the third entity, also brings the potential to address the complication and confusion altogether. The test-bench and the test-program should be placed more naturally in the verification framework. We propose two ideas for SoC verification.

Test-Program Centric Verification. The test-program, instead of the test-bench, should take the more *active* role of testcase *control*, especially parallelism management; the test-bench, not the test-program, should take the relatively *passive observation roles*.

Interaction-Oriented Verification. The *object-under-test* should be the *interactions* among components, rather than the components themselves.

2.3.1 TP-centric Verification: Reshaping the Verification Framework

While the TP has overturned the DUT-TB dualism, the relationship between TP and TB is seldom discussed explicitly. In [9], the proposed relation is that "TB controls/observes TP". Here, "TP" refers to diagnostic subroutines for HW verification purpose. The behaviours of TPs are monitored and *intercepted* by a TB. For instance, the TB could intercept a TP's read-access at a certain memory location, and modify the read data with a value generated by the TB's randomisation mechanism. Thus, the TP always gets random data from this memory location, which could be treated by TP as a random data source to configure HW components. However, using the TB to control a TP is un-natural in the sense that a

programmer's *original intention* encoded in the TP is damaged. Therefore the applicability of using the TB to control a TP would be narrow. For the above specific example, the TP could obtain a similar effect (without the TB's intervention) by accessing not a fixed memory location but an array of pre-decided random values.

The more natural TP-TB relation should be exactly the opposite: TP controls TB [38, 53, 89]. In [38], the TP-TB synchronisation is manually (and statically) specified but automatically implemented. In a test file, testcases in the form of SW snippets are *annotated* with desired TB behaviour. A custom parser reads the test file and associates snippets' program addresses with their annotated TB behaviours. At simulation, the TB is sensitive to the program-counter (PC) in the processor. Whenever the PC matches any address that has been identified by the parser, the associated TB behaviour is triggered. In this sense, the TB is under the control of the TP. This approach is useful but may not be sufficient to support more flexible TP-TB communication such as run-time parameter-passing.

It is better to let the TP control the TB more explicitly. The TB and the DUT are essentially hardware; indeed, the TB can be composed in HDL just like the DUT, and has similar reusability issue. Since a TP can control/observe the DUT through the DUT's "programming interface", namely, its control/status registers, it makes sense to allow the TP to communicate with the TB in a similar fashion. In [51], a patented idea is to connect all TB components using a central bus dedicated to verification, just as an SoC is integrated around some interconnection mechanism. If the TB is controlled by a TP as the DUT is, from the TP's point of view, the *dualism* between the "TB" and the "DUT" is reduced – there is only *one* type of entity, namely, SW-controllable HW components. Indeed, counting a HW component on the "TB" side or on the "DUT" side is not absolutely necessary. A good example is the SoC's processor, which could be represented either by an accurate but slow HDL model, or by a less accurate but faster instruction-set-simulator (ISS). Counting the processor on the DUT side or on the TB side is simply a matter of interpretation. If we do not distinguish between DUT and TB, the distinction between DUT's "external" and "internal" behaviours also loses significance, for everything now happens within the SW-controllable "DUT-TB super-system" (detailed in Chapter 5).

If we adopt this view, the TP actually moves to a more critical position than the TB. The entire verification framework could centre on TP's generation and automation, which substantially differs from the mainstream TB-centric verification practice.

2.3.2 Interaction-Oriented Verification: Redefining the Object-under-Test

As discussed in the last subsection, the term "design-under-test" (DUT) (as well as "TB") becomes *subjective*. Therefore we shall now *redefine* the term "object-under-test".

At system integration stage, the focus should be put on the interaction between components, rather than the components themselves. This suggests that the *interactions, not components, should be treated as the object-under-tests*. This view is justifiable, since at integration stage a certain degree of confidence in hardware quality should *already* be established. This assumption is especially valid for commercially available components.

Focusing on components' communication is already a common practice in SoC design/verification, such as "transaction-level modelling" (TLM) [60]. The central idea in TLM is to separate a component's computation capability from its communication capability. Nevertheless, the term "communication" is still viewed as the capability, or property, *attached* to a component [67]. The communication property is modelled as a task member attached to a component-object. It takes the form of

$$Component.Communicate(Transaction).$$

where

Component represents a hardware component;

Communicate() represents a member-task (or member-function) associated with the hardware component, and

Transaction represents the data content to be communicated.

To allow convenient connection between components in the test-bench, more components are introduced. These additional components include "ports" and "channels", whose *only* capability is to communicate. User components must hold references to ports, which are sub-components of a channel. This technique is shown in Figure 2.7. In this scheme, a user component communicates with other components via a channel typically in the form of

$$Component.Port.Communicate(Transaction).$$

As a benefit, the communicating components do not have to have full knowledge of each other. This connection technique is one major concept in the TB-centric verification methodologies.

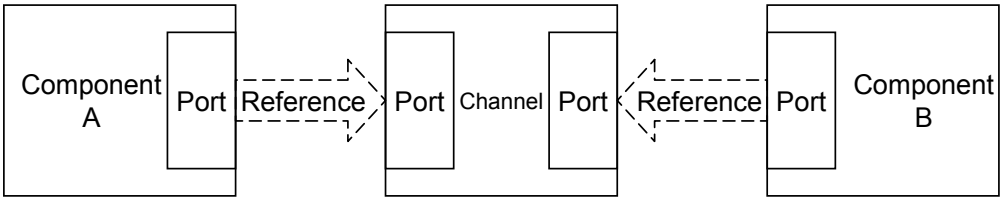


FIGURE 2.7: The object-oriented convention on how to connect components. To connect Component A and Component B, a third components called the *channel* is introduced, which further contains sub-components called "ports". A port is a placeholder for functions that perform the actual data movement. Each of A and B has a reference to the ports of the channel. The nature of communication is function-calling.

By introducing channels and ports, the component-oriented view of a system continues to be valid. However, the hardware verification community has not proceeded to migrate from the view in which interactions or communications are treated as the properties, or "capabilities", *attached* to HW components, to the view in which interactions themselves become a set of objects *independent* from HW components. We respectively refer to these two views as the "component-oriented" mindset and the "interaction-oriented" mindset.

In the component-oriented mindset, "tests" and "objects-under-test" are two *distinct* concepts (even implied by the appellations); the former is something external (and incomparable) to the latter. As a result, the component-oriented mindset inevitably faces some fundamental questions that cannot be answered by the mindset itself; these questions include: (a) what can be counted as a "test" to a component, and (b) how to generate those tests for the components.

In the interaction-oriented mindset, this "test vs. object-under-test" distinction disappears – the terms "tests" and "objects-under-test" are now referring to *one* type of entities, namely, the *interactions*. The "test generation" issue now is transformed into interaction identification issue. Although identifying interaction objects still requires non-trivial effort, the conceptual complications caused by the "test vs. object-under-test distinction" are removed.

Moreover, this mindset is coherent with the concept of *concurrency*, or *parallelism*. Treating interaction as objects, test generation now could focus on combining simpler interaction-objects into more complex scenarios in which parallel interactions compete for resources.

In a pure interaction-oriented mindset, the actual HW components' significance is reduced. If components' properties are reorganised into interactions' properties, they could even be abstracted away. The philosophical implication of interaction-oriented mindset can be found in many areas of studies. Milner [63] observed that "it is reasonable to define the behaviours

of a system to be nothing more or less than its entire capability of communication". In [86], the author quoted: "Besides *substances*, there are *processes* ... The universe should be better understood as a set of processes rather than a set of substances." Indeed, even a so-called substance is not a static matter – it is a process of being.

The interaction-oriented mindset also brings an opportunity to identify the *system-level* verification complexity. In order to mitigate the soaring verification complexity, the verification community has long been attempting to implement the intuitive idea of reducing system-level verification complexity from the *cross-product* of verifying all components, i.e.,

$$\prod C_i,$$

to the *summation* plus the *system-level* verification complexity, i.e.,

$$\sum C_i + C_{sys},$$

where the $\sum C_i$ part is already performed during component-level verification stage [74]. Treating interactions as objects-under-test is an implementation of this idea. The system-level verification complexity C_{sys} now can be expressed in terms of interaction-objects' properties, including the *temporal relations* of interactions.

2.3.3 Combining TP-Centric Approach and Interaction-Oriented Mindset

A TP-centric verification methodology is not necessarily interaction-oriented. A TP-centric but component-oriented method implies that TPs are developed to diagnose hardware components. Since testcase development for each hardware component must be very specific, we see little opportunity for automation here. One example of TP-centric but component-oriented method is described in [33], in which some lower-level software components, called low-level-device-drivers (LLDDs), are oriented at testing each hardware-components, while higher-level software components such as the test-operating-system (TOS) and the test-applications (TAs) are responsible for high-level control, including *test-initialisation*, *multi-tasking* and *result-checking*. Because the scheme is still component-oriented, it suffers critical drawbacks as follows:

- the user needs to hand-write a test-operating-system (TOS), where sophisticated synchronisation mechanism between (TOS, TAs, LLDDs) must be implemented;

- the test-generation problems is not handled; it is the user's responsibility to conceive interesting testcases of parallelism and to exploit the synchronisation mechanism;
- running the TOS becomes a huge overhead to running TAs and thus need to be done on a separate execution platform, which in turn introduces further complications about the synchronisation between platforms.

Another TP-centric but component-oriented scheme in [41] also attempts to construct parallelism between component-oriented tests by utilising existing OS. This scheme has similar issues as mentioned above, that is, (a) the hardware overhead for running the specific OS, (b) the software overhead for synchronisation and (c) human-conceived parallelism.

If a TP-centric approach is combined with the interaction-oriented mindset, the TP naturally takes the responsibility of *parallelism* management – the very role that the TB struggles to play but hardly satisfactorily in a TB-centric verification approach. Very few researches combine these two concepts. XGEN [28, 29, 37], the system level test generator developed in IBM for verification of high-end server and SoC, explicitly use "interaction" objects as building blocks to generate system level test cases. An *interaction* in XGEN is a series of communication stages known as "acts"; each act is performed by some hardware components (known as "actors"). An interleaving technique is also used to enforce the parallelism [29]. Nevertheless, hardware components are not abstracted away in XGEN. To the contrary, a library of component models, each with its properties modelled in detail, must be already available. Obviously, these part of modelling requires in depth hardware knowledge. Therefore, XGEN is not a purely interaction oriented verification tool.

This thesis proposes a test-program (software) centric and interaction-oriented methodology. The link between "TP-centric verification" and "interaction-oriented verification" is "transfer" – a software controllable interaction-object. The next chapter details the transfer model.

2.4 SoC Used in the Research

Our research is demonstrated on a simple SoC ($\sim 25,000$ lines of Verilog code) generated by the SoPC builder [1], an Altera's product which contains a library of commonly used RT-Level hardware component for general embedded applications. This SoC was also used by Cheng *et al.* in [25] for a related research in the University of Adelaide. Some comparison based on this SoC is described in Section 6.2.

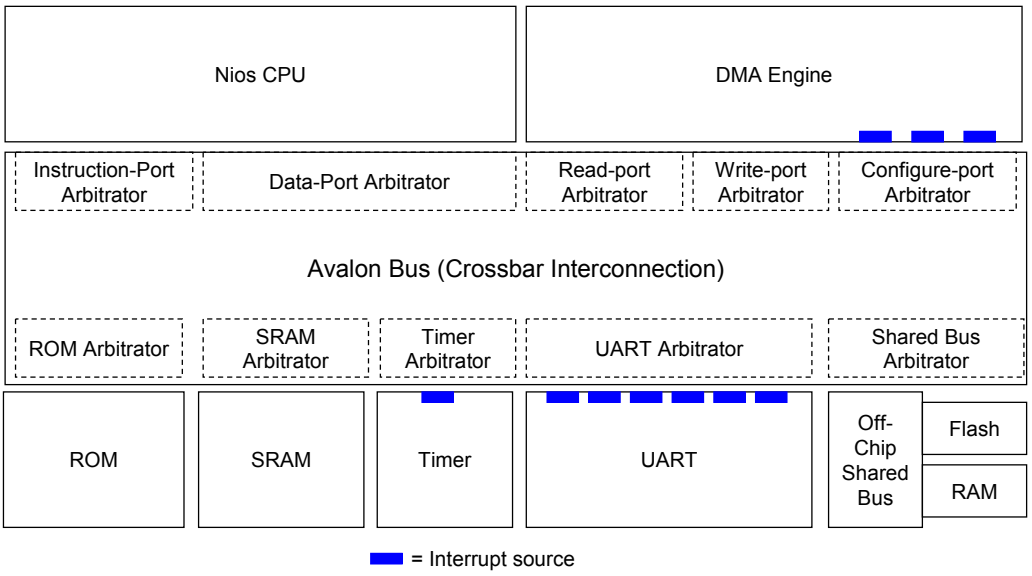


FIGURE 2.8: The System-on-Chip (SoC) under demonstration – the Nios SoC. This simple system possesses typical SoC features: multiple components interconnected by on-chip bus.

The Nios SoC [5] is shown in Figure 2.8.

- The Nios CPU is a typical five-stage pipelined RISC, including an 8KB data-cache and a 8KB instruction cache. The instruction-set contains 54 RISC op-codes. The RISC structure contains a register-window mechanism allowing fast context-switching.
- The DMA (direct-memory-access) engine can perform data transfer between any slaves, including memories and other peripherals. The max word-width is 32-bit.
- The UART (universal-asynchronous-receiver-transmitter) can interact with the external world in bit stream in *full-duplex* mode. The baud-rate is programmable.
- The ROM (read-only memory) stores instructions, size 64 KB.
- The SRAM (static random-access memory) stores data, size 1 MB.
- The RAM and Flash are the additional memory modules.
- The on-chip interconnection is the Avalon Bus [2], which is a cross-bar interconnection. It does not have a central arbitration mechanism. Instead, a mechanism called "slave-side arbitration" is provided for each slave port. There is no arbitration requirement until two masters are visiting the same slave.

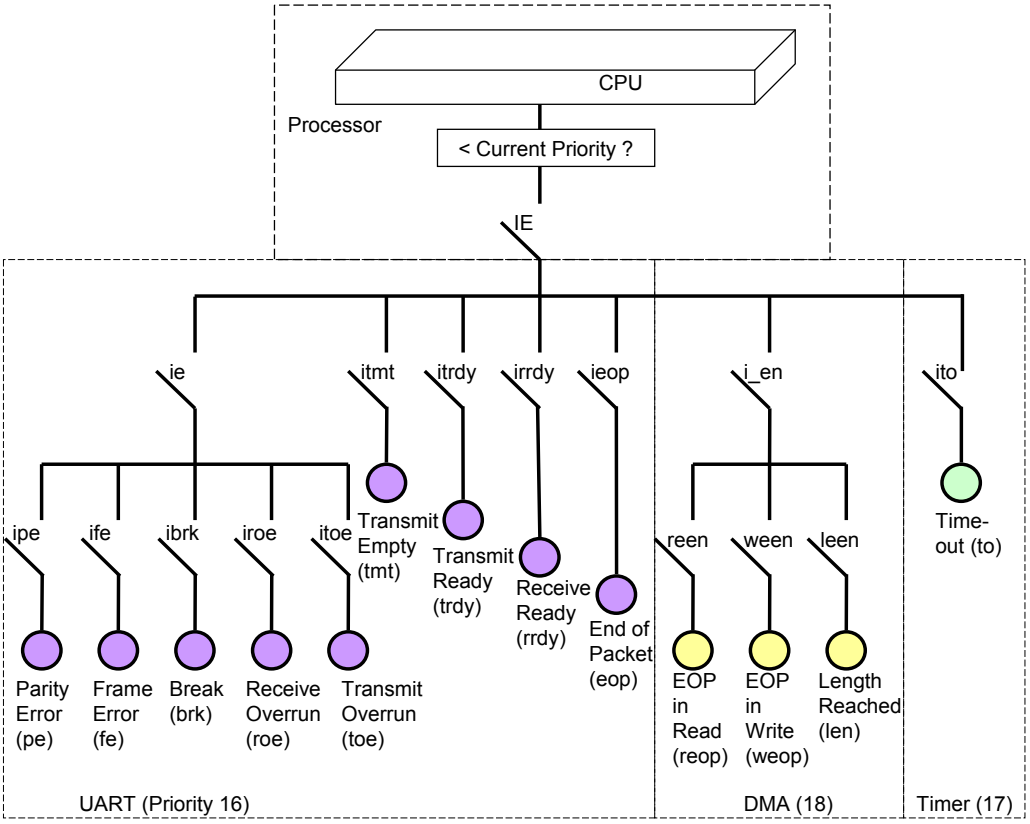


FIGURE 2.9: The interrupt sub-system of the Nios SoC. In case of simultaneous interrupt, the request with lower priority code will be serviced first. Interrupt requests may be disabled at different levels before they finally reach the CPU. Interrupt sources from a same peripheral may be completely independent.

The Nios SoC has an interrupt subsystem as illustrated in Figure 2.9. The DMA, the Timer and the UART each has an interrupt line wired to the CPU with fixed priority-code (DMA > Timer > UART, the smaller the code, the higher the priority). A number of interrupt sources exist across the system, including

- UART Receiver Ready (rrdy) – UART received a character;
- UART Transmitter Ready (trdy) – UART transmitter sent a character;
- UART Status/Errors (end-of-packet; parity, frame, overrun, etc)– Various UART status/errors;
- DMA Done – DMA engine finished transfer;
- Timer time out – the timer's internal countdown-register reaches zero.

In addition to the external interrupt sources, the Nios CPU also contributes two exceptions with highest interrupt priority: (a) Register-window-Underflow-Exception, priority-code 1 and (b) Register-window-Overflow-Exception, priority-code 2, which are raised when the register-window slides across limits. Some complications arise when the interrupt mechanism interfere with the register-window mechanism. These complications together with the solution are described in Appendix B.

This Nios SoC proves to be an ideal choice – it is simple enough for academic research and typical enough to represent a wide range of real-world SoC design. It contains adequate features of a typical system-on-chip: *multiple* components (including the CPU) interconnected by an on-chip interconnection structure. It is also a typical "system" in the very general sense – it has internal structures that could potentially co-operate or conflict with each other; it can communicate with its environment, inputting and outputting simultaneously.

An SoC verification methodology should be *independent* from verifying a specific SoC. In our research, we strictly differentiate between "SoC knowledge" and "methodology knowledge", especially in constructing the test-generator. (Refer to Appendix C for details.) We believe that the software-centric and interaction-oriented verification methodology demonstrated on this Nios SoC is generic enough to cover a large category of SoC designs.

2.5 Summary

This chapter has introduced some basic concepts and commonly adopted techniques in the verification practice. We have shown that the bottleneck to more efficient SoC design verification stems from the lack of theoretical support. It is philosophically problematic to apply the traditional test-bench centric verification methodologies to system-level verification. Furthermore, we have revealed our approach to the problem. Our "interaction-oriented" and "software-centric" verification approach gives holistic consideration to the inherent properties of a "system", including parallelism and hardware-software interactions. The next chapter starts our treatment by introducing an interaction model called transfer.

Chapter 3

Transfer-Resource Graph

3.1 Overview: Proper Abstraction Level

In a system context, the objects under test are *interactions between components*, in place of the components themselves. In contrast to a component, which has a clear boundary, an *interaction* appears to be abstract and shapeless. This chapter deals with some basic questions such as how to *identify* and *characterise* interactions as valid objects under test, and how to combine them to form legal parallelism.

For SoC verification, interactions must be modelled at a *proper* abstraction level. Interactions in a system come in different levels. There are signal-level handshakes; there are logical-level frames/packets/tokens; there are also application-level threads/processes. Lower level interactions aggregate and collaborate to become higher level interactions. The abstraction level for verification should not be too low. This is because we are to implement tests in software known as test-program (TP), which has little controllability and observability of signal-level events (e.g., Bus_Request and Bus_Acknowledge, etc). However, the abstraction level should not be too high either, because TP is supposed to vigorously stress the hardware devices. It is inappropriate yet for a TP to view hardware devices as API (application-program-interface) *services* as for an application-software programmer.

To trade-off the above considerations, the model should be readily comprehended by a device-level programmer, who understands hardware functionalities and performances, but may have little knowledge about hardware implementation. We use the term "transfer" to refer to interaction at this *specific* abstraction level. We also use the term *resource* to refer to the

hardware resource needed in interactions. Transfers and resources form transfer-resource-graph (TRG), in which testcases of parallelism can be generated.

3.2 Transfer Modelling

3.2.1 Definition of Transfer

Interactions are the focus of system-level verification. There is a challenging issue associated with modelling them, that is, interactions come in various forms, requiring different techniques to stimulate and observe them. Some interaction examples in the Nios SoC are:

(i) A Flash-to-RAM DMA transfer. It is a series of read and write operation driven by the dedicated hardware – the DMA engine;

(ii) The execution of a `sort` subroutine. The subroutine can be viewed as a pattern of memory-access performed by the CPU, driven by the execution of software;

(iii) An incoming bit-stream via the UART receiver; this stream is converted into a byte-stream and finally reaches a memory buffer. This process is mostly driven by the interrupt mechanism.

Note that these three examples are data-flows driven by heterogeneous mechanisms, which are respectively the DMA engine, the `sort` subroutine and the interrupt subsystem. While checking *each* of them is common sense, checking their *parallel* execution will greatly improve test quality, because we are able to observe not only interactions, but also *interference between interactions*. When the above three interaction examples take place in parallel, we are able to observe how the DMA engine and the CPU compete with each other for the bus access, how the UART *interferes with* their competition by frequently interrupting the sort subroutine, and how UART interrupt is nested in DMA interrupt. The concurrent execution of these behaviours brings huge indeterminism at physical level, and some extremely elusive design defects can only be exposed in such complex interaction patterns. However, it will be hard to construct such interesting testcases if we do not overcome the heterogeneity of different interaction forms.

The key to effectively construct such parallelism is to generalise heterogeneous interaction forms into a common model. We call this model *transfer-type*.

Definition 1: *Transfer-type* is a set of *programmer controlled* and *data-intensive* interaction patterns among SoC components. Its *programmer-controlled* feature means that a transfer-type has the following properties:

(i) Configuration: Transfer-types have their own *parameters*, which can be configured by some instructions. (An important part of a transfer-type's configuration is the resources to be used.)

(ii) Invocation: Transfers can be invoked by some instructions. Invocation instructions are allowed to have side-effects of configuration.

(iii) Notification: The event of transfer completion can be notified to software in some way (e.g., via interrupt) so that some software flag can be set to indicate the event.

A closely related concept is the *instance* of a transfer-type called *transfer-instance*.

Definition 2: A *transfer-instance* is a transfer-type associated with a specific configuration.

We may treat a transfer-type as a set of transfer-instances. In the case that the discrimination between transfer-type and transfer-instance is insignificant in discussion, we use the term *transfer*.

Figure 3.1 shows the life-cycle of a transfer, which includes a data-phase and a control-phase. Note that the configuration, the invocation and the notification are the *overhead* of a transfer and that the main body of a transfer is the data-flow. Also notice that the differentiation between "control" and "data" is relative (to the abstraction level). The *control* operation at transfer level, typically a register-access operation, is also a *data*-operation at instruction-level; similarly, the *data*-flow of a transfer has already implicitly involved physical-level *control*.

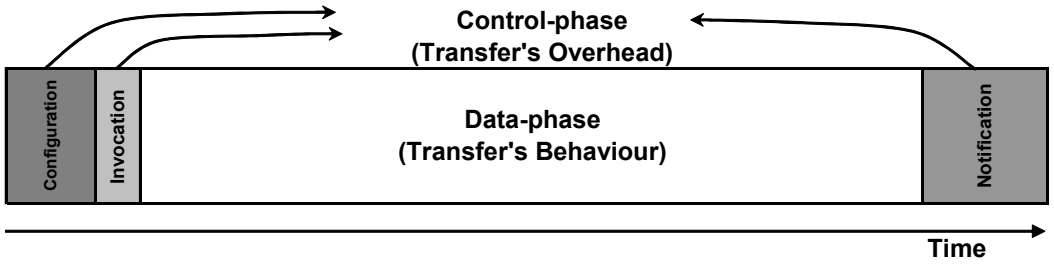


FIGURE 3.1: The life cycle of a transfer. We regard the configuration, invocation and notification as the control overhead, while the data-phase as the payload.

A *transfer* shares some similarity with a CPU instruction.

- A transfer has its type, just as an instruction has its opcode, which determines what type of operation is going to take place;
- A transfer needs to be configured with concrete parameters, just as a valid instruction needs to be filled with concrete operands;

Notification is a property not shared by an instruction, but serves as the driving force of the test-program as detaled in the next chapter.

Both transfers' running and instructions' execution can be viewed as interaction-objects. Indeed, a transfer is an interaction pattern between SoC components, just as an instruction is an interaction pattern between components inside the processor (e.g., the ALU, the register file and the pipeline). However, transfers and instructions are interactions at different abstraction level. Instructions are so short that the parallelism between them (instruction-level-parallelism, or ILP) should be handled by a dedicated hardware, namely, the processor itself; while transfers have much longer life so that it is possible for software to handle their parallelism.

We only include the configuration/invocation/notification as the basic transfer controls. It may be useful to extend the model with more control properties in control-intensive applications. For instance, in addition to configuration and invocation, we could add other control handles to transfers such as "abort", "pause" and "resume"; and transfer may also have notification events other than end-of-transfer.

3.2.2 Expressive Power of Transfer

In the early stage of an SoC design/verification cycle, the level of abstraction should be high enough to hide the differences between hardware behaviours and software behaviours [54]. Our transfer-type model meets this requirement. All the three previous interaction examples can be expressed as transfer-types.

(i) Transfer-type "Flash-to-RAM-DMA":

- Configuration: *initial-source-address, initial-destination-address*, *width* (8, 16, or 32-bit) and *length*;
- Invocation: setting the `go` bit in the DMA engine control register;
- Notification: DMA-finish interrupt.

(ii) Transfer-type "Sorting":

- Configuration: *address*, *data type*(signed/unsigned integer, etc), *length*, *sort-algorithm* and *reverse*;
- Invocation: calling subroutine `sort(address, type, length, algorithm, reverse)`;
- Notification: the return of the subroutine.

(iii) Transfer-type "UART-Rx-by-Interrupt":

- Configuration: *end-of-packet character*, *max-length*, *finish-mode* (by *max-length* and/or *end-of-packet-char*), *error-detection-mode* (`parity, frame`);
- Invocation: a STORE instruction to a special address – the test-bench/test-program interface; when this address is written, the test-bench starts to feed the SoC with a bit stream.
- Notification: the UART interrupt-service-routine detects the finish condition of the UART receiver.

More generally, the transfer-type model can express the following three categories of data-intensive interactions. The above three examples represent each of the categories.

(i) **Hardware behaviours (Hard-transfers):**

The transfer "Flash-to-RAM-DMA" models the read/write operations on the bus driven by master-devices, whose behaviours are mostly hardwired. So this kind of transfers are categorised as *hard-transfers*. Hard-transfers are intrinsically fast. But their behaviours have less variety. The configuration is done by setting up registers across the hardware devices involved in the transfer. The invocation is similar. The completion events are notified to the test-program via interrupts. The main form of interactions in hard-transfer is the read/write operations on the bus.

(ii) **Software behaviours (Soft-transfers)**:

A processor in an SoC is a valid master device, whose behaviours are programmable rather than hardwired. So its behaviours are called soft-transfers. Soft-transfers are basically in the form of subroutines and their behaviours are intrinsically flexible. A soft-transfer's configuration is done by passing arguments to the subroutine; the invocation is the "JUMP" or "CALL" instruction to a subroutine, and the completion is notified by the return of the subroutine.

There is a subtle but crucial difference between the codes in a soft-transfer and the codes in *configuring* a transfer.

- The former should be regarded as the *payload* and subject to verification; they are also application-oriented, and thus require manual development.
- The latter codes are treated as the *overhead* to build the data-phase. They should be automatically organised in a test-program.

One guideline to build soft-transfers is to compose the read/write intensive subroutines to stimulate the interactions between the CPU and slaves. However, soft-transfers do not have to transfer data *literally*; they can be computation-intensive operations to apply stress to different types of physical resources like the ALU in the CPU. For our Nios SoC, a simple recursion-intensive subroutine (`recursive_fibo`) is developed to apply stress to the register-window mechanism in the Nios CPU architecture.

(iii) **HW/SW Collaborations (Virtual-transfers)**:

In the Nios SoC, the incoming UART byte-stream is formed by the cooperation between the UART, the interrupt subsystem and the UART-receiver-ready interrupt-service-routine (ISR). Although the byte-stream is physically performed by the CPU, from a higher level of abstraction, it is functionally equivalent to perceive that a *virtual* master (also see Section 3.3.2) is conducting the stream between the receiver and a memory buffer, *independently* from the CPU which may be involved in another task (at a reduced performance). Transfers conducted by virtual masters are called *virtual-transfers*. Like a hard-transfer, configuration and invocation are in the form of writing control-registers; the notification to software is a trivial requirement for virtual-transfer, since the virtual-master (i.e., the ISR) is already software.

Unlike a soft-transfer, which explicitly requires a real CPU as its resource, a virtual transfer just requires a virtual master; therefore, we can arrange *multiple* virtual transfers (and one soft-transfer) to work "concurrently" on a single CPU. This concurrency is actually the parallelism between the CPU and the peripherals.

In a virtual transfer, the primary forms of interaction are interrupt request and response, while the traffic on the read/write bus is secondary.

Table 3.1 lists the three categories and summarises how to implement their configuration, invocation and notification.

The "transfer" is a compact and abstract model to express interactions. A tool named XGEN, developed by IBM for system-level test generation [37], also uses a model called "interaction". One such "interaction" is further divided into *acts*. For the example of a memory-to-memory DMA, XGEN models it as a *three*-act interaction: the DMA-service-request phase, the DMA copy process itself, and the DMA-finish-interrupt phase. Each act

Transfer Category	Transfer-Type Examples	Configuration	Invocation	Notification
Hard-Transfer	Any DMA transfer	Setting control-registers	Setting control-registers	Master (or slave) interrupting CPU
Soft-Transfer	UART polling; internal sorting; recursive subroutines; processor-self-test Subroutines	Setting control-registers; Passing arguments to subroutines	Calling subroutines	Subroutines' return
Virtual-Transfer	UART trdy/rrdy ISRs; Timer time-out ISR	Setting control-registers; Setting global variables	Enabling interrupt Sources	The virtual master (ISR) itself

TABLE 3.1: Implementation of hard-, soft- and virtual transfers.

needs to be modelled separately and explicitly. In contrast, we model it as *a single transfer*, which represents all three phases. Our model is more straightforward and natural.

The "features" of transfer's expressive power could also be interpreted as its "limitations" depending on the context, and vice versa. For instance, the instruction-flow cannot be modelled as an *independent* interaction-object. We may interpret this fact either as a *limitation* – we do not have a direct control over the instruction-flow, or as a *feature* – the instruction-flow is simply abstracted away and could be understood as the "noise" on the bus.

3.2.3 Transfer Complexity and Environment Complexity

In order to identify transfer-types in a given system, we need to discuss the complexity of the transfer-type model. One transfer-type's complexity is caused by its configuration. We use $\mathbf{T}$ to denote the set of transfer-types in a system, and denote T_i as each transfer-type member. For T_i, each of its parameters has a set of values to select from. Hence, T_i requires an operation $P(\cdot)$ to perform its parameterisation. Its parameters could be either totally independent or constrained with each other in various ways. In other words, each T_i's parameter-space is specific and application-oriented. Therefore, its $P(\cdot)$ should be more accurately denoted as $P_{T_i}(\cdot)$, or, from the object-oriented programming point of view, as $T_i.P(\cdot)$. The complexity of $T_i.P(\cdot)$ can represent the complexity of T_i. To let $T_i.P(\cdot)$ deterministically traverse the whole parameter-space seems neither necessary nor practical. We could implement $T_i.P(\cdot)$ using weighted and constrained randomisation.

Our transfer model is quite flexible in the sense that defining a transfer-type allows for trade-off between (a) the number of transfer-types in a system, and (b) the complexity of their $P(\cdot)$s'. To one extreme, we could model only one single transfer-type to represent all

possible interaction patterns in a system, but its $P(\cdot)$ needs to deal with a very large but also very artificially constrained parameter-space. To the other extreme, we could create a transfer-type for each possible interaction pattern *of concrete parameters*. In this case, we would have a huge number of transfer-types, while their $P(\cdot)$s' all have trivial complexity. In other words, given a system, the more generalised each transfer-type is, the fewer transfer-types are required, but at the cost of more complex $P(\cdot)$s'. Figure 3.2 conceptually displays this generalisation continuum.

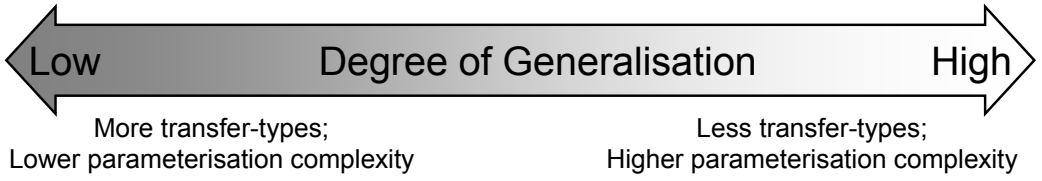


FIGURE 3.2: Generalisation of transfer-types. The transfer model allows users to trade-off between the number of transfer-types and the complexity of their parameterisation.

In practice, it is natural to adopt this guideline: generalising interaction patterns of "similar" parameterisation style as one transfer-type. Taking the example of the Nios SoC, we initially planned to model 12 transfer-types to represent DMA transactions among four source memory modules (ROM, RAM, FLASH, SRAM) and three destination memory modules (RAM, FLASH, SRAM). But later we decided to merge them into one transfer-type called "memory-to-memory DMA", with a single but stronger $P(\cdot)$ capable of assigning source and destination among all memory modules. Meanwhile, we consider it more appropriate to model "UART-Rx-by-DMA" and "UART-Tx-by-DMA" as separate transfer-types, which have very different parameters.

Another guideline is to let the *application of the SoC* to guide transfer-type identification, and let the *consideration of hardware capabilities* to guide the design of transfer-types' parameterisation spaces.

Some control variables are not appropriate to be associated with one *single* transfer-type. Instead, these variables *globally* affect concurrent transfers. For instance, in the Nios SoC, the UART baud rate affects both a receiving (RX) stream and a transmitting (TX) stream, which could run simultaneously. Another example is the data-cache and instruction-cache enabling/disabling setting, which affects all concurrent transfers from the background. These control variables are called "environment parameters", whose parameterisation is performed by $P_{en}(\cdot)$, which is not associated with any particular transfer-type. The implementation of $P_{en}(\cdot)$ is similar to $T.P(\cdot)$, i.e., via randomisation. A reasonable assumption is that

an environment parameter should not be updated until all affected transfers are inactive. The complexity of the environment space also contributes to the total complexity of SoC verification.

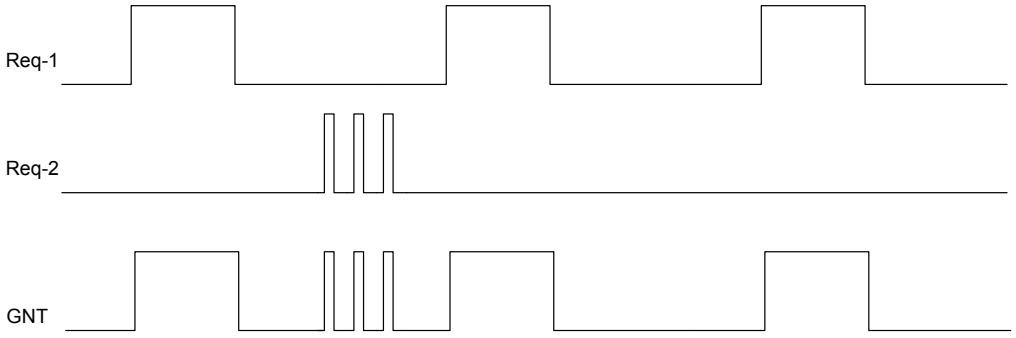
3.2.4 Transfer Temporal Granularity

To further characterise transfers, we give an estimate of their life-expectancy.

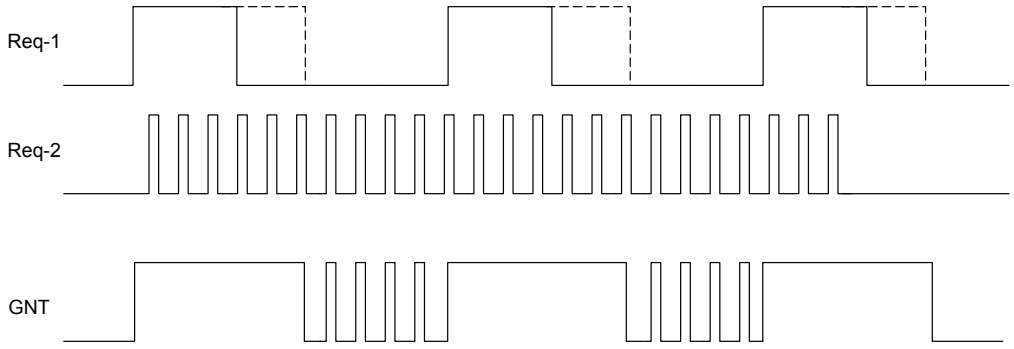
First of all, we discuss the necessity of comparable life-expectancy of all transfers. The transfer model enables us to generalise data-flows driven by various mechanisms, which could operate in a wide spectrum of data-rates. In our Nios SoC, transfer-type "ROM-to-RAM-DMA" has the rate of 33.3MB/sec; while the transfer-type "UART-RX-by-Interrupt" is operating at 14.4KB/sec (or 115,200bps baud-rate). Now the question is: how to "match" concurrent transfers in order to achieve the desired verification quality, i.e., the parallelism and resource-contention? For example, does it make sense to create a testcase in which a 1000-byte-long transfer T_1 at speed of 10MB/sec runs alongside another 1000-byte-long transfer T_2 at 10KB/sec? It appears to be a poor match, since T_1's life is only one thousandth of T_2's, meaning that the parallelism exists only 0.1% of the simulation, so the competition on the shared resource (the bus) is very little. Therefore it makes sense to configure all transfers to have comparable life-expectancies, say, within one order of magnitude of difference.

Figure 3.3 illustrates the effect of combining two transfers of different data-rates. We see that Req-1 (from one transfer) and Req-2 (from anther transfer) are using the bus at very different paces. If Req-2 is much shorter than Req-1 as shown in Figure 3.3(a), the stress put on the bus granting mechanism is completely absorbed by the bandwidth of the bus; but if the two transfers are configured with comparable life-expectancy as shown in Figure 3.3(b), these two transfers together stress the bus in a way neither of them can achieve individually.

We now consider how to estimate the optimal life-expectancy. Common sense tells us that the life-expectancy should not be too long. This is because simulation is a very time-consuming process. In the shortest time possible, we not only need to cover most configurations for each given transfer-type, but also should try its concurrent running with other transfers. On the other hand, neither can life-expectancy be too short. We regard the data-phase of a transfer as its main body, in which parallelism and resource-competition are supposed to happen; whereas the transfer's control-phase, namely, its configuration/invocation/notification, is the overhead. So it is natural to require the data-phase to be at least one order of magnitude longer than the control-phase; otherwise, a considerable portion of simulation time will be spent on the overhead. Fortunately, the length of control-phase is predictable because



(a) When bus requests from two transfers differ greatly in length, the probability of physically simultaneous bus accesses is low. In this figure, Req-2 falls completely between two continuous accesses from Req-1, therefore neither of the requests is affected by the other. The bus arbitration mechanism is stressed very little.



(b) When bus requests from two transfers has similar length, there is high probability of physically simultaneous bus accesses. The two transfers are physically interfering with each other, implying a much higher stress on the bus and a higher test quality.

