Masterarbeit

Design And Implementation Of A Register Access Verification Flow

Michael Langreiter, BSc.

Institut für Technische Informatik
Technische Universität Graz

Begutachter: Ass. Prof. Dipl.-Ing. Dr. techn. Christian Steger
Betreuer: Ass. Prof. Dipl.-Ing. Dr. techn. Christian Steger
Dipl.-Ing. Heimo Hartlieb (Infineon Technologies Austria AG)

Graz, im Juni 2013

Kurzfassung

Der Inhalt dieser Arbeit ist die Entwicklung einer Methodik zur automatischen Überprüfung von Registerzugriffen in digitalen integrierten Schaltungen. Diese kann in mehreren Implementierungsstadien verwendet werden. Das Hauptaugenmerk liegt hierbei auf der Verifikation der Schaltung vor der Produktion, wobei hier Register Transfer Ebene und Gatterebene zwei unterschiedliche Entwicklungsstufen darstellen. Ausgehend von einer Spezifikation der Register in einem Tabellenformat werden zum einen Bedingungen für eine formale Verifikation und zum anderen ein Registermodell und passende Sequenzen für die Benutzung in einer Universal Verification Methodology (UVM) Testbench erzeugt. Diese beiden Ansätze werden in Hinblick auf mehrere Metriken (Art der gefundenen Fehler, Anzahl der gefundenen Fehler, Komplexität der Methodik, Laufzeit der Analysen, Wiederverwendbarkeit etc.) analysiert und verglichen. Dies geschieht durch den Test mit mehreren Prüfimplementierungen und einer tatsächlich in der Entwicklung befindlichen Schaltung. Zusätzlich werden noch eXtensible Markup Language (XML) basierte Registerbeschreibungen für die Laborverifikation mit grafischen Benutzeroberflächen am Personal Computer (PC) beziehungsweise Visual Basic for Applications (VBA) Quellcode für die Nutzung in Routinen in automatisierten Chip-Testern erzeugt.

Abstract

The topic of this thesis is the comparison, selection and development of a methodology for automated register access verification in different implementation stages of digital integrated circuits. The main focus is on the pre-silicon verification (verification before production), where the register transfer layer implementation and the netlist (post synthesis as well as post place-and-route) are seen as the different implementation stages. Starting from a high-level register specification in a spreadsheet program the properties for a formal analysis using model checking as well as a register model for the use in a Universal Verification Methodology (UVM) testbench are generated utilizing a meta-modeling flow. The two methodologies are analyzed and compared in respect to certain metrics (e.g. number of bugs found, run time, kinds of bugs, re-usability etc.) by applying them to two test designs and ultimately to a real world design. Additionally, eXtensible Markup Language (XML) based register descriptions are generated for the use in laboratory verification with graphical user interfaces on a Personal Computer (PC) as well as Visual Basic for Applications (VBA) source code for the use in test routines in automated chip testers.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....

date (Michael Langreiter)

Danksagung

Diese Diplomarbeit wurde in den Jahren 2011-2013 am Institut für Technische Informatik an der Technischen Universität Graz sowie in der Automotive - Powertrain an Safety Abteilung bei Infineon Techologies Austria in Graz durchgeführt. Mein Dank für die Unterstützung und Betreuung bei dieser Arbeit geht an:

Infineon Technologies AG:

Betreuer: Dipl.-Ing. Heimo Hartlieb
MetaGen: Dr. Wolfgang Ecker
MetaGen: Dipl.-Ing. Michael Velten
MetaGen: Dr. Volkan Esen
SVA: Dr. Andreas Zinn

Institut für Technische Informatik:

Ass. Prof. Dipl. -Ing. Dr. techn. Christian Steger

Weiters danke ich meinen Eltern, dass sie mir dieses Studium ermöglicht haben und meiner Freundin für die Unterstützung und Aufmunterung beim Verfassen dieser Arbeit.

Graz, im Juni 2013 Michael Langreiter

Contents

List of Figures **9**

List of Tables **11**

List of Abbreviations **12**

1 Introduction **15**

1.1 Motivation . . . 15
1.2 Objectives . . . 17
1.3 Outline . . . 17

2 Theoretical Background **19**

2.1 Introduction to the Verification of Very Large Scale Integrated Circuit (VLSI) Designs . . . 19
2.1.1 Validation, Verification, Characterization, Qualification, Certification 19
Validation . . . 19
Verification . . . 19
Characterization . . . 20
Qualification . . . 20
Certification . . . 20
2.1.2 Pre-Silicon Verification of VLSI designs . . . 21
Code Coverage, Dead Code Analysis, Functional Coverage . . . 21
Formal Verification . . . 23
Functional Verification . . . 23
2.1.3 Laboratory Verification and Productive Testing of VLSI Designs . . 24
Scan Test . . . 24
Automated Test Equipment . . . 25
Other Laboratory Equipment . . . 25
2.2 MetaGen . . . 26
2.2.1 Data Models . . . 27

2.2.2 Templates . . . 30
2.3 Universal Verification Methodology . . . 33
2.3.1 Basic Verification Environment . . . 33
Environment . . . 33
Agent . . . 34
Sequence . . . 35
Test . . . 36
2.3.2 Register Model . . . 36
Bitfield . . . 37
Register . . . 38
Register Block . . . 38
2.3.3 Using the Register Model . . . 39
Access Types and Functions . . . 40
Register Sequences . . . 41
Extending Bitfield Behavior . . . 42
2.4 Formal Verification . . . 42
2.4.1 Introduction to Formal Verification . . . 43
Ideas . . . 43
BDD Solvers . . . 43
SAT Solvers . . . 44
FV in Hardware Verification . . . 46
2.4.2 Static Code Analysis . . . 46
Kripke Structure . . . 46
Temporal Logics . . . 47
2.4.3 Assertion Based Verification . . . 48
SystemVerilog Assertions . . . 49

3 State of the Art **60**
3.1 General . . . 60
3.2 Detailed Analysis . . . 61
3.2.1 Simulation Specific . . . 61
3.2.2 Formal Verification Specific . . . 63
3.2.3 Formal Verification - Simulation Combinations . . . 66
3.3 Conclusion . . . 69

4 Flow Design **73**
4.1 Current Design and Verification Flow . . . 73
4.2 Flow Extension . . . 73

4.3 UVM . . . 74
4.3.1 Automatic Generation of the Model and Tests . . . 75
Parameters Needed . . . 75
Register Model Structure . . . 76
4.4 Assertion Generation . . . 78
4.4.1 Property File . . . 80
4.4.2 Bind File . . . 80

5 Implementation **82**
5.1 UVM Implementation . . . 82
5.1.1 Starting in Excel . . . 82
5.1.2 Filling the Data Model . . . 83
5.1.3 Generation of Register Model With Mako Templates . . . 83
5.1.4 Overview of the Generated Verification Environment . . . 86
5.2 Formal Verification Implementation . . . 88
5.2.1 Starting in Excel . . . 88
5.2.2 Generating Assertion Files . . . 89
5.2.3 Usage . . . 94
5.2.4 Overview of the Generated Verification Environment . . . 95

6 Practical Results **96**
6.1 The Designs Used for Testing . . . 96
6.1.1 SPI Multiplier . . . 96
6.1.2 SimpleBus Register Pack . . . 97
6.1.3 Lithium Ion Battery Balancing IC . . . 97
6.2 Metrics for Comparison . . . 98
6.2.1 Effort and Reusability . . . 98
6.2.2 Coverage . . . 100
6.2.3 Runtime . . . 101
6.2.4 Number and Type of Bugs . . . 102
6.3 UVM Results . . . 102
6.4 Formal Results . . . 103
6.5 Comparison . . . 105
6.5.1 Comparison of Effort and Reusability . . . 106
6.5.2 Comparison of Coverage . . . 107
6.5.3 Comparison of Runtime . . . 108
6.5.4 Comparison of Number and Type of Bugs . . . 109
6.6 Conclusion . . . 110

7 Post Register Transfer Layer (RTL) Verification **112**
7.1 Post Synthesis Verification 112
7.1.1 Verifying the Synthesized Netlist 112
UVM 112
Formal Verification 113
7.1.2 Verifying the Back-Annotated Netlist 113
7.1.3 Alternative: LEC 113
7.2 Silicon Tests 114
7.2.1 VBA for Tester 114
7.2.2 XML for LabView GUI 115

8 Conclusion **116**
8.1 State 116
8.2 Current Usage 117
8.3 Outlook 117

Bibligraphy **118**

List of Figures

1.1 Moore's law for transistors in Intel® microprocessors, data is taken from the 40th anniversary of Moore's law from Intel [30] . . . 16

2.1 Meta-modeling flow for generating HW design files and documentation files 26
2.2 Simple data model for a module . . . 27
2.3 An Excel sheet with data for filling the data model . . . 28
2.4 Structure of a UVM testbench . . . 34
2.5 A UVM transaction from sequence to DUT . . . 36
2.6 Example of a UVM testbench using a register model . . . 37
2.7 Example structure of a register model . . . 39
2.8 Decision Tree for $t = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$ . . . 44
2.9 Binary Decision Diagram for $t = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$ . . . 45
2.10 Representation of a Kripke structure with $\mathcal{I}$ = s0, $|\mathcal{S}| = 4$ . . . 47
2.11 Using a bind file assertions can be added to a DUT . . . 50
2.12 An example of a typical handshake protocol, two different ways that would both be valid for the defined sequence (there are more valid possibilities), green lines show where the sequence is running, a green arrow where the sequence is successfully finished and red arrows where a sequence checking run is stopped due to a mismatch . . . 52
2.13 The scheduling algorithm for SystemVerilog (with special attention to assertions) . . . 59