FIGURE 3.3: Transfer life-expectancy affects test quality. Req-1 and Req-2 stand for the bus requests from two transfers; GNT stands for the bus granting signal.

all transfer-types' configurations, invocations and notifications are made up of instruction sequences of similar length. Hence, we assume that the following quantities are available:

- the average execution time of transfer configuration, C;
- the average execution time of transfer invocation, I;
- the average execution time of transfer notification, N.

Then we can reasonably conclude that the optimal transfer life-expectancy is simply in the range of (10 to 100) $\times$ $(C + I + N)$, which makes the overhead well under 10%.

In the Nios SoC example, $(C+I)$ requires 25 assembly instructions (mostly memory/register write accesses), or 100 SoC cycles; transfer notification is typically carried out by interrupt, which includes the time spent on context switching and ISR execution; thus the average N

is about 350 cycles. The optimal transfer life-expectancy is in the range of (10 to 100) $\times$ $(C + I + N)$, or 4,500 to 45,000 SoC cycles.

At run-time, considering the transfer-performance penalty caused by resource-competition, we shall allow for longer transfer life-expectancy than what was parameterised. Since we favour resource-competition in test scenarios, we should allow for heavy penalty, say, another ten-fold parameterised life-expectancy.

From the above discussion, we can quantitatively sense the time-granularity of "transfers". This is also the granularity of our proposed "system-level" tests. Understanding transfers' temporal granularity helps us to

- model and identify transfers, especially bias the behaviours of their $P(\cdot)$;
- understand the feature and limitations of system-level tests; and
- properly encapsulate tests in test-programs.

Compiling a TP consumes time; we should make sure that compiling a TP is a small overhead compared with executing the TP (during simulation). The actual time (wall-clock time) to simulate a transfer organised in a TP depends on the simulation platform and the SoC complexity. Our Nios SoC in RT-level Verilog model is simulated by Synopsys VCS [79]; the simulation roughly runs at the speed between 5 and 10 KHz, thus a typical transfer takes about dozens of seconds. This length justifies encapsulating *many* transfers in *one* test-program, considering that the *overhead* of test-program compilation is around one minute.

Compared with other interaction-objects, a transfer has a unique temporal granularity. The interaction-objects at lower abstraction levels, such as instructions, have a much smaller temporal granularity, which is well within 10^1 cycles. However, transfer's granularity of 10^4 cycles is still much finer than that of application-level interaction-objects, namely, the *processes*. One CPU-slice allocated to a user process is at the millisecond level, or in the order of 10^6 CPU cycles, and the *life expectancy* of a process is much longer than that.

In this sense, interaction-oriented view of a system is also a hierarchical view: *transfer* is just one intermediate interaction-form that connects other identifiable interaction-forms. Short-living interaction-forms, e.g., *instructions* and signal-level *transactions*, aggregate to become *transfers*, which in turn aggregate to become long-living *computation-treads* and *processes*, which may aggregate further to become *applications*.

Table 3.2 compares the temporal granularity of instructions, transfers and processes. The granularity decides which mechanism is used to manage the parallelism. Instructions have

Interaction Object	Temporal Granularity (cycle)	Parallelism Management	Scheduling Constraints
Instruction	10^1	Hardware (Processor)	Program Order & Resource Availability
Transfer (Test)	10^4 ~10^5	Software (Test-Program)	Logical Resource Conflict
Process	>>10^6	Software (OS)	Inter-process Communication & Resource Availability

TABLE 3.2: Different Levels of Interactions: Instructions, Transfers and Processes

such a small granularity that the instruction-level parallelism must be managed by dedicated hardware, namely, the processor. In comparison, transfers have a granularity large enough to be efficiently managed by software. On the other hand, transfers' granularity is also small enough compared with processes so that transfers exhibit much simpler behaviours than processes. As a result, a test-program, which manages transfers, enjoys substantial simplicity that is not shared by the OS, which manages processes.

(i) A transfer's resource usage is *static*; while a process uses resources dynamically. For instance, in a memory-to-memory-DMA-copy transfer, it is unnecessary, as well as impractical, to claim partially copied memory-ranges as free resources. The entire memory ranges should be regarded occupied until the copy finishes.

(ii) Concurrent transfers do not communicate with each other due to their granularity; while processes need to communicate each other. In fact, inter-process communication could be modelled as transfers.

The simplicity makes TP implementation easy and TP automation feasible. For instance, in principle, the data-structure for transfer scheduling purpose could be as simple as a *single*

bit: 1 for running and 0 for not-running. The usage of this structure is detailed in Sections 4.2.1 and 4.2.2.

Table 3.2 suggests that *resource availability* is a common factor for parallelism management. We are now in a position to model the resources for system-level verification purpose.

3.3 Resource Modelling

3.3.1 Resource-contentions and Resource-conflicts

The focus of system-level verification is parallelism. The main purpose of constructing parallelism is to observe interesting resource-competitions. Resource-competitions could happen on various mechanisms, including the on-chip interconnection subsystem, the interrupt mechanism, the CPU-time, the context-switching mechanism, memory locations, caches and buffers. Even more interesting situation is that competitions in various domains can interfere with each other, as discussed in Section 3.2.1.

Transfer model allows these competitions to be built naturally – we simply arrange multiple transfers to run concurrently. By managing transfers' configuration/invocation/notification, a test-program has considerable freedom in arranging parallelism. However, there should exist some principles to prevent the freedom from being reduced to unchecked randomness.

Our principle is to distinguish between resource-*contentions* and resource-*conflicts*. Resource-contentions represent the physical level competitions that are supposed to be resolved by hardware mechanisms (e.g. bus protocol, interrupt handling scheme, cache coherence scheme). These competitions are not just legal but also desirable. Resource-contentions are then defined as physical-level resource competitions which a programmer has no direct controllability and observability. Table 3.3 lists the typical physical resources in a system, potential contentions and the hardware mechanisms that address the contentions.

In contrast, resource-conflicts are competitions at the logical level and require a programmer's discretion to *avoid*. For example, we should allow the DMA engine to compete with the CPU for a physical memory module, but we require that the DMA transfer should never access the memory locations that are *currently* involved in a `sort` subroutine; otherwise the results of both transfers will not be predicted from their configurations. Intuitively, if the result of an interaction is dependent on its timing with respect to other interactions, resource conflicts are implied.

Resource Type	Form of Contention	Resolving Mechanism
Read/write bus or interconnection	Multiple masters requesting the bus	Bus protocol
Slave interfaces	Multiple masters accessing a same slave device	Slave interface's internal Arbitration; Providing more ports
Interrupt mechanism	Multiple devices requesting interrupt service simultaneously	Interrupt priority scheme
Cache lines	Multiple computation threads occupying the same cache line	Cache consistency strategy
General purposed registers in the CPU	Multiple computation threads running on a single-CPU system	Context switching mechanism
Buffers; FIFOs	More than one producers output to a single consumer	Queuing

TABLE 3.3: Typical physical resources and resource contentions.

Definition 3: Given a set $\mathbf{t}$ of transfer-instances t_1, t_2, ..., t_n, which are respectively instantiated from transfer-types $t_1.T$, $t_2.T$, ..., $t_n.T$, we assume that each $t_i.T$ is associated with a `pass/fail` boolean function

$$t_i.T.Check(t_i.configuration, MemRegSpace_{start}, MemRegSpace_{end}),$$

which, according to t_i's configuration, checks if t_i has caused the expected changes (between when it starts and when it ends) in the memory-register space. If, there exists a t_j in $\mathbf{t}$, whose value of $t_j.T.Check()$ varies with respect to t_j's temporal relations (sequential, overlapping) with other transfers in $\mathbf{t}$, we say there is *resource-conflict* in $\mathbf{t}$.

This definition forces some "determinism" – the *result* of each t_i should be deterministically predicted; but the determinism is also accompanied by "indeterminism" – the temporal relations between conflict-free transfers are allowed to happen in any way. If there are n conflict-free transfers, each having a *start* and an *end* event, then we shall allow for $\frac{(2n)!}{2^n}$ possible event sequences, all of which shall yield the same results in the memory/register space.

To avoid resource-conflicts is reasonable – if each transfer's result can be predicted by its configuration together with the contents in memory/register space, high level functional checkers, i.e., $T.Check(\cdot)$, can be easily implemented in the test-bench. Not enforcing this restriction on resource-conflicts is still an option; in that case, test-generator simply has more

freedom, but it loses the potential capability to predict correct results, therefore the burden of predicting correct test results is left to the users.

Once the test-generator is able to avoid resource-conflicts, no other restrictions are preventing it from constructing parallelism. In this way, resource-contentions at physical level are constructed implicitly.

3.3.2 Logical Resources

Since resource-conflict is a logical concept, we only need to model the logical resources in the system. (They are "logical" to a programmer.) Therefore, there is no need to model hardware's specific functionalities. With this simplification, we only model three categories of resources: masters, registers and memory-ranges. We will see that this modelling is not as *ad-hoc* as it may seem.

(i) **Master:** Master is defined as anything that can conduct a transfer-type. Examples of master in our Nios SoC include the read-master and the write-master of the DMA engine, and the data-master of the Nios CPU. Once modelled, a master is a trivial resource – the test-generator only needs a single bit to indicate its status: available or unavailable. However, the concept of *virtual-master* requires a little more insight into how to interpret system behaviours.

A virtual-master is an interrupt-service-routine (ISR) that cooperates with hardware to perform data-intensive operations. For example, the UART receiver-ready-ISR is a virtual-master performing transfer-type "UART-Rx-by-Interrupt". (Also see Section 3.2.2.) A virtual master is usually capable of only one transfer-type, but we can model as many virtual-masters as necessary for an SoC, independent from the number of physical CPUs. Other examples in our Nios SoC include UART transmitter-ready-ISR and timer-ISR. Once modelled, the test-generator does not distinguish virtual and real masters. In this way, the resource-contention on CPU-time can be constructed implicitly.

(ii) **Register:** Registers are also simple resources. We only need to model *data-intensive* registers visible to programmers. Examples are the UART `rxdata` and `txdata` registers.

Since control/status registers across an SoC are not suitable to be treated as data, they are not modelled as register resources. However, in fact, many control/status bits are *already* implicitly abstracted as masters.

(iii) **Memory-range:** Memory-ranges are flexible and general-purpose resources dynamically maintained by the test-generator. A memory-range is an object with properties such as base-address, size, sub-word granularity, read/write mode. From within one free memory-range, test-generator can dynamically allocate sub-ranges of suitable sizes and locations to transfers; meanwhile, the unused fragments become free memory-ranges. Allocated memory-ranges can reside in the same physical memory module, and even can overlap if they are all read-only. By this way the test-generator is able to construct resource-contentions on physical memory modules.

In our current implementation on the Nios SoC, memory-ranges do not cross physical boundaries between memory modules. But this restriction can be lifted if we view the whole memory space as a single free memory-range and allocate sub-ranges to transfers. In that case, the corner-cases in which transfers cross physical boundaries can be naturally built. However, this implementation needs to take account of miscellaneous constraints such as: ROM cannot be written; memory-mapped-registers should be excluded from the address space; different memory modules may accept different sub-word granularities, etc.

Just as transfer-types are the generalisation of similar transfer-instances (see Section 3.2.3), the above discussed logical resource types (master/register/memory-range) are the generalisation of *bit-resources*, namely, all *bits* in memory and registers accessible by a programmer. *Bit* (regardless of data-, control- and status-bit) is the finest resource object to a programmer; master, register and memory-range are simply different *aggregations* of bits. For instance, a physical master device's behaviour is controlled/observed by the bits in its control/status registers; it is actually those control/status bits that are abstracted as one logical "master" resource. Therefore, the granularity of a master resource is a few control/status bits. Similarly a register's granularity is several data bits; and a memory-range's granularity is a lot of (continuous) data bits.

Our treatment of register/memory bits in a system might appear similar to the concept of *register abstraction level* (RAL) model [34] in the TB-centric verification methodology called VMM [17]. In RAL models, bits are also organised to form hierarchical objects, including fields, registers, memories. But the philosophy of the RAL modelling is still component-oriented. Register and memory objects are treated as objects under test and it is the user who should provide tests, either from scratch or from pre-defined templates. Therefore, RAL modelling still presents the "test vs. object-under-test distinction". In contrast, in the interaction-oriented mindset, registers/memories are *resources* for tests (i.e., interactions) and simply serve as the constraints to build parallelism.

3.4 TRG for Test Generation

3.4.1 TRG Definitions

Once transfers and resources are modelled, their relationship becomes explicit: transfers need resources to run. They can be interlinked to form transfer-resource graph (TRG). TRG can be formally defined in terms of transfer-instance and bit-resource.

Definition 4: A flat *TRG* is a triple $G = (\mathbf{t}, \mathbf{r}, u)$, where:

- $\mathbf{t}$ is a set of concrete transfer-instances in a system;
- $\mathbf{r}$ is a set of bits accessible to a programmer;
- function u: $(\mathbf{t} \times \mathbf{r}) \rightarrow \{n, s, e\}$, where n, s, and e respectively represent *no-use*, *shared-use* and *exclusive-use*. Notation "$u(t, r) = n/s/e$" respectively means that transfer t will *not use*, *share* or *exclusively use* bit r during its life-time (both control-phase and data-phase).

Then "parallelism" can be formally modelled as *scenario* or concurrent transfers without conflicts.

Definition 5: Given a TRG $G = (\mathbf{t}, \mathbf{r}, u)$, a *scenario* is a subset $\mathbf{s}$ of $\mathbf{t}$ satisfying:

- $|\mathbf{s}| = 1$, or
- $|\mathbf{s}| \geq 2$ and for any two distinct $t_i, t_j \in \mathbf{s}$, for all $r \in \mathbf{r}$, $(u(t_i, r), u(t_j, r)) \notin \{(s, e), (e, s), (e, e)\}$.

Environment parameters (Section 3.2.3) can also be modelled as bit-resources in the TRG model. One "environment bit" of a scenario $\mathbf{S}$ is a control bit r that satisfies

$$\exists t, \tau \in \mathbf{S}, t \neq \tau, u(t, r) = u(\tau, r) = s.$$

Implementing a flat TRG is impractical due to the huge number of concrete transfers and bit-resources in a system. In order to visualise a TRG and generate scenarios practically, we use a different TRG definition based on transfer-types and master/register/memory-range resource models.

Definition 6: A TRG is $G = (\mathbf{T}, \mathbf{R}, U)$, where

- **T** is a set of transfer-types in a system, each transfer-type is a set of transfer-instances;
- **R** is a set of logical resources, each resource is a set of bits;
- function U: $(\mathbf{T} \times \mathbf{R}) \rightarrow \{n, s, e\}$. For each pair $(T, R) \in (\mathbf{T} \times \mathbf{R})$, if all instances of T exclusively use all bits in R, then $U(T, R) = e$; if all instances of T do not use any bits in R, then $U(T, R) = n$; otherwise, $U(T, R) = s$.

Here is a hidden assumption about the resource usage of a transfer: a transfer-instance neither releases any allocated resource nor requires any additional resource throughout its lifetime. This assumption can be justified by the discussion of transfers temporal granularity (see Section 3.2.4). This static quality of transfers contributes to automatic test-generation.

Figure 3.4 visualises an abridged TRG for the Nios SoC. Arrows represent the transfer-types and the blocks represent the resources, the letter e and s represent the access mode. Note that some ISRs are treated as master resources.

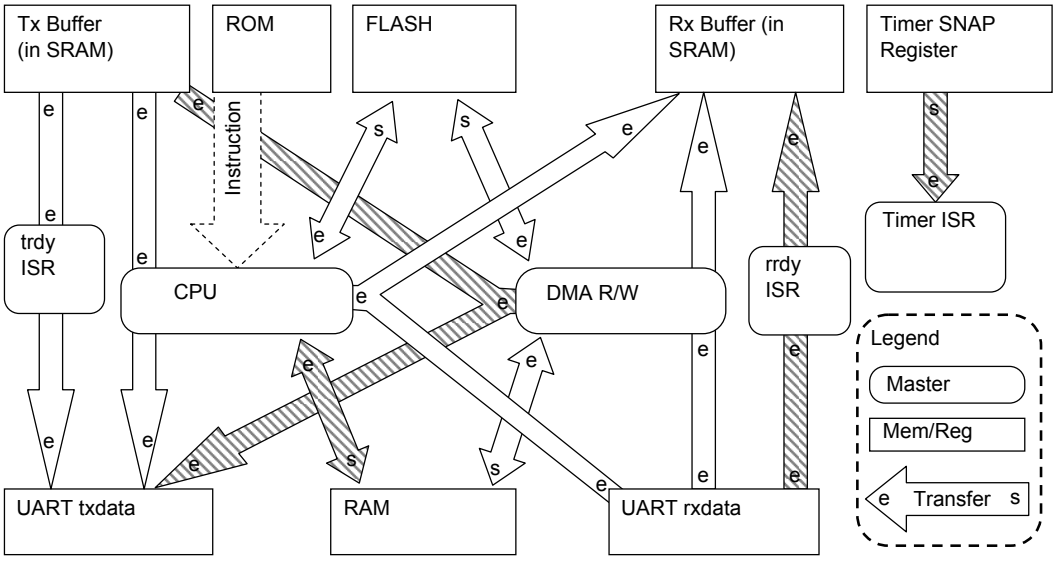


FIGURE 3.4: The abridged TRG for the Nios SoC. The shaded transfers form a valid scenario.

TRG is an interaction-oriented model in which hardware components' functionalities are totally abstracted away. The only property regarding hardware in TRG is the very generic concept of "resource availability". In fact, HW component's functional properties have not disappeared; they are reorganised to be *transfers*' functional properties. (These properties do not show up in the TRG model.) This reorganisation is the central step to migrate from a component-oriented mindset to an interaction-oriented mindset. For instance, in order to

instantiate a "UART-RX-by-DMA" transfer in the Nios SoC, with the property that "the RX stream terminates on the `eop` (end-of-packet) signal", the test-program needs to:

1. write the EOP register in the UART with a specific `eop` character;
2. setup the transactor in the test-bench to feed a byte-stream whose last byte is the same `eop` character;
3. enable the DMA engine to be sensitive on the `eop` signal.

We do not say that these three operations are "configuring three hardware devices" – this is the component-oriented interpretation; instead, we say that they are "configuring a *single* property of a transfer" – this is the interaction-oriented interpretation.

3.4.2 Implement TRG for Test Generation

We implement TRG as a couple $(\mathbf{T}, \mathbf{R})$, where members in $\mathbf{T}$ and $\mathbf{R}$ are all intelligent objects aware of resource-usage. A transfer-type T has a resource-allocation operation. This allocation is an important part of the T's parameterisation operation $T.P(\cdot)$, and it is denoted as $T.P.A(\cdot)$. The allocated *exclusive* and *total* resource-usages are respectively denoted as $T.\mathbf{U_e}$ and $T.\mathbf{U_t}$, where $T.\mathbf{U_e} \subseteq T.\mathbf{U_t} \subseteq \mathbf{R}$. Now we can re-describe the scenario generation question. To construct a *parameterised* scenario, we need to search for any non-empty subset $\mathbf{S}$ of $\mathbf{T}$ and perform $T_i.P(\cdot)$ and $T_i.P.A(\cdot)$ of each T_i in $\mathbf{S}$, so that either there is just only one transfer, or, if there are more than two transfers and for any distinct T_j and T_k in $\mathbf{S}$, $T_j.\mathbf{U_e} \cap T_k.\mathbf{U_t} = \varnothing$ and $T_j.\mathbf{U_t} \cap T_k.\mathbf{U_e} = \varnothing$.

Before we give a scenario generation algorithm, we need to introduce another internal operation of transfer-type T. Once $T.P(\cdot)$ has decided the concrete parameter-values in logical sense, the test-generator needs to *interpret* them into actual configuration/invocation instructions. This interpretation operation is denoted as $T.I(\cdot)$. Its input comes from $T.P(\cdot)$'s output; and its output is SoC instructions that implement the configuration/invocation. Separating $T.I(\cdot)$ from $T.P(\cdot)$ decouples two levels of constraints: (a) logical constraints between parameters and (b) hardware-induced constraints to implement the configuration of parameters. This decoupling makes the test-generator easy to maintain.

- $T.P(\cdot)$ resolves the logical constraints between parameters and decides the concrete parameters. It does not care about how to implement the configuration of these parameters.

- $T.I(\cdot)$ mechanically interpret the concrete parameters into configuration/invocation instructions. It is a relatively simple job. With the logical constraints between parameters having been addressed by $T.P(\cdot)$, $T.I(\cdot)$ can focus on resolving hardware-induced constraints. For instance, it may either configure *one parameter* using *multiple operations*, or combine the configuration of *multiple parameters* in *one operation*.

Since $T.P(\cdot)$ has already concretised logical parameters, the order to implement their configuration is not important. Therefore the "configuration" part of $T.I(\cdot)$'s output should not interpreted as *a sequence of* configurations, but *a set of* configurations, each of which could be implemented as an instruction-sequence. Shuffling the configurations gives the test-generator another level of freedom to output a test-program.

Given a TRG $G = (\mathbf{T}, \mathbf{R})$, let $\mathbf{R_S}$ and $\mathbf{R_E}$ respectively represent the current resources available for *shared* and *exclusive* access. The following algorithm constructs a scenario and maximises the number of transfers.

(1) $\mathbf{R_S} = \mathbf{R}$; $\mathbf{R_E} = \mathbf{R}$;

(2) Randomly select a transfer-type T_x from $\mathbf{T}$;

(3) Issue $T_x.P(\cdot)$, which in turn issues $T_x.P.A(\cdot)$, to parameterise/allocate resources to T_x so that:

- $T_x.\mathbf{U_e} \subseteq \mathbf{R_E}$, and
- $(T_x.\mathbf{U_t} \setminus T_x.\mathbf{U_e}) \subseteq \mathbf{R_S}$

(4) Issue $T_x.I(\cdot)$ to interpret the configuration and output the configuration/invocation instructions;

(5) $\mathbf{R_S} = \mathbf{R_S} \setminus T_x.\mathbf{U_e}$; $\mathbf{R_E} = \mathbf{R_E} \setminus T_x.\mathbf{U_t}$;

(6) In $\mathbf{T}$, drop any transfer-types that cannot obtain sufficient resources from the reduced $\mathbf{R_E}$ or $\mathbf{R_S}$;

(7) If $\mathbf{T}$ is empty, one scenario with maximal transfers has been generated; otherwise repeat from step (2).

In the above algorithm, transfer-type selection (step 2) is constrained by the previous round of transfer parameterisation (step 3), due to the resource usage being updated. This is different from the generation methods used in XGEN [37] and Esterel [18], in which abstract

tests are generated first and then are concretised into parameterised tests. Abstract test-generation and test-concretisation respectively requires *separate* techniques to fulfil, whereas in our approach these two tasks are uniformly blended under resource constraints.

The four shaded transfers in Figure 3.4 form a legal test scenario. Although they appear loosely distributed in the TRG, the test quality is high because all hardware components are supposed to behave concurrently in simulation: the CPU is sorting data in RAM, the DMA is transferring data from a buffer to the UART; the Timer is counting, and the UART is working in full duplex mode. (Some "noise", such as the instruction-flow on the bus, is also active.) Therefore, high degree of resource-contentions will be achieved on various physical resources such as the bus, the slave interfaces, the interrupt mechanisms and CPU-time.

The variety of tests is managed by the test-generator in several ways, including

- transfer-type selection, i.e., step (2) in the above algorithm;
- transfer-type parameterisation, i.e., $T.P(\cdot)$;
- the shuffling of configurations of each transfer;
- the *interleaving* of configurations of multiple transfers in a scenario;
- the parameterisation of *environment parameters* (see Section 3.2.3), which *globally* affect concurrent transfers.

The user can also intervene test generation by specifying a bias file, which biases most randomisation operations in the test-generator. The bias file will also be used in test generation with feedback information from post-simulation analysis. Section 6.4 provides further information. More detailed implementation details about the test-generator is described in Appendix C.

3.4.3 Features and Limitations

As a model at high abstraction level, TRG has the following features and limitations:

- TRG decouples two levels of complexity for test-generation – the complexity of each transfer-type and the complexity of generating parallelism. The former is system-specific while the latter is relatively independent from an actual SoC, which makes TRG applicable to a wide range of designs.

- Scenarios generated from TRG can serve the purpose of *functionality* test as well as *performance* test.
- While the task of generating legal scenario is left to the test-generator, the users need to manually build transfer-type (e.g. to manually compose $T.P(\cdot)$, $T.I(\cdot)$, and $T.Check(\cdot)$). Other level of automation could be introduced to help model interactions, such as the path selection mechanism used in XGEN [28];
- TRG is a method independent from the simulation platform. It is even possible to apply it to generating manufacturing tests.
- The target bugs are *not* the bugs inside each hardware component, but hard-to-detect bugs caused by close resource competitions. Therefore, hardware components are preferably free of obvious internal bugs. In fact, in the interaction-oriented mindset, bugs are associated with interactions rather than with components.
- Result-checking of transfers is by means of checking the contents in memory and registers. These checking can be implemented as high level (thus easy-to-construct) functional checkers in a test-bench. However, these high-level checkers lack the observability on low-level errors such as protocol non-conformance. Therefore, other error-detection mechanisms (e.g., HW property assertions) at lower levels should also be implemented in test-benches.

In a word, TRG as a model at a specific abstraction level has its own strengths and limitations. The clearly defined abstraction level allows the TRG method to work orthogonally with many existing verification practices.

We should realise that the "scenarios" generated by TRG are only the *snapshots* of concurrency. The TRG-based test-generator cannot deterministically predict the temporal relations between concurrent transfers. For example, the test-generator do not (and cannot) specify which transfer finish first in a scenario. The dynamic aspects of transfers must be *performed* at simulation time and be *analysed* after simulation. The next section discusses TRG's application for post-simulation analysis.

3.5 TRG for Coverage

3.5.1 Overview

The simulation-based verification of a complex VLSI like SoC requires *multiple* coverage models. Each model measures simulation completeness from a specific perspective. At system level, since the system's behaviours can be described as concurrent interactions, one coverage model is needed to enumerate all concurrent interactions and the temporal relations between them. The widely used statement-based coverages (line, toggle, conditional and local state-machine, etc) cannot give such information.

The temporal relations open up an enormous coverage space, requiring a mathematical model to deal with it. The completeness of concurrency can be quantified in terms of the temporal relations between events. In [58], Kwon *et al.* propose that users first establish a hierarchical-temporal-event-relation (HiTER) graph to represent the interactions between communicating hardware components, then an algorithm based on the graph can calculate coverage space, which will be much smaller but more meaningful than a simple cross-product coverage model. This method could generate accurate coverage space for tightly communicating FSMs. To build such a graph, the users (verification engineers) must have an accurate view of signal-level timing dependencies between components.

We choose Petri-net [69, 93] as the model because its semantics include concurrency constrained by resources.

Definition: A *Petri-net* is a directed graph represented by a 5-tuple $(\mathbf{P}, \mathbf{T}, \mathbf{F}, W, M_0)$, where,

- $\mathbf{P}$ is a set of nodes known as *places*; each place can hold *tokens*. Tokens are all identical;
- $\mathbf{T}$ is a set of nodes known as *transitions*;
- $\mathbf{F}$ is a set of directed arcs (known as *flows*) connecting places and transitions, i.e. $\mathbf{F} \subseteq (\mathbf{P} \times \mathbf{T}) \bigcup (\mathbf{T} \times \mathbf{P})$;
- Function W: $\mathbf{F} \rightarrow \mathcal{N}^+$; $W(f)$ is called the *weight* of flow f; ($\mathcal{N}^+$ denotes positive integers.)
- Function M_0: $\mathbf{P} \rightarrow \mathcal{N}$, known as *initial marking*. $M_0(p)$ is the number of initial tokens in place p. ($\mathcal{N}$ denotes non-negative integers.)

A transition t is said to be *enabled* when each of its input places has equal or more tokens than the weight of the input flow. When enabled, t can (but does not have to) *fire*, i.e., t consumes $W(f_i)$ tokens from its input place connected via flow f_i, and puts $W(f_o)$ tokens into its output place connected via flow f_o. The duration of a firing is considered zero. The firing sequence is called the *execution* of the net. The *state* of a Petri-net can be described in terms of its marking, i.e., the distribution of the tokens. A Petri-net has its *reachability graph*, whose nodes are the states (i.e. the markings) and whose directed arcs represent the transitions between states. The reachability graph can be used to define the coverage space.

3.5.2 TRG and Petri-net

TRG and Petri-net share some similarities in describing a system. Both formally define *concurrency* and *conflict*.

- In TRG, concurrency is defined as a set of n ($n \geq 2$) transfers. In Petri-net, concurrency means a transition-node has *multiple* incoming or outgoing flows;
- In TRG, conflict means that some of concurrent transfers exclusively use the same logical resources. In Petri-net, conflict means a place-node has multiple incoming or outgoing flows.

The TRG model does allow us to specify the system-level concurrency. However, TRG lacks the capability to describe the dynamics of the system. As a high level test-generation tool, TRG cannot and does not need to deterministically specify temporal relations between concurrent transfers. The rich possibilities of the temporal relations can only be realised during simulation. For example, TRG does not (and cannot) specify at which moment in transfer T_1's life, another running transfer T_2 will finish. The timing that T_2 finishes is a complex function of its configuration, its submission timing and the penalty caused by the resources-contention between T_1 and T_2.

A scenario generated from TRG only represents a *snapshot* of data-flows in a system; in contrast, the execution of a Petri-net captures the temporal aspect of a system's behaviour at logical level; the reachability graph derived from a Petri-net can be used to describe the possible execution sequences. Therefore, a Petri-net model is suitable for post-simulation analysis of the temporal aspects of a system.

Nevertheless, a desirable feature of TRG is that it can be readily converted to a Petri-net. Assuming that any transfer in TRG contributes two transitions in Petri-net, *start* and *end*, we can construct a Petri-net from a TRG by the following steps:

(1) **Converting Resources:** For each resource R in TRG, create a place P_R to represent the resource.

(2) **Converting Transfers:** For each transfer-type T in TRG,

- create two transitions T_{start} and T_{end};
- create a state-place $T_{running}$ (cf. resource-place P_R.);
- create flows of weight 1 from T_{start} to $T_{running}$ and from $T_{running}$ to T_{end}.

(3) **Connecting Transfers and Resources:**

- First, for each transfer-resource pair (T, R) that satisfies $U(T, R) = s$:
  - add one token into P_R;
  - create one flow of weight 1 from P_R to T_{start};
  - create one flow of weight 1 from T_{end} to P_R.
- Then, for each transfer-resource pair (T, R) that satisfies $U(T, R) = e$:
  - if P_R has no token, put one token in it;
  - create one flow of weight $n(R)$ from P_R to T_{start}, where $n(R)$ is the number of tokens in P_R;
  - create one flow of weight $n(R)$ from T_{end} to P_R.

The complexity of the above algorithm is linear to the size of TRG. Given a TRG $(\mathbf{T_{trg}}, \mathbf{R}, U)$, the sizes of the resulting Petri-net $(\mathbf{P}, \mathbf{T_{pn}}, \mathbf{F}, W, M_0)$ are:

- $|\mathbf{P}| = |\mathbf{T_{trg}}| + |\mathbf{R}|$,
- $|\mathbf{T_{pn}}| = 2|\mathbf{T_{trg}}|$, and
- $|\mathbf{F}| = 2|\mathbf{T_{trg}}| + 2|\{(T, R) : U(T, R) \in \{e, s\}\}|$

Once the Petri-net is generated, its reachability graph will be conveniently obtained by a Petri-net tool. Figure 3.5 shows the Petri-net constructed from the TRG.

It should be noted that both TRG and the Petri-net converted from TRG are high level abstraction of an SoC (with its application). Most resource-contentions at physical level are *invisible*, simply because the physical resources are not present in the models. Nevertheless, the Petri-net can provide useful temporal information regarding hardware-hardware and hardware-software interactions at the granularity level discussed in Section 3.2.4. The Petri-net could include even more temporal information, provided that each transfer contributes more internal states and events other than simple "start", "running" and "finish".

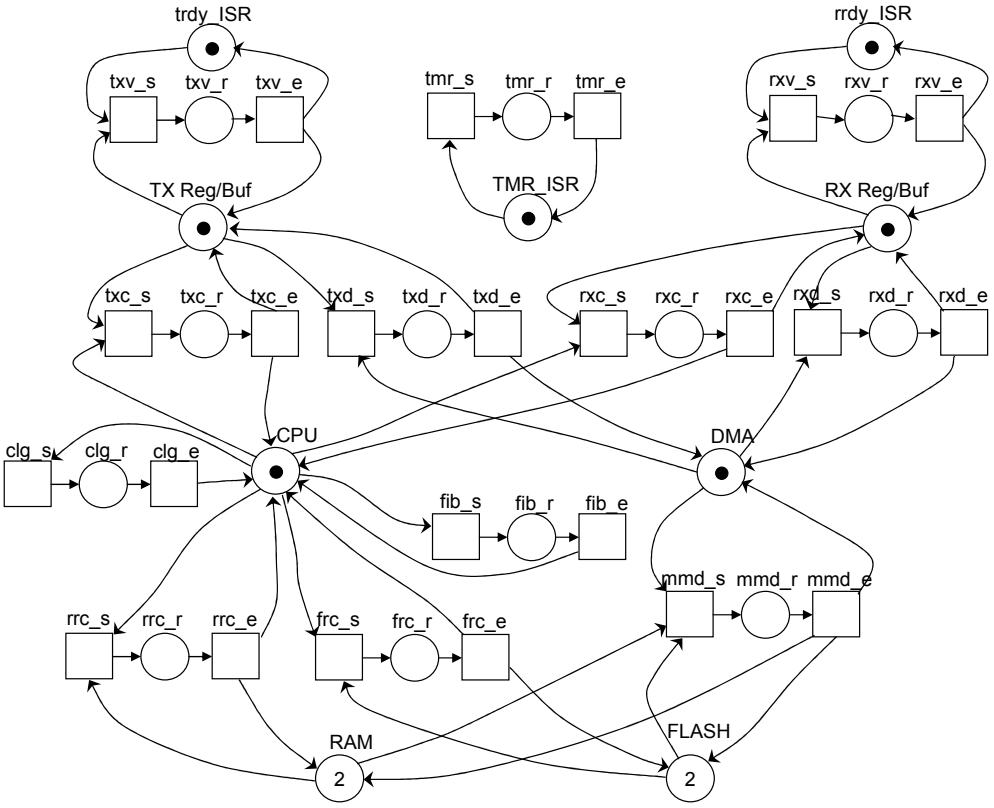


FIGURE 3.5: The Petri-net derived from the TRG of the Nios SoC. Square nodes are transitions, round nodes are places, and dots and numbers are tokens.

3.5.3 Use of Petri-net

Once a Petri-net is obtained from TRG, we can use the net in a number of ways. For instance, we could prove the liveness and boundedness of the Petri-net and then infer the similar characteristics of the TRG; we can simplify the Petri-net (but keep the reachability graph isomorphic), then we are able to map the simplification back onto the TRG. However, these theoretical treatments do not significantly contribute to a verification practice.

The derived Petri-net can indicate the total number of scenarios, because each state in the reachability graph represents a scenario. This size contributes to the total complexity of scenario-generation algorithm in Section 3.4.2.

The most practical use of the Petri-net is to define the coverage space. The coverage space is based on the reachability graph associated with the net. There are several options to define the space:

- All states in the graph (i.e., the markings);

- All state-state transitions in the graph (these transitions are different from the transitions in the Petri-net);
- All paths in the graph;
- All cycles in the graph.

These options represent the different levels of temporal details. In [93], a number of coverage-space definitions based on the reachability graph are proposed. These definitions roughly fall into: (1) state-based category, (2) transition-based category, and (3) flow-based category. Restrictions or modifications could be applied. For example, the path coverage space could be just too enormous due to the graph size and connectivity, but we could restrict the length of the path not to exceed a threshold.

To check the coverage, we need to collect transfers' start/finish event history from the simulation trace. This history can be easily collected, because each transfer has a software flag (running-flag) indicating whether it is running. The Petri-net reads the event history to replay the transition firing sequence. Its reachability graph is traversed in this manner. The traversed states, transitions and other coverage points such as cycles and paths are counted and compared with the coverage space size, then the percentages are reported.

Besides indicating the completeness of temporal relations, the coverage information can be further used to guide test generation. We have implemented test-generation with feedback at state and transition level. See Section 6.4.

3.6 Summary

This chapter details a key concept of our methodology: defining the *interactions* as the objects under test. The abstract concept of "interaction" now has a practical model – transfer. Transfers and bit-resources form the TRG model, which is used for both test generation and coverage measures. The tests generated from the TRG model can be applied to the SoC in software. The next chapter discusses the software structures that drive the TRG-based tests.

Chapter 4

Software Structures of Test-Program

Software native to a system-on-chip (SoC) could play multiple roles in system-level verification owing to the flexibility inherent to the software. In this chapter, we categorise these roles and detail their respective software structures.

4.1 Overview: Partitioning Software Roles in System-Level Verification

Running software native to the SoC for SoC verification is practiced in the following two ways, but neither of them has significantly contributed to the verification of system-level behaviours in the mainstream test-bench centric methodologies.

- Running software (SW) on a design-under-test (DUT) could be treated as the "asset" to hardware (HW) verification, since SW has certain control and observation capability over a DUT. The software that falls in this category includes diagnostic programs written by verification engineers to test the basic functionalities of a DUT. These diagnostic programs typically (a) write configuration data into the control registers in the DUT and (b) read the status registers to check the results. This practice is usually treated as an *ad-hoc* technique supplementing the test-bench (TB) centric verification. Indeed, these hand-written diagnostics are insufficient to test a system since they are either too basic or too specific; and obviously the automation of the process is poor.
- Running software on a DUT could also be interpreted as the "liability" to a HW verification team. The software that falls in this category includes device-drivers, operating

system (OS) and application software. Their owner is the software development team. The hardware team should not expect to run this set of software to verify a DUT, because

- this set of software may be simply unavailable at the integration stage;
- even if they are available, they would not put enough stress on the hardware;
- even if some anomalies are discovered, little indication is obtained about the nature of the bugs;

Running this set of software is necessary but should happen much later – *after* the DUT hardware has been substantially verified. That is why running this set of software is the "liability".

Our proposed software-centric SoC verification methodology addresses this dilemma of using software by concentrating on the *inter-transfer parallelism* in the DUT. The abstraction level of this kind of parallelism is not as high as that of the inter-process parallelism that an OS manages; meanwhile it is also high enough to abstract away the detailed hardware functionalities. The inter-transfer parallelism is the source of numerous corner-cases; by managing this level of parallelism, the test-program (TP) could greatly enhance the test quality.

"Parallelism management" is a perfect application niche for the DUT-native software, but not for the test-bench. A test-bench, being *external* to a DUT, has the inherent problems managing the parallelism *internal* to a DUT, and would incur much overhead including

- additional TB components to convey and manage synchronisation information, and
- additional languages for the user to manually specify tests of parallelism.

In the TB-centric verification methodology called VMM [17], the synchronisation information conveyor and manager are respectively called "XVC" (extensible-verification-component) and "XVC Manager"; and the language used to specify testcases is the "XVC manager test scenario description language". Again, we see that a TB is just the "vehicle" to transform abstract tests into more detailed ones; and the abstract tests must be described by human. Since everything in this XVC mechanism is not a part of the DUT, building such an external mechanism is the pure overhead to verification.

In contrast, we could utilise the interrupt subsystem and the processor embedded in the SoC-DUT respectively to convey and manage the synchronisation information; and the language

used for the management is the programming language native to the SoC DUT. Therefore, building such a software-based parallelism managing mechanism is not verification overhead at all, but a meaningful exploration of the SoC-DUT's full capabilities. Also, we have shown in the previous chapter that the testcase generation could be automated by the TRG model.

The idea to let the test-program manage the parallelism reminds us that an *operating system* (OS) shares the same concept of *parallelism management* on a general-purpose computer. Although we regard it inappropriate to run an OS for verification purpose as we have discussed above, comparing the TP with the OS will shed some light on how to partition verification responsibilities in software and where the opportunities for automation lie.

For SoC hardware verification, software can and should play multiple roles; therefore, software can be partitioned into different components.

- **Role 1**: some software components, typically interrupt service routines (ISRs), should *cooperate* with raw hardware devices to fulfil their originally intended functionalities. This role extends a system from a collection of raw hardware to a collection of usable functionalities;
- **Role 2**: some software components, such as hardware diagnostics, should *stimulate* hardware to check if they work as expected. For example, a subroutine with intensive memory access could stress memory modules; a subroutine with intensive arithmetic-logic-unit (ALU) operations could stress the ALU in the processor itself. We call such software components *soft-transfers* (also see Section 3.2.2).
- **Role 3**: some software should *manage* system-level concurrency by efficiently scheduling hardware and software behaviours.

These roles contribute differently to system-level verification. Role 1 is actually a part of a DUT, Role 2 represents some actual testcases to the DUT, and Role 3 manages testcases on the DUT. Role 3 serves as the backbone of the verification software. It enhances the test quality by arranging parallelism on a DUT, and is relatively independent from an actual SoC. In this chapter we specifically regard the software playing this role as the "test-program" (TP).

Role 1, 2 and 3 components respectively resemble the software components running on a general-purpose computer, i.e., (1) hardware drivers, which fulfil hardware functionalities, (2) user processes, which carry out the user-defined tasks, and (3) the operating system (OS), which schedules user processes. These two sets of components (ISR/soft-transfer/TP

and driver/user-process/OS) have different purposes and work on different levels. Namely, OS manages inter-process parallelism, while TP should manage hardware-hardware and hardware-software concurrency at a much finer granularity. However there are also similarities between them.

We should develop a payload-overhead view similar to that for a general-purpose computers. An OS is a complex and critical software on a computer; however, computer users always hope that their user processes occupy most CPU-time and meanwhile the OS kernel consumes a little fraction of CPU-time as the overhead to maintain inter-process parallelism. Notice that the OS is overhead only in terms of *CPU-time*, not in code size or structure. In fact, it is the smart structure that makes an OS run with minimum CPU-time usage.

Similarly, from the hardware verification point of view, the TP (Role 3 software) is only the *overhead* to manage the user-defined tests; the SoC processor should distribute most time executing the *payload* code – the code in software of Role 1 and Role 2. Therefore, the structure of the TP becomes critical since the parallelism management should not take too much CPU-time.

A general-purpose OS is event-driven software. In fact, events, typically interrupts, are the only entrance to an OS. Being event-driven makes OS work intelligently and efficiently. It is highly desirable to design event-driven test-program.

Unlike the software components on a general-purpose computer, verification SW components have some *automation requirements*.

- ISRs need to be manually developed in order to fulfil the hardware functionality properly. But there should be guidelines to compose ISRs in order to meet the requirements of the transfer model.
- Soft-transfers should also be manually developed. Like ISRs, guidelines should be provided to assist manual development. However, some existing techniques [30, 32, 56] could be adopted to automate some codes such as CPU self-testing program.
- The TP should be automated to implement scenarios generated by the test-generator. This automation requirement implies that the TP should be *regularly structured*.

This partitioning of verification software provides a valuable opportunity for the early involvement of application software. The real-world software components, such as device-drivers and applications, could respectively be decomposed and tailored into ISRs and

soft-transfers. In this way, elements from the real-world application are involved in the integration-stage verification

Although all simulation-based verification methods suffer the same intrinsic shortcoming – simulation consumes a lot of time – we alleviate the problem by adopting the smart TP structures. This alleviation is orthogonal to other techniques such as simulation accelerating technologies.

The rest of this chapter discusses the structure of Role 3 (TP), Role 1 (ISRs) and Role 2 (soft-transfers) components.

4.2 Test-Program Structure

The transfer-resource-graph (TRG) is an abstract model for test-generation, but how the resulting tests are structured in a test-program is a relatively independent issue. We have developed three flavours of TP structure.

4.2.1 Polling-Based Test-Program

Our first structure is the *polling-based* TP [90]. The test-generator identifies the legal scenarios in the TRG, then it outputs the transfers' configuration and invocation instructions in the TP. Consecutive scenarios are separated by polling statements to avoid resource-conflicts. In simulation, these statements keep polling some software flags until all transfers in the current scenario have finished, and then the TP can proceed to the next scenario. The execution of the polling-based TP is illustrated in Figure 4.4(a). Figure 4.1 is a TP fragment, which submits one scenario made up of transfers T_1 and T_2.

In this scheme, the test-generator has another level of freedom in arranging instructions by *shuffling* and *interleaving* configuration instructions of the concurrent transfers (see Section 3.4.2). This is allowed because the test-generator already guarantees that the concurrent transfers in a scenario are free of resource-conflicts.

The software flags being polled are called the "running-flags" of the transfers. However, polling is not an efficient mechanism to utilise these flags. An event-driven TP uses these running-flags much more efficiently.

```
void main(){
......
/*Scenario x: Transfer T1 and T2*/

/*Configure T1 and T2:*/
T1_configuration_instructions;
T2_configuration_instructions;

/*Invoke T1 and T2:*/
T1_Running=True;
T1_invocation_instruction;
T2_Running=True;
T2_invocation_instruction;

/*Poll T1 and T2's notification:*/
while (T1_Running or T2_Running) do_nothing;

/*Scenario x+1:*/
......
```

FIGURE 4.1: The pseudo code of a polling-based test-program. It configures and invokes two transfers T1 and T2 as one scenario. After invocation, the polling statement prevents the test-program from proceeding to the next scenario until both transfers are finished.

4.2.2 Event-Driven Test-Program

The second structure is the *event-driven* test-program also called scheduler, in which polling statements are cancelled [89]. Figure 4.2 conceptually visualises the relation between some transfers (shown as the jigsaw pieces) and the scheduler. The scheduler invokes some transfers and then exits; in turn, transfers can re-activate the scheduler at their completion (notification) event; again, the scheduler may submit (i.e., configure and invoke) new transfers since some resources must have been released by the completed transfer.

When the current scheduler finds out that a waiting transfer would conflict with a running transfer, it won't submit this waiting transfer; but the waiting transfers will eventually be submitted since the running transfers will re-activate the scheduler when they finish, and the re-activated scheduler will attempt to submit any waiting transfer again.

The event-driven execution makes a TP work more like an OS kernel. An OS kernel uses a data-structure called "process-control-block" to record each process' scheduling information; likewise, a TP needs a suitable data-structure to record the running status of transfers. Due to the TP's simplicity, in principle, the data-structure for transfer-scheduling purpose could

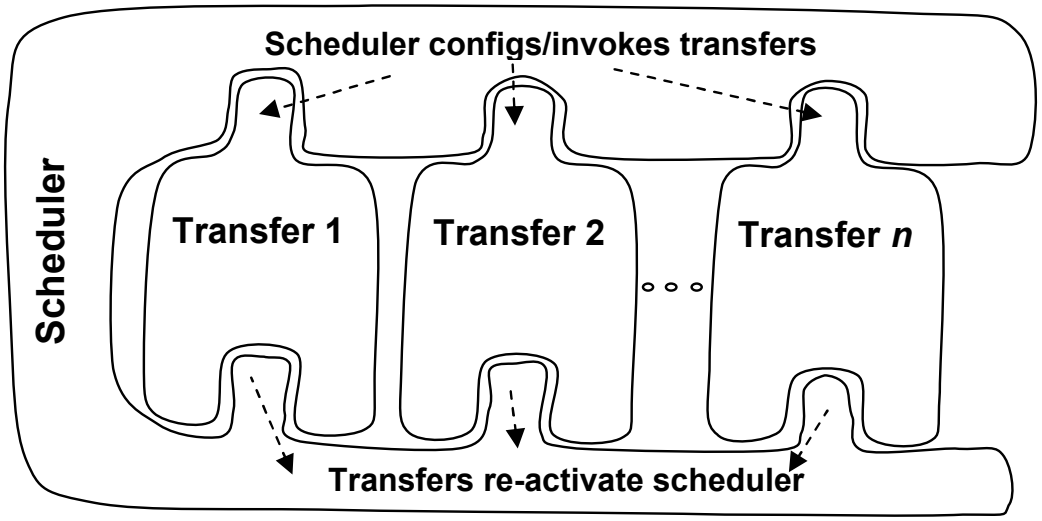


FIGURE 4.2: The relation between the scheduler (event-driven test-program) and the transfers. The scheduler submits some transfers and exits; a transfer re-activates the scheduler when it completes; in turn, the scheduler could submit some new transfers.

be as simple as a *single* bit: 1 for running and 0 for not-running. (Also see Section 3.2.4.) The running-flags naturally play this role. In the Nios SoC, we implement the running-flag as an integer associated with each transfer. Some transfers' running-flags carry other information such as error status and transfer result, while others do contain only one bit of information.

Running-flags are used by the `Scheduler()` to tell resource-conflict. The `Scheduler()` is required to avoid resource-conflicts when it submits transfers; so it seems necessary for the `Scheduler()` to have insight into resource availability, which, however, does not agree with the automation requirement of simplicity. This problem can be circumvented by exploiting the fact that transfers' resource usage is *static*. A transfer's resource usage is already known during test-generation. The test-generator (in place of the test-program) predicts resource-conflicts according to the TRG and then encode the constraints as the *submission conditions* for each transfer. The submission condition for a transfer is expressed in terms of the running-flags of its resource-conflicting counterparts. In this way, `Scheduler()` can avoid conflicts by simply checking running-flags rather than managing the actual resource usage.

The majority of `Scheduler()` function is an "action table" of transfers, implemented as one single `switch` statement, which could be very long. Each entry in the action table includes four elements:

(1) an `if` checking-statement (not a `while` *polling*-statement) implementing the submission conditions of a transfer;

(2) the configuration instructions;

(3) setting up the running-flag; and

(4) the invocation instructions.

Obviously, this regularity makes the automation straightforward. It is for this action table that the automation is possible.

The actual transfer scheduling algorithm is implemented in the function `Scheduler()`, which needs one-time manual programming effort. Compared with the action table, the size of the scheduling algorithm is negligible. Only thirty lines of of C code are for scheduling purpose in the function `Scheduler()` for our Nios SoC.

Figure 4.3 is the pseudo code of the `Scheduler()` with its action table as a separate function. In each calling, function `Scheduler()` will pass a transfer index to the function `Action_Table()`, which executes the matched entry. As the figure suggests, the condition to start the transfer T[0] is that the transfer T[7] is not running (line 4).

```
void Action_Table(int Index){  //Index: transfer ID
    switch Index{
        case 0:                //Entry 0
            if (T[7] is not running){          //Submission Condition
                T[0] Configuration_Instructions;  //Configure T[0]
                T[0] Invocation_Instructions;     //Invoke T[0]
            }
            break;
        case 1: .....
        case 2: .....
    }
}

void Scheduler(){.....
    for (each waiting transfer_instance T[id])
        Action_Table(id);
}
```

FIGURE 4.3: This pseudo code fragment shows how the event-driven test-program works. The `Scheduler()` function passes the ID of waiting transfer to the `Action_Table` function, which is one long `switch` statement. The entry of T[0] specifies that T[0] conflicts with T[7].

To start up the `Scheduler()`, an initial list of the transfer IDs are put in a first-in-first-out queue (FIFO) and then the `Scheduler()` is called. The `Scheduler()` acknowledges the

content in the FIFO and submits the transfers by calling the `Action_Table()` and then exits. When a transfer finishes, its notification instructions will

(1) reset its running-flag;

(2) put its transfer ID in the FIFO; and

(3) reactivate the `Scheduler()`.

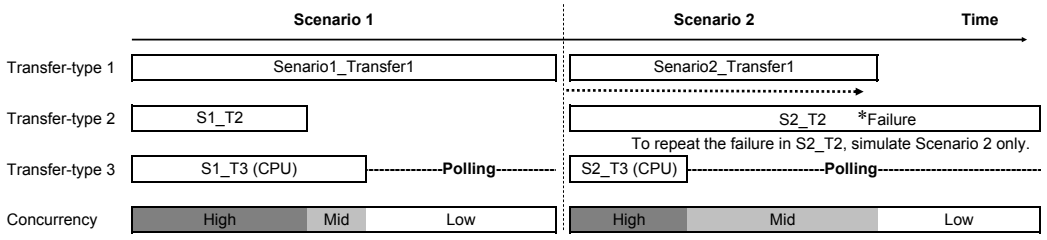
The function `Scheduler()` then attempts to submit new transfers according to the finished transfer ID.

Deadlock and starvation can be easily avoided because of the simplicity of the transfer model. The requirement to avoid deadlock in transfer-scheduling is trivial since any transfer's resource-usage is static and already predicted by the test-generator. To avoid starvation (i.e., transfers being kept waiting) is also simple – when there are more than one transfer to be submitted, `Scheduler()` submits the one having been waiting longest first.

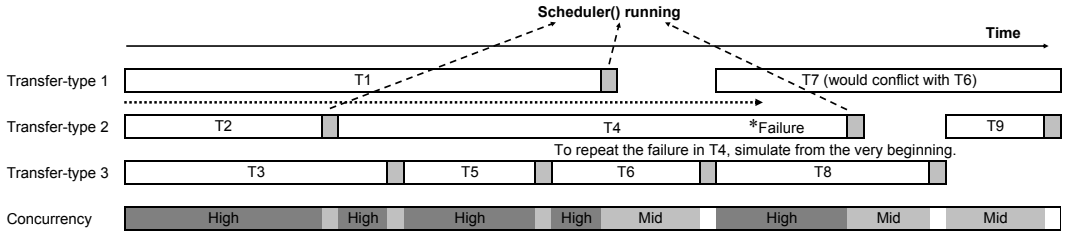
The only challenge is to handle the typical synchronisation issues in concurrent programming; these issues include (i) which parts of the TP should be atomic and which are preemptable, and (ii) the possibility of re-entrance, namely, a new copy of TP running on top of a pre-empted TP. A simple strategy is to make the entire `Scheduler()` (with its action table) atomic. However, this approach actually disables many interesting software behaviours. It is preferable to minimise the scope of atomic operations and to allow TP re-entrance. One guideline to achieve this goal is to limit the atomic operations only to operations on running-flags and the FIFO.

In the real implementation for the Nios SoC, in order to avoid the excessive checking of submission-condition, transfers using the same *master* resource are organised in one queue (hence, the number of queues is determined by the number of independent masters); and `Scheduler()` only attempts to submit the current item in each of the queues. The action table actually includes two tiers of *switch*: the outer `switch` switches on the queue ID and the inner one on the transfer ID in that queue. Also, for efficiency reasons, the action table is not a separate function but direct embedded in `Scheduler()`. The action table is mostly interruptible, while the rest of `Scheduler()` is atomic. The possibility of `Scheduler()` re-entrance is considered, and it turns out that at most two copies of `Scheduler()` could be active. Details of `Scheduler()` implementation can be found in Appendix D.2.

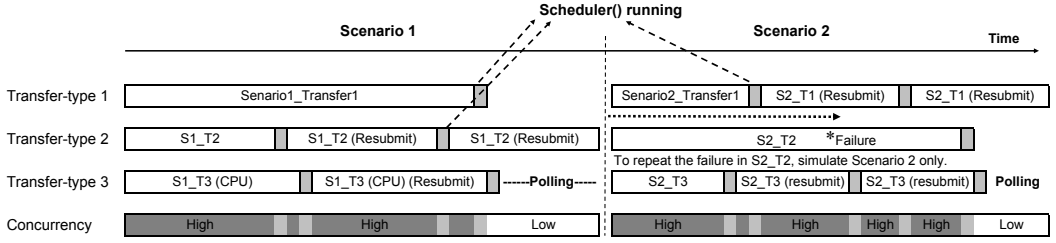
Compared with the polling-based program, the event-driven TP is more advantageous: it avoids inefficient polling statements, so the CPU could devote more time to stimulating



(a) Polling-based TP. Scenarios are separated by polling.



(b) Event-driven TP. Scheduler attempts to submit new transfers when an old one finishes.



(c) Hybrid-mode TP. Scheduler resubmits transfers in a scenario.

FIGURE 4.4: The execution of all test-program structures. In each sub-figure, the first three rows represent three transfer-types that can potentially run concurrently. The shading in the fourth row indicates the degree of concurrency. When an error happens in a polling-based or hybrid test-program, it is possible to isolate the error in one single scenario.

hardware. Meanwhile, the degree of concurrency is enhanced, because the scheduler may submit new transfers as soon as an old one finishes.

The *overhead* of the event-driven TP is small. When dealing with the overhead of TP, it is reasonable not to mix the *overhead to schedule transfers* (e.g., the FIFO operation and submission condition checking) with transfers' own *control overhead* (i.e., the configuration/invocation/notification instructions). Therefore, the real overhead for the event-driven scheme is less than the combined execution-time of the `Scheduler()` and its `Action_Table()`. From another point of view, the event-driven TP requires a robust interrupt mechanism and exposes many interrupt-related HW/SW behaviours; in this sense we shall no longer view an event-driven TP as pure overhead; instead, it directly plays a value-added part in simulation.

Nevertheless, the event-driven TP does have some shortcomings (not present in the polling-based TP) due to the fact that the simulation-time scenarios are not pre-determined at the generation-time.

- If we want to repeat a failure due to the interference among some specific transfers, we have to re-run the whole simulation from the very beginning in order to replay the exact temporal relation, even if the failure occurred near the end of the simulation. The polling-based TP does not always have such a problem (compare Figure 4.4(a) and 4.4(b)). Since scenarios are explicitly written in the polling-based TP, when a failure happens in a scenario, we can comment out all the irrelevant scenarios in the TP to directly repeat the failed scenario.
- For the event-driven TP, the only proper timing we can setup environment parameters (see 3.2.3) is before the `Scheduler()` is initially started. After that, there could be no proper timing to update the environment parameters since there always could be some transfers running. In contrast, polling-based TP can update environment parameters between consecutive scenarios.

These shortcomings lead us to a solution using a hybrid structure.

4.2.3 Hybrid Test-Program

To overcome the above shortcomings, we combine two TP structures into a hybrid scheme. The test generator still predetermines scenarios, and the TP runs each scenario in an event-driven manner. That is, whenever a transfer in one scenario has finished, the scheduler is reactivated and simply re-submits the finished transfer. This process is repeated until a certain condition is met, e.g., until each transfer in one scenario has completed at least once.

The execution of the hybrid TP is shown in Figure 4.4(c). In our current implementation, the polling mechanism is reserved only for a special reason and the polling only works when the scheduler no longer re-submits transfers. There is no principle difficulty in removing all polling statements. The TP is “hybrid” not because the polling is reserved, but because the scenarios are pre-determined.

Re-submitting a finished transfer is meaningful because more temporal relations among the given set of transfers can be traversed. At the logical level, temporal relations specify the order of logical events experienced by the concurrent transfers, such as which transfer starts first or finishes first. At the physical level, temporal relations have finer granularity, up to

a single clock cycle. Notice that most temporal relations (especially physical ones) *cannot* be pre-determined at the generation-time, but can only be "performed" during simulation. Rare temporal relations imply high-quality tests. For example, simultaneous bus accesses and nesting interrupts are relatively rare relations but they are excellent circumstances to verify whether the hardware and software can behave correctly.

Another advantage of the hybrid TP is that the scheduler has even less overhead than its counterpart in the pure event-driven scheme. The scheduler now does not have to check resource-conflicts when it re-submits one transfer, because the current scenario has already been identified as conflict-free in the TRG. We use the hybrid TP structure substantially in the research.

4.3 Interrupt and Interrupt Service Routine

4.3.1 Overview: The Semantics of Interrupts

Verifying the interrupt mechanism is an important aspect of system-level verification. The most famous problem caused by incomplete verification of the interrupt mechanism happened in the Apollo Project – flood of interrupts from radar overloaded the CPU on the Lunar Landing Module, putting the whole project in danger [77].

The interrupt mechanism glues hardware components with software components, introducing rich semantics in the behaviours of a computer system. The detailed description about the interrupt mechanism is detailed by Hills in [46]. We understand the position of the interrupt mechanism in a computer system in the following aspects.

Communication versus Computation

Nowadays, mainstream computer systems are still von-Neumann structured, in which the processor executes instructions and accesses data from memories. Theoretically, von-Neumann structure is just an approximate implementation of the ultimate computation-model – the very simple but powerful Universal Turing Machine (UTM) model [88]. However, the interrupt mechanism implemented in any real-world computer system endows a CPU with a capability that is even not modelled by the UTM. A processor equipped with an interrupt mechanism can sense the events in its environment and reactively jump around independent computation threads. It should be viewed that the von-Neumann structure contributes to the *computation* capability of a processor, while the interrupt mechanism contributes to its *communication* capability.

Parallel versus Sequential

Interrupt sources across a system imply the existence of parallelism (the parallelism within the system as well as the parallelism between a system and its environment). The interrupt mechanism enables a processor, which is essentially a sequential device, to manage the parallelism inherent to a hardware system. Besides hardware-level parallelism or concurrency, interrupts also contribute to *software-level* parallelism. The driving force to run multiple processes "in parallel" on a single-CPU computer is the interrupt from a timer hardware. In fact, interrupt is the *only* entrance to an operating system kernel to manage process-level parallelism.

Hardware versus Software

The interrupt mechanism straddles the hardware-software boundary. The hardware part includes the interrupt issue/arbitration/response subsystem; the software part refers to *interrupt-service-routines* (ISRs). Intelligent peripherals require simple ISR services, whereas primitive peripherals need smart ISR services. Also, the *context switching* associated with an interrupt can either be serviced by hardware or by software. All these phenomena suggest that the interrupt mechanism is where HW-SW transition takes place.

Because of these rich semantics, verifying the interrupt mechanism crosses the boundary between hardware verification and software verification. Methods in [43, 91] focus on the hardware side. These methods treat interrupt handling as the capability *attached* to the processor. An important aspect is to verify the interference between the processor's interrupt-response behaviours and the processor' pipeline behaviours. Interrupt could also be verified from software point of view. Researches in [71, 72] attempt to encapsulate the *interrupt verification* into *thread verification*, based on the observation that there exist some similarities between the semantics of interrupt and thread, and *thread* is already a well defined object-under-test in the field of software verification.

Since interrupt is a phenomenon crosses the HW/SW boundary, verifying it either purely in the hardware domain or purely in the software domain is flawed. Encapsulating interrupts as threads could suffer fidelity and coverage issues from hardware point of view; and such approaches need OS support, which is impractical at the early integration stage. On the other hand, treating the interrupt mechanism as signal-level request/response fails to verify the new properties caused by HW-SW synergy. These new properties include

- the basic interrupt handling *functionality*,
- *performance* issues such as latency and turn-around time, and

- *reliability* issues such as nesting, re-entrance and overloading.

The verification of the interrupt mechanism can be incorporated into our interaction-oriented methodology. Again, we do not treat interrupts as the capability or property *attached* to hardware components. The view that (a) some components have the capability to issue interrupts and (b) some other components have the capability to handle interrupts is inherently component-oriented. Instead, we interpret interrupts as properties associated *interaction*-objects and incorporate the interrupt mechanism *seamlessly* into the transfer model.

4.3.2 Incorporating Interrupts into Transfer Model

Interrupts are incorporated into the transfer model in two forms; these two forms do *not* cancel each other.

1. *Data-intensive* interrupts are modelled as *virtual masters*, which involves in virtual transfers. (Also see Section 3.2.2.) In this case, the corresponding ISRs should be coded to mimic a master devices' behaviours. Treating a data-intensive ISR as a (virtual) master is simply a reasonable reflection of the decision made at the HW-SW partition stage that no hardware (real) master is dedicated to the data-flow. Figure 4.5 illustrates the idea of treating one data-intensive interrupt (`rrdy`, or UART-receiver-ready) as a virtual master in the Nios SoC, separately from the CPU's normal operations.

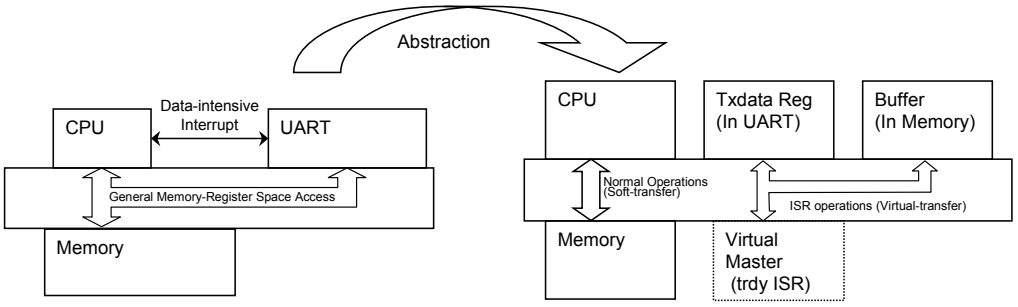


FIGURE 4.5: Modelling the UART transmission as a virtual-transfer. The CPU's behaviour in the UART interrupt-service-routine (ISR) is separated from the normal operation of the CPU. The latter is modelled as a soft-transfer, and the former is modelled as a virtual-transfer. The resources used by the ISR are also separated.

2. *General interrupts* that would be triggered in a transfer, regardless of data-intensive or control-intensive, are regarded as *events* to that transfer; enabling/disabling these events could be implemented as that transfer's parameters. In this spirit, whenever

possible, ISRs should be coded to *map* physical (hardware) events to logical (transfer) events, and then perform proper operations *in the name of the mapped transfer*. Again, this mapping reminds us to migrate from the component-oriented mindset to the interaction-oriented mindset, namely, to *bind events to transfers rather than to hardware components*.

Table 4.1 lists how each interrupt source in the Nios SoC is incorporated into the transfer model.

ISR		Virtual Master	Transfer Event	Applicable Transfers
UART	Parity Error	No	Transfer Error	RX-by-DMA/Interrupt/Polling
	Frame Err	No	Transfer Error	RX-by-DMA/Interrupt/Polling
	Break	No	Transfer Error	RX-by-DMA/Interrupt/Polling
	Receive Overran	No	Transfer Error	RX-by-DMA/Interrupt/Polling
	Transmit Overrun	No	Transfer Error	TX-by-DMA/Interrupt/Polling
	Transmit Empty	No	Transfer Status	TX-by-DMA/Interrupt/Polling
	Transmit Ready	Yes	Transfer Status (always disabled)	TX-by-DMA/Interrupt/Polling
	Receive Ready	Yes	Transfer Status (always disabled)	RX-by-DMA/Interrupt/Polling
	End of Packet	No	Transfer Status	Any TX/RX transfers
DMA Engine	Read EOP	No	Transfer Finish	RX-by-DMA
	Write EOP	No	Transfer Finish	TX-by-DMA
	Length Reached	No	Transfer Finish	Any DMA transfers
Timer	Time-out	Yes	Transfer Status	Timer-countdown

TABLE 4.1: Incorporating Nios system interrupts into the transfer model. Control-intensive interrupts are treated as events to transfers; and data-intensive interrupts are treated as (virtual) transfers as well as events.

Both the above treatments are coherent with the transfer model. The first treatment gives us a layer of abstraction to model data-intensive HW-SW interactions, while the second is even more beneficial: once an interrupt is mapped to a transfer event, any subsequent code in the ISR could be regarded as the extension of that transfer's behaviour. In other words, transfers are equipped with the "calling capability" to call other codes.

Hence, transfers are no longer *passive* objects merely subject to arrangement, but become *active* building blocks which are potentially inter-connectible. Actually, the event-driven TP described in Section 4.2.2 is realised by taking advantage of transfers' calling capabilities at their completion event. In Figure 4.6, an enhanced transfer model is conceptualised as a jigsaw piece with more convexes and concaves. The convexes represent transfers' "control-handles" (including the configuration and the invocation), and the concaves represent their

"calling-capabilities" at their events. With these convexes and concaves, it is possible to interconnect transfers in more creative ways.

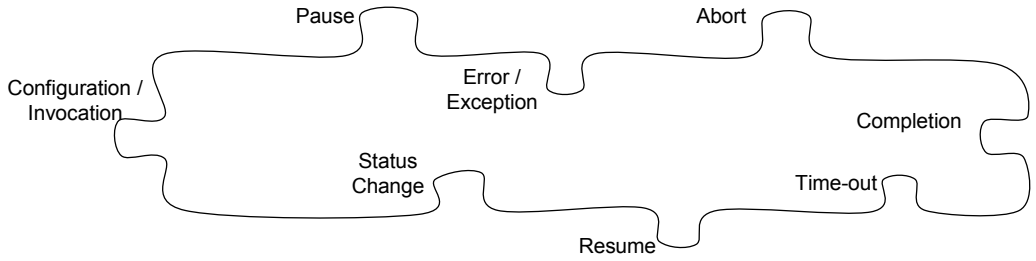


FIGURE 4.6: Enhanced transfer model. A convex represents a "handle" to control the transfer; and a concave represents the "calling capability" of the transfer. Therefore, transfers can potentially be interconnected in various ways, deserving the name of the "building-blocks" of test-programs.

For instance, we currently treat "timer-counting" as a simple and stand-alone transfer which uses dedicated resources. It is a peer to other transfers and never conflict or logically interact with them (see Figure 3.4); but we could exploit this transfer's calling-capabilities at its time-out events to pause/resume and invoke/abort other transfers. This scheme is actually implementing a time-division multi-tasking mechanism on a single CPU system. However, if we implement a test-program in such a sophisticated manner, we are actually leaving the domain of hardware verification and entering the domain of software verification. Here, we see that the transfer model connects the two domains, and the transfer model simply with configuration/invocation/notification is a good choice for SoC verification at the integration stage.

In order to map a physical *hardware-event* into a logical *transfer-event*, an ISR should associate the hardware-event (i.e., the interrupt firing) with a transfer. The relation between a hardware-event and a transfer is *many-to-many*.

- One hardware-event may occur in many transfers. For example, the event "DMA Completed" could happen in (1) the transfer "Memory-to-Memory-by-DMA" and (2) the transfer "UART-RX-by-DMA".
- One transfer can have many hardware-events. For example, the transfer "UART-RX-by-ISR" has (1) the event "RX Data Ready" and (2) the event "RX Parity Error".

Table 4.1 also infers the many-to-many relation between hardware-events and transfer-types. However, in order to convert a hardware-event to a transfer-event *deterministically*, there is

a hidden assumption or restriction about the many-to-many relation. That is, one hardware-event should correspond to *only one currently active* transfer-type. In other words, *concurrent* transfers in a valid scenario should *not* share interrupt sources. (In fact, this restriction could still be modelled by the TRG model if we treat interrupt sources as exclusive resources.)

We believe that this assumption is reasonable. The following discussion shows that the only violation of this assumption in the Nios SoC has caused negative consequences.

The only violation is the UART `end-of-packet` (EOP) interrupt, which happens either (1) when the `txdata` register has be written with an EOP character – the same character stored in the `eop` register, or (2) when the `rxdata` register has received the same EOP character.

Although it is intuitive to treat a UART RX transfer-type and UART TX transfer-type as two independent and concurrent interactions, the shared EOP interrupt caused some complications in the UART ISR, since it is unclear whether an EOP event is caused by the receiving (RX) or by the transmission (TX) or RX/TX *simultaneously*. In fact, it is quite problematic for an ISR to differentiate these cases (and work accordingly).

Since the RX and the TX are two independent streams, it is always possible, though rare, that an EOP is received and transmitted *simultaneously*; so the `eop` ISR must check RX and TX using *two independent checks*, which is obviously an undesired overhead considering the rareness of the case. When checking whether the EOP event is happening during RX, we might want to let the ISR compare the `rxdata` register with the EOP character. However, reading the `rxdata` register has a *side-effect* – the `rrdy` (receiver ready) signal will be reset by a `rxdata` read-operation. This side-effect will disrupt a normal RX stream. A working-around is to let the RX stream leave a copy of `rxdata` in a variable, (which is an overhead again) and let the `eop` ISR read the copy instead of the actual `rxdata` register. This solution would work only if we also could make sure that the copy is *strictly* synchronised with the `rxdata` register. This brings further overhead codes – the protection instructions to guarantee that the copy is atomic. Even if we finally come up with a functionally correct ISR to handle all these complications, we are already violating the general principle of composing ISR of minimum turn-around time. This EOP issue is not just inconvenient for modelling in the verification process; the real-world UART application will eventually face the same dilemma: to sacrifice performance for functionality, or even worse – the functionality may still be flawed even we agree to sacrifice performance. In Appendix A.1, plausible problems related to this EOP issue are discussed in detail.

The most appropriate solution is to separate an `RX-EOP` event from a `TX-EOP` event *in hardware* by providing additional hardware resources – either two independent interrupt sources

or two separate EOP status bits in place of one. Therefore, we believe that this EOP modelling issue reveals the inadequacy of the hardware design rather than the invalidity of the assumption in the interrupt modelling.

4.3.3 General Form of Interrupt Service Routines

Unlike the TP, which is to be generated automatically, ISRs need to be manually programmed. However, since the interrupt mechanism seamlessly merges into the transfer model, we are able to give strong guidances on how to compose ISRs. As a result, ISRs, irrespective of the underlying hardware, could share a general form.

First, we discuss the general requirements to an average ISR.

Functionality:

- An ISR should map the hardware interrupt to a transfer-event and reflect the status-change in the transfer's running-flag. These mapping operations implement the interaction-oriented mindset.
- An ISR should perform proper operations on the hardware to fulfil the design functionality intended for the hardware. The mapping described above could help an ISR to decide which operations are appropriate.
- An ISR may serve as the entrance to the `Scheduler`. This facilitates the "event-driven" scheme.

Performance:

- Latency – the codes that service the interrupt source should be executed as soon as possible.
- Turn-around time – the execution of an ISR should be brief enough compared with a normal function. Our verification interest is more about the *interaction* between the interrupt subsystem and other hardware/software components than about the ISRs themselves. So we expect *frequent interrupt firings and short ISR executions* instead of infrequent firings and long ISR executions.

Reliability:

- Preemption and re-entrance. An ISR should allow preemption by other interrupts *without losing its functionality*. This will allow us to focus on the *interactions* between interrupt subsystem and other HW/SW components.

- Overloading – An ISR, should be stressed under extreme circumstances in order to reveal potential HW/SW hazards not detectable in normal situations.

Some of these requirements to build ISRs may compete with each other. For instance, a reliable ISR could gain safety by performing many checks before hardware operations, sacrificing the latency; and when an ISR preempts another ISR, the former could achieve a good latency at the cost of the longer turn-around of the latter. However, under the principle of "pushing everything to the extreme of what is allowed", we should not put too much artificial constraints in composing ISRs, such as (i) disabling preemption and (ii) extensive safety checks. Execution of ISRs should be reasonably fast, safe, independent and allowing preemption.

The TRG-based test-generation method allows ISR-composing to meet the above requirements. This is because, a *scenario* in a TRG is formally defined as concurrent transfers without contradictory resource usage; the *resources* here include any control/status bits that will be accessed throughout the transfers' life cycle, including the interrupt service stage. Therefore, concurrently active ISRs are guaranteed to be logically un-interfering. As a result, there is no need to resort to the arbitrary assigned interrupt priority scheme for correct ISR execution.

For instance, in the Nios SoC, two concurrent transfers "UART-RX-by-ISR" and "UART-TX-by-DMA" will respectively generate UART interrupts and DMA interrupts. Although both the UART ISR and the DMA ISR will read-modify-update the same UART control register, they do not *functionally* interfere with each other because each of them is accessing a *different set of bits* in that register. The fact that the two sets are physically located in the same register is the only reason to require the read-modify-update operations in both ISRs to be atomic to each other. Meanwhile the read-modify-update operations are unnecessary to be "atomic" to other irrelevant interrupts, e.g., interrupts from the Timer, which will never touch the UART registers. This instance shows that the scope of atomic operation can be minimised.

In a word, since the TRG has already avoided logical conflicts *at the test-generation stage*, the ISRs can avoid excessive overhead of safety operations *at the simulation stage*; the codes in the ISRs now can focus on their due functionalities. Also, the scope of atomic operations is minimised, allowing interesting preempting behaviours to happen.

It is possible to follow a guideline to compose low-overhead ISRs in a general form, regardless of the underlying hardware generating the interrupts. The simplicity and uniformity are eventually due to the interaction-oriented TRG method. Figure 4.7 shows the proposed

general ISR structure. The context-switching part is not included in the structure since it could be implemented by hardware or firmware. The general ISR structure works in the following manner.

1. The ISR firstly copies the control and status registers from relevant hardware to local variables. The benefit is two-fold – (1) the subsequent ISR code could work faster with the local copy, and (2) the subsequent operations on the copy do not need to be atomic. Then the ISR identifies the physical reason of the interrupt.

2. The ISR maps the interrupt into transfer-event by checking transfers' running-flags, and obtains the transfer ID *tid*.

3. The ISR performs proper operations to service the interrupt. If the interrupt is data-intensive, this is the place where the virtual-master's behaviours are coded. Otherwise, the proper control operations are written to the local copy of registers.

4. Optionally, the ISR could "log" the event, either as a physical event or as an logical event bound up with transfer *tid*. The actual logging activity is performed by the test-bench; the ISR simply issues a command to the test-bench. (The SW-TB relationship will be discussed in the next chapter.)

5. The ISR proceeds to exit:

   - If this event is not a transfer-finish event, then the ISR updates the real control and status registers with the modified bits in the local copy and exits.
   - Otherwise (transfer finishes), the ISR (1) resets the running-flag of transfer *tid*, (2) resets the appropriate bits in the real control/status registers, then (3) pushes *tid* into a FIFO, and (4) finally hands over the control to function `Scheduler()`.

In this general form, ISRs only perform necessary operations and many of them are preemptable. The overhead (shaded boxes) caused by the transfer-model is reasonable, including the mapping to *tid*, the *optional* logging and the *possible* calling to the `Scheduler()`. The protected operations (shown as the underscored statements in Figure 4.7) include (1) the operations on the real control registers; (2) resetting the running-flag; and (3) the FIFO push operation.

Appendix D.3 lists the implementation of the ISR of the UART, which is the most interrupt-intensive hardware in the Nios SoC.

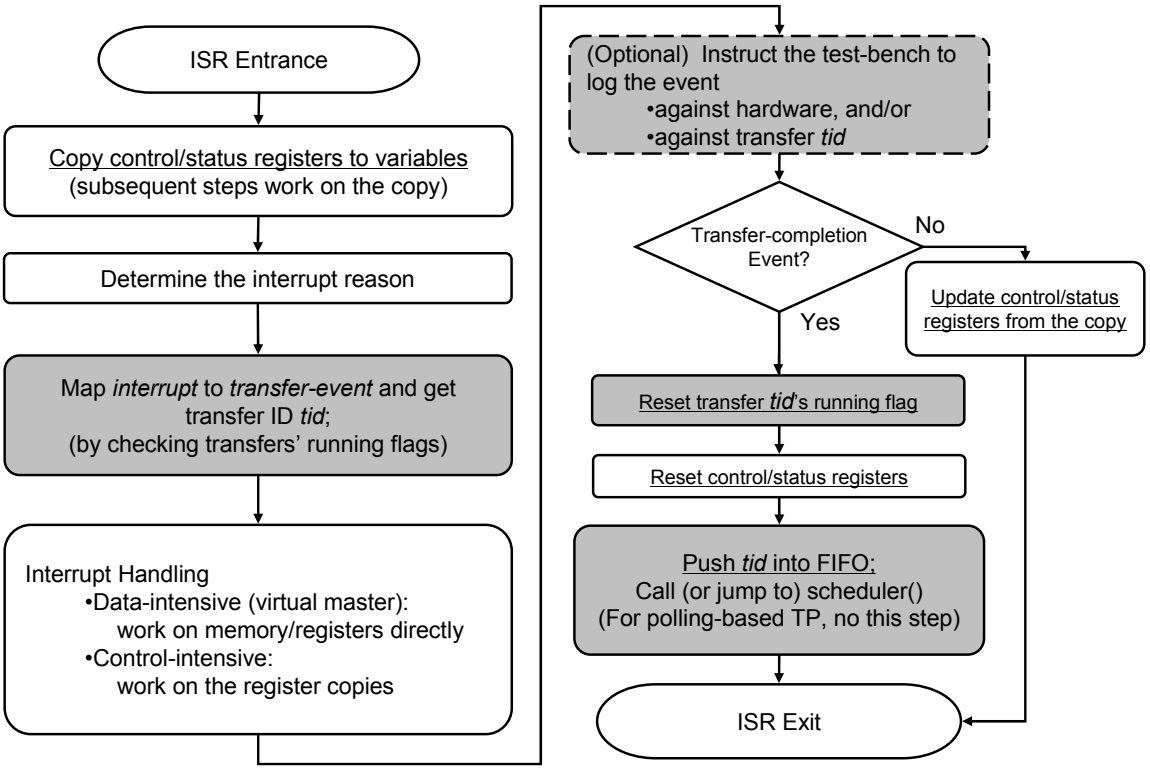


FIGURE 4.7: The general interrupt-service-routine (ISR) structure. The underlined operations need to be protected. The shaded blocks represent overhead operations induced by the methodology; even if these operations are removed, the remaining code is still sensible as interrupt service.