4.1 A simplified overview of the currently used design flow . . . 74
4.2 A simplified overview of the new design flow using meta modeling and code generation . . . 75
4.3 Multi-view register extension . . . 77
4.4 Register model file structure . . . 78
4.5 Register access type extension . . . 79
4.6 The structure of the property files . . . 81

5.1 The registers of the SPI multiplier 83
5.2 Excel description of registers of the SPI multiplier 84
5.3 The filled data model represented in the Essence component builder 85

6.1 Block diagram and register overview of the SPI multiplier test design 97
6.2 Block diagram and register overview of the register pack test design with a simple parallel bus and a fault injection mechanism 98
6.3 Diagram of the blocks important for this thesis including the binding hierarchy for the assertion modules and the cutpoint for access to the internal bus for formal verification 99

List of Tables

2.1 Sequence operators 53

6.1 Comparison of the two verification methodologies on designs of different complexity in respect to runtime 105
6.2 Comparison system 106
6.3 Comparison of the two verification methodologies in respect to (initial) effort and reusability 107
6.4 Comparison of the two verification methodologies in respect to coverage . . 108
6.5 Comparison of the two verification methodologies in respect to runtime . . 109
6.6 Comparison of the two verification methodologies in respect to bugs 110

List of Abbreviations

ADC Analog to Digital Converter

ATE Automated Test Equipment

BDD Binary Decision Diagram

BGA Ball Grid Array

CAN Controller Area Network

CNF Conjunctive Normal Form

CTL Computation Tree Logic

DFT Design for Test

DNF Disjunctive Normal Form

DPI Direct Programming Interface

DUT Design Under Test

ECO Engineering Change Order

EDA Electronic Design Automation

EEPROM Electrically Erasable Programmable Read Only Memory

EMC Electromagnetic Compatibility

ESD Electrostatic Discharge

FPGA Field Programmable Gate Array

FSM Finite State Machine

FV Formal Verification

GE Gate Equivalent

GUI Graphical User Interface

HDL Hardware Description Language

HTML Hypertext Markup Language

HW hardware

IC Integrated Circuit

IBCB Inter Block Communication Bus

IEEE Institute of Electrical and Electronics Engineers

IEV Incisive Enterprise Verifier

IFV Incisive Formal Verifier

INF If-then-else Normal Form

IP Intellectual Property

LEC Logic Equivalence Check

LIN Local Interconnect Network

LoC Lines of Code

LRM Language Reference Manual

LTL Linear Temporal Logic

NVM Non Volatile Memory

OBDD Ordered Binary Decision Diagram

OVM Open Verification Methodology

PC Personal Computer

PCB Printed Circuit Board

PSL Property Specification Language

PXI PCI eXtensions for Instrumentation

RAL Register Abstraction Layer

RAM Random Access Memory

ROBDD Reduced Ordered Binary Decision Diagram

ROM Read Only Memory

RTL Register Transfer Layer

SAT Satisfiability

SDV Static Driver Verifier

SoC System-on-Chip

SPI Serial Peripheral Interface

STA Static Timing Analysis

SVA System Verilog Assertions

SVG Scalable Vector Graphic

Tcl Tool Command Language

TLM Transaction Level Modeling

UML Unified Modeling Language

USB Universal Serial Bus

UVM Universal Verification Methodology

VBA Visual Basic for Applications

VC Verification Component

VHDL Very High Speed Integrated Circuit Hardware Description Language

VIP Verification Intellectual Property

VLSI Very Large Scale Integrated Circuit

VMM Verification Methodology Manual

XML eXtensible Markup Language

XVP eXecutable Verification Plan

Chapter 1

Introduction

This thesis is about the development of a design verification flow, which utilizes known and new methodologies for an automated register access verification in digital integrated circuits. These established methodologies are the Universal Verification Methodology (UVM) and formal verification (model checking) using System Verilog Assertions (SVA).
The automation is supported by a framework for code generation, which was developed at Infineon and is called MetaGen, with which it is possible to generate the register tests from a high level description written in a spreadsheet program. The description is usually not written by the designer himself/herself, but by the concept engineer or application engineer, who is typically not familiar with the exact implementation in Hardware Description Language (HDL) code, but specifies the behavior of the bits in the registers. From the high level description other useful files can be generated, e.g. for the automatic Integrated Circuit (IC) testers or automated laboratory equipment. So there is a single source file from which the tests for several implementation stages of the design can be generated. This method is also called 'single source approach'). The thesis was written in the context of automotive integrated circuits, but is applicable to most other applications as well, as the methodologies used are generally usable for digital integrated circuit designs. Some other fields of design like microcontrollers, which have a lot more registers, would even benefit more from this approach.

1.1 Motivation

Digital designs tend to grow more and more complex. The well known law of Moore shows this increasing complexity over time with the amount of transistors that can be packed into a single chip. According to this law the complexity of ICs doubles within approximately 18 months. The graphic in Fig. 1.1 shows this for microprocessors, but this is true for almost all designs. With increasing complexity it gets harder to fully verify the

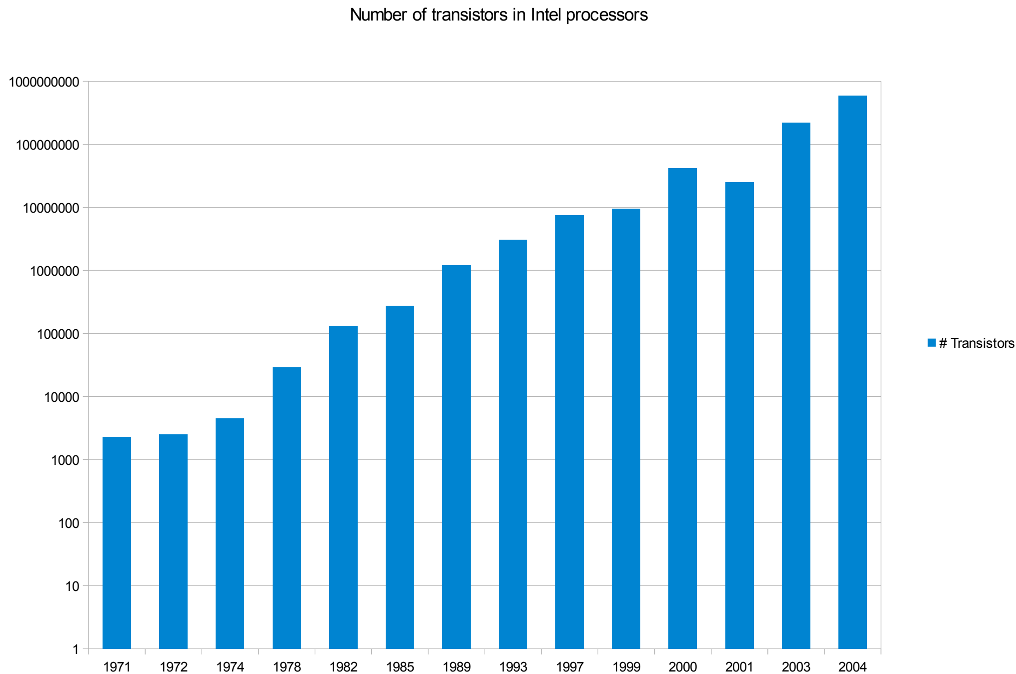

Figure 1.1: Moore's law for transistors in Intel® microprocessors, data is taken from the 40th anniversary of Moore's law from Intel [30]

correctness of such a design. Even to this day designs are often verified using functional directed tests. Such a test covers exactly one special sequence of settings for a Design Under Test (DUT). This might be acceptable to prove one of the typical use cases right, but leaves aside possibly infinite numbers of signal combinations and sequences, that might show design bugs. It is of course possible to write lots of tests for different situations, but this is a lot of work and it still won't be enough to get a good test coverage.
Typically the way to be sure the code is working as expected is to prove it correct mathematically using formal verification. The problem with this approach is that it would take much too long to check a complete design. A typical in-between solution would be to check the systems behavior with several random stimuli applied to the DUT. This will usually cover lots of cases one has possibly not thought about in the design phase. This thesis does not focus on a complete verification of a system but only the subset of register accesses via an interface, so that the part of the system is small enough to also try formal verification on it.

The increasing complexity mentioned above is also visible in the increasing amount of registers in an IC. What is special about the registers is that they will look similar because of the interface for access. The behavior of bits can also be expressed quite well, for example read-only, write-only, read-write or similar. This makes it possible to create a model and tests from a high-level description. With the result of this thesis it will be possible to create a push-button solution for register tests.

1.2 Objectives

The main objective of this thesis is to create a register access verification flow that can be used throughout the design and verification process of an IC. The pre-silicon flow (pre-silicon refers to the time before the fabrication of the IC starts; this includes design, verification and layout) has to be integrated as a sub-flow to the existing digital design flow and the silicon verification part of the flow needs to be usable in the typical verification environment in the laboratory (which is explained in further details in chapter 7.2).
The construction of the design sub-flow has to be done based on the results of the evaluation of two different verification approaches: Functional and constrained random verification using the UVM and formal verification using SVA with the Incisive Formal Verifier (IFV). The resulting methodology is chosen by comparing these methodologies using different metrics like implementation effort, number and kind of bugs found and run-time of the tests.

1.3 Outline

The thesis is split into several parts explaining the tools and methodologies used, as well as the pre-silicon verification part and the laboratory verification part.
The first part is an introduction to MetaGen, because most of the code generation and automation in this flow is based on it. It is essential to understand the way this works to get a feeling for the effort that has to be put into a special part of the verification and why some coding is done in a rather simple way with longer code instead of using highly sophisticated class hierarchies.
The second part is a rather detailed explanation of the UVM, because this methodology has just been released (version 1.0p1 came out in February 2011) and thus it is new to most people.
The next part focuses on formal verification. It has a short introduction on what formal verification is and how it works in general and the way it is used in hardware verification. Following the explanation of the methodologies a comparison is made by using them on designs with different complexity and evaluating the results based on several metrics.