4.4 Guidelines to Soft-Transfers

The Role 2 software components are supposed to *stress* the hardware; they literally implement the idea to "use software to test hardware". This nature makes them very dependent on hardware and generally requiring manual development. This section briefly discusses how to seamlessly incorporate these components in the interaction-oriented verification methodology.

These software components could be modelled as transfers and hence referred to as the "soft-transfers" (see Section 3.2.2). Therefore they should follow the general guidelines for transfers, including (1) implementing data-intensive computation and/or communication to stress relevant hardware, (2) having suitable life expectancy, (3) using resources in a static manner, and (4) allowing interrupts.

Software inherently has greater flexibilities than hardware. But we should *prevent* soft-transfers from meddling with the roles of other software components. For instance, a soft-transfer should not determine its own behaviour based on the behaviours of other transfers; neither should it control other transfers' behaviours. In fact, a soft-transfer is not supposed to be aware of the existence of other transfers. Transfer management is the responsibility of the TP (Role 3 software). Another requirement is that a soft-transfer should not disable interrupt unless absolutely necessary; if it disables interrupt for some atomic operations, it must re-enable it as soon as possible; otherwise, other concurrently active transfers will be disrupted.

It is trivial to implement the "calling-capability" for a soft-transfer. Being software by itself, a soft-transfer can call any other code right away. But it is an ideal situation to use a different calling mechanism, i.e., the `TRAP` instruction, in place of a normal `CALL` instruction. A `TRAP` instruction – also referred to as the *software-induced interrupt* – bears the similar semantics as a hardware-induced interrupt.

- The `TRAP` instruction is associated with an *interrupt vector* just like a hardware-induced interrupt.
- The `TRAP` instruction will be serviced by a service routine specified by the vector, just like a hardware-induced ISR. At the end of the service routine, the control returns to the soft-transfer.
- For advanced processors equipped with hardwired security features, the `TRAP` instruction, just like the hardware-induced interrupt, switches the processor from the "user mode" to the "supervisor mode". In contrast, a normal `CALL` instruction does not.

Therefore, using the `TRAP` instruction as soft-transfers' calling capability reflects the rationale to treat some software behaviours and hardware behaviours *equivalently* as "transfers". In addition to the opportunity to test the basic functionality of the `TRAP` instruction, taking into account the parallelism we build in test-programs, we are also able to test the interactions between the `TRAP` mechanism, which is the software-induced interrupt, and hardware-induced interrupts.

Soft-transfers, explicitly using the processor as its resource, may still appear special compared with hard- and virtual-transfers at the *invocation*. The program control remains in the TP after TP invokes a non-soft transfer; while invoking a soft-transfer implies that the program control moves from the TP to the soft-transfer itself. For the Nios SoC, our test-generator, and thus the test-program as well, does consider this particular fact and treats soft-transfers

slightly differently: when multiple transfers need to be invoked, the soft-transfer is always invoked last.

As illustrated by Figure 4.8, this specialty can be abstracted away so that non-soft transfers and soft-transfers can be invoked in any order. At the entrance of a soft-transfer, we could arrange a special event – the "TRAP-to-TP" event. As the name suggests, this event is simply a `TRAP` instruction which activates the TP. Thus, right after the soft-transfer is invoked, the "TRAP-to-TP event" happens, and the TP regains the program control to finish any operations left unfinished by the last invocation of the soft-transfer. After this new execution of TP finishes its job, the program control returns to the soft-transfer.

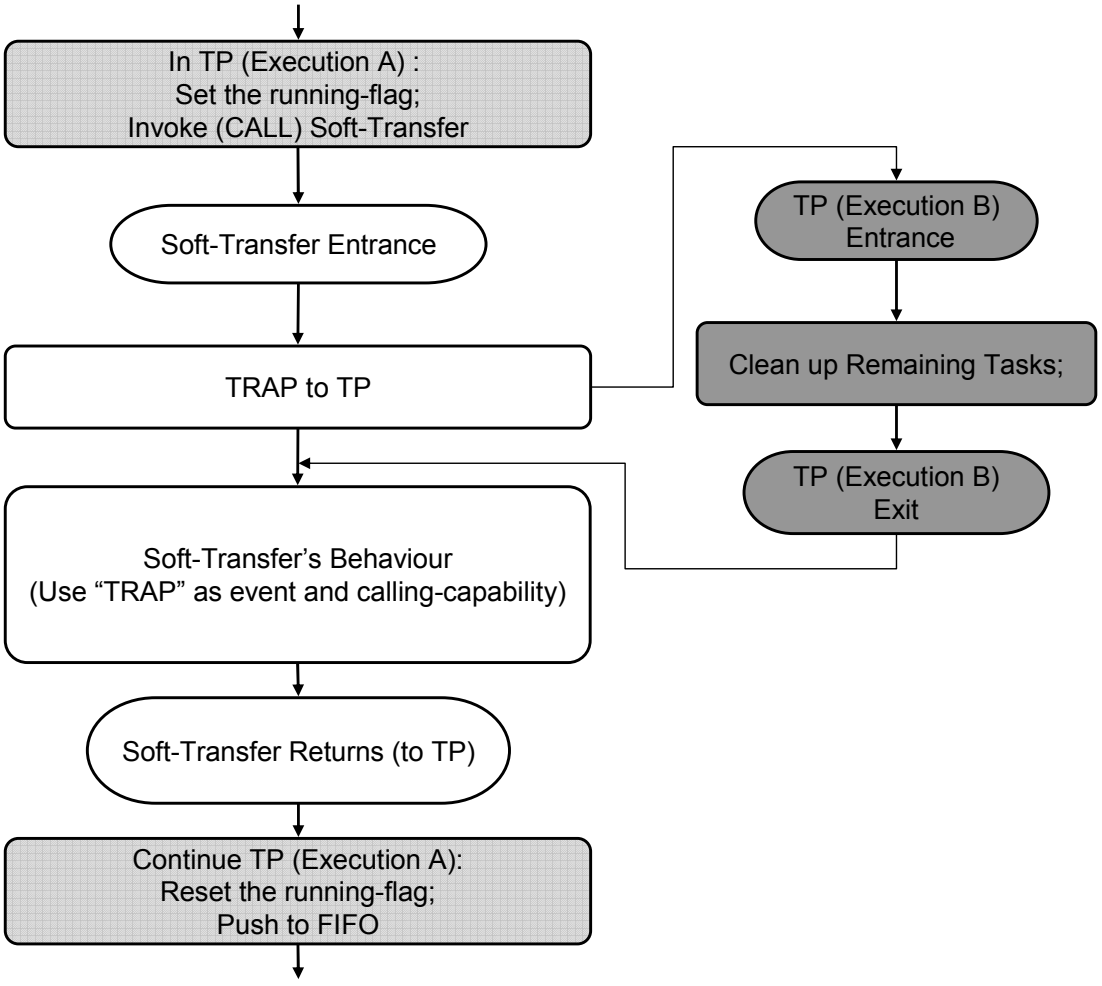


FIGURE 4.8: The general soft-transfer structure. The test-program (TP) sets the running flag and invokes the soft-transfer using a normal CALL instruction. The soft-transfer uses TRAP instruction to activate a second execution of the TP. It also uses TRAP instructions as its calling capabilities and events. When the soft-transfer finishes, the original TP resumes control. It resets the running flag and performs FIFO operations.

In this scheme, the `TRAP` instruction is used to move the control from the user-defined payload code (the soft-transfer) to user-independent overhead code (the TP). In the real-world application, the `TRAP` instructions is used in a similar fashion: a user-mode application uses `TRAP` to request OS services, which operate in supervisor mode.

Soft-transfers could come from various sources. They could be manually developed by verification engineers, or come from legacy hardware diagnostics. Soft-transfers for the purpose of CPU-self-testing can be generated under another level of automation (beyond the scope of this thesis), for which a number of methods have been proposed. For example, randomisation techniques [14] could be good sources of soft-transfers. Soft-transfers can also be decomposed and tailored from the real SoC application software so that application software can have an earlier entrance in the SoC verification process.

Appendix D.4 lists the implementation of a soft-transfer for the Nios SoC which exercises the CPU-to-memory read/write operations.

4.5 Summary

Software execution is the driving force of the TP-centric verification. Owing to its flexibility, software could take various roles. We have partitioned software into three independent roles, and proposed automation schemes or development guidelines for these roles. These roles render the interaction-oriented mindset from different perspectives.

Although it is the test-program that drives the simulation in our methodology, the test-bench, which is the driving mechanism in the traditional simulation-based verification, is still needed. What is the relationship between the test-program and the test-bench? How do they interact? The next chapter discusses these questions and shows how to place test-programs and test-benches harmoniously in the verification framework.

Chapter 5

Test-Bench and Post-Simulation Support

5.1 Overview: Unifying the TP, the TB and the DUT

In test-bench centric verification methodologies, it is the test-bench (TB) that provides both stimulation and observation facilities to the design-under-test (DUT). However, the software (SW) native to the SoC DUT is not taken into account, therefore the software's position in verification has not been well recognised. Although there is growing awareness that software could take significant responsibilities in SoC verification and there are proposed techniques about SW-TB interaction [20], the overall SW-TB relationship has not been clearly discussed. This is because *both* the test-bench and the software can control and observe the SoC DUT, and this overlap in responsibilities gives rise to conceptual confusions about their relationship.

In our methodology, the test-program (TP, the software native to the SoC) drives the simulation process. Although a test-bench is still needed, we now have an opportunity to resolve the conceptual confusions about the SW-TB relationship. This chapter discusses the test-bench's roles in our TP-centric SoC verification methodology and gives an overall solution to the question about TB-TP-DUT relationship.

The control and observation capabilities provided by the TP and the TB have their respective strengths and limitations as listed in Table 5.1. It should not be too surprising to see that the TB's strengths exactly complement the TP's limitations and vice versa. This is because the TP is software, slower but flexible, and the TB is hardware (in the simulated world), faster but less flexible.

	DUT Control	DUT Observation
TP	•Logical level write-operation; •Native to DUT; •Sequential but highly orchestrated.	•Limited to read-operation and interrupt mechanism; •Affecting DUT's behaviour.
TB	•Physical level signal-feeding; •Brute-force to DUT; •Parallel but poorly coordinated.	•Ubiquitous observation; •Un-instructive to DUT's behaviours.

TABLE 5.1: Comparison between test-benches and test-programs' capabilities to control and to observe.

Therefore, the TP and TB should cooperate with each other in control and observation. In our TP-centric methodology, the TP replaces the TB for the more active tasks of parallelism *control*; while the TB is particularly suitable for the *observation* tasks. The term "ubiquitous observation" in Table 5.1 refers to TB's potential to observe anything, i.e., any register or wire, at anytime, i.e., at any simulation cycle. Moreover, this type of observation is *non-intrusive* to the DUT's behaviour. In contrast, software could observe hardware only through read operations and interrupts; and these limited observation mechanisms also modify the DUT's behaviour.

In Section 2.3.1, we have already presented the idea to treat the TB and the DUT uniformly as a "DUT-TB super-system". But this is still not the full picture. The boundary between the software and the DUT is not clear-cut either; hence, we could treat the TP and the DUT collectively as a "HW-SW super-system". The idea of "super-system" in each case is reasonable since

- for the DUT-TB super-system, a hardware component could be counted either on the TB side or on the DUT side. For instance, an extensively verified DUT component could work as a bus-function-model (a TB component) to another DUT component;
- for the HW-SW super-system, an interrupt-service-routine (ISR, a software component) collaborating with the hardware actually fulfils the hardware's intended functionality. So this ISR could also be interpreted as one part of the DUT.

Therefore, from the pure simulation point of view, the DUT, the TP and the TB could be collectively understood as *one continuum* rather than three stand-alone entities. Figure 5.1 illustrates this concept.

Nevertheless, different portion of the continuum should have different responsibilities in verification. Since the TP is taking the more active "control" tasks, it is very natural

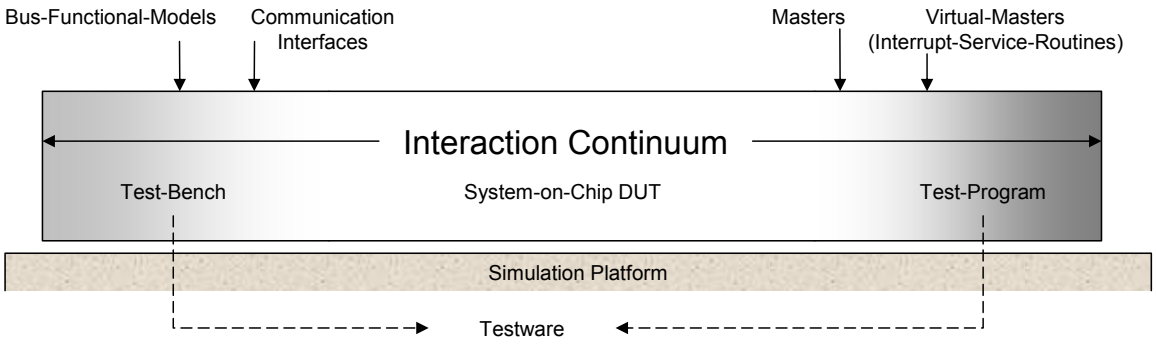


FIGURE 5.1: The TP-TB-DUT Continuum. The TP is continuous with the DUT in the sense that some software components work like a hardware component and are subject to verification; the test-bench is continuous with the DUT in the sense that treating a component on the DUT side or on the TB side is quite subjective. This continuum connects the *software phenomena in the real world* (the TB) and the *software phenomena in the simulated world* (the TP).

to "observe" the TP's behaviour, or more generally, the software's behaviour, in order to understand the quality of the control. The TB is the right vehicle for software observation (detailed in Section 5.3.2).

The overall TB-TP-DUT relationship, together with their responsibilities, can be visualised in Figure 5.2. A *single* "DUT-TB super-system" is under the TP's *unified control*; meanwhile, a *single* "HW-SW super-system", which is made of the SoC and its TB, is under the TB's *ubiquitous observation*. By this scheme, each of the test-program and the test-bench is exerting its strengths to make up the other's limitations.

The construction of test-benches is already the central topic of the mainstream verification methodologies; therefore, this chapter does not intend to discuss the common facilities a test-bench should provide. Instead, we will focus on

- how to realise the "DUT-TB super-system" by implementing the TP-control-TB mechanism, and
- how to realise the "HW-SW super-system" by implementing the TB-observe-TP mechanism and the post-simulation facility.

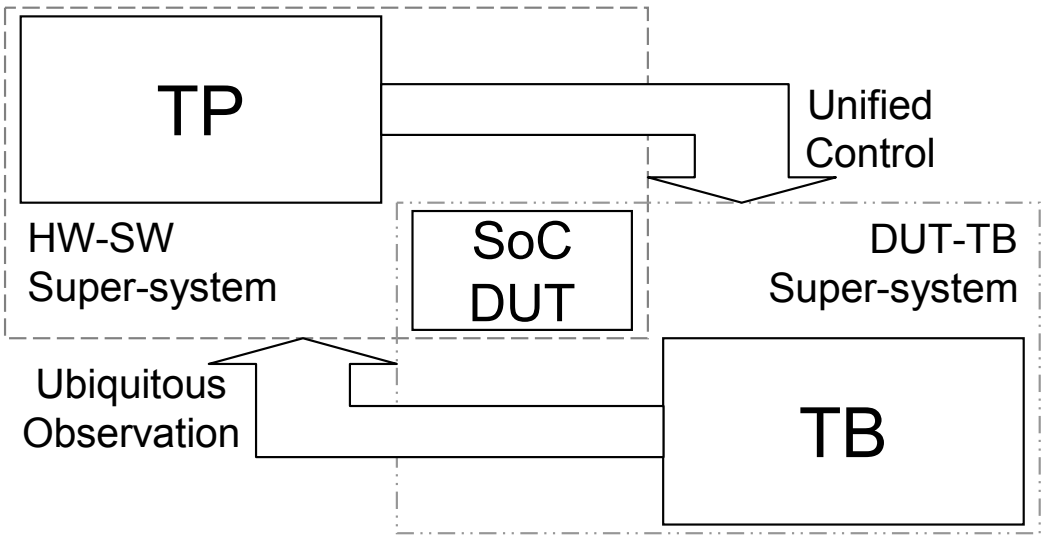


FIGURE 5.2: Position the test-program and the test-bench in the verification framework. The test-program and the test-bench co-operate with each other. The test-program takes the creative high-level control, including controlling the test-bench; while the test-bench is suitable for the onerous but mechanical observation tasks, including observing the test-program.

5.2 TP Controls TB

A DUT and its TB are becoming two very different entities in TB-centric verification methodologies; developing a TB and developing a DUT require very different techniques. Costly propriety hardware-verification-languages (HVLs), (alongside immature hardware-description-verification-languages, or HDVLs, such as SystemVerilog), are replacing hardware-description-languages (HDLs) in constructing complex TBs; HVLs provide sophisticated software constructs, allowing TBs to be composed under the true object-oriented-programming (OOP) paradigm; while the DUT is still required to be constructed with the basic *synthesisable subset* of HDL constructs. In this sense, the TB-centric verification methodologies promoted by EDA vendors have *enforced* the "DUT-TB dualism", which we intend to *eliminate*.

From the pure simulation point of view, a DUT and a TB share a common nature: they are both hardware entities in the simulated world, or the virtual world; meanwhile, they are also software entities in the *real world*. The current TB-centric methodologies tend to interpret TBs as pure software phenomena in the real world and provide extensive support to TBs' OOP capabilities and even beyond [16]. We argue that the ultimate purpose of simulation is to study the DUT's behaviours in the *simulated world*. Hence, we should not over-invest on the TB's software-aspect in the real-world, but understand the TB largely as hardware

in the simulated world just as the DUT is. We should instead invest more efforts on the software in the *simulated world*, for instance, on the test-program (TP).

Under the philosophy to treat a DUT and its TB as peers, the TP should be able to communicate with the TB, just as it is able to communicate with the DUT through the "programming interface" of the DUT. It follows that a *TP-TB interface* similar to the DUT's programming interface should be provided.

5.2.1 TP-TB Communication

A TP-TB interface helps to hide the distinction between the components in a DUT and the components in its TB, validating the concept of "DUT-TB super-system". For instance, to configure the transfer "UART-RX-by-DMA", the TP must set up the RX transactor, which is in the TB, to generate the bit stream, just as the TP needs to set up the UART and the DMA engine, which are in the DUT, to convey and receive the stream. In this sense, the TP does not differentiate components in the DUT and components in the TB and treat them uniformly. The TP programmer only sees one *single* system, i.e., the DUT-TB super-system, under the TP's unified control.

Nevertheless, the fact that TB components are *not native* to a DUT applies some constraints to the TP-TB interface implementation:

- the TP cannot communicate with each TB component *directly*, since TB components don't provide their own programming interface;
- the TB-to-TP communication is harder than the TP-to-TB communication, since TB components cannot interrupt the CPU as many DUT components can.

These constraints suggest the need to implement a *central* control in the TB and let TP communicate with it through a centralised interface. Figure 5.3 illustrates this TP-TB communication mechanism. Using some reserved memory locations as the TP-TB interface, the TP issues logical-level commands and parameters to the *TB-central-control* (TBCC), which in turn autonomously manages the rest of the TB *without* the TP's further interventions.

This communication mechanism is mostly used to configure the bus-functional-models' (BFMs) behaviours; but it also allows the TP to gain some controllability over the simulation environment, including the simulation process itself. For instance, if the TP detects an abnormal event such as unexpected software failure, it has the option to instruct the TB to pause the

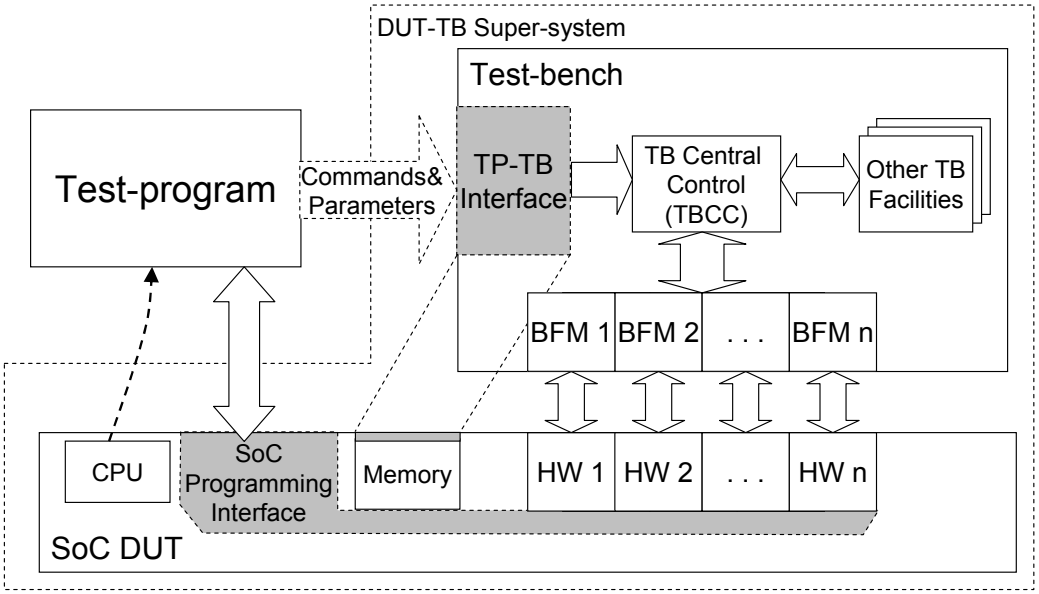


FIGURE 5.3: The test-program controls the test-bench through an interface (the upper gray box). The interface is implemented as a few reserved locations in the SoC memory. Meanwhile, the test-program controls the SoC itself via the "programming interface" (i.e., program-accessible registers in the system). With these interfaces, the test-bench and the SoC could be viewed as one single system, which is under the control of the test-program.

simulation. This usage provides valuable debugging opportunities *without* the support of a software debugger, which is unavailable as well as inappropriate at the hardware integration stage since it (if available) only freezes the software but not the hardware.

The cost to implement such a centralised TP-TB interface is low. For simulation, the interface simply consumes a few memory locations. If the TB and the DUT are in the form of hardware prototypes, the interface requires a couple of dedicated registers. For the Nios SoC used in the research, we implemented the interface as 16 32-bit words at the beginning of the on-chip RAM. Eight of them are for the TP-to-TB communication; and the other eight, reserved for TB-to-TP communication, are not used. The C codes of the interface is listed in Appendix D.1.

5.2.2 TB's Control Facilities

What kind of "control facilities" should the *TB* provide in a *TP-centric* verification approach? Some of them are already shown in Figure 5.3 and described as follows.

- The TB-central-control (TBCC) module should implement the "autonomous control" over various TB components. The TBCC does not need too much intelligence, since

it simply performs whatever the TP instructs it to do. (The intelligence in the TP is automated by the TRG-based test-generator.) In contrast, TB-centric verification methodologies require additional TB layers and components for high-level control and coordination, making a TB very sophisticated. For example, VMM [17] introduces additional TB components called Extensible Verification Components (XVC) and XVC manager [7] to schedule the user-specified test scenarios. In the end, it is still the user, not the TB, who is responsible to generate abstract test scenarios.

- Each TB component performs whatever the TBCC tells it to do. Again, this kind of mechanical tasks does not need too much intelligence. On the other hand, this is the right place to exploit the "constrained-randomisation mechanism" in the hardware-verification-language (HVL) to concretise properties that are not specified by the TP. For example, via the TBCC, the TP may instruct the UART RX transactor to provide an RX stream with the property "length equal to 50 and the last byte is 0x127". Then the RX transactor could create and feed an instance of byte-sequence with this property using the constrained-randomisation mechanism. Here we see that we should not misuse the TP's control capability to control every aspect of stimulation. The TP should provide logical control over the TB and the TB should provide stimulation at the physical-level.

To thoroughly verify a DUT, the TB could inject errors into the physical-level stimulation. The purpose is to simulate a real-world environment that is not always ideal for the DUT. The DUT may respond the error with a rare sequence of hardware/software behaviours. Combined with the parallelism constructed by the TP, the deliberately injected errors could lead to un-conceivable (butterfly-effect) failure modes, greatly improving the test quality. The error-injection by the TB may or may not be modelled as transfer parameters, giving some flexibility in the TB development.

For the Nios SoC, a TP-configurable (up to 10% off) baud-rate is used to inject errors in the UART RX stimulation; and this configurable bad baud-rate is treated as a parameter of all UART RX transfer-types (RX-by-Interrupt, RX-by-DMA and RX-by-polling). This deliberately bad baud-rate intends to check (1) the robustness of the UART receiver in receiving an out-of-pace bit stream, and (2) the RX-error (`parity, frame, break` errors) detecting mechanism in the UART. Interestingly, this error-injection targeting at the UART eventually allows us to discover a strange behaviour in the DMA engine, which is detailed in Appendix A.3.

To summarize, in a TP-centric verification approach, since the more active *control* responsibilities, especially the parallelism management, are already taken by the TP, the TB is

not required to have too much intelligence in providing stimulations. Nevertheless, the TB should exert its powerful observation capabilities to observe the TP.

5.3 TB Observes TP

Software native to a DUT may still be interpreted as the behaviours *attached* to the hardware, namely, the processor. However, considering software's inherent flexibility, this interpretation does not help to increase the abstraction level. Some software components *cooperate* with hardware components to fulfil or even extend the latter's design intentions, while some other software components *manage* software- or hardware-natured behaviours. It is the "HW-SW partition", which has been decided in the earlier design stage, that determines which functionalities are implemented in hardware and which in software. Therefore, it is reasonable not to treat the software as the behaviour "attached" to the processor, but independent entities comparable to hardware components. It follows naturally that we need some mechanisms to observe the software's behaviours as well as the hardware's behaviours.

Test-bench is the right vehicle to observe the software's (i.e., the TP's) behaviours. To observe the TP's behaviours from the TB is both (a) *necessary* since the TP's behaviours could suggest the test efficiency and quality by showing how close the TP is working with hardware, and (b) *possible* since the "software's behaviour" is derived from the processor's behaviour, which is definitely under the test-bench's ubiquitous observation.

5.3.1 Software's Behaviours

What kind of phenomena should be counted as the "software's behaviours"? Since software is made of *instructions* and *data*, software's behaviour could be interpreted as its instruction-flow (i.e., the program-control) and its data-flow.

On a general-purpose computer, *profiling* is a powerful tool that gives software engineers visibility into software's instruction-flow, helping engineers to improve software quality. Profiling provides critical software performance information including the CPU-time spent in executing user-mode code and supervisor-mode code, the CPU-time on each function, and the calling relations among functions. This kind of software profiling needs support from the OS (e.g., time-stamping). Likewise, in TP-centric verification, if a certain profiling facility is provided, verification engineers can closely monitor software components' behaviours, improve software quality and find coverage holes and anomalies. However, to perform TP profiling

through OS support is infeasible. (Remember that the OS is unavailable and inappropriate at the integration stage.) Also, software profiling should not be confused with the hardware profiling facility [79] provided by the simulation environment. The hardware profiling gives information about the real-world time spent in simulating each hardware component. The software profiling information we expect is about the simulated world, including

- the calling relations among software functions,
- the function execution time, and
- interrupt-related information.

Another type of information about the instruction-flow refers to TP code coverage during simulation. Like TP profiling, TP code coverage has a counterpart on a general-purpose computer. Again, TP code coverage information can only be obtained by TB's observation capability. The coverage information will be especially useful when TP components (i.e., ISRs and soft-transfers, see Section 4.1) come from the real-world software.

For the data-flow, the software behaviour refers to the software's accesses to critical data structures. For easier implementation, these critical data should be declared as global or static variables in the TP, so that their addresses are determined before software is executed. For our transfer-oriented TP, critical data includes:

- transfer-related information, i.e., the running flags,
- scenario-related information,
- environment-related information, and
- the FIFO used by the ISRs and the `Scheduler()`.

A variety of coverage information could be extracted from these data. For instance, the running-flags' history contains the information regarding high-level concurrency. Section 6.4 in the next chapter gives the application of the concurrency information.

5.3.2 TB's Observation Facilities

Figure 5.4 shows the way we implement the software observation mechanism.

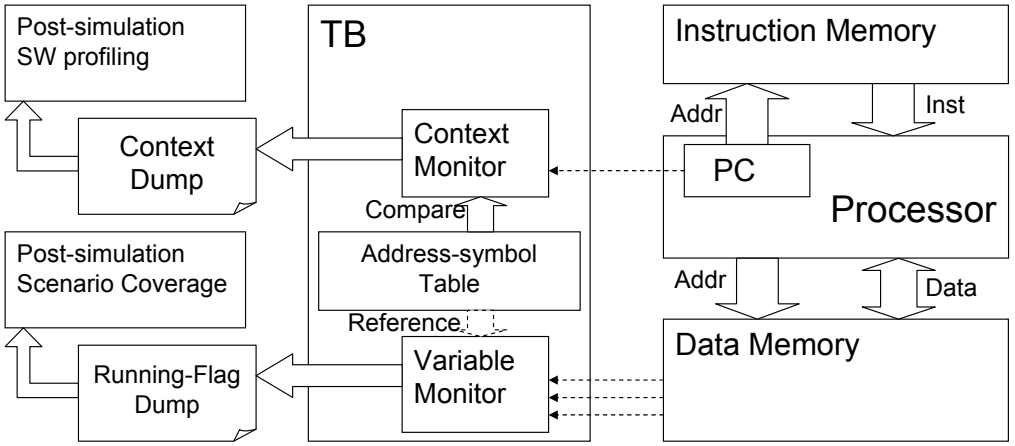


FIGURE 5.4: The monitors in the test-bench can observe the software's behaviours by monitoring memory locations and context registers in the CPU, including the program-counter (PC). The context monitor uses the addresses information from the address-symbol table to transform the instruction-level changes into the function-level changes, allowing program profiling and coverage. The actual analysis of data could be performed by post-simulation utilities.

The program-counter (PC) in the SoC processor holds most information about instruction-flow. Other registers in the processor could contain some auxiliary information. The context-monitor is sensitive to PC's changes, and transforms the instruction-level changes into function-level changes by comparing the PC content with the functions' entrance/exit addresses. The address information could be obtained from the address-symbol table, which is the output of the TP compilation/link tool set. The TP profiling data for the Nios SoC is discussed in Section 6.5.2.

Similarly, the TB could also implement variable-monitors to observe software's operations on critical data structures. Again, the address information comes from the address-symbol table. A variable-monitor is sensitive to the modifications to the variable and dumps these modifications. It uses the address information statically to *refer to* memory locations, whereas the context-monitor uses the addresses for dynamic comparison.

It is worth mentioning that both kinds of the monitors in Figure 5.4 could be constructed *automatically* from the information in the address-symbol table.

Equipped with the "ubiquitous and non-intrusive" observation capabilities, TB is competent for many other *observation* tasks. The typical "observation" responsibilities which the TB should provide include:

- test result checking,

- coverage data collection,
- hardware property assertion, and
- event dumping.

We will not elaborate on these observation responsibilities, which are well practiced in the mainstream TB-centric methodologies. Instead, we make two general points about using the TB in the TP-centric methodology.

Using software-based result-checking. While it makes sense to let the TB observe the DUT at physical level, we might wonder whether the software can act as a proper logical-level observer. For instance, one transfer "DMA-copy 1,000 bytes from address 0xa000 to address 0xf000" takes 2,000 DUT cycles to complete; in order to check the transfer result, is it appropriate to compare 1,000 destination and source locations, *byte-to-byte in software*, which may consume 20,000 cycles? Obviously, software is not in the position to perform such an observation task on a large amount of data. One typical memory-read operation native to an SoC, called *front-door* access, not only consumes 1 to 10 DUT cycles (the *simulated time*), but also wastes *simulation time* (i.e., the wall-clock time). In contrast, a behavioural checker implemented in the TB reads data two to three orders of magnitude faster than software in terms of *simulation time*, and consumes strictly zero *simulated time*. This kind of memory/register access is called *back-door* access. Result checking via back-door access is extensively used and supported by the TB-centric verification methodologies [34]. The conclusion is that the TB-based test checking should be extensively used. The SW-based checking may supplement the TB-based checking in some special cases where no massive data operations are needed; it may also find its applicability when the TB and the DUT are prototyped as true hardware.

Retaining test-benches' synthesisability. From the synthesis point of view, to let the TB focus on mechanical observation tasks is especially reasonable and practical since mechanical tasks can be easily synthesised into true hardware. For instance, El Shobaki *et al.* proposed a HW/SW event monitor in true hardware for the purpose of FPGA-based SoC verification [76]. In contrast, high-level control mechanism implemented in a TB is not synthesisable. To let the TP (in place of the TB) take the responsibility of high-level control avoids this problem and retains the TB's synthesisability. A synthesisable TB can accompany its DUT into *lower* abstraction levels. In this sense, we could argue that TB synthesisability is also one form of TB reusability. This form

of reusability is *opposite* to the reusability proposed by the mainstream verification methodologies, in which TBs are constructed at *increased* abstraction-levels and thus lose synthesisability. (See Section 2.1.2.2 and Section 2.1.2.4)

5.3.3 TB and Offline Support

The mainstream TB-centric approaches emphasise the online (i.e., simulation-time) stimuli-generation, online information-extraction and online feedback between observation and generation. This scheme unavoidably requires a complex TB structure.

In contrast, in the TP-centric verification approach, since the more active role of parallelism management is taken by the TP, which is automatically generated *before* simulation, we can be less dependent on the online stimuli-generation. The requirement of the online information-extraction of raw observation is also removed. The feedback loop can be achieved *offline*.

The intelligence needed for test-generation and information-extraction does not need to be expressed in HDLs/HVLs in the form of test-benches. Instead, the intelligence could be expressed in any general-purpose languages in the form of offline tools, including the pre-simulation test-generator and post-simulation data-analysers. This arrangement not only matches the overall profile of current HDLs/HVLs, but also implements the concept of language-independent verification [27].

The software observation mechanisms described in Figure 5.4 is a good example of where the intelligence is required. The monitors in the Verilog test-bench simply collect and dump data. The intelligence really goes to (a) the post-simulation data analysers, and (b) the scripts to generate the Verilog monitors from the address-symbol table . For the Nios SoC, both of the above facilities, together with the TP generator, are written in Python language – a high-level script language not related to VLSI verification.

Powerful post-simulation analysers can extract rich information about simulation thoroughness buried in dump files. We implement the analysers for the Nios SoC as follows.

- The context analyser extracts the program-control from the dump file and reports the actual profiling information (see Section 6.5);
- The running-flag analyser extracts the concurrency information from the dump file and reports concurrency coverage based on Petri-net (Section 3.5); and further generates a

parameterisation file to bias the next round of TP generation, completing the offline feedback loop (see Section 6.4).

These post-simulation analysers written in languages other than HDL/HVL greatly relieve the observation responsibilities required in a test-bench. The test-bench now simply performs the raw data-collection.

5.4 Summary

This chapter has addressed the conceptual confusions about the relationship between the test-program and the test-bench. The test-program provides high-level control over the test-bench, and the test-bench returns the statistics about the test-program's behaviours. Test-benches may still require considerable programming efforts; however, the level of intelligence required in TB programming is significantly reduced; meanwhile, the software's behaviours are vigorously taken into consideration. This arrangement solves the problem of TB-TP relations naturally and harmoniously. Also we could treat the test-bench, the test-program and the SoC as one continuum rather than disparate entities.

Here we see the defining difference between our methodology and the mainstream methodologies. The latter focus on test-bench construction and thus require enormous investment on the *real world* software phenomena, i.e., building sophisticated test-benches under the object-oriented-programming paradigm (and even beyond). In contrast, our methodology allows the verification efforts to be concentrated on the *software phenomena in the simulated world*. Checking the behaviours in the *simulated world* is the ultimate reason for simulation; in this sense, our TP-centric approach to SoC verification captures the nature of *simulation* more accurately.

Chapter 6

Experiments

6.1 Overview: The Verification Environment

This chapter demonstrates some quantitative aspects of the test-program centric verification methodology. While the experiments are directly applied to the Nios SoC, our goal is not about verifying this specific SoC, but about the the validity and feasibility of the test-program (TP) centric and interaction-oriented methodology. In this sense, the construction of the whole verification environment should also be regarded as an experiment to prove the validity and feasibility. Figure 6.1 shows the main components and processes of the verification environment.

One full cycle of the work flow includes several stages, most time being spent on simulation.

(1) Test Generation (approx. 1 second)

- Input: a list of transfer-types (Python); a parameterisation-bias file (Python);
- Output: a test-program (C);
- Tool: the TRG-based test-generator (Python).

(2) Software Compilation (approx. 1 minute)

- Input: the test-program (C); interrupt-service-routines (C); soft-transfers (C);
- Output: the executable TP (ROM.dat); the address-symbol table (ROM.nm);
- Tool: the Nios software development tool kit, including the C compiler and linker.