With this the pre-silicon part is finished and the laboratory verification is shown as the next part. Finally a conclusion about the results is given, followed by the current state and what might be future extensions to this flow.

Chapter 2

Theoretical Background

2.1 Introduction to the Verification of VLSI Designs

This chapter is meant to give an introduction to what verification is, a short description of different terms used in integrated circuits testing and to the different ways of verifying digital integrated circuits.

2.1.1 Validation, Verification, Characterization, Qualification, Certification

Validation

There are two often cited definitions for validation, the first one is quite formal, the second one is simple yet easy to remember. The definition according to the Institute of Electrical and Electronics Engineers (IEEE): "Confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled." The definition according to Barry Boehm [16, pg. 75]: "Are we building the right system?" So validation is about finding out if the system fits for the intended use. The correctness of the implementation of the system is not of concern, but only the specification is of interest. As later on in a design all test-cases, measurement setups and other means of testing and verification are based on checks against the specification an error in this will most likely not be found by these methods as long as it is consistent within itself.

Verification

For verification there are also two definitions from the same institutions and people. The definition of verification according to the IEEE: "Confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled." The definition according to Barry Boehm [16, pg. 75]: "Are we building the system right?" Verifica-

tion is about finding out if the implemented system fulfills the requirements as denoted in the specification. Sometimes implicit requirements have to be verified too, as they are dependent on the implementation (e.g. special buffer full/empty conditions). An additional problem arises from this verification against the specification - the human factor [12, pg. 6ff]. The designer reads a specification and implements the design according to his interpretation of the specification - and will afterwards verify against his interpretation!

Characterization

The characterization of an integrated circuit is the process of evaluating all necessary electrical parameters in different operating conditions (temperature ranges, supply voltages etc.). The minimum, typical and maximum ratings for the parameters in the specification are found with this extensive evaluation method. These parameters include for example voltages, currents, timings and EMC behavior. The evaluations are done on wafer level as well as on packaged devices.

Qualification

The process of qualifying an integrated circuit is reliability testing. This is done on a large amount of samples to collect useful statistical information and these tests include special stress tests and processes like aging to predict the average lifetime of chips. Faster aging is simulated by applying high temperatures on the IC over a longer period of time. There are several different qualification standards for different applications. For integrated circuits in automotive applications, the field I am currently working in, this is the “AEC-Q100 Stress Test Qualification For Integrated Circuits”. This standard defines the requirements for the lots (e.g. they should be from non-consecutive wafers and not be assembled consecutively), the required size of samples, pre- and post-stress test requirements, failure conditions and all other parameters influencing the tests itself and the statistical results as well as the measurement setups and equipment. It provides a qualification flow, guidelines for requalifications and the set of tests needed for a certain device. These tests include latch-up tests, Electromagnetic Compatibility (EMC) tests and Electrostatic Discharge (ESD) tests, but also package tests (e.g. solder ball shear tests for Ball Grid Array (BGA) packages), special tests for devices containing memories (Random Access Memory (RAM), Read Only Memory (ROM), Electrically Erasable Programmable Read Only Memory (EEPROM), flash etc.) and many others.

Certification

The definition of certification according to the glossary of the ISO 9000 standard is: “Formal procedure by which an accredited or authorized person or agency assesses and verifies

(and attests in writing by issuing a certificate) the attributes, characteristics, quality, qualification, status of individuals or organizations, goods or services, procedures or processes, events or situations, in accordance with established requirements or standards." This means that for getting an attest of compliance to a certain standard (e.g. an interface standard like Universal Serial Bus (USB), Local Interconnect Network (LIN) or Controller Area Network (CAN), but also a functional safety standard like ISO 26262) a procedure called certification has to be performed by an authorized institution (usually a company or an university institute that has received this authorization from the issuer of the standard in some way).

2.1.2 Pre-Silicon Verification of VLSI designs

Code Coverage, Dead Code Analysis, Functional Coverage

The following subsections describe the details of two coverage metrics - code coverage and functional coverage. Both measure what amount of the design has been checked, but in very different ways. Code coverage checks what amount of the code has been seen and used in the verification. It can be done automatically and yields a result of what percentage of the code has been covered by the current verification tests and which parts of the code were never executed[12, pg. 46]. Functional Coverage measures what percentage of the design functionality according to the specification has been executed in the verification. This can not be done automatically, as it is needed to specify what contributes to the functional coverage in which way [12, pg. 55].

Code Coverage

The following information is based on [12, pg. 46-54], which is also well worth a look as it shows examples and gives some additional background information, which would go beyond the scope of this thesis.

Statement Coverage

Statement coverage shows how many lines were executed and which were and which were not executed. Usually this is the more important information. Most current simulation environments provide means of browsing through the source code with the coverage information visible. Not all lines will be executed in a normal design (e.g. default statements). Statement coverage only shows that a certain line was executed, not why and how it got there.

Path Coverage

Path coverage shows if every possible path through the code was taken (and which ones were taken/not taken). This means that it checks if every possible combination of entering if or else branches or cases has been observed (not that every expression in the conditions has been used to trigger it). If a piece of code with its conditions was depicted as a tree,

one hundred percent path coverage would mean that every path through this tree has been taken at least once.

Expression Coverage

Expression coverage checks which of possibly several expressions were used to enter the if/else/case branch. Path coverage will show that a branch has been used, expression coverage that the branch was entered in all possible ways (and which entry conditions were never met).

Finite State Machine (FSM) Coverage

A finite state machine is usually coded as a switch-case statement. Verifying that each state has been used means verifying that each case has been used. The second part of FSM checking is to verify whether all possible state transitions have been seen.

In addition the the checks mentioned above, which run with a dynamic verification, also automated static checks for some of these items exist. These are formal methods that abstract the design to a state transition model (these will be explained in more detail in the chapter about formal verification) and on this reachability of certain code can be checked (dead code analysis), as well as the reachability of FSM states and some other properties. These checks prove that there is a way to reach the code or states, but not necessarily the one intended by the designer.

Functional Coverage

The in-detail description of the following coverage types as well as some examples and background information can be found in [12, pg. 55-61]. Functional coverage can (almost always) only be defined manually. A number of specification items get mapped to a number of coverage items (not necessarily the same number), where the occurrence of these items in a simulation is noted. What this occurrence means has to be defined manually too, as it can be quite complex, e.g. it can just be the occurrence of a “fault”-signal on an output port of the design, but also a failure bit set within a reply frame in a bus protocol as a reaction to a bad checksum.

Item Coverage

Item coverage is a measure for different events, states and transactions that occurred from, to or within the design. For example data transfers to and from the design are always monitored. This may be a lot of different communication items, but that a readout of a diagnosis value showed e.g. an over-temperature situation would be a relevant item that one wants to have been seen in some verification. If this occurs it is known that the design can flag the over-temperature. If a lot of constrained random tests are done this kind of coverage can be even more interesting, as the outcome of a test can vary within certain limits. It can then be monitored that buffer overflow situations, all different command types of a certain command set and so on have been seen while running a big set of verification tests. Also for a non-random test suite this might be interesting, as many items

can be defined on a higher level already (e.g. in concept phase) and the completeness of the test suite in respect to the set of possible states can be measured.
Cross Coverage
Cross coverage specifies the combined occurrence of two or more items. Often it will not be sufficient to know that one event happened, but it is needed to know that this happened while something other was also true, e.g. that this over-temperature diagnosis bit was set while an output driver was on.
Transition Coverage
Transition coverage specifies the sequence of items that occurred. Often it is interesting that a certain design behavior shows in a certain sequence or is stimulated in a certain sequence.
Cross coverage and transition coverage operate on the same items as item coverage and can be found also in post-processing of the data. Most dedicated verification languages feature ways to specify them and also have it checked while the simulations are running.

Formal Verification

The term formal verification in the context of Very Large Scale Integrated Circuit (VLSI) development refers to two types of verification, which are equivalence checking and model checking. The first one is an automatic proof of the equivalence between designs in different abstractions(usually RTL versus gate-level netlist) or description languages(e.g. design written in Very High Speed Integrated Circuit Hardware Description Language (VHDL) compared to a rewritten version of it in SystemVerilog). This is present in all digital design flows nowadays, especially because it requires nearly no additional effort (only a few scripts). The second one is an automated way of proofing logically that a design meets some specified requirements. This requires quite some additional effort, as the model (high level specification) has to be written manually. This is done using properties specified in a certain language in addition to the normal design(examples for these languages are Property Specification Language (PSL) and SVA). Model checking is explained in detail in chapter 2.4.

Functional Verification

The aim of functional verification is to prove the correct operation of a module or system in a certain operating mode. The tests are done simulating the design in the way it would be operated in the real application. Parts of the system that are not part of the design are modeled in an adequate degree of complexity to be close to the real behavior without trading in too much simulation time (e.g. for mixed signal systems the analog parts are modeled for the digital simulation). Certain operating conditions can be achieved easier

in the simulation, where the model of the environment is easily controllable, than in the verification of the fabricated IC in the laboratory.

Directed Testing

For directed testing a scenario is run step by step. Everything is deterministic, the points in time when some signal is set is known, as well as the time and reaction of the system. Tests are usually built in a way that a signal is set, after a certain wait time an output is checked and so on. Checks for different operation speeds or communication speeds have to be explicitly specified. A full verification suit of directed tests usually features one typical use case test and additional tests with settings that have been identified to be corner cases for this design. The immediate drawback is that corner cases that have not been identified as such will usually not be tested. If functional coverage is collected, it is only used to check the completeness of the verification (i.e. has a needed test been forgotten?).