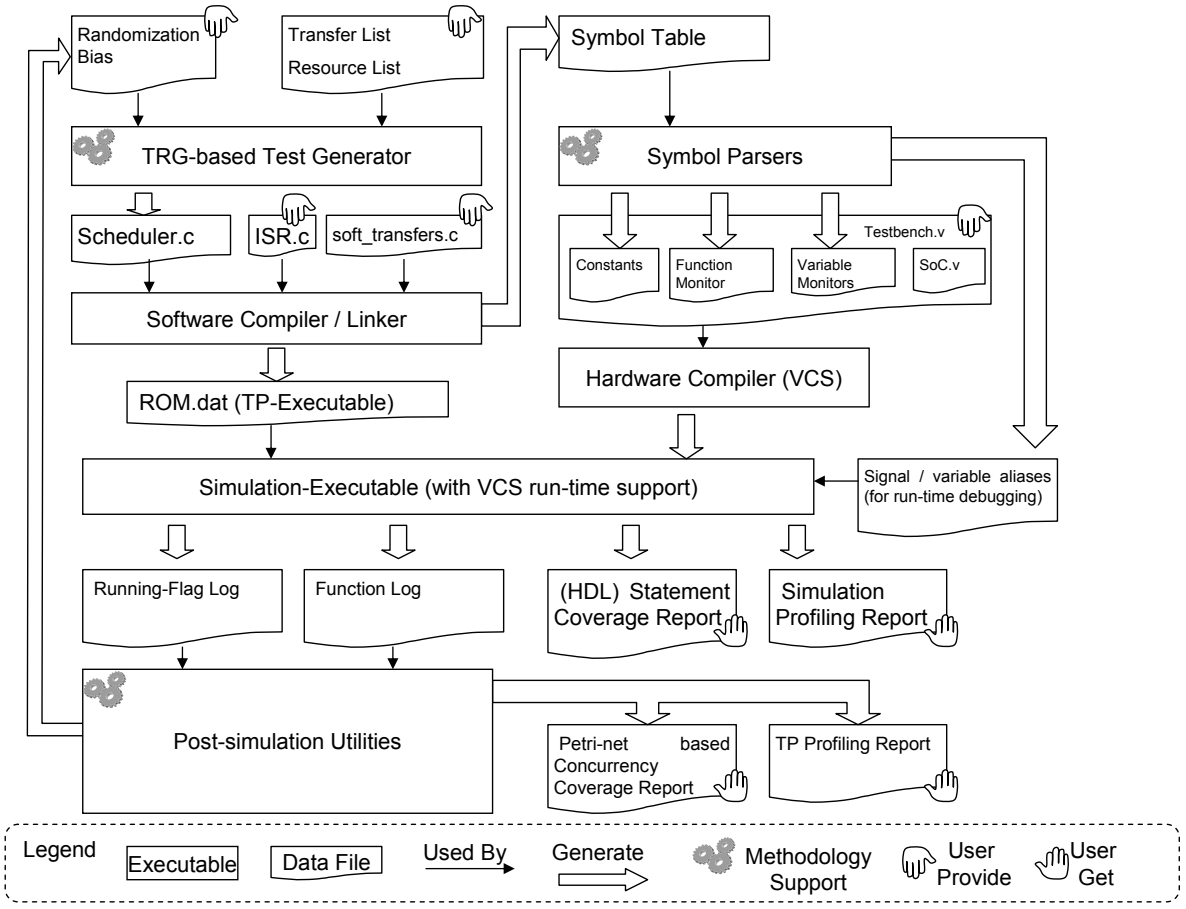


FIGURE 6.1: The components and processes in the verification environment. The entire process can be divided into these stages: test-generation, compilation (including software compilation, hardware compilation and symbol parsing), simulation and post-simulation. The feedback happens when the post-simulation utility updates the randomisation strategy file. Except for the inputs provided by the user, the entire process can be automated.

(3) Software Symbol Parsing (approx. 2 seconds)

- Input: the address-symbol table (ROM.nm);
- Output: a list of constants (Verilog constants); the program-counter monitor (Verilog); the running-flag monitor (Verilog); an alias table for simulation debugging (text);
- Tool: the address-symbol-table parsers (Python).

(4) Hardware Compilation (approx. 2 minutes)

- Input: the system-on-chip design (Verilog); all test-bench components (Verilog);
- Output: a simulation executable program (simv);

- Tool: Synopsys VCS.

(5) Simulation (minutes to days, but typically controlled between 30 minutes to two hours)

- Input: the simulation executable (simv); the TP executable (ROM.dat);
- Output: a statement coverage report; a hardware profiling report; log files (text);
- Tool: Synopsys VCS.

(6) Post-Simulation Analysis (approx. 1 minute)

- Input: the log files (text);
- Output: the TP profiling report; the concurrency coverage report; the updated parameterisation bias file (Python);
- Tool: the post-simulation analysers (Python).

The main commercial tool being used is the Synopsys's VCS for Verilog simulation (Stages 4 and 5). The software development tool kit (used in Stage 2) comes together with the Nios SoC package. Our methodological contribution (in Stages 1, 3, 6) includes the TRG-based test-generator, custom makefiles, symbol parsers and post-simulation utilities; they are mostly written in Python language. The feedback loop is completed when the post-simulation utility updates the randomisation-bias file. The time consumed by our methodology is insignificant to that consumed by the simulation.

The experiment data in the remainder of this chapter come from the outputs of the above stages.

6.2 Statement-Based Coverages

Statement-based coverages (or code coverages) are substantially used as indications of the verification completeness in the realm of software verification. The term *statement* refers to the program statement written in the programming language. Coverages of this type are borrowed from the area of software verification to hardware verification, since hardware design can also be described in programming languages, namely, HDLs. The coverage facilities should be provided by the simulator. In our cases, the simulation tool VCS provides this facility. To use the facility, the proper coverage options should be specified when compiling the DUT and the TB using VCS. The outcome of compilation is a simulation executable

(program `simv`) that will dump coverage information into a working directory. After simulation, a coverage viewer reads out the coverage information. Three types of coverages are supported by VCS version 7.1.1:

- Line Coverage – the percentage of statements that have been activated in the simulation.
- Toggle Coverage – the percentage of registers and wires that have toggled from 0 to 1 and 1 to 0;
- Conditional Coverage – the percentage of branching statements (e.g., `if/else` and `case`) that have been taken.

The statement-based coverages are independent of the nature of the stimulation, providing a general way to determine how a DUT has been exercised by a stimulation. It serves as one indicator of test completeness as well as a platform to compare different test generation methods.

We have observed that reasonable coverages can be achieved by the test-programs generated by the TRG method. Since another software-centric test generation methodology called SALVEM [25] is demonstrated on the same Nios SoC, we compare the results of SALVEM tests with the test results of TRG-based tests. For each set of tests, 12 hours of simulation (with all coverages enabled) was performed. Figure 6.2 shows the comparisons of the statement coverages. (The line coverage was not compared because both methods achieve full coverage).

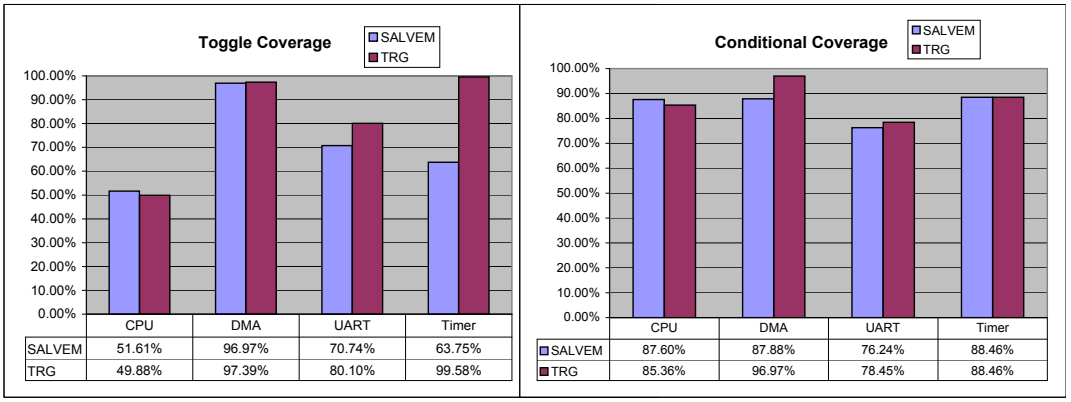


FIGURE 6.2: Toggle and conditional coverage comparison. TRG-based tests have higher coverages on peripherals but lower on the CPU. This may be due to the fact that we have not spent too much effort in stressing the CPU itself. Stressing the CPU itself can be achieved by a different level of automation.

For the toggle and conditional coverages, the TRG method has higher coverages on some components but is marginally lower on the Nios CPU. The CPU has 11,000 lines of code and is the most complex component in the system. The lower coverages on CPU using TRG method may be attributed to the fact that we have not put too much effort in *manually* creating subroutines stressing the processor itself, which could be addressed by another level of automation. We believe that the TRG method imposes no restrictions on achieving reasonably high statement-based coverages.

The statement-based coverage serves as the minimum requirement of *component-level* verification. However, they should not be the driving metrics for the *system-level* verification. This is because there is no obvious correlation between the statement-based coverages and the system-level concurrency. (For instance, we could achieve very high statement-based metrics by stressing each component *one after another*.) Therefore we need some metrics that can reveal information regarding the system-level concurrency.

6.3 State Space Traversing

The statement-based coverage measures give little information regarding the system-level concurrency and the resource-contention [18]. Therefore, we attempt to indicate this information using the "system state space". We define the system state space as the space made by the *concatenation* of the major control and status registers in the SoC components (CPU, DMA, UART and Timer). We implement a simple module with a 64-bit input port tapped to those registers. This module does nothing but log the state-changes and the corresponding time stamps. The state traversing information is extracted by a post-simulation utility.

Since the theoretical size of the state-space is 2^{64}, it is impractical for any tests to traverse it exhaustively. However, we can statistically measure how fast the states can *change* and how fast the *new* (i.e., unprecedented) states will emerge. These values are useful since the system states can give information regarding the concurrency. For example, from the traversed states, we can tell if *all* peripherals have requested interrupts *simultaneously*.

We compare the capabilities to traverse the state-space between two sets of TPs. The TPs in one set contain the scenarios made of one or two transfers, and the TPs in the other set contain the scenarios made of maximum number of (i.e., three, four or five) transfers. Figure 6.3 shows the rate of the *state-change*. The high-concurrency TPs have a state-change rate roughly two times the rate of the low-concurrency TPs. The faster state-change rate indicates that more events are happening simultaneously in the system.

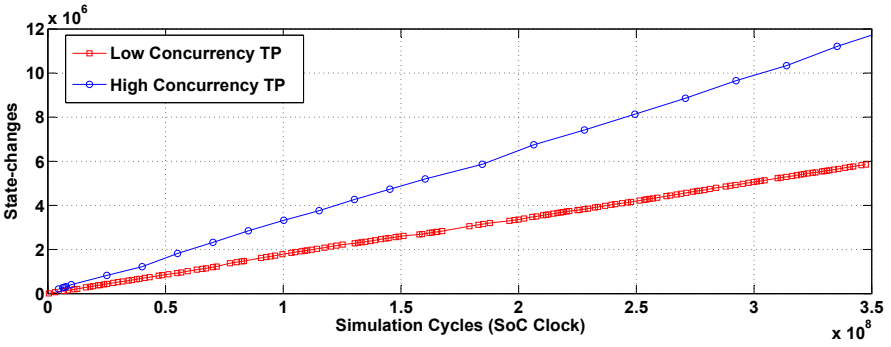


FIGURE 6.3: State-changes against simulation cycles. TRG-based tests has an almost double rate of state-change, implying that the design is behaving more actively during simulation.

However, the faster state-change rate does not necessarily mean the efficient state-space traversing. This is because any state may recur many times. We further compare how fast the *unprecedented* states emerge in simulation. Our experiments show that the low-concurrency TPs have traversed about 10^5 distinct states in 420 million SoC cycles (12 computing hours on a 3.2 GHz workstation); in comparison, the high-concurrency TPs can traverse 10^6 distinct states in the same simulation duration. In Figure 6.4, each data-point represents one simulation of a TP. We observe that the high-concurrency TPs produce new states at a much faster speed, and the speed is insensitive to the number of the known-states. This encouraging comparison shows that concurrency is the key to efficient exploration of the state-space.

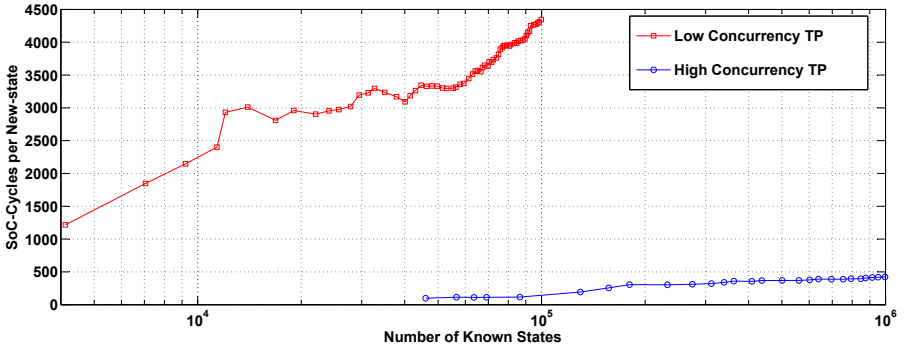


FIGURE 6.4: New-state emergence rate against the number of known-states. An overall lower position of the curve of high-concurrency test-programs indicates that new states are more easily to be exposed in high level of concurrency. This proves that maintaining parallelism is important to verification efficiency.

Like the statement-based coverages, system state-space traversing is a metric independent from the test-generation method. Therefore, it can serve as a fair comparison platform for

stimuli from different sources. However, it is difficult to accurately separate a "feasible" state-space from the simple concatenation of registers. There is no *coverage* to report because we do not known the denominator of the coverage. Another limitation is that the system state changes at a very fine temporal granularity. In the high-concurrency experiment (Figure 6.3), the state changes every 30 SoC-Cycles, consuming both simulation performance and the dump size. For these reasons, state-space traversing is only suitable for methodology characterisation, but not for verification completeness indication.

6.4 Petri-Net Based Coverages

In order to give the coverage information about the concurrency completeness of the simulation, a Petri-net model is used. The Petri-net is derived from the TRG used for test-generation based on transfer model (see Section 3.5). The coverage space is defined based on the reachability graph associated with the Petri-net; the reachability graph can be obtained by a Petri-net tool [73]. The TRG-to-Petri-net conversion could be done automatically using the algorithm introduced in Section 3.5.2. However, for our simple Nios SoC, we chose to do it manually.

We model a TRG with 12 major transfer-types for the Nios SoC. Our generator can exhaustively (but randomly) produce 139 unparameterised scenarios. This is well predicted by the reachability graph, which has 140 states, with the additional state representing the *empty* scenario. The reachability graph also contains 772 transitions.

We have achieved the test-generation with feedback at state-level and transition-level. We insert a running-flag monitor in the test-bench as described in Figure 5.4 in Section 5.3.2, which monitors the accesses to the software variables called running-flags and logs their changes in a dump file. Also, a post-simulation running-flag analyser is developed. The analyser is responsible for the following tasks:

(1) detecting states and transitions from the trace of the running-flags and counting them,

(2) comparing the counts with the total states and transitions in the reachability graph, and then reporting the coverage,

(3) identifying the target (i.e. uncovered or less frequent) states and transitions, and

(4) in a bias file (see Section 3.4.2), adjusting the randomisation biases about transfer selection and parameterisation.

The state-level feedback is straightforward because a state in the reachability graph simply represents a scenario in the TRG. Once a target scenario is identified, in the bias file, we simply increase the selection weights of the transfer-types which make up the target scenario. Thus the test-generator will be more likely to generate the target scenario.

The transition-level feedback requires additional consideration. A transition in the reachability graph is a transfer-start event T_s or a transfer-end event T_e. A transition separates two scenarios S_1 and S_2, i.e., $S_1 \xrightarrow{T_s} S_2$ or $S_1 \xrightarrow{T_e} S_2$. Therefore, the analyser needs to manage both target scenario (S_1 or S_2) and target event (T_s or T_e).

First, we identify the target scenario:

- In case of $S_1 \xrightarrow{T_s} S_2$, the target scenario is S_2;
- In case of $S_1 \xrightarrow{T_e} S_2$, the target scenario is S_1;

Once the target scenario is identified, we can apply the same mechanism as that used in state-level feedback in order to make the target scenario more likely to happen.

Second, we need to make the target event happen earlier in the current scenario in order to enter or leave the target scenario; otherwise the current scenario changes. For each transfer-type, we define one of its parameters as its *life-expectancy*, which controls how long that transfer will be running. For example, for a transfer-type "MEM-to-MEM DMA", the parameter "DMA length" is the life-expectancy parameter. The analyser then adjusts the randomisation ranges of the life-expectancy parameters in the bias file: it *reduces* the life-expectancy of the target transfer and/or *extends* the life-expectancy of the rest transfers in the target scenario. Therefore, in simulation, the target event has more chance to fire earlier to enter or leave the target scenario.

Figure 6.5 includes the accumulative state-coverage and transition-coverage comparisons between two sets of 20 simulation-runs, one with feedback and the other without feedback but with randomly generated scenarios. Each set needs approximately 15 computing hours on a 3.2 GHz 1G RAM workstation. The figure shows that, with feedback, all states and transitions are covered in the first several runs. For the 20 runs without feedback, the state-coverage space is traversed 5 times slower; and the transition coverage space cannot be traversed in 20 runs.

Like all feedback techniques, our feedback scheme only targets at one type of coverage. It should also be noted that the fast traversing on states and transitions does not mean that the whole verification process is complete. If more detailed temporal relations (e.g.,

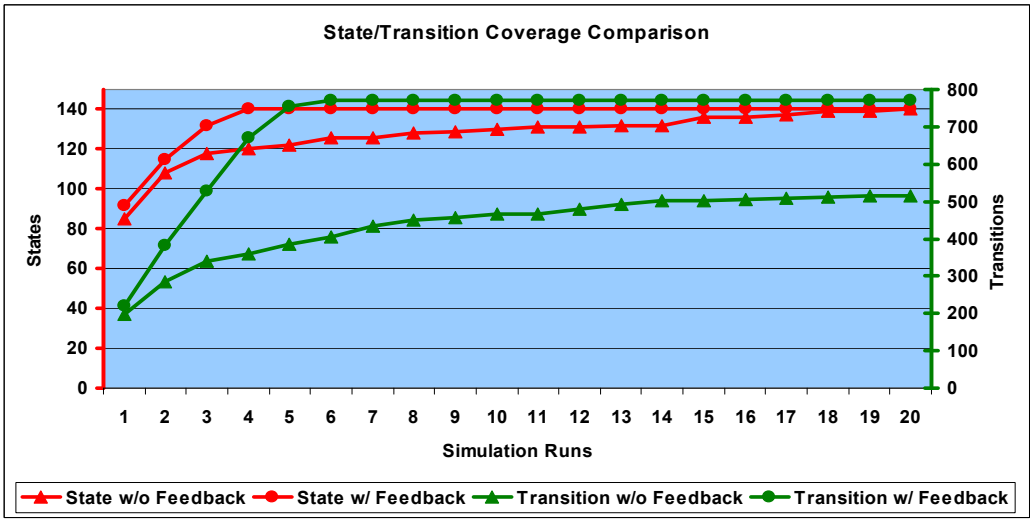


FIGURE 6.5: Petri-net based state-coverage and transition-coverage with and without feedback. With feedback information, both state and transition coverages converge rapidly.

paths and cycles) are taken into account, more scenarios are needed. Nevertheless, the fast traversing does give us a chance to focus on other coverage areas. We could cross this scenario-level feedback mechanism with the management on other test variations, including transfer parameterisation and environment parameterisation.

Although the Petri-net based coverages can indicate the test completeness from a specific angle, these metrics are strongly coupled with the TRG-based test-generation method (thus cannot be used to compare different methodologies). In fact, the "verification completeness" is a common problem faced by all simulation-based verification. No single coverage metric can serve as the absolute criterion for simulation completeness.

6.5 Profiling: Simulation Efficiency

6.5.1 Overview: Profiling in Two Worlds

Accompanying the coverage issue is the efficiency issue since simulation is a time-consuming process. We want simulation to be efficient – behaviours being simulated should mostly contribute to the meaningful verification job. The efficiency can be evaluated by appropriate *profiling mechanisms*, which give objective statistics on *software components*' performances.

There are two categories of software components in a simulation, i.e., the components written in HDLs/HVLs and the components written in languages native to the DUT. The former is the software in the real-world (but the hardware in the simulated-world; notice that the true hardware components do not have profiling issues since they are parallel); while the latter is the software in the simulated-world. Therefore, there are two categories of profiling mechanisms:

- the real-world profiling, which reports the real-world CPU-time (in seconds) spent in computing the behaviours of the simulated-world hardware components, and
- the simulated-world profiling, which reports the simulated-world CPU-time (in DUT cycles) spent in executing the simulated-world software components.

The former is already provided by the simulation tool (VCS in our case). Figure 6.6 is a top-level profiling of one simulation run of duration 2735.04 seconds (wall-clock time), which simulated an event-driven TP execution of 42 million SoC Cycles (simulated time). We see that the TB components `test_bench` and `function_logger` sum up to 1.87% of the simulation time. Although we did not specifically develop a TB-centric approach to compare with the TP-centric approach, this low enough profiling matches our expectation that the TP-centric approach does not introduce large overhead for computing TB's behaviour.

The rest of this section focuses on the second type of profiling, which requires (a) a context monitor inserted in the TB to collect raw data during simulation, and (b) a post-simulation utility to extract various profiling information from the raw data. This kind of profiling gives users the insight into how close the SoC software is interacting with the SoC hardware.

6.5.2 TP Profiling: Insight into the System Behaviour

The TP profiling experiments we show in this section serve two purposes:

- to validate the idea to use the TB and post-simulation utilities as a powerful TP observers, by showing *what kind of data* can be profiled and how to interpret and use these data;
- to validate the idea to use the TP as the efficient controller of parallelism, by showing the *profiling data themselves*.

```
//      Synopsys VCS 7.1.1 Simulation profile: vcs.prof Simulation Time:    2735.040 seconds
                                        TOP LEVEL VIEW
============================================================================
                           TYPE       %Totaltime
----------------------------------------------------------------------------
                            PLI          0.00
                            VCD          0.00
                         KERNEL         11.25
                         DESIGN         88.75
                                        MODULE VIEW
============================================================================
Module(index)                %Totaltime    No of Instances   Definition
----------------------------------------------------------------------------
altsyncram            (1)      31.46          6             altera_mf.v:238.
CPU_32bit_instructio
n_decoder             (2)       8.51          1             CPU_32bit.v:5826.
Ext_Shared_Bus_avalo
n_slave_arbitrator    (3)       3.26          1             soc.v:2486.
CPU_32bit_alulogic    (4)       3.19          1             CPU_32bit.v:3495.
CPU_32bit_control_re
gister_unit           (5)       3.18          1             CPU_32bit.v:4200.
CPU_32bit_pipeline    (6)       2.78          1             CPU_32bit.v:8688.
CPU_32bit_instructio
n_master_arbitrator   (7)       1.62          1             soc.v:682.
CPU_32bit_cpu_instru
ction_fifo_fifo_modu
le                    (8)       1.61          1             CPU_32bit.v:537.
CPU_32bit_skip_unit   (9)       1.35          1             CPU_32bit.v:4074.
DMA_1                (10)       1.32          1             DMA_1.v:480.
test_bench           (11)       1.31          1             TESTBENCH.v:51.
CPU_32bit_constant_g
enerator             (12)       1.30          1             CPU_32bit.v:1516.
CPU_32bit_data_maste
r_arbitrator         (13)       1.30          1             soc.v:31.
CPU_32bit_op_fetch   (14)       1.23          1             CPU_32bit.v:2751.
CPU_32bit_op_b_mux   (15)       1.17          1             CPU_32bit.v:2398.
CPU_32bit_index_matc
h_unit               (16)       1.05          1             CPU_32bit.v:1923.
CPU_32bit_dcache     (17)       0.96          1             CPU_32bit.v:10525.
DMA_1_read_master_ar
bitrator             (18)       0.89          1             soc.v:1362.
CPU_32bit_aluadder   (19)       0.86          1             CPU_32bit.v:3085.
CPU_32bit_reg_index_
calculator           (20)       0.84          1             CPU_32bit.v:1749.
CPU_32bit_byteshift_
control_unit         (21)       0.82          1             CPU_32bit.v:3939.
CPU_32bit_instructio
n_receive            (22)       0.70          1             CPU_32bit.v:734.
Uart_1_tx            (23)       0.70          2             Uart_1_without_stimulation.v:66.
on_chip_ROM_s1_arbit
rator                (24)       0.67          1             soc.v:5265.
CPU_32bit_register_b
ank_a_module         (25)       0.65          1             CPU_32bit.v:5421.
CPU_32bit_icache_vfi
fo_module            (26)       0.65          1             CPU_32bit.v:9801.
CPU_32bit_alushifter (27)       0.63          1             CPU_32bit.v:3738.
Timer                (28)       0.59          1             Timer.v:26.
CPU_32bit_data_maste
r                    (29)       0.57          1             CPU_32bit.v:2225.
function_logger      (30)       0.56          1             functionlogger.v:92.
CPU_32bit_subinstruc
tion_unit            (31)       0.56          1             CPU_32bit.v:978.
CPU_32bit_icache     (32)       0.52          1             CPU_32bit.v:10138.
CPU_32bit_compact_al
u                    (33)       0.51          1             CPU_32bit.v:4929.
----------------------------------------------------------------------------
```

FIGURE 6.6: Simulation profiling data generated by Synopsys VCS. This report gives wall-clock time consumed in simulating each hardware components. The report can tell how much computation capability is spent on the hardware components in a DUT and its test-bench.

Figures 6.7 to 6.10 demonstrate a variety of software performances extracted by profiling. The DUT in the experiment is the Nios SoC (see Figure 2.8) in register-transfer-level Verilog. The TP is made of a set of functions, total size 49 KB-executable, including a 37 KB, event-driven function `scheduler` (automatically generated using the TRG method). The TP took 42 million DUT-cycles to run, simulated in 2735.04 sec on a 3.2 GHz workstation (the *same* simulation run as in Figure 6.6). It produced a reasonably sized (12 MB) dump of text format, containing 355k context-switchings, analysed in 25 seconds by our profiling utility written in Python language. Note the time-unit in these figures is "simulation-cycle"; one SoC-Cycle equals 20 simulation-cycles.

Figure 6.7 provides basic information associated with each function.

- Column 1 lists the names of functions to be profiled.
- Columns 2-5 count the program-control's switchings: entrances, interrupts, callings and resumes.
- Columns 6 reports each function's CPU time, *including* the time spent on callings and interrupts.
- Columns 7 reports each function's CPU time, *excluding* the time spent on callings and interrupts. It is this column that *partitions* the CPU-time.
- Columns 8 (Prof(%)) is the CPU-time share for this function, based on Columen 7. The entire table is sorted according to column 8. This report can tell how the profiling of the payload and overhead (Role 1–3 functions, see 4.1) matches our expectation. This particular report says that
  - the Role 3 function (`scheduler` only) takes 4.44%;
  - the Role 1 functions (all trap handlers, ISRs and ISR-wrappers) amounts to 28.75%; and
  - the Role 2 functions (all remaining functions except `main`) accounts for 47.01%.

  We thus conclude that the payload/overhead profiling basically matches our expectation.
- Columns 9-12 are information derived from the previous columns. From different angles, they report the average duration the program-control stays in each function.
  - Column 9 (Run_Avg), which equals Column 6 ÷ Column 2, tells how long each function has the program-control for each entry, callings and interrupts are included.

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8
Function	**Entry**	**Interrupted**	**Callings**	**Resumed**	**Exec w/ calls**	**Exec w/o calls**	**Prof (%)**
main	1	2654	141	2794	826947860	163721980	19.80%
memoryblkrevbyCPU	1098	5714	235	5949	174466360	130329820	15.76%
do_logic	233	5632	0	5632	160023220	111371640	13.47%
uartISR	17489	894	1106	1283	168592040	108945940	13.17%
recursive_fibo	129356	4501	128678	133134	17610769620	102495880	12.39%
CWPUnderflowTrapHandler	1194	0	0	0	60685220	60685220	7.34%
scheduler	2169	629	2051	2680	662509920	36686160	4.44%
CWPOverflowTrapHandler	1194	0	0	0	26475040	26475040	3.20%
uartISR_wrapper	17489	894	17489	18383	213993300	25007320	3.02%
txdata1pollingbyCPU	32	618	0	618	27560660	22703540	2.75%
rxdata1pollingbyCPU	10	318	0	318	19655500	15876180	1.92%
TimerISR	3202	730	1123	730	37719960	10296280	1.25%
TimerISR_wrapper	3202	564	3202	3766	69227060	4597580	0.56%
Unknown	75	0	0	0	2419900	2419900	0.29%
dmaISR	269	39	259	39	4640460	1356700	0.16%
fibonacci	678	26	678	704	238660720	1350020	0.16%
TerminateRunningRX	389	0	0	0	856200	856200	0.10%
PrepareNextScenario	70	0	84	84	3206680	740260	0.09%
__mulsi3	235	21	0	21	719140	576720	0.07%
dmaISR_wrapper	269	114	269	383	6828020	408960	0.05%
UpdateEnvironment	14	0	14	14	2159560	26420	0.00%
caching	14	0	5	5	2133140	20100	0.00%
Totals	**178682**	**23348**	**155334**	**176537**	**Meaningless**	**826947860**	**100.0%**
Totals Excl. Recursion	**49326**	**18847**	**26656**	**43403**	**Meaningless**	**724451980**	**87.60%**

(a) Function profiling: raw data and profiling. Program-control movement (Col 2 to 5), program-control time (Col 6, 7) and profiling (Col 8)

Col 1	Col 9	Col 10	Col 11	Col 12
Function	**Run_Avg**	**Ent_Avg**	**Cnt_Avg**	**Int_Avg**
main	826947860	163721980	58576	61666
memoryblkrevbyCPU	158894	118697	18494	19132
do_logic	686794	477989	18989	18989
uartISR	9639	6229	5803	5926
recursive_fibo	136141	792	390	766
CWPUnderflowTrapHandler	50825	50825	50825	50825
scheduler	305444	16913	7565	13112
CWPOverflowTrapHandler	22173	22173	22173	22173
uartISR_wrapper	12235	1429	697	1360
txdata1pollingbyCPU	861270	709485	34928	34929
rxdata1pollingbyCPU	1965550	1587618	48402	48403
TimerISR	11780	3215	2618	2619
TimerISR_wrapper	21619	1435	659	1221
Unknown	32265	32265	32265	32265
dmaISR	17250	5043	4404	4405
fibonacci	352006	1991	976	1918
TerminateRunningRX	2201	2201	2201	2201
PrepareNextScenario	45809	10575	4806	10575
__mulsi3	3060	2454	2252	2253
dmaISR_wrapper	25382	1520	627	1068
UpdateEnvironment	154254	1887	943	1887
caching	152367	1435	1057	1436
Totals	**113723**	**4628**	**2328**	**4093**
Totals Excl. Recursion	**54930**	**14687**	**7813**	**10627**

(b) Function profiling: derived data.

FIGURE 6.7: Basic test-program profiling. This report gives statistics associated with each software function. These statistics could be used to characterise the overload/payload ratio of the test-program and direct further software improvement.

- Column 10 (Ent_Avg), which equals Column 7 ÷ Column 2, tells how long each function has the program-control for each entry, callings and interrupts are excluded.
- Column 11 (Cnt_Avg), which equals Column 7 ÷ (Column 2 + Column 3 + Column 4), tells how long each function retains the program-control until interrupted or calling other functions.
- Column 12 (Int_Avg), which equals Column 7 ÷ (Column 2 + Column 3), tells how long each function retains the program-control until interrupted.

The difference between Column 10 and 12 reveals how each function's execution is *fragmented* by interrupts. This information implies the degree of parallelism in the simulation. This particular report confirms that most Role 2 functions are strongly fragmented by interrupts (e.g., `memblkrevbyCPU` is fragmented to 16%), due to the parallelism.

Two total lines give gross total/average data for each column. (Since the recursion-intensive function `recursive_fibo` is special, the second total line excludes its impact.) The information can be used to characterise our TP. For instance,

- The gross average of 14678 (simulation-cycles) on Column 10 tells that an average function requires program-control for 14678/20 = 733 SoC-Cycles, which is already fine-grained software behaviour, but this required length of program-control is still fragmented by interrupts: the average of 10627 Column 12 tells that the actual program-control reduces to 10627/20 = 531 SoC-Cycles due to interrupt, an impact of $(531 - 733) \div 733 = -27.6\%$.
- This report confirms that the granularity of *continuous* program-control is in the order of 10^2 to 10^3 SoC-Cycles. It is a reasonably fine granularity when Role 1, 2 and 3 software components are considered altogether, considering that (a) a typical UART ISR turn-around time is about 350 SoC-Cycles [3], and (b) the life-expectancy of transfers, including soft-transfers, is in the range of 10^4 to10^5 SoC-Cycles (see Section 3.2.4). Therefore, a soft-transfer will be interrupted at least dozens of times when it is active – implying that the SoC is constantly busy during simulation.

This profiling could guide us to further improve the software structure. For instance, the `main` function accounts for 19.8%, in which the CPU is basically locked (until interrupted) in a strictly infinite loop of doing nothing (C code: "`while 1;`") to emulate CPU *idling*.

The CPU is idle because the rest of the system is so busy that the next soft-transfer has to wait for the release of resources. The idle time can be exploited to contribute to hardware verification: we could replace "`while 1;`" with "`while 1 CPU_self_test();`". (Looping the self-testing is meaningful, since each loop will experience different interrupt situations.) This profiling tells us, if this scheme would be carried out, the meaningful verification job on the CPU would increase by 19.8% .

Another instance about improving ISR-wrappers' performance using the profiling information is discussed in Appendix B.

Although the CPU is essentially a sequential device, due to interrupts, we still can obtain information regarding parallelism from SW profiling. Figure 6.8 indicates how many times each function is preempted by ISRs. From this report we can identify a coverage hole in the parallelism: the DMA ISR had never preempted function `__mulsi3`. (Many other zeros are reasonable and don't represent coverage holes.) The report may also be useful to identify abnormal software behaviours. For instance, the `CWPUnderflowTrapHandler` and `CWPOverflowTrapHandler` (see details in Appendix B) should be happening in pairs, one in the calling function, and one in the called function. This report confirms this property.

If the dump includes a "program-counter (PC) when interrupted" field, we could obtain a similar report but with more fine-grained "PC-interrupt cross" data.

The function-interrupt cross report does not give information regarding the *nesting* behaviours of interrupts, i.e., an ISR preempting another ISR and so on. Figure 6.9 shows such information by summarising the CPU time spent on each nesting depth. The Column "INT-DEP" is the depth of interrupt nesting; Column "T-Prof(%)" reports the CPU-time spent on each depth. This report helps us to determine the impact of interrupt to the system behaviour from another angle.

Figure 6.10 reports more detailed information regarding ISR nesting: all exact interrupt nesting sequences that have occurred in the simulation. The report says that the deepest interrupt nesting sequence first occurred at simulation cycle 640220650, when a recursive function was preempted by the DMA ISR, which was in turn preempted by the Timer ISR, which was in turn preempted by the UART ISR, which was finally preempted by a register-window underflow handler. That this extremely rare corner-case has happened three times in this simulation-run (and that it is allowed to happen) without software-induced error is eventually the result of our TP-centric and interaction-oriented TRG method. Compared with all possible nesting sequences, this report could also serve as a coverage measurement, telling how much the parallelism has traversed the space of nesting relations.

Report 2: Function-Interrupt Cross
(Warning: Function names may be truncated)

Func/INT	dmaISR	TimerI	uartIS	CWPUnd	CWPOve
main	34	111	2509	0	0
memoryblkrevbyCPU	77	1205	4432	0	0
do_logic	62	866	4704	0	0
uartISR	0	0	0	894	0
recursive_fibo	69	694	3398	170	170
CWPUnderflowTrapHandler	0	0	0	0	0
scheduler	11	84	534	0	0
CWPOverflowTrapHandler	0	0	0	0	0
uartISR_wrapper	0	0	0	0	894
txdata1pollingbyCPU	9	188	421	0	0
rxdata1pollingbyCPU	7	30	281	0	0
TimerISR	0	0	608	122	0
TimerISR_wrapper	0	0	442	0	122
dmaISR	0	7	24	8	0
fibonacci	0	2	24	0	0
TerminateRunningRX	0	0	0	0	0
PrepareNextScenario	0	0	0	0	0
__mulsi3	***0***	6	15	0	0
dmaISR_wrapper	0	9	97	0	8
UpdateEnvironment	0	0	0	0	0
caching	0	0	0	0	0

FIGURE 6.8: Function-interrupt cross. The column headings are interrupt-service-routine (ISR) names; the row headings are function names. The cross data represent how many times a function has been preempted by an ISR.

Report 3: Interrupt Depth Profiling

5 interrupt-Depths switched 46697 times during 826947860 time units

INT-DEP	Occ'd	Exist'd	S-Prof(%)	T-Prof(%)	Avg-Dur
0	20100	570531040	43.04%	68.99%	28384.63
1	22762	162729320	48.74%	19.68%	7149.17
2	3243	67225520	6.94%	8.13%	20729.42
3	586	26329020	1.25%	3.18%	44930.07
4	6	132960	0.01%	0.02%	22160.00

FIGURE 6.9: Interrupt nesting depth profiling. This report gives statistics associated with each "nesting depth".

```
Report 5: Interrupt Nesting Sequences

94 Interrupt Sequences Encountered.
Warning: Function names in the history may be truncated.

Occur LEV      Debut@ Stack_History(INT'EE>INT'ER)..
 2509   1     6158890 main>uartISR_wr
   34   1    27536290 main>dmaISR_wra
    2   2   267711570 main>dmaISR_wra..scheduler>TimerISR_w
  111   1     6727650 main>TimerISR_w
  170   1    43234970 recursive_>CWPOverflo
  170   1    43207290 recursive_>CWPUnderfl
 3398   1    43254430 recursive_>uartISR_wr
  609   2    43256390 recursive_>uartISR_wr..uartISR>CWPUnderfl
  609   2    43288350 recursive_>uartISR_wr..uartISR_wr>CWPOverflo
   69   1    63326090 recursive_>dmaISR_wra
    2   2   640600130 recursive_>dmaISR_wra..uartISR_wr>CWPOverflo
. . .
    3   3   640195150 recursive_>dmaISR_wra..dmaISR>TimerISR_w..TimerISR_w>CWPOverflo
    3   3   640219010 recursive_>dmaISR_wra..dmaISR>TimerISR_w..TimerISR_w>uartISR_wr
    3   4   640220650 recursive_>dmaISR_wra..dmaISR>TimerISR_w..TimerISR_w>uartISR_wr..uartISR>CWPUnderfl
    3   4   640249890 recursive_>dmaISR_wra..dmaISR>TimerISR_w..TimerISR_w>uartISR_wr..uartISR_wr>CWPOverflo
    3   3   640164930 recursive_>dmaISR_wra..dmaISR>TimerISR_w..TimerISR>CWPUnderfl
    3   3   640185950 recursive_>dmaISR_wra..dmaISR>TimerISR_w..TimerISR>uartISR_wr
. . .
```

FIGURE 6.10: Exact interrupt nesting sequences. A nesting sequence is a list of (interruptee, interrupter) pairs.

To summarise, the rich information about software performances extracted by the profiling mechanism illustrate that (a) the TB (with the post-simulation analyser) is a powerful observer of software behaviours, and (b) the TRG-based TP is an efficient controller of concurrent HW/SW behaviours. The profiling mechanism provides valuable opportunities to monitor and improve simulation quality. Furthermore, since this profiling mechanism is *independent* of the TP-generation method, it can serve as a *fair comparison platform* for different TP-generation methods. One example is to see whether a method can easily produce a profiling in which all possible interrupt nestings are covered. The next section introduces another experiment using the profiling mechanism: comparison of the efficiency of different TP structures.

6.5.3 TP Structure Efficiency: Application of Profiling

In this experiment we compare the efficiency of the polling-based, the pure event-driven and the hybrid TP structure. We expect that the event-driven TP should be much more efficient than the polling-based TP; but we are not sure how different it will be when the event-driven TP is compared with the hybrid TP. In order to compare them fairly, we need to reasonably define the *payload* and *overhead*.