Constrained Random Verification (Coverage Driven Verification)

Constrained random simulation also follows some real sequences. The difference to directed testing is that parts of this sequence are randomized, for example random data can be sent to random addresses via the communication bus and so on. To keep this within a meaningful range all randomized variables have some constraints set that define the limits within which the values can be. Randomization adds the problem that the state cannot be checked like in direct testing, as at the time of writing the test the state of the design at a certain point in time is not known. This is the reason, that a different kind of pass/fail checks have to be used. This verification makes a vast use of functional coverage. Transaction items are collected and evaluated within a certain module (called "scoreboard"), which evaluates if a certain item is showing the correct behavior of the system in regard to other items it has received before or at the same time and also if the content of the item itself is correct (e.g. a data transfer with a checksum). This means that the scoreboard contains some kind of high-level model of the design. Running a test several times with a different random seed will most likely produce different results, this means that the information about the functional coverage (and also code coverage) is used to judge if the verification is done.

2.1.3 Laboratory Verification and Productive Testing of VLSI Designs

Scan Test

The scan test is used to find production defects in the logic of the IC. It is also used by the failure analysis department to locate defects that showed up during the lifetime of a chip. For example long time drift and aging can lead to problems on ICs that were not visible at production time. Instead of normal flip-flops with just a D input special scan flip-flops are used, which feature an additional input used for scan. All the flip-flops that

should be scan-able are chained up into one ore more chains by connecting the data out pin (either the normal Q pin or sometimes a dedicated scan-out pin) with the scan in pin of the next one to form a shift register. Using an (usually external) clock the current state of all registers can now be shifted out, a pattern can be applied at the first flip-flop, shifted into the design and then by doing one clock cycle with D instead of scan-in selected as the input the combinatorial logic between the flip-flops produces the next data, which is then shifted out. These patterns, called input and output vectors, are pre-calculated and the output can be compared against this to find defects. The scan test is very common among VLSI designs, as it can be implemented nearly automatically, the drawback is an increased amount of area for the flip-flops. For more information on scan test please refer to [32, pg. 324f].

Automated Test Equipment

In principle every automated measurement device is called Automated Test Equipment (ATE), but the focus in this subsection is especially on the ones used for integrated circuit testing, often called chip testers. These very complex devices are used in the automated verification and characterization of ICs as well as in the test and trimming of devices in the production. The already mentioned scan tests are performed and also other functional checks and trim procedures (to get rid of process variation dependent deviations of analog values) and other device initializations like erasing Non Volatile Memories (NVMs). The ATE can be used in different states of the chip production, e.g. on the wafer or the final bonded and packaged device. This depends on different needs, for example trim values that are sensitive to the package stress should be set after packaging.

Other Laboratory Equipment

The verification and characterization done in the laboratory is usually accomplished by using a wide variety of different measurement equipment. Apart from the well known voltage/current sources, voltmeters, ampere-meters, waveform-generators and oscilloscopes an increasing number of automated measurement equipment is used in combination with these other devices. The verification Printed Circuit Boards (PCBs) are often controlled by a microcontroller or Field Programmable Gate Array (FPGA) for automated measurements. The National Instruments PCI eXtensions for Instrumentation (PXI) system is another example of an automated measurement equipment for the laboratory. It is a special Windows-PC with various measurement and stimulus cards that can be used to read or force values and the measurement automation is programmed in LabVIEW (a graphical programming language developed by National Instruments).

2.2 MetaGen

A rather new methodology (for hardware development) originating from the software development domain is the so called "Meta-Modeling". Using a meta-model and data that fills this model with information textual output e.g. source code can be generated instead of typing it manually. This code can be virtually anything: C++, Java, Pascal or any other type of programming language, Hypertext Markup Language (HTML), XML, LaTeX,VHDL, Verilog, plain text or anything else that can be expressed using text (also graphics might be expressed as text, e.g. Scalable Vector Graphics (SVGs) are described using a syntax similar to XML).
So how is that possible? A meta-model usually is represented as some kind of class hierarchy. This is typically modeled using Unified Modeling Language (UML) or a similar high-level modeling language. See 2.2.1 for more information about such models. This model represents the structure of data. Data itself is filled into the structure from a spreadsheet source using an import filter that generates instances of the classes defined in the model and populates them with this information.A template engine is then used to fill a language-specific (language in the meaning of programming language or other file format language) template with information read from this filled data model. See 2.2.2 for a deeper insight into templates and template engines.
MetaGen is a framework for generating the whole meta-modeling infrastructure. It was

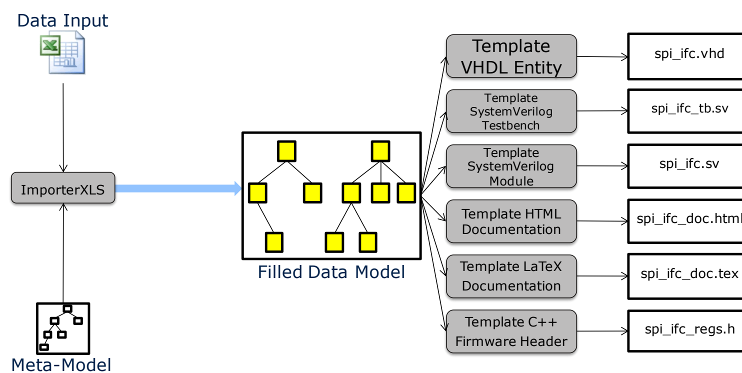

Figure 2.1: Meta-modeling flow for generating HW design files and documentation files

developed by the design support and methodology department of Infineon in Munich [25].
MetaGen provides everything necessary to create meta-models and importers for the model data and write templates and exporters for different formats. Additionally it features a library, which already includes lots of predefined importers, exporters and templates.
Another special part of MetaGen is that it includes a predefined data model for hardware

modules, already featuring ports, bus interfaces, registers, memories and other components that are needed to define and generate HDL code. This model was created even some time before MetaGen itself and is called the Essence Data Model. It is the model that is used as the base from which all the register models, assertions and VBA code in this thesis is generated. The data input to this model is a predefined Excel workbook standardized by the digital design community at the Infineon development center in Graz. It includes the information necessary for typical applications in the automotive chip design.

2.2.1 Data Models

A data model represents the hierarchy and dependencies of classes of a given system. This is best explained using an example. This example will be used throughout this chapter to clarify what a simple meta-flow looks like. As this thesis is about the verification of HDL code, the data model will be the definition of a hardware (HW) module. It should include the things needed in order to generate a simple VHDL entity or Verilog module (to keep it simple only ports are used, additional parameters like generics are left out). The corresponding model is shown in figure 2.2. The root class is a module and it has a name

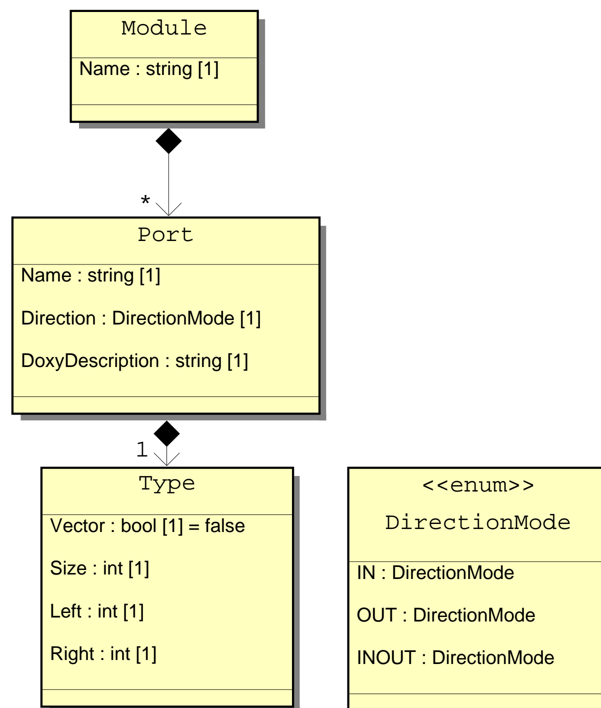

Figure 2.2: Simple data model for a module

(which will be the entity/module name). It can have an infinite numbers of ports. Every

port has a name, a direction which is defined by the enumeration visible in the figure and a description which will be used for code documentation. Every port has exactly one type object, which defines if it is a vector type (this is not really needed, as this could be seen from the size, but the template will be easier to understand), the size of this port and the left and right indices of the vector.

After the modeling of the data structure has been done data itself is needed. The module specification is written in Excel (Excel is a commonly used tool to write specifications like pin lists, register maps and so on throughout the industry). Most important is that the input has to obey a well defined structure or syntax in order to serve as a data source. A short demonstration sheet is given in figure 2.3. A productive module often has more than hundred ports, this simple model would also work for such long lists. Now everything

	A	B	C	D	E	F	G	H
1	ModuleName	my_module						
2		Name	Direction	Vector	Size	Left	Right	Description
3	Port	clk	in	FALSE	1	0	0	Clock
4	Port	reset_n	in	FALSE	1	0	0	reset
5	Port	my_scalar_output1	out	FALSE	1	0	0	scalar output port
6	Port	my_scalar_output2	out	FALSE	1	0	0	another scalar output port
7	Port	my_scalar_input1	in	FALSE	1	0	0	scalar input port
8	Port	my_scalar_input2	in	FALSE	1	0	0	another scalar input port
9	Port	my_vector_output	out	TRUE	4	4	1	a vector output
10	Port	my_vector_input	in	TRUE	4	3	0	a vector input

Figure 2.3: An Excel sheet with data for filling the data model

is ready to combine data with the model. In order to do so, an import filter capable of reading the Excel sheet has to be written. This has to be done only once as long as the input file structure is not changed. This is a good reason to set up standardized sheets for a special purpose, so the extra effort of writing the importer has to be done only one time.

In this flow this is done using the Python scripting language. As the model itself is simple and small so is the importer:

```
1  class module_xls2xml:
2      def __init__(self,**args):
3          self.__dict__.update(args)
4          # do abort if severity error
5          self.Api.setSeverityExit("error")
6          # get instance of xls input plugin
7          xls_in = self.Api.getInput("xls_in",0)
8          # unmarshal input plugin if required
9          if xls_in.workBook == None:
10             xls_in.unmarshal()
11         # get unmarshaled workBook from plugin
```

```
12            workBook = xls_in.workBook
13            # call conversion member function
14            self.readXLS(self.dataInput,workBook)
15
16        def readXLS(self,Module,workBook):
17            sheet_cnt = 0
18            for sheet in workBook.sheets():
19                sheet_cnt = sheet_cnt + 1
20                if sheet_cnt == 1:
21                    for row in range(sheet.nrows):
22                        line_type = str(sheet.cell(row,0).value).strip()
23                        if line_type == "ModuleName":
24                            Module.setName(sheet.cell(row,1).value)
25                        elif line_type == "Port":
26                            Port = Module.addPort()
27                            Port.setName(sheet.cell(row,1).value)
28                            Port.setDirection(sheet.cell(row,2).value)
29                            description = str(sheet.cell(row,7).value)
30                            description = description.replace("\r\n", "; ")
31                            Port.setDoxyDescription(description)
32                            pType = Port.createType()
33                            pType.setVector(sheet.cell(row,3).value)
34                            pType.setSize(sheet.cell(row,4).value)
35                            pType.setLeft(sheet.cell(row,5).value)
36                            pType.setRight(sheet.cell(row,6).value)
37                        else:
38                            self.Logger.warn("Unknown line kind: "+str(line_type
                                 ))
39                else:
40                    self.Logger.warn("More than one sheet is not allowed")
```

Listing 2.1: Importer code

The code itself should be rather self-explanatory. It iterates through the rows of the sheet and uses functions defined by the data model to fill the model with data using access functions like setName or setSize.

Now the model and data are combined. This is stored in an XML file, an excerpt is given here:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Module>
3   <Name>
4     my_module
5   </Name>
6   <Port>
7     <Int_Class_ID>
```

```
8         514899624
9       </Int_Class_ID>
10      <Name>
11        clk
12      </Name>
13      <Direction>
14        in
15      </Direction>
16      <Type>
17        <Int_Class_ID>
18          514899696
19        </Int_Class_ID>
20        <Vector>
21          False
22        </Vector>
23        <Size>
24          1
25        </Size>
26        <Left>
27          0
28        </Left>
29        <Right>
30          0
31        </Right>
32      </Type>
33      <DoxyDescription>
34        Clock
35      </DoxyDescription>
36    </Port>
37    ...
38  </Module>
```

Listing 2.2: XML code of the filled data model

2.2.2 Templates

Now the filled model can be used to generate various files. This is done by writing templates for the different file types needed. A template in this context is a file filled with partially fixed text and text taken from the filled model and controlled generation of text depending on information from the model. For this example the Mako template engine [10] is used. It uses Python for special transformations and simple Python like-statements for simpler actions. Templates can call sub-templates, so some can be written as function libraries and can be reused.

The simple "top" template for the VHDL entity generator looks like this:

```
1 <%namespace name="vhdl" file="vhdl_templates.mako" import="*"/>\
2 ${vhdl.Entity(dataInput)}
```

Listing 2.3: Top template for the entity generator

The ${vhdl.Entity(dataInput)} prints data returned from the sub-template "Entity", which is defined in the vhdl namespace (the template functions in the vhdl_templates.mako file are all defined in this namespace) and it gets the data input as argument. The data input in this case is the filled data model as Python object hierarchy.
The interesting part is defined in the sub-template:

```
1  <%def name="Entity(Module)">\
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  entity ${Module.getName()} is
6  ${Port(Module.getPorts())}\
7  end ${Module.getName()};
8
9  </%def>\
10 <%def name="Port(Ports)">\
11 % if len(Ports) > 0 :
12    port(
13 <%
14    max_len = 0
15    for plc in Ports :
16      if len(plc.getName()) > max_len :
17      max_len = len(plc.getName())
18 %>\
19 % for Port in Ports:
20      ${(Port.getName()).ljust(max_len)} : ${(str(Port.getDirection())).ljust
        (5)} \
21 %   if Port.getType().getVector():
22 %     if (Port.getType().getLeft() - Port.getType().getRight()) + 1 != Port.
      getType().getSize():
23 <% print "ERROR: Size and Left/Right do not match" %>\
24 %     endif
25   std_ulogic_vector( \${Port.getType().getLeft()} downto \${Port.getType().
      getRight()})\
26 %   else:
27   std_ulogic\
28 %     endif
29 % if Port == Ports[-1]:
```

```
30  \
31 % else:
32 ;\
33 %   endif
34  --! ${Port.getDoxyDescription()}
35 % endfor
36    );
37 % endif
38 </%def>\
```

Listing 2.4: sub-template for the entity generator

This simple template generates the following entity definition:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity my_module is
5    port(
6      clk                 : in    std_ulogic; --! Clock
7      reset_n             : in    std_ulogic; --! reset
8      my_scalar_output1   : out   std_ulogic; --! scalar output port
9      my_scalar_output2   : out   std_ulogic; --! another scalar output port
10     my_scalar_input1    : in    std_ulogic; --! scalar input port
11     my_scalar_input2    : in    std_ulogic; --! another scalar input port
12     my_vector_output    : out   std_ulogic_vector( 4 downto 1); --! a vector
           output
13     my_vector_input     : in    std_ulogic_vector( 3 downto 0)  --! a vector
           input
14   );
15 end my_module;
```

Listing 2.5: VHDL entity generated from the template

Just by writing an additional template it is now also possible to create a testbench, which instantiates the entity and creates sources for every port without having to create a new input file or model.
It should be clear now how easy it is using the meta-modeling approach to create code automatically for different applications. For the case that a quite dedicated and standardized meta-model already exists the effort is even reduced much further. Exporters from this format are always independent of the input source style (the Excel table or else) and are reusable for everyone who works with this model. This is exactly what the Essence data model is used for. The code generation examples in the UVM and formal verification chapters are based on it.

2.3 Universal Verification Methodology

The Universal Verification Methodology (UVM) is a new verification methodology and features a SystemVerilog library to implement test environments according to it in SystemVerilog directly and via Direct Programming Interface (DPI) also in SystemC and similar HDLs. The UVM is based on the Open Verification Methodology (OVM) version 2.2.1, which has been used for several years. The UVM extends the OVM with new features, the most interesting one for this thesis is the register layer. In the OVM register tests needed additional packages like the RegMem package from Cadence. Another big advantage of the UVM is that it is supported by all of the big tool vendors like Cadence, Mentor Graphics and Synopsys. The OVM was not supported by all vendors and there was the Verification Methodology Manual (VMM) as competing methodology as well as several other ones. The VMM defined a register layer (called RAL - register abstraction layer) which served as the base for the UVM register layer.

2.3.1 Basic Verification Environment

This chapter serves as an introduction to the structure of a UVM test environment and the components it typically includes. Some are not always required or are sometimes implicitly included in special classes. This is very similar to the OVM environment structure, but will still be explained here, as understanding this structure is essential for understanding how the methodology works, how parts of this can be generated and how easily components can be exchanged with other components without having to adapt the whole testbench. In figure 2.4 an overview of the core structure of a testbench is given. The components in this will be explained in detail in the following sections.

A main aim of the UVM (and also OVM) is to improve Verification Intellectual Property (VIP) reuse. This is a reason why this structure might seem a bit complicated compared to a typical testbench written in plain VHDL/Verilog style or in Tool Command Language (Tcl). It might also seem strange at first that the verification environment is instantiated in the test and not the other way around. This is also due to the reusability approach. This will be explained further in 2.3.1.

Environment

As mentioned in [20, pg. 66] the environment includes the main methodology components like the agents (which include the monitor, driver and sequencer) and the scoreboard, but it can also include environments, which allows building hierarchical verification environments. It does not include the tests, but is instantiated in the tests. In [3, pg. 4f] it is stated, that the main purpose of the environment is to "model behavior by generat-