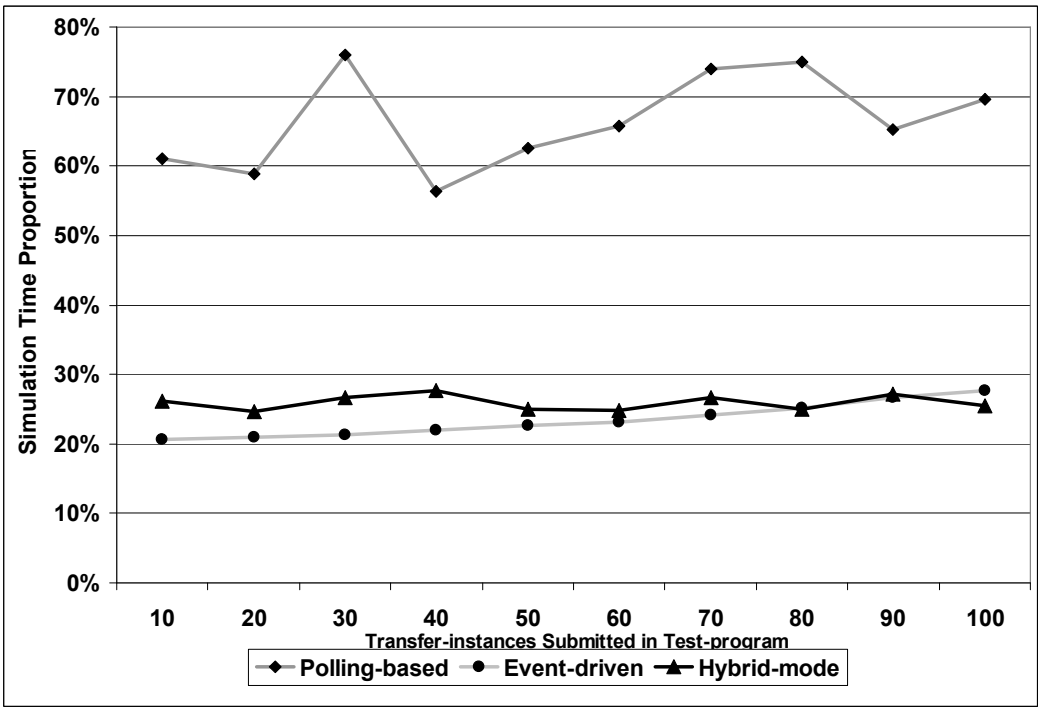


FIGURE 6.11: Using profiling to compare different test-program structures.

As mentioned in Section 4.2.2, a TP's scheduling overhead, i.e., the FIFO and running-flag operations, should not be confused with a transfer's control overhead, i.e., the configuration/invocation instructions.

- For the polling-base TP, the overhead refers to the running-flag-polling statements in function `main`;
- For the pure event-driven TP, the overhead refers to the submission-condition checks in function `Action_Table`, the FIFO operations dispersed in the ISRs and the `Scheduler`, and the infinite-loop-of-doing-nothing statement (C code: `while 1;`) in `main`.
- For the hybrid-based TP, the overhead refers to the FIFO operations dispersed in the ISRs and the `Scheduler`, and the polling statements in `main` at the end of each scenario.

Once the overhead operations are identified, they are organised into functions to be profiled. Figure 6.11 plots the CPU-time profiling of the overhead versus the transfer-submissions in the TP.

We observe that the polling-based TP wastes a high percentage of the simulation time in overhead, but the percentage is independent from the number of submitted transfers; the event-driven TP consumes much less time (still have room to improve, as mentioned in Section 6.5.2). However, as the number of transfers increase in the TP, there are more resource dependencies between the transfers. As a result, more time is consumed in checking running-flags. This explains the increasing overhead. For the hybrid TP, both the event-driven and the polling mechanisms are used, so the percentage is comparable to the pure event-driven case but is marginally higher. Like a polling-based TP, the percentage is independent from the number of transfers. This is because no running-flag checking is needed for hybrid TP (see Section 4.2.3). Hence, the hybrid mode TP is the most efficient among the three TPs when a large number of transfers need to be submitted. Also because of its advantages for debugging (see Section 4.2.3), we use the hybrid mode TP extensively in our research.

6.6 Summary

This chapter has quantitatively revealed various aspects of the proposed TP-centric and interaction oriented verification methodology, focusing on the simulation *completeness* and simulation *efficiency*. Although the criterion to decide simulation completeness remains to be an intrinsic problem, we could monitor and improve the simulation efficiency to increase the confidence level. While the mainstream methodologies are concentrating on hardware-based coverage (for completeness) and profiling (for efficiency), our methodology extends the coverage and profiling measurements to the software domain, allowing a coherent practice of hardware-software co-verification.

Chapter 7

Conclusion

7.1 Thesis Summary

This thesis reviews the general practices in electronics design verification, and proposes a novel but natural simulation-based approach to verifying one category of designs, namely, system-on-chip (SoC), which features (a) integrated components and (b) mixed hardware and software behaviours.

These two central features make an SoC subject to hard-to-conceive failure modes. The mainstream verification practices, represented in the form of test-bench-centric verification methodology promoted by EDA vendors, are not oriented toward these two specific SoC features. Instead, they focus on simulation infrastructure, complex test-bench structures and test-bench construction conventions. While proved to be *productive* for component-level verification, these practices are not *creative* enough to address the SoC verification challenges. In fact, they are experiencing conceptual confusions when being applied to SoC verification. These confusions include vague verification emphasis and unclear responsibilities of software in verification.

In our methodology, the efforts for system-level verification focus on the following:

- interactions between components;
- concurrency and the associated resource competition; and
- the roles of software (SW) in verification.

These seemingly loosely related topics are systematically articulated into a uniform verification methodology under the concepts of "interaction-oriented verification" and "software-centric verification".

In *interaction-oriented verification*, we abandon the component-oriented view of a system, and adopt a view in which the system is made of a collection of dynamic objects – interactions. "Interactions" are *tests* as well as *objects-under-test*.

- As "tests", interactions stress the communication mechanisms between design components. Moreover, by constructing concurrent interactions, the whole system can be exercised vigorously, greatly improving the test quality.
- As "objects-under-test", interactions have their own properties to be tested. Some properties are simply the reorganisation of properties that were originally associated with hardware components; interactions also introduce new properties – their temporal relations.

The interaction-oriented mindset overturns the distinction between the "test" and the "object-under-test", which is deeply rooted in the conventional TB-centric methodologies.

In *software-centric verification*, it is the software native to the system, not the test-bench (which is external to the system), that drives the simulation. Software does not have a proper position in the mainstream TB-centric methodologies. In contrast, our software-centric approach harmoniously positions the software and the test-bench in the verification framework; the test-bench external to the DUT and the software native to the DUT cooperate with each other harmoniously:

- the software provides high-level *control* over both the DUT and the test-bench;
- the test-bench provides powerful *observation* to both the DUT and the software.

As a result, we not only have a simpler test-bench, but also allow the early involvement of application software in system-level verification, whereas the TB-centric verification is dominated by the construction of complicated test-benches and the involvement of software has to wait till much later.

The link between "interaction-oriented verification" and "software-centric verification" is the "transfer" – a *software-controllable interaction-model*. *Transfer* is a model of interactions *at a certain temporal granularity*. The configuration-invocation-notification model of transfer is

(a) intuitive to understand, (b) simple to implement and (c) expressive enough to generalise heterogeneous interaction forms, including hardware-natured behaviours, software-natured behaviours and HW-SW collaborations.

Hardware properties are reorganised into transfer properties. As a result, details about hardware implementation can be abstracted away; only very generic property of "bit-resource availability" is left, consistent with a programmer's view of the system.

Based on transfers and resources, the whole system can be modelled as a *transfer-resource-graph* (TRG). A TRG could serve both as the test-generator of concurrency and as the coverage model of concurrency. By providing the test-generation method and the coverage measures for SoC designs, the TRG model marks the semantic difference between the interaction-oriented "methodology" proposed in this thesis and those "methodologies" promoted by EDA vendors, which are simply conventions to construct test-benches (TBs).

The transfer model has a huge impact on the structure of software. Software's roles are partitioned into independent components; these components respectively work as (i) transfer participants, (ii) transfers themselves, and (iii) the transfer manager, each contributing to the interaction-oriented mindset from a different perspective. This partition shows its rationale when compared with the software phenomena found in general-purpose computer system, namely, operating systems and user applications. Indeed, these partitioned components can be naturally organised into an *event-driven* test-program, which intelligently manages parallelism somewhat like an operating system.

To summarise, under the concepts of interaction-oriented verification and software-centric verification, which are linked by the transfer model, we now gain an insight into system-level behaviours as well as a new approach to driving these behaviours.

7.2 Application, Implication and Future Direction

Our methodology will have a wide range of applications in simulation-based verification due to its independence of the verification *infrastructure* (see Section 2.1.2.1). It is independent from

- hardware abstraction levels (transaction level, RTL, gate level, etc),
- hardware modelling languages (HDLs, HVLs, HDVLs) and

- simulation platforms (cycle-accurate simulators, event-driven simulators, HW-SW co-simulation, hardware prototyping, etc).

As a result, our methodology is applicable throughout an SoC implementation-verification cycle.

In fact, our methodology is not limited to the application of "verification", which largely refers to finding functional bugs in RTL implementation; it is applicable throughout the whole design-manufacturing cycle, including the early specification validation and the manufacturing-test. The reason is that the methodology focuses on the central phenomenon of a hardware system, namely, the *parallelism*, which is the source of various potential problems in the whole design-manufacturing span.

- In functional verification, *parallelism* is the source of functional bugs and corner cases.
- In the earlier design stage, e.g., hardware-software partition, *parallelism* and the associated resource-competition should be taken in to vigorous consideration to evaluate the "performance penalty" [19]. Therefore, methods to generate various scenarios of parallelism are required.
- In SoC manufacturing, *parallelism* can expose manufacturing defects unique to the deep-sub-micron manufacturing technologies, such as *crosstalk* [12] (i.e., signal interference among adjacent wires) and the associated extra power dissipations [94]. These defects only occur in vigorous parallel behaviours. The problem of self-testing for crosstalk and other technology-related defects at SoC level has not been addressed by the existing methods [94].

Therefore, focusing on *parallelism*, our methodology is applicable to these areas. Especially for the area of SoC manufacturing test, our software-centric methodology is consistent with the idea of software-built-in-self-test (SBIST) [84].

Our methodology, which is demonstrated on a single-processor SoC, can be readily extended to verifying other categories of VLSI designs, since the idea to construct resource-constrained parallelism is very generic. For instance, the TRG model does not restrict that tests must be organised by software (native to the DUT). The following design categories could adopt verification methodology similar to ours.

- For multi-processor SoC (MPSoC), the concepts about transfers and resources are still applicable. Our TRG method is able to make parallelism prevalent on a single-processor SoC; there is no reason why parallelism cannot be constructed similarly on

an MPSoC. We simply have more options to do that. A conceivable scheme is to let multiple processors share the responsibility of transfer scheduling by executing a globally accessible action-table.

- For designs with multiple components but without on-chip processor, we could introduce a *processor for verification purpose* and then use the same methodology described in this thesis. This *verification processor* can be regarded as a TB component, and the software running on the processor can be regarded as the stimulation from the TB to the DUT. Again, we see that the distinction between the TB, the TP and the DUT is a matter of interpretation; they should be understood as one *continuum* (see Section 5.1).

- For component-level verification, where software is not present, the concept of "interaction-oriented verification" still applies and the TRG model is still valid; but it is now natural to resort to a test-bench for interaction scheduling. We should realise "TP- or TB-centric" and "component- or interaction-oriented" are two independent differentiations, as suggested in Table 7.1;

Methodology	Component-oriented	Interaction-oriented
TP-centric	•Not very exciting common practice •Hardware diagnostics	•Emerging, this thesis •SoC verification
TB-centric	•Traditional •Component-level verification	•Consciously practiced? •Features? Applications?

TABLE 7.1: Methodology Differentiation.

Moreover, since the system-on-chip is not fundamentally different from the system-on-board except the encapsulation and packaging, our methodology could find its application on a wide range of digital designs.

The "interaction-oriented mindset" has simply reconfirmed the same philosophical implication derived from many other fields of scientific studies – a system is more a collection of "processes" than a collection of "substances". (The term "interaction" is even a better appellation than "process": it is a "process" with "substance" implicitly considered.) This view provides insights into the nature of the universe. For instance, the fundamental matters in the universe seem to be more like *entanglement* than *particles*; the *life* is more the process

of *metabolism*, i.e., the controlled flow of chemicals/energy/information, than the structure of *organisms*.

The interaction-oriented mindset and software-centric verification will simply become more dominant in the future VLSI design and verification. The International Technology Roadmap for Semiconductors [49] suggests that (a) programmability and (b) communications are the two sources of verification complexity – *the underlying reasons for this unmanageable complexity lie in the inability of verification to keep pace with highly integrated system-on-a-chip (SoC) designs and parallel chip-multiprocessor systems, paired with highly interconnected communication protocols implementing distributed computation strategies.* Indeed, MPSoC and Network-on-Chip (NoC) are the two important technology-nodes in the roadmap. Designing and verifying an MPSoC require more vigorous involvement of software since an MPSoC's behaviours are software intensive; while an NoC requires rigorous verification of its intelligent on-chip communication mechanism. *Software-centric verification* and *interaction-oriented verification* respectively satisfy the requirements from MPSoC (programmability) and NoC (communication). Since an MPSoC could simultaneously have an NoC, the combination of the software-centric verification and the interaction-oriented verification proposed in this thesis is simply natural.

Appendix A

Major Bugs in the Nios SoC

This appendix provides the detailed description of three problems identified on the Nios SoC used in our research. They are typical *system-level* bugs since they all occur during interactions between multiple components; and it is sometimes arguable to blame a single component to be responsible for a bug. Instead, the bug is revealed as ill-matched behaviour between components. And interestingly, very often a problem is revealed in a seemingly unrelated arrangement. In some sense, the term "bug" is very subjective – it simply refers to the situation in which the simulated reality does not match human expectation; and the human expectation cannot always be exhaustively expressed *before* the mismatching happens. That is why the formal methods are not at the position for system-level verification and we have to resort to the simulation approach.

A.1 Weak end-of-packet (EOP) Arbitration

Symptom 1: A Memory-Memory-DMA transfer terminates unexpectedly before the desired transfer length is reached.

Conditions:

- The MEM-MEM-by-DMA transfer is running concurrently with a UART-RX(TX)-by-Interrupt transfer;
- In addition to the DMA engine's `len` bit, its `ween` or `reen` bit is also set. (`len`, `reen` and `ween` terminate the DMA engine respectively when (a) the length is reached, (b) the

`eop` is sensed by the DMA-read master and (c) the `eop` is sensed by the DMA-write master.)

Explanation/Cause: The root cause of the failure is due to the following problematic assignments in the modules called DMA read/write master arbitrators (in file soc.v).

```
assign DMA_1_write_master_endofpacket = Uart_1_s1_endofpacket_from_sa;

assign DMA_1_read_master_endofpacket = Uart_1_s1_endofpacket_from_sa;
```

Although the UART is the only possible source of `eop` signal for the DMA engine, the DMA engine should not be sensitive to the `eop` signal from the UART when the engine is *not* accessing the UART. But these two statements do not match our expectation.

Workaround: Make sure that the `ween` (write-eop-enable) and `reen` (read-eop-enable) bit in the DMA engine's control-register are de-asserted if the DMA engine is not working with the UART (e.g. in a memory-memory DMA copy).

But this workaround is not general enough. It assumes that the UART is the only device in the system that could synchronise with the DMA engine on the `eop` signal.

Proposed Fix: We attempted to fix the problem in hardware by ANDing the `eop` with the UART-grant signal, i.e.,

```
assign DMA_1_write_master_endofpacket =
       Uart_1_s1_endofpacket_from_sa & DMA_1_write_master_granted_Uart_1_s1;

assign DMA_1_read_master_endofpacket =
       Uart_1_s1_endofpacket_from_sa & DMA_1_read_master_granted_Uart_1_s1;
```

After modification, the unexpected memory-memory-DMA-copy termination issue is addressed.

Symptom 2: A UART-RX-by-DMA transfer terminates unexpectedly before the desired length is reached or the `eop` character is received.

Conditions: The UART-RX-by-DMA transfer is running concurrently with a UART-TX-by-Interrupt transfer.

Explanation/Cause: The problem can be illustrated in Figure A.1. The UART is working in full-duplex fashion. The DMA and the CPU are accessing the UART respectively for the RX and the TX services. The UART arbitrator grants their access-requests by fine-grained interleaving. The unexpected RX-by-DMA termination happens when an `eop` character is sent in the TX stream. When this character is sent by the CPU, the UART asserts its `eop` signal on the bus; when the arbitrator grants the DMA engine the UART access, the `eop` signal makes the DMA engine terminate. Worse, the DMA engine may be out of synchronisation and trapped in an illegal status, in which both the `busy` and the `done` bits are set.

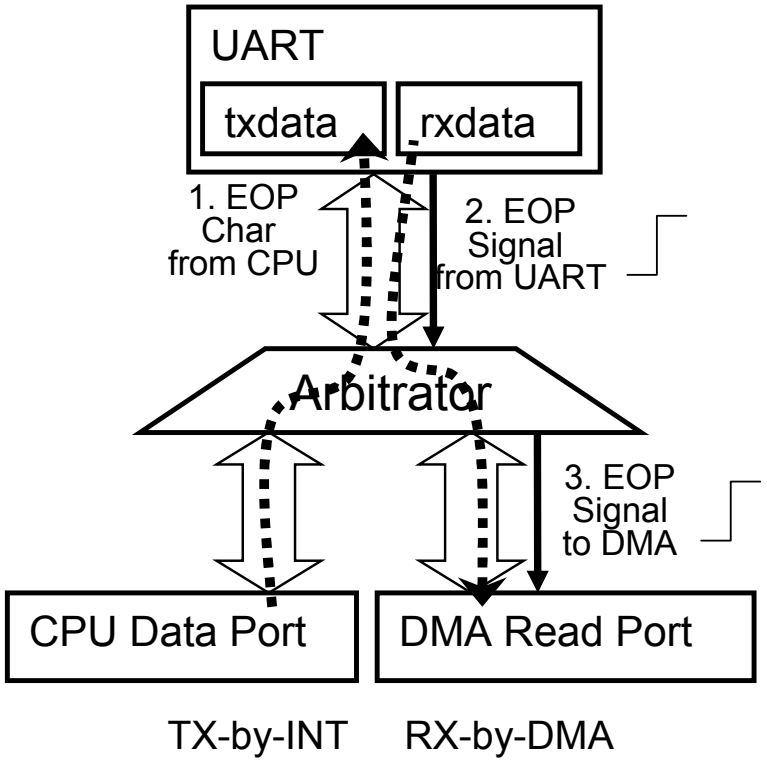


FIGURE A.1: The arbitration problem of the end-of-packet (EOP) signal. During a full-duplex scenario, where the RX stream and the TX stream are conducted by different masters, the EOP signal in the TX (or the RX) stream will be mistakenly propagated to the RX (or the TX) stream.

Workaround: It seems that any workaround in software will introduce some complications or limitations. For instance, we could impose a limitation that the UART should not be accessed by hardware masters when it is working in full-duplex.

Possible Fix: There seems no particular culprit in this ill-behaved scenario. It appears that the proper modification could be done on both the UART arbitrator and the UART itself, i.e.,

- let the UART generate `tx-eop` and `rx-eop` signals separately; and
- let the arbitrator work logically on the `eop` signals – remembering which master is responsible for which `eop` signal.

But this solution would be too complicated, since the AVALON bus only defined *one* `eop` signal for each peripheral, while the UART is capable of two independent streams. As we mentioned in Section 4.3.2, the shared EOP actually reflects the inadequacy in the UART design; and therefore letting the arbitrator, which is supposed to be simple, to handle both physical-level and logical-level arbitration is not reasonable.

The best solution is to separate the RX and the TX functionality in two peripherals; this radical solution addresses the problem both at the physical level and the logical level:

- logically, each of the TX and the RX functionality has a dedicated interrupt and `eop` signal, and
- physically, there will be no need for interleaving-based arbitration since the DMA engine connects the RX and the CPU connects the TX peripheral (or vice versa).

A.2 Transient Interrupt Request

Symptom: During an RX-by-DMA transfer, an DMA-ISR is triggered. But strangely, the ISR only finds that the transfer is still in progress; none of the possible triggering events (`reop` and `len`, respectively means that an `eop` is read and the transfer length has reached) has occurred. In short, the DMA-ISR finds that it is triggered for no reason. And the DMA-ISR is triggered again and again until the transfer finally finishes.

Condition: For the RX-by-DMA transfer with this strange behaviour, the `irrdy` control bit in the UART is set. The `irrdy` allows the `rrdy` interrupt (one of the UART interrupts) to fire when the `rxdata` register in the UART has received a new character. This `irrdy` bit must be set for an RX-by-ISR transfer, since it enables the virtual master, i.e., the `rrdy` ISR, to conduct the RX-by-ISR transfer, but it is not required to be set in an RX-by-DMA transfer. However, in order to implement the concept of treating a general interrupt as a transfer *event* (see Sectionsubsec:modellinginterrupt), we also allow the `rrdy` ISR to fire in an RX-by-DMA transfer; we expected that `rrdy` ISR could map the `rrdy` interrupts as events to transfer RX-by-DMA. This should not slow down the RX-by-DMA transfer, since it is the CPU that executes the ISR while the DMA is conducting the real transfer simultaneously.

However, we did not see any `rrdy` event logged by the `rrdy` ISR; instead, we observed that the DMA-ISR was repeatedly triggered for no reason.

Explanation/Cause: The cause of this strange behaviour is due to the fact that the DMA engine reads the UART's `rxdata` register very fast (as it should). Whenever the `rxdata` register is ready, the `rrdy` interrupt is asserted to the interrupt-subsystem, and the UART's `rrdy` ISR is supposed to response to handle this interrupt request; however, since the DMA also senses this `rrdy` event via the AVALON bus and immediately (in two SoC-cycle) read out the `rxdata` register, the `rrdy` interrupt request is quickly de-asserted in two cycles. (In contrast, during a normal RX-by-ISR transfer, the `rrdy` lasts more than 150 SoC-cycle before the ISR reads the UART.) It is this transient (two cycles) UART interrupt-request that fools the interrupt-subsystem.

When any interrupt happens, the interrupt-subsystem checks which peripheral is requesting the interrupt; and each peripheral is associated with an ID called "irqnum", or irq number, which is used as the index to its entry in the interrupt-vector-table. The UART has a smaller (higher-priority) irqnum of 16 and the DMA engine has a greater (lower-priority) irqnum of 19. Actually the DMA engine's irqnum is the greatest in the Nios SoC. When an interrupt request fires, the interrupt-subsystem is triggered to check the irqnum; but if the request is too short, the corresponding irqnum would not be correctly captured. What would be capture instead is the greatest irqnum in the Nios SoC, which, unfortunately, is the DMA's irqnum 19. The exact Verilog coding is as follows:

```
assign CPU_32bit_data_master_irqnumber =
       (Uart_1_s1_irq_from_sa)? 16 :  //16: UART IRQ number
       (Timer_s1_irq_from_sa)? 17 :   //17: Timer IRQ number
       19;  //19: DMA IRQ num. But the IRQ source is not checked.
```

This assignment states: if the interrupt request is not from the UART (16) or the Timer (17) in that order, it must come from the DMA engine(19). Note that the right-hand-side *does not actually* check the DMA engine as it checks the UART and the Timer. That is the root cause why the DMA-ISR is triggered instead of the UART-ISR, and why the DMA-ISR finds it is triggered for no reason. This problematic behaviour does not agree with the document [3], which states that any transient interrupt request will be ignored. This illegal behaviour is illustrated in Figure A.2.

Workaround: Enabling/disabling the `rrdy` event is a parameter to the transfer-type "RX-by-DMA". With the root cause clearly known, we simply disable this `rrdy` event by setting the corresponding randomisation weight of enabling `rrdy` event-reporting to zero.

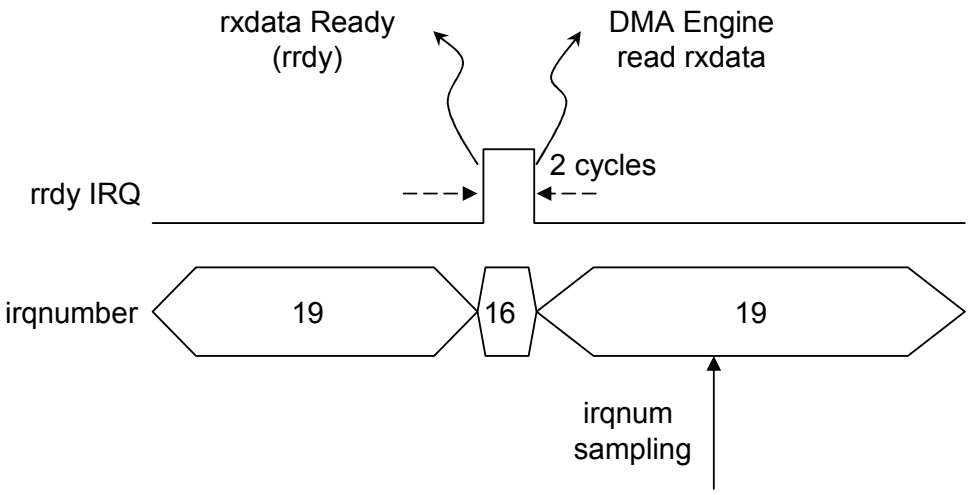


FIGURE A.2: The transient interrupt-request problem. If the `rrdy` interrupt-request (IRQ) fires but does not maintain long enough, the transient request is NOT ignored as it should be. Instead, a wrong IRQ number representing DMA IRQ is sampled; consequently, the DMA interrupt-service-routine is triggered illegally.

Implication: We should note that this mis-fired DMA ISR has nothing to do with the DMA engine, who *happens* to be conducting a normal transfer; it also has nothing to do with the UART, whose behaviour is reasonable. The mis-fired ISR *happens* to be the DMA-ISR simply because the questionable implementation of the interrupt-subsystem. We could generalise the risk in this way: any transient interrupt request could trigger a peripheral's ISR who owns the last irqnum in the right-hand-side of the above assignment. The mis-fired ISR is potentially very hazardous.

Proposed Fix: To finally fix the problem in hardware, the assignment should explicitly enumerate all HW peripheral's irqnums, and include a default and "invalid" irqnum as shown below; therefore, when a transient interrupt happens, the invalid irqnum is selected and the CPU has a chance to choose to ignore this hardware event.

```
assign CPU_32bit_data_master_irqnumber =
       (Uart_1_s1_irq_from_sa)? 16 :  //16: UART IRQ number
       (Timer_s1_irq_from_sa)? 17 :   //17: Timer IRQ number
       (Irq_From_DMA)? 19 :           //19: DMA IRQ number
       255;                           //255: the invalid IRQ number.
```

To summarise, this issue shows a typical "corner-case" bug which reveals itself in ill-matched behaviours among hardware components (the UART, the DMA engine and the interrupt-subsystem in the CPU). Although in the real application, the `irrdy` bit would not be enabled

if we decide to use the DMA engine to serve the HW `rrdy` events, the idea to treat interrupts as transfer event is logically valid and proves to be helpful to detect rare corner-cases.

A.3 Weak DMA Control

Symptom: After a failed RX-by-DMA transfer, a Memory-Memory-DMA-copy transfer runs totally out of control –

- the DMA transfer starts ghostly and illegally BEFORE the invocation instruction (setting the `go` bit in the DMA control-register) is issued;
- the source and destination addresses during the illegal transfer are not as configured;
- the `busy` status bit is NOT set during the illegal transfer; and
- when the illegal transfer finally stops, NO interrupt is raised.

Condition: The disastrous DMA behaviour happens when the DMA-engine is being reconfigured for a new DMA transfer *after* a failed RX-by-DMA transfer. The reason why the previous RX-by-DMA transfer has failed is the result of error injection to the RX stream. The UART is able to detect four possible errors, (1) frame error, (2) parity error, (3) break, (4) and receiver overrun. They are typical error modes for real-world asynchronous communication. In order to inject these errors easily, we let the RX transactor in the test-bench to operate at a "bad" baud-rate which is different from the nominal baud-rate configured for the UART in the SoC. The deviation is configurable and max 10% off the nominal value set for the UART. Our intention is to (a) test the UART's robustness in receiving a stream that is not perfectly in pace with the nominal baud-rate, and (b) test if the UART' error detection mechanism could work as expected. When any error happens in an RX transfer, it is treated as an "abortion-event" to that transfer. The transfer should be terminated by software, i.e., the UART-error ISR. If the RX transfer is conducted via interrupt, the termination operations only involves the resetting of the UART receiver; when the RX transfer is conducted by the DMA-engine, the termination operation should also terminate the DMA engine's behaviour; this is because the RX transactor (in the test-bench) may be out of pace with the UART receiver (in the SoC), and then the DMA engine could be waiting for transactions that are never to happen.

Explanation/Cause: However, the DMA implementation seems not robust enough to handle the termination operation. In fact, the DMA engine does not have an "awareness

for errors". To start up a DMA transfer, it is sufficient to set the `go` bit; then the DMA engine asserts its "busy" bit. And the user is supposed to expect the "busy" bit to be de-asserted at the end of a successful DMA transfer. The Nios document [4] does not specify the situation in which DMA termination is needed for an unsuccessful DMA transfer. We have to implement the termination operation using common sense, i.e., to reset the control register (which holds the `go` bit) and the `length` register, and verify the `busy` bit is reset. Indeed, by using this commonsense abortion operation, the DMA transfer stops and the `busy` bit is reset; however, the DMA engine's internal control logic and data-path are *not* really purged. In short, the DMA is now in an illegal status. At this moment, when there is a further write operation to any DMA register (which is exactly what happens in configuring a subsequent memory-to-memory-DMA-copy transfer), the DMA engine could fire illegally.

Workaround: To forbid this illegal DMA transfer using software seems not reliable – resetting all five registers of DMA may or may not help. Using the simulation tool VIRSIM, we still observe that the illegal DMA transfer could happen secretly after the control register is zeroed out. We have to accept the fact that the illegal DMA transfer would fire, and attempt to nullify its effect. To do that, in addition to zeroing out the control/status/length registers, the content in the source/destination registers should be set to point to the ROM; so that even if the DMA engine fires illegally, no RAM/register could possibly be corrupted. These termination operation are organised in a special subroutine called `TerminateRunningRX()`.

Proposed Fix: Obviously, the root cause is that the DMA engine's implementation has no awareness of errors and does not provide enough controllability for software. The best solution is to allow an explicit reset-bit in the control register to allow clean purge of the hardware.

Appendix B

Address the Complications in the Register-Window Mechanism

To facilitate fast context switching, the Nios CPU uses a register-window mechanism. The processor is configured with 256 general-purpose registers; but only 24 of them (as one register-window) are currently available in the current execution context. There are another 8 general-purpose registers for global information, so altogether a programmer has access to 32 general-purpose registers. When a function (the caller) calls another function (the callee), the register-window slides down 16 registers. Now the current window contains 16 new registers for the callee to use, and the overlapping 8 registers being the communication mechanism between the caller and the callee, i.e., arguments from the caller to the callee and return-values from the callee to the caller. When the callee returns, the register-window slides up 16 registers to restore to the old window.

Since a new window uses 16 new registers, 256 registers contain at most 16 register-windows. When the register-window is about to go over the limit, the underflow (i.e., no new window to open) or the overflow (i.e., no old window to restore) exception is raised. These two exceptions have the highest interrupt priority (priority code 1 and 2 respectively), and are serviced by respective exception handlers supplied in the Nios software package. The underflow handler saves the all register-windows to reserved memory locations; and the overflow handler restores the register-windows from the reserved memory locations. With these handlers, in most cases, a programmer does not have to worry about the details of the register-window mechanism.

It seems that the original intention of the Nios CPU instruction-set-architecture allows and prefers *interrupts* to be serviced in a new register-window just like a normal function call.

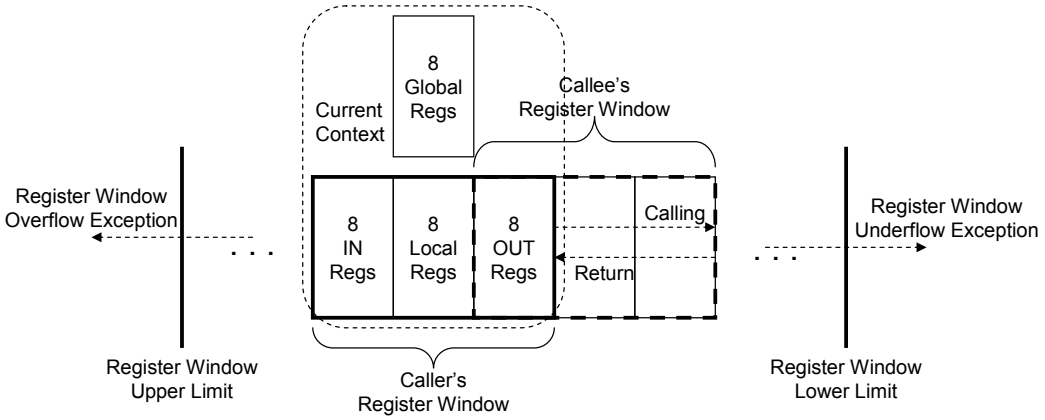


FIGURE B.1: 256 registers in the Nios SoC are organised as 16 register-windows. The current function occupies 24 continuous registers as one register-window. When the function calls or returns from another function, the register-window slide down or up 16 registers. The 8 overlapping registers are used for arguments/return values. There are another 8 registers for global access. When the register-window slides out of range, underflow or overflow exception will be triggered.

When an interrupt happens, the register-window automatically slides down 16 registers just like a normal function call; so an interrupt is also possible to cause the register-window underflow. Unfortunately, such an underflow caused by interrupt will *not* be triggered. Hence, the ISR programmer should explicitly handle a possible underflow situation. To circumvent this complication, the programmer may choose to wrap his/her ISR within an ISR-wrapper, which is shared by all user-ISRs. This ISR-wrapper is also provided in the Nios package. This wrapper, for safety reasons, performs very special operations to circumvent the using of the new register-window.

(1) The wrapper firstly uses special instructions to *undo* the new register-window already automatically opened for the interrupt, in order to recover from an possibly missed underflow-exception.

(2) The wrapper then saves the general- and special-purpose registers that could be over-written by the user-ISR to reserved memory locations;

(3) It `CALL`s the user-ISR. Using a normal `CALL` allows the possible underflow-exception to be captured.

(4) When the user-ISR returns, the wrapper restores the saved context from the memory locations.

(5) Finally, the wrapper uses special instructions to artificially open a new window only in order to close it and return the interruptee using the `IRET` (interrupt-return) instruction, emulating a true interrupt-return.

In a *normal* function, sliding the register-window is automatically accompanied with the passing of program-control, i.e., the jump in program-counter; but the ISR-wrapper is so special that it modifies the current window at step 1 and 5 without losing program-control. Using the ISR-wrapper, the problem of possible un-attended underflow exceptions is addressed, *transparently* to the user-ISR. However, the cost includes 30 memory-operations (14 STOREs, and 16 LOADs) in the wrapper, making the wrapper a severe overhead to run the user-ISR. The overhead is around 55% if it is defined as

$$wrapper\_time \div (wrapper\_time + user\_ISR\_time).$$

The overhead data is obtained from the TP profiling mechanism described in Section 5.3.2.

It is this high overhead that motivates us to reduce it by composing a custom ISR-wrapper. The new wrapper firstly checks the register-window status: if the underflow is about to happen (1/16 chance), the same special treatment described above is carried out; otherwise (15/16 chance), it is safe to utilise the wrapper's register window to save the interruptee's context. In this case, the interruptee's 8 OUT registers are *already* secured in the wrapper's 8 IN registers; and the 16 new registers in the wrapper's window can be used to save the global and special-purpose registers before the user-ISR is `CALL`ed. Now, for 15/16 chance, the 30 memory-operations are reduced to only 18 register-operations. This treatment agrees with the original intention of the processor design by fully exploiting the register-window. As a result, the overhead caused by the ISR-wrapper decreases to about 20%.

The optimised ISR-wrapper works correctly in extensive simulation-runs. Therefore, the wrapper not only helps to improve simulation efficiency, it is also proved to be correct code ready for real application.

Appendix C

Test Generator Implementation

Conceptually, the TRG-based test-generator consists of the following two types of information.

- SoC-independent information, which we also refer to as methodology-knowledge. This part basically includes (a) class definitions (transfer-types, transfer-instances, resources, scenarios, etc) and (b) the scenario generation code that implements the algorithm described in Section 3.4.2.
- SoC-dependent information, which we also refer to as system-knowledge. This part includes the data and functions that will be instantiated as transfer-type-objects and resource-objects.

For instance, transfer-types-objects' $P()$'s, $I()$'s are system-knowledge, whereas the method to organise all $I()$'s outputs into the action-table (see Appendix D.2) belongs to methodology-knowledge.

Decoupling system-knowledge from methodology-knowledge makes it easy to maintain the generator; and the generator could be easily applied to another system with minimum modification. Since the TRG methodology is intuitive and simple, for our Nios SoC, the ratio between methodology-knowledge and system-knowledge is roughly 1:2 in terms of code size (Python language). The test-generator combines the system-knowledge and the methodology-knowledge in the form of object-instantiation.

In addition to the system-knowledge and the methodology-knowledge, a user is allowed to specify randomisation strategies (i.e., biases) that are either system-dependent or system-independent.

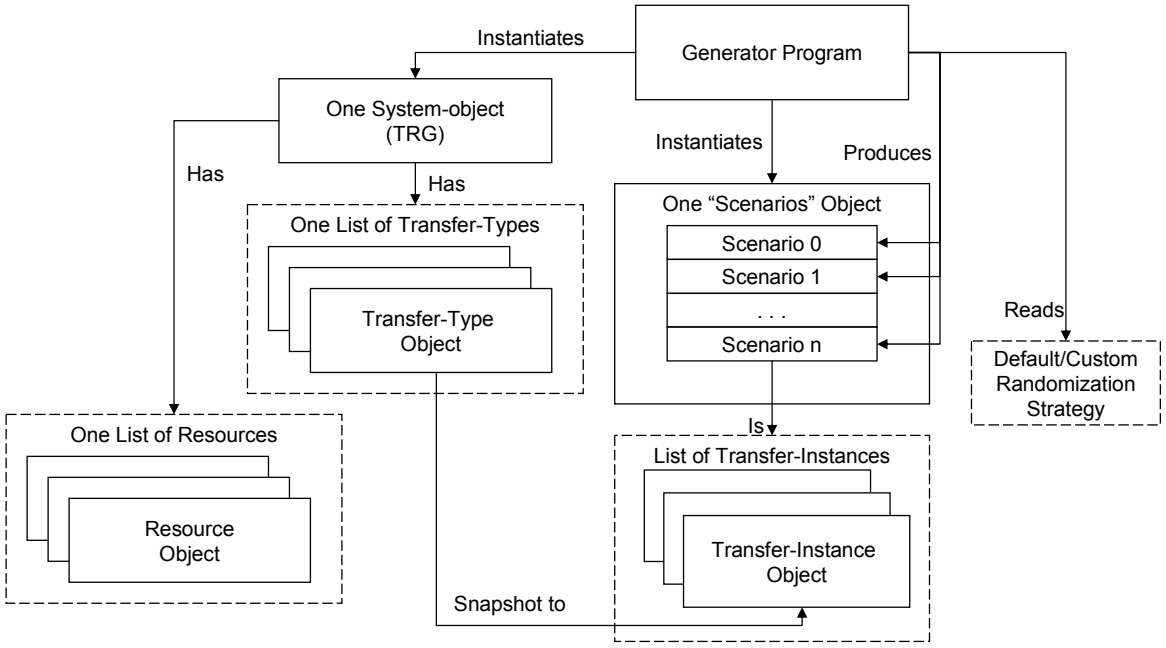


FIGURE C.1: Implementation of the test-generator. Solid boxes are objects instantiated from the classes contributed by the methodology; dashed boxes are objects of types built-in the Python language. Arrows represent the relation between objects.

Figure C.1 illustrates our implementation of the generator (for the hybrid-mode test-programs). Dashed-boxes are Python's built-in data objects, and solid boxes contain methodology/system-knowledge; here we briefly introduce some important objects and their data/method members. Their functionalities can be easily implied from their names.

(A) A resource-object has the following important methods:

- $InquireResourceAvailability()$: return a data-structure representing available resources in this object.
- $AllocateResourceToTransfer()$: allocate an available resources and update the internal resource usage.
- $ReturnUnrealisticTransferTypes()$: return a list of transfer-types that cannot obtain sufficient resources from this object.

(B) A transfer-type-object has the following data and method members:

- $MinimumResourceRequirment$: the minimum resource requirement of this transfer-type.