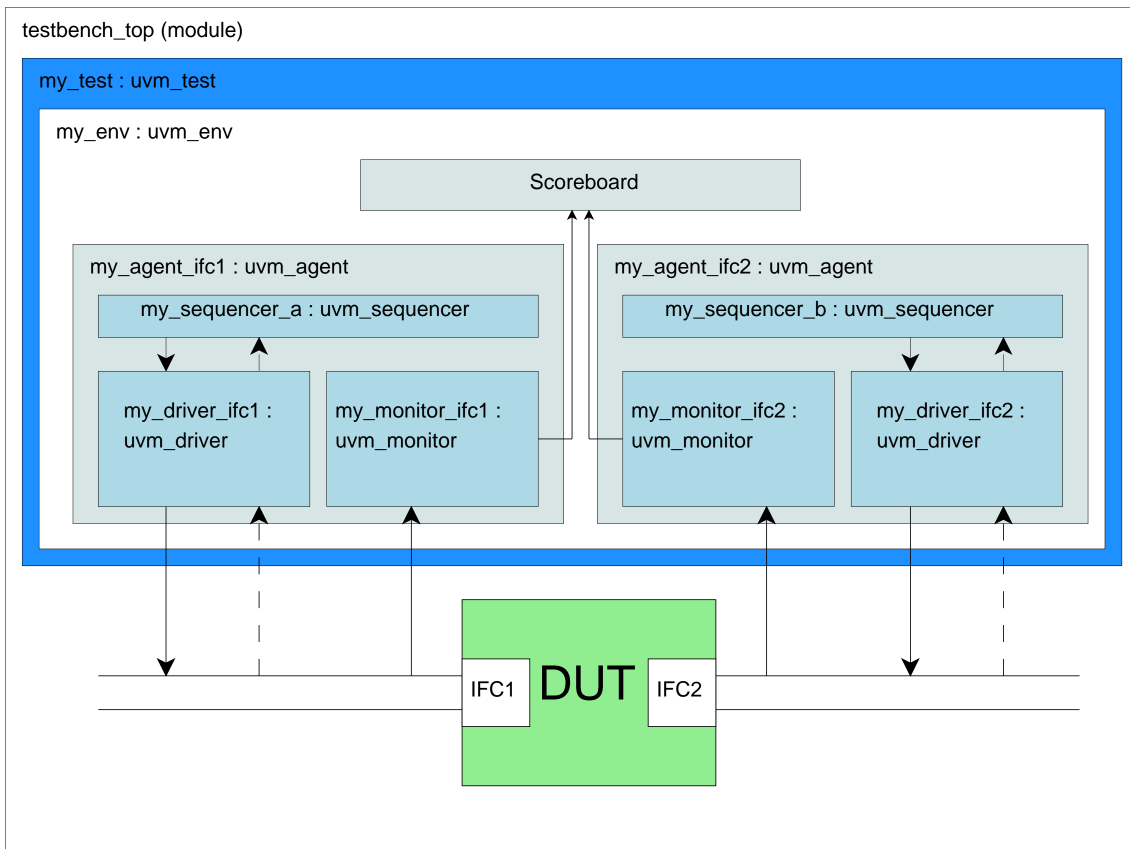

Figure 2.4: Structure of a UVM testbench

ing constrained-random traffic, monitoring DUT responses, checking the validity of the protocol activity, and collecting coverage".

Agent

The agent works as container for interface specific components that logically belong together (e.g. a CAN driver and its sequencer and a CAN monitor) and can be reused as this bundle in other environments. It can be stored in VIP repositories to facilitate reuse [3, pg. 2]. It does not need to include a driver. If only a monitor is used it is a passive component that only reads the bus. This should rather be done by building a configurable agent that uses configurations to decide if the driver should be instantiated(see [20, pg. 14]). The agent itself has not much functionality other than deciding this active/passive state and instantiating and creating the connections between its contained components. These include the real functionality and are explained in detail in the following sections.

Monitor
A monitor is a component that reads data on an interface. It, thus, has to be able to decode the specific protocols at least on the data link layer. Further information processing might happen in an upper layer of the hierarchy, e.g. the scoreboard or a custom decoder component in between, but for passing typical transaction items (i.e. sequence items) it usually does the full decoding for this. A monitor does not drive signals so all the ports in the interface connecting to the DUT are inputs. If it is connected to a scoreboard it uses an analysis export to send data via a Transaction Level Modeling (TLM) channel to it. For coverage driven verification (what the OVM/UVM standard is about) the monitor is also used to check coverage. It is not the only Verification Component (VC) that checks coverage though. It is mentioned in 2.3.2 that the register model defines its own covergroups and coverage constraints for functional coverage. As this is not directly used in this thesis for more information about coverage driven/coverage based verification a short introduction can be found in [3, pg. 1], for further information refer to [8, pg. 235ff] or [12, pg. 55ff].

Driver
A driver is the VC responsible for communication with the DUT. It drives the signals and in some cases also reads them. This backward path is used for protocols like the Serial Peripheral Interface (SPI). A driver receives items from a sequencer (which is also part of the agent and will be covered in the next section) via a TLM port and transforms this abstract item into a hardware access to the DUT via a virtual interface (a virtual interface is an instance of a normal SystemVerilog interface that only exists at runtime, as it is created in an object context).

Sequencer
A sequencer takes items from a sequence (usually generated in the test) and passes them to the driver. This can be done in pull-mode (the driver requests the item from the sequencer) or in push-mode (the sequencer pushes the item into the driver). Sometimes the sequencer is a simple typedef that adapts a uvm_sequencer base class to the used sequence_item type. It can also handle the randomization of items and other functions. See [3, pg. 3] for more information on this topic.

Sequence

A sequence is a structured series of data called sequence items that are sent to the sequencer for use in the driver. It can describe directed tests as series of predefined transactions or generate constrained random data. It is separated from the hardware specific representation of data (which is created by the driver from these items) and can be reused for testing the same data sequences on different interfaces connecting to a DUT without

having to change anything but the hardware-specific part (i.e. the driver or the whole agent). Figure 2.5 shows this communication from the sequence to the DUT. For more detailed information refer to [20, pg. 177ff].

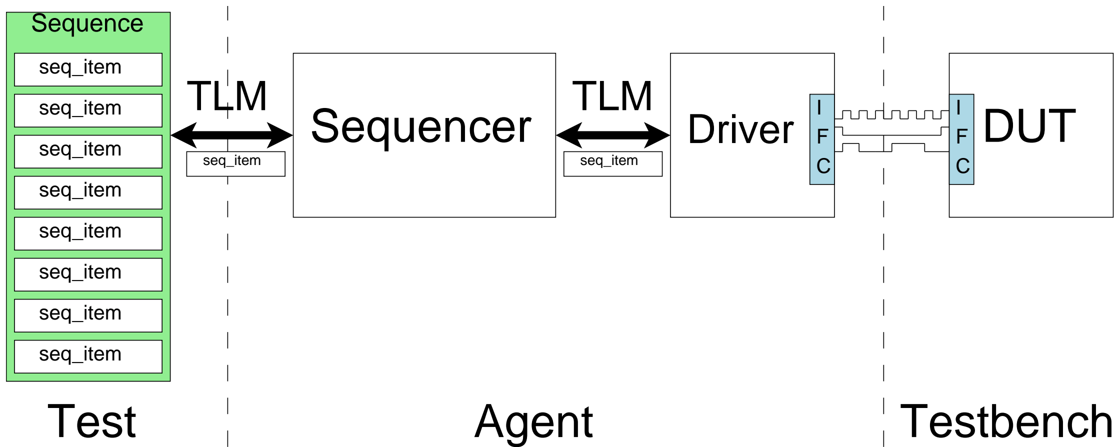

Figure 2.5: A UVM transaction from sequence to DUT

Test

The test is the object, whose execution gets called in the top testbench module using run_test(), but it is not explicitly instantiated. Although they can be defined in the testbench, it is better to define the tests in separate files and include them in the testbench file. The command line argument +UVM_TESTNAME=class_name_of_test_to_run chooses the test and executes it in this testbench. This way it is easier to use such automatic tests if e.g. overall coverage is checked and collected with another program like the Verisity vManager [29] or similar software or self written verification scripts. The test instantiates the sequences and the environment. In this way the same test can be done using different environmental setups e.g. other interfaces or differently connected interface for a new version or different configuration of the DUT.

2.3.2 Register Model

The UVM register model is based on the VMM Register Abstraction Layer (RAL) [5]. It describes the registers (and memories) in a class hierarchy. Parts of the following explanations of the register modeling can be used in a similar way for modeling memories, but as this is not in the scope of this thesis it will be omitted. For further information on how to model memories refer to [20] and [3] and for tutorials and code examples the Mentor Graphics OVM/UVM cookbook in the verification academy [26] is recommended,

note that registration with a company email address is needed for access to this. In the following sections the basic building blocks of the register model will be explained, but the focus is on the register types used in this thesis (there are some more ways of modeling the registers). In fig. 2.6 an example testbench using the register model can be seen.

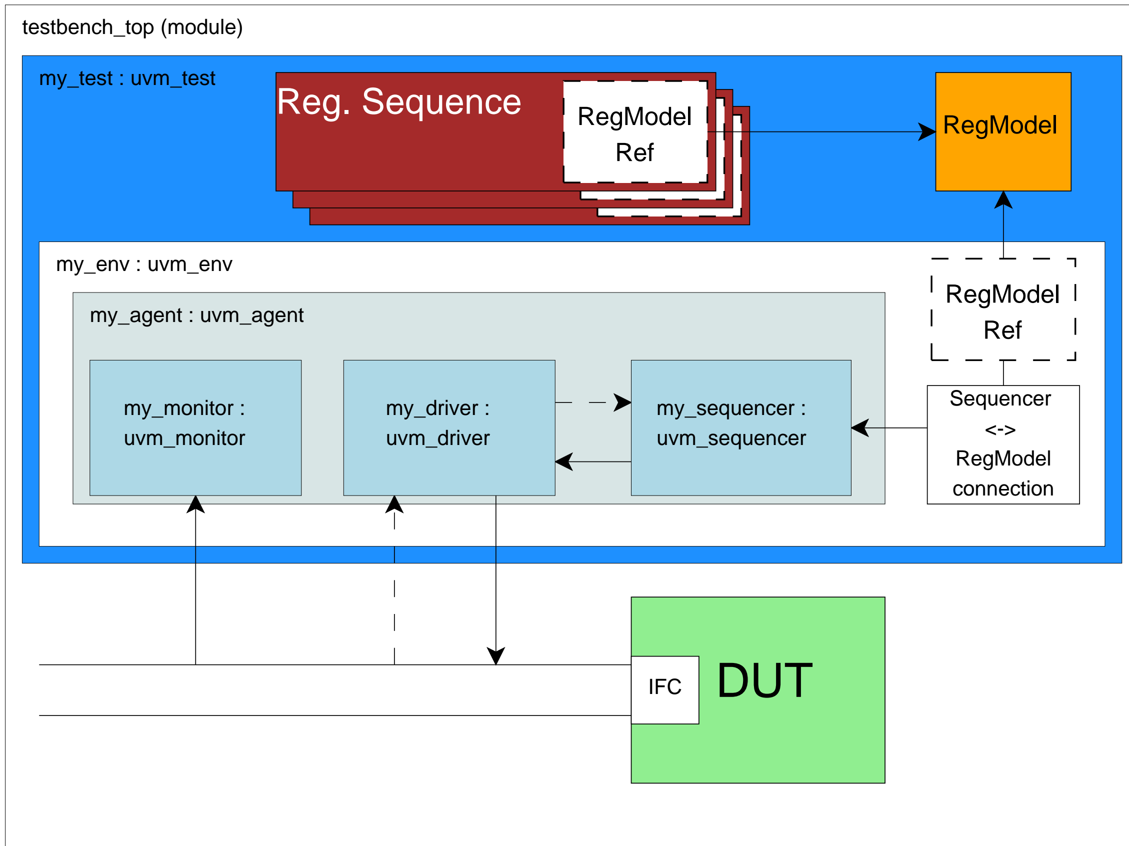

Figure 2.6: Example of a UVM testbench using a register model

Bitfield

The bitfield is the smallest element of the register model, and registers consist of these bitfields. It describes a set of bits that logically belong together. The base class is uvm_reg_field, which normally is not subclassed, but used the way it is. For special purposes it might help deriving an own class from this, e.g. for implementing special access policies like config-once [3, pg. 93]. A bitfield can be configured with the following parameters [2, pg. 548]:

- size: the size in bits

- lsb_pos: the position of the LSB of this field in the containing register
- access: a string selecting a predefined access type (see [3, pg. 89f])
- volatile: the volatility of the field
- reset: the reset value (standard: power-on-reset value, but several resets possible)
- has_reset: defines if the field has a reset
- is_rand: activate/deactivate randomization (should be set to 1)
- individually_accessible: defines if the field is accessible individually in the register

The access policy is one of the essential settings for automated access checking. It defines how the bits should behave on read and write operations. The most common types are "RO" (read-only), "RW" (read-write) and "WO" (write-only). The UVM defines several other types, see [3, pg. 89f] for a list of the predefined policies. These policies can be extended to match user needs. A typical application would be a field that is read-only as long as a special bit is not set (guarded field).

Register

A register is the element grouping a number of bitfields and configuring them (with the settings mentioned in 2.3.2). The register class is derived from uvm_reg and filled with the design specific information. This means that the bitfields get declared and then constructed via the factory in the build phase (the UVM uses a design pattern called "factory" defined by the Gang Of Four to provide reconfigurability without having to alter the testbench code [39]). In this register class custom cover groups and coverage settings can also be defined. Some coverage can be collected automatically (by passing arguments like UVM_CVR_FIELD_VALS in the constructor of the class, see [20, pg. 142] for more). For this thesis custom cover items will not be used, the predefined register sequences already include checking the information needed for general access verification.

Register Block

A register block, derived from uvm_reg_block, declares the registers and constructs them via the factory. It also handles the mapping from different interfaces to the registers. The interface mappings can have different base addresses, byte widths and endianess and the register can be mapped into these interfaces on different addresses and with possibly different access types (read-write, read-only or write-only). The HDL path needed for backdoor access (see section 2.3.3) is also specified here (optionally, only needed if it is used, which is the case for some sequences used in this thesis). This is a possible top level

of the register model, but blocks may also be included in blocks. When instantiating the model in the testbench a check if the parent is null is used to see if it is really a top level block. See figure 2.7 for an example structure of a register model.

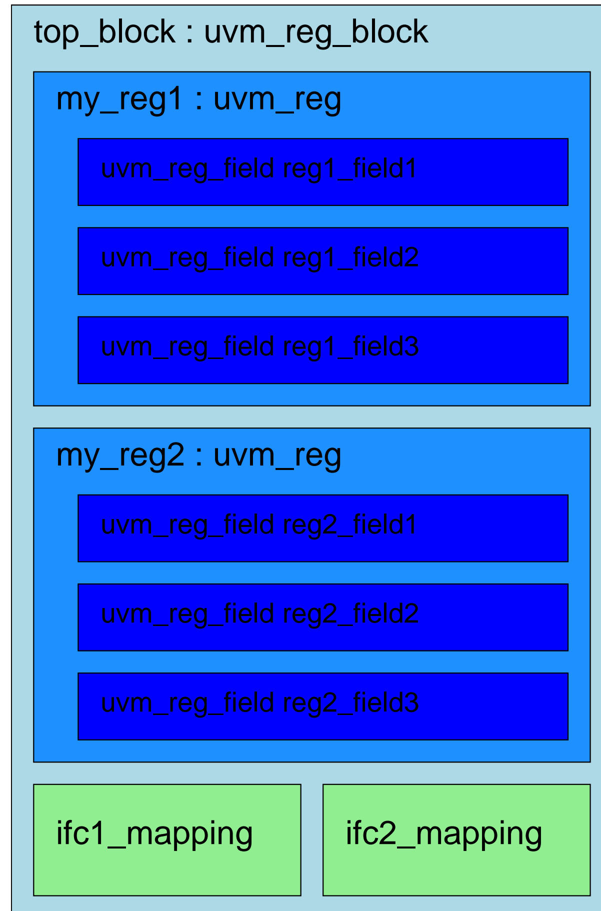

Figure 2.7: Example structure of a register model

2.3.3 Using the Register Model

This section is about the usage of a register model after it has been created like described in the previous sections. The model itself gets instantiated in the test, as it can be seen in figure 2.6. After connecting it to the sequencer in the agent for the used bus interface the driver can access the registers in the DUT like it would be done on the real IC, as described in 2.3.3. The alternative to accessing a register in a device in a testbench, which does not take simulation time (in the meaning of simulator time steps) because it bypasses the normal communication and directly writes into the signals is called backdoor access,

described in 2.3.3. After successfully connecting everything register sequences can be used to drive the signals. These sequences can be custom ones utilizing the register model to e.g. create directed tests (maybe a chip startup sequence) or also constrained random testing or mixed ones, like starting up the chip normally, then switching to constrained random values sent to the DUT. The UVM also predefines some sequences that perform the checks that are most commonly done on designs.

Access Types and Functions

As mentioned, there are two major ways of accessing any device in a testbench - a normal access via an interface and a backdoor access.

Frontdoor Access

A frontdoor access is the method used to communicate to the chip after production. Data is written to or read from the device via an interface like a LIN, SPI, CAN or any other one suitable for the design. This can not happen in zero time, as data will be transferred in one or more clock cycles. For the pre-silicon verification this means that this will take simulation time, but reflects a real access in the best way. It is essential to check this kind of access to detect interface design errors as well as errors in the registers.

Backdoor Access

A backdoor access is an access that bypasses the real interface communication and thus can be done in zero simulation time. This of course does not reflect how the real communication works, but is a good tool to see what data arrived in a register after accessing it via the frontdoor or writing known data into a register and check what arrives on the interface after a read access. In SystemVerilog such an access is quite easy, as signals can be accessed from the outside of the module by specifying the hierarchical path to it in a fashion like dut.module1.submodule1.signal. In the register model these paths can be specified to allow the predefined sequences to use it. The frontdoor access is independent of the actual position of the register in the hierarchy of the design as the access works via the interface, but the backdoor path changes if the register is moved between blocks. For automatically generating the backdoor path it is thus needed to know the whole system structure. The good thing for the solution in this thesis is that the Essence data model is capable of describing the whole design with hierarchies and registers. As this meta-modeling flow is a new approach in design used at Infineon Graz only for a year now not everything has been moved to a full description via a meta model, so the backdoor paths get specified manually for now, but this will be changed to automatic generation later.

Register Sequences

Sequences are used as a high-level interface independent description of data to be sent over an interface (or via backdoor). For the register sequences these use the register model to create fixed or random data fitting to these registers. For extensive testing of the design quite a lot of sequences will be needed, some of them as directed tests for checking the defined behavior and some as randomized sequences to check the handling of correct and possibly faulty settings. Using randomized sequences results in automatically checking the design's behavior on lots of different register accesses, some maybe not considered while designing, without having to explicitly specify hundreds of sequences. This randomization can of course be constrained to take only a special set of values or values in a specified range or similar, but one has to keep in mind not to constrain too much or possibly faulty stimuli might not be generated. The UVM already specifies some sequences that check part of the design and use the register model's constraints like access type and bit width to automatically drive signals and compare the results to the mirror (a modeling of register data predicting what will have to happen to the bits according to their specification).

Built-In Sequences

The UVM has predefined sequences for several types of register and memory accesses. As it is not in the scope of this thesis the memory part will be omitted, see [3, pg. 129f] for information on this. For the register sequences there are some that work on a single register and according ones that do the same for all registers in the model. The first sequence, that should be run is the reset value checking sequence (which is only defined for the whole model) that verifies the "hard" reset values (set in the model) after a reset of the DUT. It is called uvm_reg_hw_reset_seq. The full list is available in [3, pg. 129f].

Custom Sequences

Custom sequences can be used for directed tests (e.g. sending two operands and checking the correct result of a multiplier) but also for random stimulus or general register tests that are not covered by the default sequences. It is done by deriving a class from uvm_sequence and creating a reference to the register model there. In the body task functions like reg_model.REG_NAME.write or reg_model.REG_NAME.read can be used to communicate with the DUT and still have the mirror check the behavior. The bus addresses of the registers are also not needed in the test as the access functions are working on the register model and get the address from there. This means that a change in the register addresses has no influence on the sequence and thus reusability is better and it is more robust against design/concept changes. Backdoor access can also be used in custom sequences, but it has the problem with the dependency on the design hierarchy (and namings), which makes it weak against the design/concept changes, but allows better insight into the DUT.