- *SetupInstructions*: the *static* instructions that setup this transfer-type, not configuration.
- *ParameterizationAlgo*: this is a *reference* to the function that implements the $P()$ operation.
- *InterpreteToConfigurationAlgo*: this is a *reference* to the function that implements the $I()$ operation, which interpret transfer-parameters into configuration instructions.
- *InterpreteToInvocationAlgo*: similar to *InterpreteToConfigurationAlgo*, but for invocation instruction.
- *Parameterization*(): this method uses the reference *ParameterizationAlgo* for parameterisation.
- *InterpreteToConfiguration*(): this method uses the reference *InterpreteToConfigurationAlgo* to generate configuration-instructions.
- *InterpreteToInvocation*(): this method uses the reference *InterpretToinvocationAlgo* to generate invocation-instructions.

Here, we use a programming technique to avoid a conceptual pitfall. Note that we do *not* implement $P()$ and $I()$ directly as the *method-members* in the *class definition* of transfer-type, this is because $P()$ and $I()$ are conceptually associated with the *objects* of transfer-type, not the transfer-type *class* i.e., the *model* called "transfer-type". We treat $P()$ and $I()$ as *data-members* (e.g., the references *ParameterizationAlgo*) rather than *method-members*. To invoke them, the *class* uses the formal method-members (e.g. *Parameterization*()) which use the *reference* to call the actual $P()$ and $I()$. In this way, all transfer-type-objects *formally* share an exactly the same class, having the same formal data-members and method-members, but have different actual $P()$'s and $I()$'s. One class is enough – this class is the transfer-type "model". We think that this "reference technique" is conceptually important, because the methodology-knowledge (the class definition) and the system-knowledge (the information for object instantiation) are clearly separated.

Alternatively, it is possible to directly treat $P()$ and $I()$ as method-members of a class, however, since each object's $P()$ and $I()$ are specific, we have to resort to the inheritance technique, i.e., for each transfer-type, we *derive* a class from a base class (in which $P()$ and $I()$ are declared but not implemented) and implement the real $P()$ and $I()$ in the derived class. This "inheritance technique" does not naturally reflect the transfer-type as a "model", since now each transfer-type-object is instantiated form a *different*

derived-class, and each such class only instantiates one object. Obviously, the system-knowledge (the objects) and the methodology-knowledge (the classes) are mixed in the inheritance technique.

(C) A transfer-instance-object are only the containers of the *results* of a transfer-type's parameterisation and interpretation operations, snapshot from the corresponding transfer-type-object; so the transfer-type-object can be reset for the next round of scenario-generation. Transfer-type-objects are not bound up with scenarios, but transfer-instance-objects are. For a hybrid-mode test-generator, a transfer-instance do not have method-members that support test-generation.

(D) The system-object implements the transfer-resource-graph (TRG). It includes:

- $TransferTypeObjs$: a list of transfer-type-objects.
- $ResourceObjs$: a list of resource-objects.
- $LinkTransfersAndResources()$: link resource-objects and transfer-type-objects using transfer-type-object's $MinimumResourceRequirment$. This method actually constructs the TRG.
- $SelectOneTranferInstanceAndUpdate()$: a method that generates a transfer-instance from the current available resources and update resource-usage and transfer-type validity; this is the central step of scenario generation, incorporating many services provided by transfer-types and resources.
- $Reset()$: reset the TRG for the next round of scenario generation.

We may choose to implement the complete scenario-generation algorithm as this system-object(TRG)'s method; however, since scenario-generation should allow for users' intervention, to make the TRG, which is suppose to be a abstract model, dealing with the user-defined options is not very appropriate. Therefore, the complete scenario-generation algorithm is implemented in the top-level generator program (F).

(E) The "scenarios" object is basically a placeholder for scenarios, each of which is in turn the placeholder for transfer-instances. And this object provides the method that actually organises configuration/invocation instruction fragments into the action-table. Thus this object essentially has

- $ScenarioList$: a list of scenarios, each scenario is a set of transfer-instances, and
- $GenerateActionTable()$: this method assembles all transfer-instances' configuration/invocation instructions into the action table – a single `switch` statement, which contains a lower level of `switch` statements.

(F) the top-level generator program interacts with the system-object (D) to produce scenarios, and place them into the scenarios-object (E), which finally returns the action-table. In this process, it applies the default or custom randomisation strategy. It also generates the code that updates environment parameters. The program eventually generates `hybrid.c`, `hybrid.h` and `scheduler.c` files, which will be detailed in Appendix D.

Finally, in the following program-listing (in Python language), we show all system-knowledge necessary for instantiating the transfer-type-object "UART_TX_by_DMA". Comments are lead with "#'; and "//" marks the comment in C, which is treated as normal C code.

```
1
2  ......
3  # Part 1: Information to instantiate the transfer-type-object.
4
5  TransferDefinitionList.append(('TXBUF2TXDATAbyDMA',         #The internal-name for this transfer-type-object
6                      ['DMA1READ','DMA1WRITE'],    #The required resources -- masters
7                      ['TXBUF1','TXDATA1'],        #The required resources -- memory-ranges and registers
8                      ['temp=na_DMA_1->np_dmacontrol; //Read-modify-update DMA Control_reg\n'+\
9                       'temp&=~(np_dmacontrol_rcon_mask+np_dmacontrol_wcon_mask+np_dmacontrol_byte_mask+\
10                       np_dmacontrol_hw_mask+np_dmacontrol_word_mask);\n'+
11                      'temp|=(np_dmacontrol_wcon_mask+np_dmacontrol_byte_mask);\n'+
12                      'na_DMA_1->np_dmacontrol=temp; //Finished setting DMA rcon/wcon=0/1;b/hw/w=1;',
13                      'na_DMA_1->np_dmareadaddress=(int) txbuf;//set DMA Read_addr=int(char*(txbuf));',
14                      'na_DMA_1->np_dmawriteaddress=(int) &(na_Uart_1->np_uarttxdata); //set DMA Write_addr'],
15                      #The above ugly list contains the static instructions to set up this transfer-type
16                      P_D_TXBUF2TXDATAbyDMA,       #The reference to P()
17                      P_I_TXBUF2TXDATAbyDMA,       #The reference to the configuration part of I()
18                      S_H_TXBUF2TXDATAbyDMA,       #The reference to the invokation part of I()
19                      R_F_TXBUF2TXDATAbyDMA))      #Not used for event-driven/hybrid TP. (For RF polling.)
20 ......
21
22 #Part2: The Randomisation Strategy of the transfer-type: a couple of (SelectionWeight, ParameteraisationStrategy).
23 TransferStrategyTable['TXBUF2TXDATAbyDMA']=(1,{                              #Selection Weight = 1
24     'ws_finish_mode':(['LENGTH','WEOP','LENGTH_OR_WEOP'],[1,1,1]),           #How this transfer finishes.
25     'wc_enabled_INT':(['leen_m','ween_m','ieop_s','itrdy_s','itmt_s','itoe_s'],[1,1,1,1,1,1],3), #Event enabling
26     'rs_length':(1,512),                      #Length, valid only when finish_mode='LENGTH' or 'LENGTH_OR_WEOP'
27     'ws_RFclear_INT':(['enable','disable'],[3,0])}) #Always enabled in event-driven/hybrid TP.
28
29 ......
30 #Part 3: Thea actual implementation of P() and I() of the transfer-type.
31
32 #Part 3.1: The P().
33 #Input: (a): parameterisation strategy; (b) ALL resource objects
34 #Output: (a): concrete parameters; (b) allocated resource (Not used in polling/hybrid TP!)
35
36 def P_D_TXBUF2TXDATAbyDMA(PARA_STRAT,RES_OBJ_LIST):
37     value_dict={}; resource_dict={'MEMORYBLOCK':[],'DATAREGBIT':[],'MASTER':[]}
38
39     #now decide the parameter: finish_mode
40     finish_mode=WeightedChoice(PARA_STRAT['ws_finish_mode'])
41     value_dict['finish_mode']=finish_mode
42     if finish_mode=='NONE_CHOICE':
43         print 'zero weight!!!'
```

```
44          sys.exit(1)
45
46      #now decide the parameter enabled_INT; firstly intercept the strategy table for one constraint consideration
47      INTlist=PARA_STRAT['wc_enabled_INT'][0]
48      Weightlist=PARA_STRAT['wc_enabled_INT'][1][:]
49      BaseWeight=PARA_STRAT['wc_enabled_INT'][2]
50      #there is an constraint between enable_INT and finish_mode: if finish_mode
51      #is 'LENGTH', (length only); 'ween_m' should NOT be included in enabled_INT. modify the Weight for ween to 0
52      if  finish_mode=='LENGTH':
53          Weightlist[1]=0     #ween is on position 1 in the list
54
55
56      enabled_INT=WeightedCombination((INTlist,Weightlist),BaseWeight) #free parameter
57      value_dict['enabled_INT']=enabled_INT
58
59      #The next parameter to be decide is the eop character, however,
60      #since eop is more appropriate to be an environment variable than a transfer parameter, so the
61      #next 'if' is simply passed;
62      if finish_mode=='WEOP' or finish_mode=='LENGTH_OR_WEOP':
63          #dependent parameter;
64          pass
65
66      #Now decide the parameter length, it is dependent on finish_mode;
67      if finish_mode=='LENGTH' or finish_mode=='LENGTH_OR_WEOP':
68          length=randgen.randint(PARA_STRAT['rs_length'][0],PARA_STRAT['rs_length'][1])
69      else:#when finish_mode is 'REOP' there will be no length being specified
70          #however, length should still be set to the maximum buffer size to make the transfer safe
71          length='TXBUF_SIZE'
72      value_dict['length']=length
73
74      #now decide the parameter, RFclear_INT, which is heavily dependent on finish_mode
75      RFclear_INT_enable=WeightedChoice(PARA_STRAT['ws_RFclear_INT'])
76      RFclear_INT=([],0)
77
78      if finish_mode=='LENGTH_OR_WEOP':
79          if RFclear_INT_enable=='enable':
80              RFclear_INT=(['ween_m','leen_m'],3) #bit 0 indicate leen(dma); bit 1 indicate ween (dma);
81
82      elif finish_mode=='LENGTH':
83          if RFclear_INT_enable=='enable':
84              RFclear_INT=(['leen_m'],1)
85
86      else: # finish_mode=='WEOP': in this case leen_m should still be enabled to make sure at the end of buffer
87          if RFclear_INT_enable=='enable':
88              RFclear_INT=(['ween_m','leen_m'],3)
89
90      value_dict['RFclear_INT']=RFclear_INT
91       #bit 0 indicate leen(dma); bit 1 indicate ween (dma);
92
93      return (value_dict,resource_dict)
94
95  #Part 3.2 -- The configuration part of I()
96  #Input: concrete parameters;
97  #Output: a SET of instructions (in form of a list)
98
99  def P_I_TXBUF2TXDATAbyDMA(PARA_VAL_DICT):
100
```

```
    instruction_list=[]

    instruction_list.append('FM[TXBUF2TXDATAbyDMA]='+PARA_VAL_DICT['finish_mode']+';//finish_mode')

    dmaie=0;dmamask=0
    uartie=0; uartmask=0

    dmamask |= 1<<7
    if 'leen_m' in PARA_VAL_DICT['enabled_INT'][0]:
        dmaie |= 1<<7   #leen
        dmaie |= 1<<4   #i_en
        dmamask |= 1<<4
    dmamask |= 1<<6
    if 'ween_m' in PARA_VAL_DICT['enabled_INT'][0]:
        dmaie |= 1<<6   #ween   bit 6
        dmaie |= 1<<4   #i_en
        dmamask |= 1<<4

    if 'ieop_s' in PARA_VAL_DICT['enabled_INT'][0]:
        uartie |= 1<<12                                  #bit 12 is ieop
        uartmask |=1<<12
    uartmask |=1<<6
    if 'itrdy_s' in PARA_VAL_DICT['enabled_INT'][0]:
        #uartie |= 1<<6                                  #itrdy bit 6
        pass
    uartmask |=1<<5
    if 'itmt_s' in PARA_VAL_DICT['enabled_INT'][0]:
        uartie |= 1<<5                                  #itmt bit 5
    uartmask |=1<<4
    if 'itoe_s' in PARA_VAL_DICT['enabled_INT'][0]:
        uartie |= 1<<4                                  #itoe bit 4; also need enable ie bit
        uartie |= 1<<8
        uartmask |= 1<<8
    instruction_list.append('IE[TXBUF2TXDATAbyDMA]='+hex(PARA_VAL_DICT['enabled_INT'][1])+';')

    dmaie |= 1<<7   #leen
    instruction_list.append('na_DMA_1->np_dmalength='+str(PARA_VAL_DICT['length'])+';')


    if PARA_VAL_DICT['finish_mode']=='WEOP' or PARA_VAL_DICT['finish_mode']=='LENGTH_OR_WEOP':
        dmaie |= 1<<6   #ween

    dmamask |= 1<<4
    if PARA_VAL_DICT['RFclear_INT'][1]:
        dmaie |= 1<<4   #i_en bit

    instruction_list.append('temp=na_DMA_1->np_dmacontrol; //now setting up the DMA interruptions\n'+
                            'temp &='+hex(dmamask^0xffff)+'; temp |='+hex(dmaie)+'; \n'+
                            'na_DMA_1->np_dmacontrol=temp; //completed setup the DMA interruptions')
    instruction_list.append('temp=na_Uart_1->np_uartcontrol; //now setting up the UART interruptions\n'+
                            'temp &='+hex(uartmask^0xffff)+'; temp |='+hex(uartie)+'; \n'+
                            'na_Uart_1->np_uartcontrol=temp; //completed setting up the UART interruptions')

    return instruction_list

#Part 3.3 -- The invocation part of I()
#Input: concrete parameters;
```

```
158 #Output: a SEQUENCE of instructions (in form of a string).
159
160 def S_H_TXBUF2TXDATAbyDMA(PARA_VAL_DICT):
161     if PARA_VAL_DICT['RFclear_INT'][1]!=0:
162         handle='(na_Uart_1->np_uartstatus)=0;(na_DMA_1->np_dmastatus)=0;'
163         #this is because the hardware constraint:
164         #uart eop will trigger DMA reop or weop even before dma really start.
165
166         handle+='RF[TXBUF2TXDATAbyDMA]='+hex(PARA_VAL_DICT['RFclear_INT'][1])
167         handle+=';(na_Uart_1->np_uartcontrol) |= np_uartcontrol_itrdy_mask;'
168         return handle
169     else:
170         handle='(na_Uart_1->np_uartstatus)=0;(na_DMA_1->np_dmastatus)=0;' #reset the EOP signal
171         handle+='RF[TXBUF2TXDATAbyDMA]=128;(na_Uart_1->np_uartcontrol) |= np_uartcontrol_itrdy_mask;'
172
173         return handle
```

Appendix D

Software Structure Implementation

In this appendix, some codes implemented for the hybrid-mode TP are listed. Except for the omission of very lengthy automatically generated parts , the codes are ready for compilation. The four sections respectively list

- the function `main()` implemented in `hybrid.c` and `hybrid.h`.
- the Role 3 function `scheduler()` implemented in `scheduler.c` and `scheduler.h`.
- a Role 1 function `uartISR()` implemented in `ISR.c` and `ISR.h`, and
- a Role 2 function `memoryblkrev()` implemented in `soft_transfer.c`.

Our implementation does treat soft-transfers differently (especially in the `scheduler()` function). But this specialty can be generalized. See the discussion in Section 4.4.

D.1 The `main()` Function

The `main()` function (which contributes to Role 3 SW) is implemented in the file `hybrid.c`; while some supporting declaration/typedefs are in the `hybrid.h`. We list both files here with detailed comments. For the `hybrid.c` file, it would be easier to directly read `main()`'s implementation at the end of the file before referring to the functions defined earlier. The `hybrid.h` file includes some supporting macros and constants, especially the implementation of the TP-TB interface, and the pre-defined commands that control the test-bench.

The following is the content of file `hybrid.c`.

```
1
2  /*****************************************************************************************/
3  /* Auto-commenting of the generation parameters passed to the generator.*/
4  /*****************************************************************************************/
5
6  //Generated with options: "-p default_parameters -s 60" :
7  //Generating 60 scenarios, with the default randomisation strategies.
8
9  /*****************************************************************************************/
10 /* The Head Files this file depends on*/
11 /*****************************************************************************************/
12
13 #include <excalibur.h>   //excalibur.h: Symbols of Constants defined in the Nios SoC
14 #include "hybrid.h"      //hybrid.h: the head of this c file
15 #include "scheduler.h"   //scheduler.c is the main
16 #include "ISR.h"         //
17 #include "isr_wrapper.h"
18
19 /*****************************************************************************************/
20 /* A Table for TP to decide which transfer-types are soft-transfers.*/
21 /* When multiple transfers needed to be submitted; soft-transfer needs to be submitted last.*/
22 /*****************************************************************************************/
23
24 int IsSoftTransfer[TRANS_TYPE_NUM]={1,0,0,0,0,0,1,1,1,1,0,1,0,1,0};
25
26 /*****************************************************************************************/
27 /* A table for TP to initialising scenarios
28 (1) set up the current scenario running-flag;
29 (2) set the minimum transfer-repetition in the current scenario.
30 Also, transfer-instance's description is auto-commented here;
31 together with the auto-comment of environment.*/
32 /*****************************************************************************************/
33
34 Scenario_t Scenario_List[SCENARIO_NUM]={
35
36 //Scenario 0;
37 //TransType RXDATA2RXBUFbyDMA; bad_baud: 512; RFclear_INT: (['reen_m', 'leen_m'], 3); ......
38 //TransType Fibonacci; fib: 11;
39 //TransType TIMERCountingDown; count: 8; continuous: 1; period: 100;
40 //TransType TXBUF2TXDATAbyV_Master; RFclear_INT: ([], 0); length: 48; finish_mode: ......
41
42 //Using environment 0
43 {0xc24,1},
44
45 /* ........ Scenario 1 to 58 are omitted in this representation.*/
46
47 //Scenario 59;
48 //TransType M2MDMA; length: 5939; dest_Addr: 3941862; src_Addr: 223232; ......
49 //TransType RXDATA1POLLINGbyCPU; bad_baud: 512; length: 0; finish_mode: REOP; enabled_INT: ([], 0);
50 //TransType TXBUF2TXDATAbyV_Master; RFclear_INT: (['ieop_s'], 1); length: 56;......
51 //TransType TIMERCountingDown; count: 3; continuous: 1; period: 557;
52
53 //Using environment 5
54 {0x4604,1}};        //End of Scenario Initialisation Table.
55
56 /*****************************************************************************************/
```

```
57 /* A table of environment settings, used by the TP to change environment settings */
58 /* The auto-commenting for each setting is also included. */
59 /*******************************************************************************************/
60
61 Environment_t Env_List[ENV_NUM]={
62
63 //Environment 0:
64 //baud rate=115200;eop=0x83 [131];(d_caching,i_caching)=(0, 0)
65
66 {289,0x83,0, 0},
67
68 /* ........ Environment 1 to 4 are omitted in this representation.*/
69
70 //Environment 5:
71 //baud rate=225828;eop=0xe5 [229];(d_caching,i_caching)=(1, 1)
72
73 {148,0xe5,1, 1}};   //End of Environment Table.
74
75 /*******************************************************************************************/
76 /* The following is the definition (not declaration) of other globally visible variables
77 for various purposes.*/
78 /* Volatile ones circumvent cached behaviours, so could be detected by the TB. */
79 /* Firstly, The definition of critical data-structure for
80 (1) scheduling purpose;
81 (2) TP-ISR communication. */
82 /*******************************************************************************************/
83
84 int RF[TRANS_TYPE_NUM]; //The running-flag of each transfer-type. Critical.
85 int IE[TRANS_TYPE_NUM]; //The interrupt-enable flag. Implementation-dependent.
86 int FM[TRANS_TYPE_NUM]; //The flag for virtual-transfers to determine their behaviour.
87 int RepetitionCounter[TRANS_TYPE_NUM]; //The counter for each transfer in a scenario. Necessary.
88 int done_trans_FIFO[MAX_FIFO_LENG];    //The FIFO used by the scheduler. Critical.
89 int FIFO_P=0;                  //The pointer to the FIFO. Critical
90 int min_repetition;        //The minimum repetition of a transfer in a scenario. Important.
91 int ScenarioNotDone=0;     //The flag that decides re-submitting transfers in a scenario. Important.
92 volatile int ScenarioRF=0; //The (bit-mask) flag that marks if any transfer is running. Critical.
93 int CurrentScenario=0-1;   //The current scenario number.
94 int EnvironmentCounter=0;  //The current environment setting.
95
96 /*******************************************************************************************/
97 /* Then the start-point of the general-purpose memory-range in the data-memory (SRAM of the SoC).*/
98 /* This GP-Buffer size is 512KB -- the 2nd half of the 1MB SRAM, not exactly, because --
99 -- the interrupt vector table (256B) is at the tail of the SRAM. */
100 /* This GP-Buffer is volatile since the DMA-engine could dump raw data in it.*/
101 /*******************************************************************************************/
102
103 volatile void* GP_BUFFER = (void*) (((int)nasys_vector_table) - 512*1024);
104
105 /* End of Definition of All Global Variables. */
106
107 /*******************************************************************************************/
108 /* The Polling Mechanism. It polls the running-flag of the current SCENARIO (not transfer's RF).
109 When the scenarioRF is False, we attempt to update environment, and initialise the next scenario.
110 It is not really an idling operation but a polling operation. The name of "Idle_CPU" is a legacy
111 of the pure event-driven TP; in which Idle_CPU is implemented as "while 1;".*/
112 /* It is PrepareNextScenario() that detects the last scenario and terminates the simulation.*/
113 /*******************************************************************************************/
```

```
114
115 #define Idle_CPU {\
116   while(1){\
117     if (!ScenarioRF) {\
118       SHUT_SYS_INT;\
119       PrepareNextScenario();\
120       scheduler();\
121       OPEN_SYS_INT;\
122     }\
123   }\
124 }
125
126 /*******************************************************************************/
127 /* The Implementation of UpdateEnvironment and PrepareNextScenario.
128 We could choose to implement them as macros instead of subroutines.
129 But implementing them as subroutines makes them profilable by the TB.
130 PrepareNextScenario() detects the need to finish simulation.*/
131 /*******************************************************************************/
132
133 void UpdateEnvironment(){
134   Environment_t E = Env_List[EnvironmentCounter++];
135
136   //EOP
137   na_Uart_1->np_uartendofpacket=(int) E.eop; //Set the eop physically in the UART
138   EOP= E.eop;                                //set the eop softly for the virtual masters
139   na_Uart_1->np_uartdivisor=E.baud;          //Set the baud-rate physically in the UART
140   caching(E.d_caching, E.i_caching);         //Enabling/Disabling the D-cache/I-cache.
141 }
142
143 void PrepareNextScenario(){
144   Scenario_t S;
145   int mask;
146   int TransID=0;
147
148   CurrentScenario++;                                    //Proceed the current scenario
149
150   if (CurrentScenario==SCENARIO_NUM) FINISH_SIMULATION; //When all scenarios are done, terminate simulation.
151
152   if (CurrentScenario%CHANGE_ENV==0){                   //Update environment.
153     UpdateEnvironment();
154   }
155
156   S=Scenario_List[CurrentScenario];                     //Begin to initialise current scenario.
157
158   ScenarioNotDone =S.S_mask;    //The bit-mask that reflect which transfer-types are running
159   ScenarioRF =S.S_mask;
160   min_repetition =S.Min_Repetition;
161
162   mask = S.S_mask;
163   while (mask){                 //According to the mask, push transfers into the FIFO.
164     if (mask & (1<<TransID)){
165       RepetitionCounter[TransID]=0;
166       push_FIFO(TransID);
167       mask &=(~(1<<TransID));
168     }
169     TransID++;
170   }
```

```
171 }
172 /*******************************************************************************************/
173 /* The Implementation of installation of user-ISR. We do not use the default one
174 provided by Altera since we are going to install our faster customised ISR_wrappers.
175 It is a single in-line assembly instruction.*/
176 /*******************************************************************************************/
177
178 #define installisr(irq,prog)\
179 asm volatile (\
180     "ST [%0], %1;"\
181     :\
182     :"r"((irq<<2)+(int)nasys_vector_table),"r"(prog)\
183     :"memory")
184
185 /*******************************************************************************************/
186 /* The main() function. It sequentially perform these tasks:
187 (1). Reset data-structures (running-flags, the FIFO);
188 (2). Install ISRs;
189 (3). Prepare scenario 0;
190 (4). Invoke the scheduler();
191 (5). Poll the current scenario's running-flag and proceed to the next scenario.
192 */
193 /*******************************************************************************************/
194
195 main(){
196   //(0). Define the Working Variables.
197   int dma_control_reg=0;
198   int uart_control_reg=0;
199   int timer_control_reg=0;
200   int temp;
201
202   START_SIMULATION;//Give TB the signal to start logging. Macro defined in hybrid.h.
203
204   //(1). Reset data-structures (running-flags, the FIFO);
205   for (temp=0;temp<TRANS_TYPE_NUM;temp++){
206     RF[temp]=0;
207     IE[temp]=0;
208     FM[temp]=0;
209     RepetitionCounter[temp]=0;
210   }
211   for (temp=0;temp<MAX_FIFO_LENG;temp++)
212     done_trans_FIFO[temp]=-1;
213
214   //(2). Install ISRs;
215   SHUT_SYS_INT;
216   SHUT_UART1_INT;
217   SHUT_DMA1_INT;
218   SHUT_TIMER_INT;
219   installisr(na_Uart_1_irq,(nios_isrhandlerproc) uartISR_wrapper);
220   installisr(na_DMA_1_irq,(nios_isrhandlerproc)dmaISR_wrapper);
221   installisr(na_Timer_irq,(nios_isrhandlerproc)TimerISR_wrapper);
222
223   //(3). Prepare scenario 0; (4). Invoke the scheduler(); (5). Polling for next scenario.
224   PrepareNextScenario();
225   scheduler();
226   OPEN_SYS_INT; //Now main() is ready for polling. Make sure interrupt can happen.
227   Idle_CPU;     //Not really idling. Polling and proceeds to a new scenario.
```

```
228 }
229
230 //End of File hybrid.c
```

The following is the content of file `hybrid.h`.

```
1
2  /* hybrid.h: Most of hybrid.h content is fixed.*/
3  //Only two constants are inserted by the Test Generator:
4
5  #define TRANS_TYPE_NUM 15
6  #define SCENARIO_NUM 60
7
8  //Typedefs for TB-control data-structure.
9
10 //Firstly, re-interpret a 32-bit word in three alternative ways;
11 typedef struct{  //One word = one integer
12     int i;
13 }t_int_word;
14
15 typedef struct{  //One word = 2 short integers
16     short h0;
17     short h1;
18 } t_short_word;
19
20 typedef struct{  //One word = 4 bytes
21     char b0;
22     char b1;
23     char b2;
24     char b3;
25 } t_char_word;
26
27 //Secondly, put these interpretations in one union to represent a flexible 32-bit word.
28 typedef union{  //One flexible word = one integer, two short-integers OR, four bytes.
29     t_int_word int_para;
30     t_short_word short_para;
31     t_char_word char_para;
32 } flex_word;
33
34
35 //Thirdly, typedef the TB-control structure: 1 command (int) and 7 flexible-words;
36 typedef struct
37 { int command;
38   flex_word p1;
39   flex_word p2;
40   flex_word p3;
41   flex_word p4;
42   flex_word p5;
43   flex_word p6;
44   flex_word p7;
45 } tb_control;
46
47 // Finally, locate the TB-control structure at the first 8 word of the on-chip RAM
48 #define tb_control_p (volatile tb_control *)(na_on_chip_RAM)
49
50 //Now, we define constants/macros that use the above TB-control structure.
```

```
51 //The Following negative numbers are command-code for the TB to understand:
52 #define SIMULATION_FINISH (-1)
53 #define DEBUG_INFO (-2)
54 #define RX_STIMULATION_SETUP (-3)
55 #define RX_STIMULATION_START (-4)
56 #define RX_STIMULATION_STOP (-5)
57 #define SIMULATION_STOP (-6)
58 #define SIMULATION_START (-7)
59
60 //The next macro is defined for the TB to execute a specific command;
61 //but it is not recommended to use it directly by the TP.
62 #define EXECUTE(ins) {tb_control_p->command=0;tb_control_p->command=(ins);}
63
64 //The following user-level macros are used for TB-control by TP.
65 //They use the above EXECUTE(ins) macro.
66 #define FINISH_SIMULATION EXECUTE(SIMULATION_FINISH)
67 #define DISPLAY_DEBUG_INFO(d,nm) (tb_control_p->p1).int_para.i=(d);(tb_control_p->p2).char_para.b0=(nm);\
68 EXECUTE(DEBUG_INFO)
69 #define SETUP_RX_STIMULATION(BB,L,EM) (tb_control_p->p1).short_para.h0=(BB);(tb_control_p->p1).short_para.h1=(L);\
70 (tb_control_p->p2).char_para.b0=(EM);EXECUTE(RX_STIMULATION_SETUP)
71 #define START_RX_STIMULATION EXECUTE(RX_STIMULATION_START)
72 #define STOP_RX_STIMULATION EXECUTE(RX_STIMULATION_STOP)
73 #define STOP_SIMULATION EXECUTE(SIMULATION_STOP)
74 #define START_SIMULATION EXECUTE(SIMULATION_START)
75
76 //Typdef the content of an environment entry;
77 typedef struct{
78   int baud;
79   unsigned char eop;
80   char d_caching;
81   char i_caching;
82 }Environment_t;
83
84 //Typedef the content of a FIFO entry: a bit-mask and a minimum transfer iteration.
85 //Typedef  the content of a scenario entry;
86 typedef struct{
87   int S_mask;
88   int Min_Repetition;
89 }Scenario_t;
90
91 //FIFO-related macros and constants.
92 #define MAX_FIFO_LENG 32    //be large enough....
93 #define FIFO_NOT_EMPTY (FIFO_P)
94 #define pop_FIFO (FIFO_NOT_EMPTY?done_trans_FIFO[--FIFO_P]:-1)
95 #define push_FIFO(tid) if(FIFO_P<MAX_FIFO_LENG)done_trans_FIFO[FIFO_P++]=(tid)
96
97 //Miscellaneous but critical system-level macros for ISR/scheduler/main to use.
98 #define SHUT_SYS_INT asm("PFX 8");asm("WRCTL %g0")  //disable system-wide interrupt.
99 #define OPEN_SYS_INT asm("PFX 9");asm("WRCTL %g0")  //enable system-wide interrupt
100 #define IDLE_CPU while(1)   //The real CPU-Idling operation. Not used in the hybrid mode.
101
102 //Now publish (declare) all globally accessible symbols defined in hybrid.c
103 extern int RF[TRANS_TYPE_NUM];                //For the scheduler and ISRs
104 extern int IE[TRANS_TYPE_NUM];                //For ISRs
105 extern int FM[TRANS_TYPE_NUM];                //For ISRs
106 extern int IsSoftTransfer[TRANS_TYPE_NUM];    //For the scheduler
107 extern int RepetitionCounter[TRANS_TYPE_NUM]; //For the scheduler
```

```
108 extern int done_trans_FIFO[MAX_FIFO_LENG];    //For the scheduler and ISRs
109 extern int FIFO_P;                           //For the scheduler and ISRs
110 extern int CurrentScenario;                  //For the scheduler and ISRs
111 extern int ScenarioNotDone;                  //For the scheduler
112 extern volatile int ScenarioRF;              //For the scheduler
113 extern volatile void* GP_BUFFER;             //For the scheduler
114 extern int min_repetition;                   //For the scheduler
115 extern Scenario_t Scenario_List[SCENARIO_NUM];//For the scheduler and ISRs
```

D.2 `scheduler()` – the "Test-Program"

The `scheduler()` function is the main body of the "TP". Since it is a hybrid-mode TP, it simply (re-)submitted the transfers specified by the transfer-IDs, until all transfers in the current scenario have completed at least `Min_Repetition` times. This is a simple scheduling scheme, very much like a juggler who throws, receives and re-throws several balls, each ball with different travel-duration in the air. This part of code is fixed and account for only dozens of lines of C code.

The majority of `scheduler()` is single, but probably very long, `switch` statement which we call the "action-table". Essentially, it switches against a transfer-ID and executes the matching entry. The action-table is actually the main output of the test-generator.

For efficiency reasons, we make the following arrangements:

- the action-table is a normal `switch` statement directly embedded in the `scheduler()` rather than a subroutine.
- the transfer-instance-ID is in fact a pair of (a) the scenario-ID and (b) the transfer-type-ID. We assume that transfer-instances of the same transfer-type use the same master resource, so a scenario cannot contain two instances of the same type; it follows that a transfer-type-ID is enough to identify a transfer-instance in a given scenario.
- Accordingly, the action-table has two tiers of `switch` statements.
  - The outer tier is a single `switch`, switching against the scenario-ID, and
  - its entry is a second tier of `switch`, switching against the transfer-type-ID; and
  - the entry of the second tier is the transfer's configuration/invocation sequence.

Because the action-table is a single `switch` statement, `scheduler()` returns very fast although its size could be quite big.

Before we list `scheduler()`'s implementation code, we describe how `scheduler()` interaction with other software components. `scheduler()` has three types of entrance-points, namely,

- being called from `main()`;
- being called from an ISR which notifies a transfer-finish event; and
- returning from a finished soft-transfer (i.e., the notification of soft-transfers).

In any case, `scheduler()` receives transfer-IDs from the FIFO (`done_trans_FIFO`).

`Scheduler()` has three types of exit-points:

- returning to `main()` with transfer(s) being submitted;
- returning to the caller ISR either (a) with transfer(s) being re-submitted, or (b) with re-submission cancelled (to run-down a scenario).
- invoking (i.e., calling) a soft-transfer.

In any case, `scheduler()` guarantees the FIFO is cleared before it loses control.

The following is the code of one `scheduler()` with 60 scenarios; but only one entry is shown in the action-table.

```
1  //Generated with options: -p default_parameters -s 60
2
3  #include <excalibur.h>
4  #include "hybrid.h"
5  #include "CPUTransTypeImple.h"
6  #include "ISR.h"
7  #include "scheduler.h"
8
9  //Remember: the caller of scheduler must have already disabled system-wide interrupts (for atomic FIFO operation).
10 //The scheduler is responsible to re-enable interrupts when appropriate.
11
12 void scheduler(){    //scheduler receives its input from the FIFO.
13
14     /*Local working variables*/
15     register int temp; //The general-purpose working variable. Typically used in read-modify-update operations.
16     int TransType;     //The current Transfer Type; combined with the global CurrentScenario, forms the Transfer-ID.
17     int RestartTransfers[TRANS_TYPE_NUM]; //Used for remembering which transfers need to be (re-)submitted.
18     int RestartTransferCount=0;           // And its pointer.
19     int tid;       //A working variable. Not important.
20
21
22     do{
23         //This top-level do-while loop guarantees that when scheduler exits, the FIFO is empty.
```

```
24          //This loop executes only once unless a soft-transfer is (re-)submitted.
25          //This is because submitting a soft-transfer is simple CALL to that soft-transfer.
26          //When this happens, we regard that the scheduler has finished its current running.
27          //But since it the submission is just a normal CALL, when
28          //the soft-transfer returns, it leaves its ID in the FIFO, but the control now
29          //returns to scheduler. That is why this do-while loop exists: it emulates that
30          //the scheduler is triggered by the soft-transfer that just completed.
31
32          RestartTransferCount=0;    //Resetting how many transfers to submit.
33          while (FIFO_NOT_EMPTY){
34
35              // This while loop move entries from the FIFO to a list for (re-)submission.
36              // For most cases, there is only one entry in the FIFO.
37              // The exception includes the beginning of a new scenario.
38
39              TransType = pop_FIFO;           //Pop the FIFO,  get the Transfer-type-ID.
40
41              RepetitionCounter[TransType]++; // Repetition-counting.
42
43              if (RepetitionCounter[TransType]<=MAX_REPETITION) // MAX_REPITITION prevents too many re-submissions.
44                  RestartTransfers[RestartTransferCount++]=TransType; // The popped transfers recorded.
45              else ScenarioRF&=~(1<<TransType); // This finished transfer won't be re-submitted.
46
47
48              if (RepetitionCounter[TransType]>min_repetition)  // min_repetition prevents too few re-submission.
49                  ScenarioNotDone&=(~(1<<TransType));           // Flag that this finished transfer is "done", i.e.,
50                                                                  // it is nice but not required to re-submit
51          } //End of popping all FIFO entries (the while loop).
52
53          /* At this stage, all FIFO is popped. */
54
55          /* Now judge if re-submission is needed. If not, exit scheduler.*/
56          /* The still running transfers will trigger scheduler at their notification time. */
57          /* When that happens, the scheduler also exits here.*/
58          /* It is the main() function that detects ScenarioRF and proceed to the next scenario.*/
59
60          if (!ScenarioNotDone) {                             //If ALL transfers in the scenario have been "done",
61              for (temp=0;temp<RestartTransferCount;temp++)   //the FINISHED transfers will NOT be re-submitted;
62                  ScenarioRF&=(~(1<<RestartTransfers[temp])); //so they are flagged as NOT physically running;
63              OPEN_SYS_INT;                                   //Re-enable system-wide interrupts before --
64              return;                                         //-- exiting scheduler with re-submission cancelled.
65          }
66
67          /* At this stage, the scheduler decides to proceed to re-submission.*/
68
69          /* The following for-loop moves a soft-transfer to the end of the re-submission list, if it exists.*/
70          /* For most of time, this loop is skipped altogether since only one transfer need be re-submitted.*/
71          /* The exception is the beginning of a new scenario.*/
72
73          for (tid=0;tid<RestartTransferCount-1;tid++){
74              if (IsSoftTransfer[RestartTransfers[tid]]) {
75                  int tid2;
76                  int SoftTransferID=RestartTransfers[tid];
77                  for (tid2=tid+1;tid2<RestartTransferCount;tid2++)
78                      RestartTransfers[tid2-1]=RestartTransfers[tid2];
79                  RestartTransfers[RestartTransferCount-1]=SoftTransferID;
80                  break;
```

```
81              }
82          }
83
84          /* At this stage, it is safe to (re-)submit transfers */
85          /* The following for-loop does the (re-)submission. Again, for most cases, this loop execute only once.*/
86
87          for (tid=0;tid<RestartTransferCount;tid++){ //Proceed to (re-)submit transfers!
88
89              /* The transfer-instance-ID is made of CurrentScenario and TransType */
90
91              TransType=RestartTransfers[tid];
92
93              /* !!! Here is the ENTRY POINT of the ACTION-TABLE !!! */
94              /* All comments in the action-table are automatically scripted.*/
95
96              switch (CurrentScenario) {  //The 1st-tier switch.
97
98                  case 0:                 //Scenario 0
99
100                     switch (TransType) {//The 2nd tier switch. TransType was popped from the FIFO
101                         case RXDATA2RXBUFbyDMA:
102
103                             //Here Are the Setup Configurations -- non-parametric:
104                             temp=na_DMA_1->np_dmacontrol;  //now setting DMA rcon/wcon=1/0; b/bw/w=1;
105                             temp&=~(np_dmacontrol_rcon_mask+np_dmacontrol_wcon_mask+
106                                 np_dmacontrol_byte_mask+np_dmacontrol_hw_mask+np_dmacontrol_word_mask);
107                             temp|=(np_dmacontrol_rcon_mask+np_dmacontrol_byte_mask);
108                             na_DMA_1->np_dmacontrol=temp;  //completed setting DMA rcon/wcon=1/0; b/hw/w=1;
109                             na_DMA_1->np_dmareadaddress=(int) &(na_Uart_1->np_uartrxdata);//DMA Read_addr=0x910800;
110                             na_DMA_1->np_dmawriteaddress=(int) rxbuf; //set DMA Write_addr=int(char*(rxbuf));
111
112                             //Here Are the Configurations -- parametric:
113                             FM[RXDATA2RXBUFbyDMA]=REOP; //finish_mode
114                             IE[RXDATA2RXBUFbyDMA]=0xf7;
115                             na_DMA_1->np_dmalength=RXBUF_SIZE; //set DMA leng_Reg=RXBUF_SIZE;
116                             temp=na_DMA_1->np_dmacontrol; //now setting up the DMA interruptions
117                             temp &=0xff4f; temp |=0xb0;
118                             na_DMA_1->np_dmacontrol=temp; //completed setup the DMA interruptions
119                             temp=na_Uart_1->np_uartcontrol; //now setting up the UART interruptions
120                             temp &=0xee70; temp |=0x110f;
121                             na_Uart_1->np_uartcontrol=temp; //completed setting up the UART interruptions
122                             SETUP_RX_STIMULATION(512,RXBUF_SIZE,REOP);
123
124                             //Now start this transfers instance;
125                             (na_DMA_1->np_dmastatus)=0;RF[RXDATA2RXBUFbyDMA]=0x3;
126                             (na_DMA_1->np_dmacontrol) |= np_dmacontrol_go_mask; START_RX_STIMULATION;
127
128                             break; //end of case: Instance ID=(0,RXDATA2RXBUFbyDMA);
129
130                         case Fibonacci://Transfer_Type Case
131
132                             //Here Are the Setup Configurations:
133                             fib_result=0;
134                             fib_level=0;
135
136                             //Here Are the Parameterisations:
137                             expected_fibonacci_result=144;
```

```
138
139                         //Now start this transfers instance;
140                         //since it is a soft-transfer, we are leaving the scheduler!!!
141                         RF[Fibonacci]=1;
142                         OPEN_SYS_INT;
143                         RF[Fibonacci]=fibonacci(11);
144                         // Now we are back to scheduler from the soft-transfer!!!
145                         //Since an Soft-Transfer is finished:
146                         SHUT_SYS_INT; //Need to re-shut the interrupt from here;
147                         push_FIFO(Fibonacci); //Need to feed the scheduler;
148
149                         break; //end of case: Instance ID=(0,Fibonacci);
150
151                     case TIMERCountingDown:     //Transfer_Type Case
152
153                         //Here Are the Setup Configurations: Empty
154
155                         //Here Are the Parameterisations:
156                         na_Timer->np_timerperiodh=0x0; //setting the periodh register
157                         na_Timer->np_timerperiodl=0xd04; //setting the periodl register
158                         Timer_Counter=8; //setting how the count down of the timer ISR
159
160                         //Now start this transfers instance;
161                         RF[TIMERCountingDown]=1;na_Timer->np_timercontrol =0x7 ;//#bit 0-3
162
163                         break; //end of case: Instance ID=(0,TIMERCountingDown);
164
165                     case TXBUF2TXDATAbyV_Master:    //Transfer_Type Case
166
167                         //Here Are the Setup Configurations:
168                         last_txdata=~EOP; tx_counter=0;   //clearing the virtual master
169
170                         //Here Are the Parameterisations:
171                         FM[TXBUF2TXDATAbyV_Master]=LENGTH;
172                         IE[TXBUF2TXDATAbyV_Master]=0x4; //for ISR
173                         txLength=48; //for ISR
174                         temp=na_Uart_1->np_uartcontrol; //now setting up the UART interruptions
175                         temp &=0xff8f; temp |=0x20;
176                         na_Uart_1->np_uartcontrol=temp;    //completed setting up the UART interruptions
177
178                         //Now start this transfers instance;
179                         RF[TXBUF2TXDATAbyV_Master]=128;
180                         (na_Uart_1->np_uartcontrol) |= np_uartcontrol_itrdy_mask; //UART itrdy=1;
181
182                         break; //end of case: Instance ID=(0,TXBUF2TXDATAbyV_Master);
183
184                 }  //end of tier-2 switching (on transfer-types);
185
186                 break;  //end of Scenario ID=0;
187
188             /* .......Scenario 1 to 59 are omitted........*/
189
190         }  //end of tier-1 switching (on scenarios);
191
192         /* !!! END of the ACTION-TABLE!!! */
193
194     } // end of the for-loop; each loop is for a (re-)submission.
```

```
195
196     } while (FIFO_NOT_EMPTY);  //End of the do-while loop.
197
198     /* Repair to exit the scheduler -- re-enable the system-wide interrupts before exit. */
199     OPEN_SYS_INT;
200 }   //Normal end of scheduler -- (re-)submission is performed.
```

D.3 uartISR() – an interrupt-service-routine

ISRs are critical codes that provide service to hardware interrupts. The UART in our Nios SoC is the most interrupt-intensive device, with both data-intensive sources and control-intensive ones. Therefore we choose to list `uartISR()`'s implementation code, in order to see how to implement the idea of "general ISR structure" described in Section 4.3.3. The `uartISR()` is a function that deals with all interrupt sources from the UART, so it actually encapsulates multiple "ISRs". They perform system-dependent behaviours. We do not suggest readers to grasp every lines of the function. The important thing is the overall structure of the function and the flag/FIFO operations which contributes to the execution of the entire TP.

```
1  #include <excalibur.h>
2  #include "ISR.h"
3  #include "hybrid.h"
4  #include "scheduler.h"
5  #include "isr_wrapper.h"
6
7  void uartISR(){
8
9      /* Overview: this function handles all 9 possible UART interrupt reasons.
10     Reason  Description          Modelling
11     ------------------------------------------------------------------------------
12     rrdy:   receiver-ready:      virtual-master/general transfer-event
13     trdy:   transmitter-ready:   virtual-master/general transfer-event
14     pe:     parity error:        transfer-abortion-event (error in RX-streams)
15     fe:     frame error:         transfer-abortion-event (error in RX-streams)
16     brk:    RX-strem break:      transfer-abortion-event (error in RX-streams)
17     roe:    RX overrun:          transfer-abortion-event (error in RX-streams)
18     toe:    TX overrun:          transfer-abortion-event (error in TX-streams)
19     tmt:    TX empty:            transfer-event (warning in TX-streams)
20     eop:    end-of-packed:       transfer-event (in RX or TX stream, Problematic)
21
22     The mapping from a HW-interrupt to an transfer-event is done by checking the running flags.
23     Matched code can be regarded as transfers' extended behaviour, or, transfers are
24     equipped with the "calling capabilities".
25     Only the transfer-finish or transfer-abortion events currently use the capabilities
26     (to reactivate the scheduler). General events can also trigger the logging behaviour,
27     which, for efficiency reason, is not implemented in the code.
28
29     There are some treatments out of efficiency considerations.
```

```
30
31      A. It is rare (in normal situation) that the uartISR is triggered for more than
32      one reason, so after servicing one interrupt reason, we attempt to exit uartISR.
33
34      B. ISRs behave themselves on UART registers. We minimise read/write operations
35      on physical registers using such a technique (with reasonable exceptions):
36
37          1. copy registers to variables at uartISR entrance;
38          2. make modifications to the copies;
39        3. update the physical registers with the modified copy BEFORE uartISR hand over the control.
40
41      Soft flags used by ISRs:
42      A. RF: transfers' running-flag;
43      B. IE and FM: Other transfer variables controlling ISRs' behaviours.*/
44
45      register int uart_control_reg=(na_Uart_1->np_uartcontrol); //Copy the control-register
46      int uart_status_reg=(na_Uart_1->np_uartstatus);            //Copy the status-register
47      register int status_vs_control = (uart_control_reg & uart_status_reg); //A handy working variable.
48
49      /*The rrdy ISR*/
50      if (status_vs_control & np_uartstatus_rrdy_mask){
51
52      /*Mapping: Checking the running flags*/
53
54          if (RF[RXDATA2RXBUFbyV_Master]>0){ //Mapping: is it an RX-by-Virtual_master transfer?
55            //Yes. Virtual-master behaviour: read from rxdata and send to rxbuf, leave a copy for the EOP issue.
56
57              last_rxdata= (unsigned char) na_Uart_1->np_uartrxdata;  //Read to the copy first;
58              rxbuf[rx_counter++]=last_rxdata;                        //Send the copy to receiving buffer.
59
60              //Virtual-master behaviour: detect the finish event of the RX stream and perform notification.
61              if ((rx_counter>=RXBUF_SIZE)||
62                 ((rx_counter == rxLength) && (FM[RXDATA2RXBUFbyV_Master]&1))||
63                   ((RF[RXDATA2RXBUFbyV_Master]==128)&&(last_rxdata==EOP) && (FM[RXDATA2RXBUFbyV_Master]&2))){
64                     //Begin notification!
65                   int rx_err=(na_Uart_1->np_uartstatus) & UART_RX_ERR_MASKS; //Error-awareness
66                   uart_control_reg&=(~UART_RX_INT_MASKS); //Operation on the fake control register
67                   RF[RXDATA2RXBUFbyV_Master]=-rx_err;     //Running-flag operation!
68                   if (rx_err) STOP_RX_STIMULATION;        //Error-awareness.
69                   IE[RXDATA2RXBUFbyV_Master]=0;
70                   SHUT_SYS_INT;
71                   push_FIFO(RXDATA2RXBUFbyV_Master);      //FIFO operation!
72              }
73          } //End of mapping to a transfer.
74
75          else if (RF[RXDATA2RXBUFbyDMA]>0){ //Mapping: is it an RX-by-DMA transfer?
76
77              if (na_DMA_1->np_dmastatus & np_dmastatus_done_mask)
78              uart_control_reg&=(~np_uartcontrol_irrdy_mask);
79          } //End of mapping to a transfer.
80
81          else; //End of Mapping: no other transfer could possibly cause rrdy!
82
83          /* Now rrdy ISR is done; attempt to quit uartISR: is rrdy the only reason for this interrupt? */
84          if (!(status_vs_control &= (~np_uartstatus_rrdy_mask))){
85          //Yes. So we begin the exit sequence.
86              UPDATE_UART1_INT;         //Update the physical register.
```

```
87              SHUT_SYS_INT;
88              if (FIFO_NOT_EMPTY) scheduler(); //Re-activate scheduler() if there is something in FIFO.
89              OPEN_SYS_INT;
90              return;                   //The quick exit of uartISR
91          }
92      } //End of rrdy ISR.
93
94
95      /* The trdy ISR */
96      if (status_vs_control & np_uartstatus_trdy_mask){   /
97          //Begin Mapping.
98          if (RF[TXBUF2TXDATAbyV_Master]>0) { //Mapping: is it a TX-by-Virtual_Master transfer?
99          //Mapped. It is TX-by-ISR. Starting Virtual-master's behaviours.
100         //The uart-transmitter could be empty (tmt) as well as ready (trdy). In that case,
101         //we should be able to send 2 bytes instead of 1, saving one interrupt without risk of overrun.
102
103             int repetition=(uart_status_reg & np_uartstatus_tmt_mask)?2:1;//Initialise a short loop.
104             while (repetition){
105                 //Begin sending a byte to the txdata (transmitter).
106                 //Considering the EOP issue, make a copy of the sent byte.
107                 last_txdata=txbuf[tx_counter++];            //Make a copy of the sent byte first;
108                 na_Uart_1->np_uarttxdata=(int) last_txdata; //Now send the copy physically.
109
110                 //Now, a shortcut to reset eop status set; do not wait for the eop ISR.
111                 if ((last_txdata==EOP) && (uart_control_reg & np_uartcontrol_ieop_mask))
112                 status_vs_control |= np_uartstatus_eop_mask;
113
114                 //Virtual master checking transfer-finish event.
115                 if (((RF[TXBUF2TXDATAbyV_Master]==128)&&(last_txdata==EOP) && (FM[TXBUF2TXDATAbyV_Master]&2))||
116                   (tx_counter>=TXBUF_SIZE)||((tx_counter == txLength) && (FM[TXBUF2TXDATAbyV_Master]&1))){
117                     //It IS an TX-finish event! Do as many preparation before lose control.
118
119                     uart_control_reg&=(~UART_TX_INT_MASKS); //Disable trdy! Otherwise the CPU will be choked!
120                     RF[TXBUF2TXDATAbyV_Master]=0;           //Clear running-flag!
121                     IE[TXBUF2TXDATAbyV_Master]=0;
122                     SHUT_SYS_INT;
123                     push_FIFO(TXBUF2TXDATAbyV_Master);     //FIFO operation!
124                     break;
125                 }//end of TX-finish judgment
126                 repetition--;
127             } //end of the short while loop.
128         } //end of one mapped entry.
129
130         else if (RF[TXBUF2TXDATAbyDMA]>0) {  //Mapping: the TX-by-DMA transfer.
131
132             //Disable trdy itself (since it is an TX-by-DMA), unless this trdy is a general transfer-event.
133             //This will cause the Transient DMA interrupt problem. Let dmaISR() handle that.
134             if (!(IE[TXBUF2TXDATAbyDMA]&8)) uart_control_reg&=(~np_uartcontrol_itrdy_mask);
135
136             //What the scheduler has done to invoke the TX-by-DMA is only enabling the trdy. Therefor,
137             //here is the REAL invocation: let the first trdy INT to physically invoke the RX-by-DMA.
138             //set DMA to 'go'; (Configuration is already done in the scheduler.)
139             if (!(na_DMA_1->np_dmacontrol & np_dmacontrol_go_mask))
140                 na_DMA_1->np_dmacontrol|=np_dmacontrol_go_mask;
141
142             else if (!(na_DMA_1->np_dmastatus & np_dmastatus_busy_mask))
143                 uart_control_reg&=(~np_uartcontrol_itrdy_mask);
```

```
144             //this 'else if' works around such a failure: t(r)rdy is enabled during DMA transfer;
145             //however, after DMA is done, DMA interrupt cannot take control to disable UART interrupt,
146             //since UART has higher interrupt priority than DMA.
147         }
148 
149         else; //Mapping end.
150 
151     //trdy ISR done. The quick exit of uartISR.
152 
153         if (!(status_vs_control &= (~np_uartstatus_trdy_mask))){
154             UPDATE_UART1_INT;
155             SHUT_SYS_INT;
156             if (FIFO_NOT_EMPTY){scheduler();}
157             OPEN_SYS_INT;
158             return;
159         }
160     }
161 
162     /* The eop ISR */
163     if (status_vs_control & np_uartstatus_eop_mask){
164 
165         status_vs_control |= (uart_control_reg & UART_ERR_MASKS & (na_Uart_1->np_uartstatus));
166 
167         //The following is a write to the REAL status-register (not the copy) of UART.
168         //This is a requirement of the nature of the interrupt: the eop bit need to be reset.
169         na_Uart_1->np_uartstatus=0;
170 
171         //Begin Mapping. But use independent "if"s instead of "else if".
172         //It is possible to have an RX-eop AND a TX-eop simultaneously
173         //But compare with the COPY of rxdata/txdata instead of the real rxdata/txdata.
174         //And perform some finishing jobs, since eop means the end of a transfer.
175 
176         if (RF[TXBUF2TXDATAbyV_Master]==1) { //1: Expect the eop ISR to finish the TX transfer.
177             if ((last_txdata==EOP)||(tx_counter>=TXBUF_SIZE)){
178                 uart_control_reg&=(~UART_TX_INT_MASKS);
179                 RF[TXBUF2TXDATAbyV_Master]=0;   //Running-flag operation.
180                 IE[TXBUF2TXDATAbyV_Master]=0;
181                 SHUT_SYS_INT;
182                 push_FIFO(TXBUF2TXDATAbyV_Master);//FIFO Operation
183             }
184         }
185 
186         if (RF[RXDATA2RXBUFbyV_Master]==1){ //1: Expect the eop ISR to finish the RX transfer.
187             if ((last_rxdata==EOP)||(rx_counter>=RXBUF_SIZE)) {
188             uart_control_reg&=(~UART_RX_INT_MASKS);
189             RF[RXDATA2RXBUFbyV_Master]=0;   //Running-flag operation.
190             IE[RXDATA2RXBUFbyV_Master]=0;
191             SHUT_SYS_INT;
192             push_FIFO(RXDATA2RXBUFbyV_Master);//FIFO operation
193             }
194         }
195 
196         //The quick exit of uartISR
197 
198         if (!(status_vs_control &= (~np_uartstatus_eop_mask))){
199         UPDATE_UART1_INT; //Update the REAL register
200         SHUT_SYS_INT;
```

```
201         if (FIFO_NOT_EMPTY){ scheduler();}
202         OPEN_SYS_INT;
203         return;
204         }
205     }//End of eop ISR.
206
207     /* ISRs to handle RX/TX errors */
208     if (status_vs_control & np_uartstatus_e_mask){
209
210         //Firstly, checking RX errors.
211         int rx_err = status_vs_control & UART_RX_ERR_MASKS;
212         if (rx_err){ // Error awareness code.
213              int tid=TerminateRunningRX(-rx_err); //Function TerminateRunningRX will:
214              //1) stop the RX external stimulation;
215              //2) find out which RX transfer-type is running (the mapping)
216              //3) reset UART and reset DMA if the transfer is rx-by-DMA;
217              //4) reset the running flag RF, etc;
218              //5) return the ID of the mapped transfer-type as tid.
219
220              na_Uart_1->np_uartstatus=0; //Reset the REAL uart status reg err bits;
221              uart_control_reg&=(~rx_err);
222              last_rxdata= (unsigned char) na_Uart_1->np_uartrxdata;//Read-off the garbage data in rxdata
223
224              if (TidIsValid(tid)){
225                  SHUT_SYS_INT;
226                  push_FIFO(tid);            //FIFO. Aborted transfer is also a finished transfer.
227                  OPEN_SYS_INT;
228              }
229
230             //The quick exit of uart ISR.
231             if (!(status_vs_control &= (~UART_RX_ERR_MASKS))){
232                 UPDATE_UART1_INT;
233                 SHUT_SYS_INT;
234                 if (FIFO_NOT_EMPTY) { scheduler();}
235                 OPEN_SYS_INT;
236                 return;
237             }
238
239         } // End of RX errors
240
241         //Secondly, Checking TX error. (only toe is possible)
242         if (status_vs_control & np_uartstatus_toe_mask){
243             //Since the error resetting is a common task already done in treating RX errors,
244             //no HW is operation needed.
245             if (!(status_vs_control &= (~np_uartstatus_toe_mask))){
246                 UPDATE_UART1_INT;
247                 SHUT_SYS_INT;
248                 if (FIFO_NOT_EMPTY) { scheduler();}
249                 OPEN_SYS_INT;
250                 return;
251             }
252         } //End of TX error.
253     } // End of error handling
254
255     if (status_vs_control & np_uartstatus_tmt_mask){
256         // tmt (transmitter empty) is not a critical event. It happens either after
257         // the last byte is physically sent or when the bit stream outruns the byte stream.
```

```
258         // Since tmt ISR is a UART interrupt, which has higher priority of DMA interrupt,
259         // the tmt ISR should disable tmt itself when a TX-by-DMA transfer finishes.
260         // Otherwise, even if the TX-by-DMA finishes physically, DMA interrupt could not
261         // take control and reset tmt interrupt, and the CPU is choked with tmt interrupt.
262             if  ((RF[TXBUF2TXDATAbyDMA]>0) && (IE[TXBUF2TXDATAbyDMA]&16)&&
263               !(na_DMA_1->np_dmastatus & np_dmastatus_busy_mask) &&
264               (((int) na_DMA_1->np_dmareadaddress) > (int) txbuf))
265                 uart_control_reg &= ~(np_uartcontrol_itmt_mask);
266     }//End of handling tmt interrupt
267
268     //Now exit the uart ISR
269     SHUT_SYS_INT;
270     UPDATE_UART1_INT;
271     if (FIFO_NOT_EMPTY) { scheduler();}
272     OPEN_SYS_INT;
273
274 } //End of uartISR.
```

D.4 `memoryblkrevbyCPU()` – a soft-transfer

This section demonstrates a simple soft-transfer in the form of subroutine. This soft-transfer reverses the content in a memory range. The width of reversion could be 8-, 16- or 32-bit. For debugging visibility, configuration is passed to the subroutine by global variables instead of by subroutine arguments.

A soft-transfer should allow interrupt, and it should not modify the interrupt settings unless absolutely necessary. This subroutine does modify the interrupt settings but just momentarily; the reason is hardware-dependent – in the Nios SoC, the current interrupt-priority will disable interrupts of lower priorities. Therefore, the subroutine must lower its interrupt-priority to allow other interrupts. The reason why a subroutine may have a high priority is because of the event-driven nature of the test-program – the soft-transfer is invoked by the `Scheduler()` function, which in turn could be invoked by an ISR, which may have a high interrupt priority; hence the soft-transfer inherits the interrupt priority.

Also notice that soft-transfers do not have to perform operations on their running-flags and the FIFO in themselves. These operations can be automated into their invocation instructions.

```
1 /* The soft-transfer named memory-block-reversion is implemented as a subroutine.
2 It reverse the content in a specified memory block. accepting three parameters:
3 1. the width: rev_width (1=byte, 2=short, 4=integer)
4 2. the length: rev_length
5 3. the raw, starting memory address: rev_start_a
6
7 The subroutine originally accepts its parameters as function arguments. However, in order to
```

```
8  let the TB has a visibility of the parameters (for debugging purpose), function
9  arguments are replaced by volatile global variables. Being volatile, TB can observe
10 non-cached accesses to these parameters.
11
12 As a soft-transfer, the subroutine should allow interrupt for most of its execution. */
13
14 volatile char rev_width;
15 volatile int rev_length;
16 volatile int rev_start_a;
17
18 void memoryblkrevbyCPU() {
19
20     int i_priority; //used by the macro SAVE_AND_RESET_I_PRIORITY.
21
22     // The following three macros allows the subroutine to be interrupted.
23     // Due to a Nios-specific reason,
24     // enabling system-wide interrupt is not sufficient to allow interrupt.
25     // The current interrupt-priority also needs to be reset.
26     SHUT_SYS_INT;
27     SAVE_AND_RESET_I_PRIORITY;
28     OPEN_SYS_INT;
29
30     revcounter=0;       //Another global for debugging purpose.
31
32     if (rev_width==1){ //Reverse in byte
33         volatile char* sa=(volatile char*) rev_start_a; //Head and Tail
34         volatile char* ea=(volatile char*) (rev_start_a+(rev_length-1));
35         char temp;
36         while (sa<ea){      //The kernel operation of the soft-transfer.
37             temp=*sa;
38             *sa=*ea;
39             *ea=temp;
40             sa++;
41             ea--;
42             revcounter++;
43         }
44     }
45     else if (rev_width==2) { //Reverse in short-integer.
46         volatile short* sa=(volatile short*) rev_start_a; //Head and Tail
47         volatile short* ea=(volatile short*) (rev_start_a+(rev_length-1)*rev_width);
48         short temp;
49         while (sa<ea){      //The kernel operation of the soft-transfer.
50             temp=*sa;
51             *sa=*ea;
52             *ea=temp;
53             sa++;
54             ea--;
55             revcounter++;
56         }
57     }
58     else {//rev_width ==4; reverse in integer
59         volatile int* sa=(volatile int*) rev_start_a; //Head and Tail
60         volatile int* ea=(volatile int*) (rev_start_a+((rev_length-1)<<(rev_width>>1)));
61         int temp;
62         while (sa<ea){      //The kernel operation of the soft-transfer.
63             temp=*sa;
64             *sa=*ea;
```

```
65              *ea=temp;
66              sa++;
67              ea--;
68              revcounter++;
69          }
70      }
71      SHUT_SYS_INT;
72      RESTORE_I_PRIORITY;
73      OPEN_SYS_INT;
74  }
```

Bibliography

[1] Altera. *SOPC Builder PTF File Reference Manual*. Altera Inc., September 2002.

[2] Altera. *Avalon Bus Specification Reference Manual*. Altera Inc., 2.2 edition, May 2003.

[3] Altera. *Implementing Interrupt Service Routines in Nios Systems*. Altera Inc., January 2003.

[4] Altera. *Nios DMA Datasheet*. Altera Inc., January 2003.

[5] Altera Inc. *Nios Hardware Development Tutorial*, December 2004.

[6] A. M. Amory, L. A. Oliveira, and F. G. Moraes. Software-Based Test Integration in a SoC Design Flow. In *Proceedings of IEEE Latin American Test Workshop, 2003*, 2003.

[7] T. Anderson, J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale. SystemVerilog Reference Verification Methodology: ESL. EETimes Design News, http://www.eetimes.com/news/design/showArticle.jhtml?articleID=188703275, December 2006.

[8] T. Anderson, J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale. SystemVerilog Reference Verification Methodology: RTL. EETimes Design News, http://www.eetimes.com/news/design/showArticle.jhtml?articleID=187001913, May 2006.

[9] J. Andrews. Unified Verificaton of SoC Hardware and Embedded Software. Chip Design Online Magazine, http://www.chipdesignmag.com/display.php?articleId=1267, 2007.

[10] B. B, M. Bidoit, A. Finkel, F.Laroussinie, A.Petit, L. Petrucci, P. Schonoebelen, and P.Mckenzie. *Systems and Software Verifiation Model-Checking Techniques and Tools*. Springer, 2001.

[11] F. Bacchini, G. Moretti, H. Foster, J. Bergeron, M. Nakamura, S. Mehta, and L. Ducousso. Is Methodology the Highway Out of Verification Hell? In *DAC '05: Proceedings of the 42nd Annual Conference on Design Automation*, pages 521–522, New York, NY, USA, 2005.

[12] X. Bai, S. Dey, and J. Rajski. Self-test Methodology for at-Speed Test of Crosstalk in Chip Interconnects. In *DAC '00: Proceedings of the 37th Conference on Design Automation*, pages 619–624, New York, NY, USA, 2000.

[13] B. Bailey. Verification Languages and Where They Fit. in Proceedings of edaForum 2003, http://www.edacentrum.net/edaforum/downloadables/dateien/mentor-seminar.pdf, November 2003.

[14] K. Batcher and C. Papachristou. Instruction Randomization Self Test For Processor Cores. In *VTS '99: Proceedings of the 1999 17TH IEEE VLSI Test Symposium*, page 34, Washington, DC, USA, 1999.

[15] M. Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, and Y. Wolfsthal. A Study in Coverage-Driven Test Generation. In *DAC '99: Proceedings of the 36th ACM/IEEE Conference on Design Automation*, pages 970–975, New York, NY, USA, 1999.

[16] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, second edition, 2003.

[17] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer Science Business Media, Inc., 2005.

[18] G. Berry, L. Blanc, A. Bouali, and J. Dormoy. Top-level Validation of System-on-Chip in Esterel Studio. In *HLDVT '02: Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop (HLDVT'02)*, page 36, Washington, DC, USA, 2002.

[19] A. Bobrek, J. J. Pieper, J. E. Nelson, J. M. Paul, and D. E. Thomas. Modeling Shared Resource Contention Using a Hybrid Simulation/Analytical Approach. In *DATE 04: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2004.

[20] M. Boini. Complex SoC Verification Using ARM Processor . Design & Reuse Magazine Online Article, http://www.design-reuse.com/articles/15351/complex-soc-verification-using-arm-processor.html, 2007.

[21] L. M. V. Bolzani, E. E. Sanchez, and M. S. Reorda. A Software-based Methodology for the Generation of Peripheral Test Sets Based on High-Level Descriptions. In *SBCCI '07: Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design*, pages 348–353, New York, NY, USA, 2007.

[22] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[23] Cadence. Accelerated Hardware/Software Co-verification. Technical report, Cadence, 2004.

[24] A. Cheng, C.-C. Lim, Y. Sun, H. He, Z. Zhou, and T. Lei. Using Genetic Evolutionary Software Application Testing to Verify a DSP SoC. *DELTA '08: Proceedings of the Fourth IEEE International Workshop on Electronic Design, Test and Applications*, 0:20–25, 2008.

[25] A. Cheng, A. Parashkevov, and C. Lim. Verifying System-on-Chips at the Software Application Level. In *Proceedings of IFIP-WG Conference on Very Large Scale Integration System-on-Chip*, 2005.

[26] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Form. Methods Syst. Des.*, 19(1):7–34, 2001.

[27] B. Cohen. Transaction-based Verification in HDL. http://members.aol.com/vhdlcohen2/vhdl/veriflang.pdf, February 2002.

[28] S. Copty, I. Jaeger, and Y. Katz. Path-based system level stimuli generation. In *Hardware and Software Verification and Testing*, pages 1–13. Springer-Verlag, 2006.

[29] S. Copty, I. Jaeger, Y. Katz, and M. Vinov. Intelligent Interleaving of Scenarios: a Novel Approach to System Level Test Generation. In *DAC '07: Proceedings of the 44th Annual Conference on Design Automation*, pages 891–895, New York, NY, USA, 2007.

[30] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero. Evolutionary Test Program Induction for Microprocessor Design Verification. In *ATS '02: Proceedings of the 11th Asian Test Symposium*, page 368, Washington, DC, USA, 2002.

[31] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero. Fully Automatic Test Program Generation for Microprocessor Cores. In *DATE 03: Proceedings of the Conference on Design, Automation and Test in Europe*, page 11006, Washington, DC, USA, 2003.

[32] F. Corno, M. S. Reorda, G. Squillero, and M. Violante. On the Test of Micro Processor IP Cores. In *DATE 01: Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pages 209–213. Design, Automation and Test in Europe Conference and Exhibition, 2001, 2001.

[33] R. J. Devins, M. E. Kautzman, K. A. Mahler, and D. W. Milton. Method for Efficient Verification of System-on-Chip Integrated Circuit Designs Including an Embedded Processor, July 2002.

[34] K. Dikramanjian. An Introduction to the VMM Resgister Abstraction Layer. SoC Central, July 2007.

[35] R. Dubey. Elements of Verification. Chip Design Online Magazine, http://www.soccentral.com/results.asp?CatID=488&EntryID=12420, March 2005.

[36] C. V. Eijk. Sequential Equivalence Checking Based On Structural Similarities. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 19(07):814–819, July 2000.

[37] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin. X-Gen: a Random Test-Case Generator for Systems and SoCs. In *HLDVT '02: Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop (HLDVT'02)*, page 145, Washington, DC, USA, 2002.

[38] S. Ezer and S. Johnson. Smart Diagonistics for Configurable Processor Verification. In *DAC '05: Proceedings of the 39th Conference on Design Automation*, pages 789–794, 2005.

[39] F.Hunsinger, S.Francois, and A.A.Jerraya. Definition of a Systematic Method for the Generation of Software Test-Programs Allowing the Functional Verification of System On Chip(SoC). In *Proceedings of the 4th International Workshop on Microprocessor Test and Verification*, pages 11–16, 2003.

[40] L. Fournier, Y. Arbetman, and M. Levinger. Functional Verification Methodology for Microprocessors Using the Genesys Test-Program Generator. In *DATE '99: Proceedings of the Conference on Design, Automation and Test in Europe*, page 92, New York, NY, USA, 1999.

[41] R. Francard, H. Singh, V. lhommeDesages, F. Delguste, and H. Keding. Using System Level Modeling to Enhance SoC Verification Lead-Time. In *Synopsys User Group 2005*. Synopsys, Boston, 2005.

[42] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal. Coverage-Directed Test Generation Using Symbolic Techniques. In *FMCAD '96: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 143–158, London, UK, 1996. Springer-Verlag.

[43] A. Gluska, R. Fourner, R. Gewirtzman, and R. Nisser. Automatic Verification of External Interrupts. US Patent 5,592,674, January 1997.

[44] R. Goering. Ten 2008 Trends in System and Chip Design. SCD Source Online Article, http://www.scdsource.com/article.php?id=68, February 2008.

[45] R. Gupta, S. Rawat, S. Shukla, B. Bailey, D. Beece, M. Fujita, C. Pixley, J. O'Leary, and F. Somenzi. Formal Verification - Prove It or Pitch It. In *DAC '03: Proceedings of the 40th Conference on Design Automation*, pages 710–711, New York, NY, USA, 2003.

[46] T. Hills. Structured Interrupts. *SIGOPS Oper. Syst. Rev.*, 27(1):51–68, 1993.

[47] A. Hosseini, D. Mavroidis, and P. Konas. Code Generation and Analysis for the Functional Verification of Micro Processors. In *DAC '96: Proceedings of the 33rd Annual Conference on Design Automation*, pages 305–310, New York, NY, USA, 1996.

[48] IEEE. IEEE Approves SystemVerilog and Verilog Standards for Electronic Design. IEEE Standard, http://standards.ieee.org/announcements/pr_p1364-1800.html, November 2005.

[49] ITRS. International Technology Roadmap of Semiconductors 2007 Edition: Design. The International Technology Roadmap for Semiconductors, http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_Design.pdf, 2007.

[50] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. Automatic Test Program Generation for Pipelined Processors. In *ICCAD '94: Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided design*, pages 580–583, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[51] C. R. Johns, D. G. Mihal, and D. A. Pierce. Architecture for Simulation Testbench Control. US Patent 6,651,038 B1, November 2003.

[52] S. Katz, O. Grumberg, and D. Geist. "Have I written enough Properties?" - A Method of Comparison between Specification and Implementation. In *CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 280–297, London, UK, 1999.

[53] J. Kenny. Using a Processor-Driven Test Bench for Functional Verification of Embedded SoCs, 04 2006.

[54] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transaction on Computer-Aided Design*, 19(12), December 2000.

[55] Z. Kirshenbaum. Understanding the "e" Verification Language. EETimes Online Article, http://www.design-reuse.com/articles/5646/understanding-the-e-verification-language.html, May 2003.

[56] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian. Effective Software Self-Test Methodology for Processor Cores. In *DATE 02: Proceedings of Design, Automation and Test in Europe Conference and Exhibition, 2002*, pages 592–597, Paris, France, 2002. Design, Automation and Test in Europe Conference and Exhibition.

[57] N. Kranitis, G. Xenoulis, A. Paschalis, D. Gizopoulos, and Y. Zorian. Application and Analysis of RT-Level Software-Based Self-Testing for Embedded Processor Cores. *IEEE International Test Conference*, 00:431, 2003.

[58] Y.-S. Kwon, Y.-I. Kim, and C.-M. Kyung. Systematic Functional Coverage Metric Synthesis from Hierarchical Temporal Event Relation Graph. In *DAC '04: Proceedings of the 41st Annual Conference on Design Automation*, pages 45–48, New York, NY, USA, 2004.

[59] S. Y. Liao. Towards a New Standard for System-Level Design. In *CODES '00: Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, pages 2–6, New York, NY, USA, 2000.

[60] Mentor. Advanced Verification Methodology Cookbook. Mentors Inc. Document, 2005.

[61] Mentor and Synopsys. Open Verification Methodology. Mentor & Synopsys Whitepaper, 2007.

[62] D. Mills. SystemVerilog 3.1 The Hardware Description AND verificiaotn Language. SNUG 2003, http://www.sutherland-hdl.com/papers/2003-SNUG-paper_SystemVerilog.pdf, 2003.

[63] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[64] G. Mosensoson. Practical Approaches to SoC Verification. Proceedings of DATE User Forum, 2002.

[65] A. Mulper. Use Co-Simulation for the Functional Verification of RTL Implementations. Chip Design Magazine Online Article, http://www.chipdesignmag.com/display.php?articleId=69&issueId=9, March 2005.

[66] Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura. A Fast Hardware/Software Co-Verification Method for System-on-a-Chip by Using a C/C++ Simulator and FPGA Emulator with Shared Register Communication. In *DAC '04: Proceedings of the 41st Annual Conference on Design Automation*, pages 299–304, New York, NY, USA, 2004.

[67] G. Nicolescu, S. Yoo, A. Bouchhima, and A. A. Jerraya. Validation in a Component-Based Design Flow for Multicore SoCs. In *ISSS '02: Proceedings of the 15th International Symposium on System Synthesis*, pages 162–167, New York, NY, USA, 2002.

[68] C. A. Papachristou, F. Martin, and M. Nourani. Microprocessor Based Testing for Core-Based System on Chip. In *DAC '99: Proceedings of the 36th ACM/IEEE Conference on Design Automation*, pages 586–591, New York, NY, USA, 1999.

[69] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[70] P. Rashinka, P. Paterson, and L. Singh. *System-on-a-Chip Verification, Methodology and Techniques*. Kluwer Academic Publishers, 2001.

[71] J. Regehr. Thread Verification vs. Interrupt Verification. In *Proceedings of the Multithreading in Hardware and SOftware: Formal Approaches to Design and Verification (TV'06), Federation Logic Conference (FloC'06)*, pages 106–110, 2006.

[72] J. Regehr and N. Cooprider. Interrupt Verification via Thread Verification. *Electron. Notes Theor. Comput. Sci.*, 174(9):139–150, 2007.

[73] S. Roch and P. H. Starke. INA Integrated Net Analyzer Version 2.2 Manual, 1999.

[74] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, C. Grecu, and A. Ivanov. System-on-Chip: Reuse and Integration. In *Proceedings of the IEEE*, pages 1050–1069, 2006.

[75] J. Shen and J. A. Abraham. Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation. In *ITC '98: Proceedings of the 1998 IEEE International Test Conference*, pages 990–999, Washington, DC, USA, 1998.

[76] M. E. Shobaki and L. Lindh. A Hardware and Software Monitor for High-Level System-on-Chip Verification. In *ISQED '01: Proceedings of the 2nd International Symposium on Quality Electronic Design*, pages 56–61, Washington, DC, USA, 2001.

[77] S. Siewert. *Real-Time Embedded Components and Systems*, chapter 1: Introduction to Real-Time Embedded Systems. Charles River Media, Inc., 2006.

[78] K. Silver. 'Divide and Conquer' with Verification IP. EETimes Online Article, http://www.eetimes.com/news/design/columns/eda/showArticle.jhtml?articleID=26807279, August 2004.

[79] Synopsys. VCS/VCSi User Guide, V7.0. Synopsys User's Manual, 2003.

[80] Synopsys. Verification of a NET+ARM Processor with VCS and Vera. Synopsys Online Document, http://www.synopsys.com/products/success/va_netsilicon_ss.pdf, 2004.

[81] Synopsys. The Unified Verification Methodology. Synopsys Whitepaper, 2005.

[82] Synopsys. *OpenVera Language Reference Manual: Testbench*. Synopsys, 2007.

[83] Synopsys and ARM. Advanced Verification Methodology Cookbook. Synopsys & ARM Whitepaper, 2005.

[84] M. H. Tehranipour, S. M. Fakhraie, Z. Navabi, and M. R. Movahedin. A Low-Cost At-Speed BIST Architecture for Embedded Processor and SRAM Cores. *J. Electron. Test.*, 20(2):155–168, 2004.

[85] Veresity. Functional Verification Automation for IP. Verisity Inc. Whitepaper, http://www.verisity.com/resources/whitepaper/ipwhitepaper.html, 1999.

[86] L. von Bertalanffy. *General System Theory: Foundations, Development, Applications*. Harmondsworth : Penguin, 1973.

[87] W. Walker. Verification Reuse Ensures Predictable Design. EETimes Online Article, http://www.eetimes.com/isd/coverstory/OEG20020103S0050, March 2002.

[88] D. Wood. *Theory of Computation*. Harper & Row New York, 1987.

[89] J. Xu and C.-C. Lim. Exploiting Concurrency in System-on-Chip Verification. In *IEEE Asia Pacific Conference on Circuits and Systems APCCAS 2006*, pages 836–839, December 2006.

[90] J. Xu and C.-C. Lim. Modelling Heterogeneous Interactions in SoC Verification. In *IFIP International Conference on Very Large Scale Integration 2006*, pages 98–103, 2006.

[91] F.-C. Yang, W.-K. Huang, and I.-J. Huang. Automatic Verification of External Interrupt Behaviors for Microprocessor Design. In *DAC '07: Proceedings of the 44th Annual Conference on Design Automation*, pages 896–901, New York, NY, USA, 2007.

[92] ZAiq. Methodology and Code Reuse in the Verification of SoCs. ZAiq Inc.

[93] H. Zhu and X. He. A Methodology of Testing High-Level Petri Nets. *Information and Software Technology*, 44:473–489, 2002.

[94] Y. Zorian, S. Dey, and M. J. Rodgers. Test of Future System-on-Chips. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design*, pages 392–399, Piscataway, NJ, USA, 2000.