Extending Bitfield Behavior

There exist a lot of predefined behaviors, as can be seen in [3, pg. 89f], but for a typical design the point at which none of those fits for some bitfields will be reached soon. There are virtually infinite possibilities for behaviors, so there is a mechanism to implement custom ones. One example would be a bit "valid" that gets set after an Analog to Digital Converter (ADC) finishes the conversion and places the measured value in a register. No description fits the "set by design" behavior which that is. It would be very hard to generally specify this without losing the ability to check this bit. But with knowledge of the design one is able to tell that this bit will be set in an amount of clock cycles after another bit (the "start conversion" bit) was set. Another quite usual behavior is the guarded field, which is only accessible for write operations after a special guard bit has been set. To enable such a behavior two ways are possible [3, pg. 92f]:

- Derive a custom class from uvm_reg_field in which the methods for pre/post read and write are extended.
- Create a custom callback method.

As such bitfield behaviors might be quite usual in the designs one is working with it is best to create a library for such special bitfields to enable reuse.

2.4 Formal Verification

Formal verification is a term that has been around for more than 30 years [8]. Its use is quite common in software development nowadays, for example the Microsoft SLAM and Static Driver Verifier (SDV) [15] [24], where formal methods are used for static verification of driver code written for Microsoft Windows. There are also operating system kernels that claim to have been fully verified using formal methods [22]. For hardware development formal verification has been in use for some time too. The most typical and widely used formal verification type in hardware design verification today is equivalence checking (starting by the late 1990s [8, pg. 174]), i.e. checking two implementation stages (or designs) against each other for logic equivalence. This can be for example the comparison of RTL code and gate level netlist, the comparison of two netlists after a small fix has been made in one of them or the comparison of a design written in one HDL to the same design after moving it to a different language [12, pg. 9f]. The logic equivalence check has become an integral part of today's design implementation and verification flows. The second one, which is in the focus of this thesis, is called model checking. The usage of model checking has increased in the last years, but it originates from the 1980s. In 1981 Emerson and

Clarke independently developed temporal model checking, also Queille and Sifakis did this in 1982 [21]. The focus of this approach was on finite state machines and transitions and it utilized Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [7, pg. 16, 314f] respectively. Details on these are given in the following sections.

2.4.1 Introduction to Formal Verification

Ideas

The basic idea behind all types of formal verification is to use a mathematical proof for the correctness of a system. To do so, it is needed to abstract the given system from C code or HDL code or any other type of implementation (also netlists etc.) to a formal representation. In software this is typically a boolean representation (see [24] for a an approach with an incrementally refined boolean model in software), for hardware this is a state transition structure (Kripke structure). For equivalence checking now just another system abstracted in the same way is needed to check against (and probably a rule set that defines the equivalence of certain structures). For model checking, which is the main topic in this thesis, this is not enough. A formal specification of the system, referred to properties, is needed additionally. These properties define the behavior of the system using sequential statements (sequences), implications and time invariant expressions. The properties are then asserted in certain conditions (e.g. reset, system running, special operation mode). The formal verification tool uses this specification to check the abstract model for correctness. This already shows a potential problem - the results of the formal verification will only be as good as the properties. It is one of the reasons to automatically create the properties from the specification itself in this thesis, as this reduces the risk of human errors in generating them (anyone who has ever written formal assertions for a system, no matter if hardware or software, knows what a complex task this can be even for simple behavior and thus it is likely to be error prone).

BDD Solvers

A binary decision diagram is a directed, acyclic graph, that is a representation of if-else decisions. To obtain such a graph from a Boolean expression the If-then-else Normal Form (INF) needs to be obtained. Boolean expressions are commonly written either in Disjunctive Normal Form (DNF) or Conjunctive Normal Form (CNF), but it can be proved, that any Boolean expression can be written in INF by using the Shannon expansion [6, pg. 8f]. The result can be represented as a tree (called decision tree). The nodes of this tree represent the decisions, the leaves (terminal nodes) are the constants 0 or 1. An example for the boolean expression $t = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$ is given in figure 2.8. If the number of subexpressions in this expression (or tree) is reduced by combining the

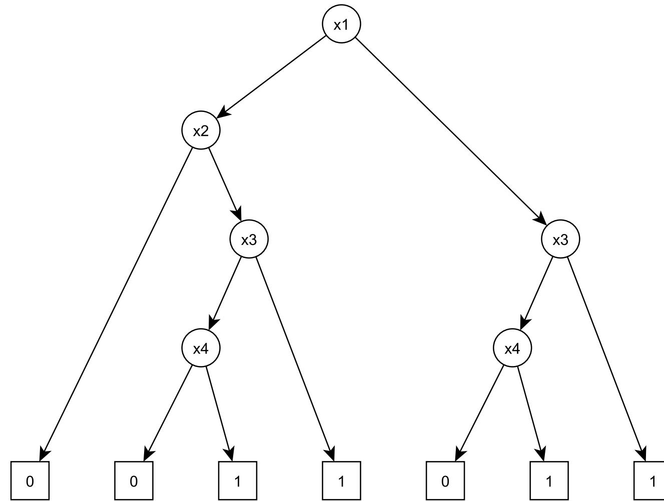

Figure 2.8: Decision Tree for $t = (x_1 \lor x_2) \land (x_3 \lor x_4)$

equal subexpressions the result is the binary decision diagram. For the given example the resulting Binary Decision Diagram (BDD) is given in figure 2.9. The BDDs used for calculations are often special types: Ordered Binary Decision Diagram (OBDD) or Reduced Ordered Binary Decision Diagram (ROBDD) [6, pg. 12], where ordered means that the variables are used in a certain order through all paths of the graph and reduced means that all nodes are unique (no two nodes of the same variable have the same *then* and *else* path) and no redundant tests (*then* and *else* path are the same) are used. One major use case is in formal verification [6, pg. 29]. For ROBDDs efficient algorithms for logical operations on these exist [6, pg. 12] and their nature also implicates that the diagram for two equivalent boolean functions has the same nodes (and is thus the same ROBDD)[6, pg. 13]. With this the benefit of these diagrams in equivalence checking is obvious. For model checking it is checked if for a model M and a set of properties P: $M \vdash P$ (M satisfies P), where those properties are typically specified in CTL (see 2.4.2) and both the model and the properties can be brought into the form of an OBDD.

SAT Solvers

Satisfiability (SAT) solvers work based on the boolean satisfiability problem. This problem describes the following [23, pg. 157]: There exists a formula φ (in our case the property or its inversion); the satisfiability problem on this is the question if a variable assignment on φ exists, which lets it evaluate to true. If it exists φ is satisfiable else it is unsatisfiable. This problem is a known NP-complete problem. NP complete means nondeterministic-

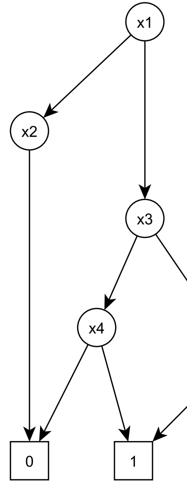

Figure 2.9: Binary Decision Diagram for $t = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$

polynomial complete and is a complexity class for problems typically used in informatics. Polynomial problems can be solved in polynomial time (a polynomial function of the input size), for NP-complete problems a given solution can be verified in polynomial time, but finding this solution might take a lot more time. The NP-complete class is a subclass of NP problems, if any of these could be proven to be solved in polynomial time all would be solvable in polynomial time.
So why care for an algorithm, that will take virtually forever to solve? In practical use, there are lots of additional constraints for certain problems. In hardware everything is finite sized, e.g. memory can not grow in run time and has its size defined by design. There are several other simplifications that can also be applied to hardware problems.
For φ, which is a boolean formula, the typical representation is the conjunctive normal form (CNF). Most SAT algorithms are built up following a similar scheme (see [27]), with which the algorithms search through the state space, trying all variable settings and deducing satisfiability or unsatisfiability from it. SAT based model checking is able to handle very large designs (a lot larger than BDD based) in a very efficient way, but it can usually not provide completeness (it is bound by the length of the counterexample). It is often fast at discovering bugs [13]. This was very well visible when doing the tests with the formal approach, where finding the bugs often took only minutes but proving the correctness took hours or never finished due to reaching the state space exploration limit.

FV in Hardware Verification

As mentioned before, this thesis is about hardware verification. Though formal verification of hardware and software share some ideas, special solutions have been found for the problems in HW verification.

2.4.2 Static Code Analysis

The term static code analysis already implies that there is no real simulation running, i.e. the code does not get executed. It is rather checked in a static state. For equivalence this stays static. For model checking, the start of the state space exploration is this first point in time. The exploration runs on the abstracted model, not the code. From this starting point, the formal static checker tries to find a way to a state, that proves a given assertion wrong. The structure representing this abstract model and the logic types used to disprove the assertion in formal hardware verification will be explained in the following subsections.

Kripke Structure

The abstract representation of the RTL code is, as mentioned above, a state-transition model, usually a Kripke structure. It can be seen as a kind of finite state automaton.
According to [36, pg. 45], a Kripke structure $\mathcal{K}$ for a finite set of variables is defined as $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$, where $\mathcal{I}$ is a set of initial states, $\mathcal{S}$ is the (finite) set of all states, $\mathcal{R}$ is a transition relation (a collection of all possible state transitions $s_i \to s_j$ within this model) and $\mathcal{L}$ is a label function, that maps each state to a set of variables. In figure 2.10 an example of such a structure is shown. It has the following attributes:

- $\mathcal{I}$ = s0 (it starts in state s0, denoted in this figure by a double circle)
- $\mathcal{S}$ = (s0, s1, s2, s3)
- $\mathcal{R} = \{(s0, s1), (s1, s2), (s2, s1), (s2, s3), (s3, s3)\}$
- $\mathcal{L} = \{(s0, \{a, b\}), (s1, \{a\}), (s2, \{a, b\}), (s3, \{b\})\}$.

In some model checking papers and approaches the Kripke structures will not look exactly like the one described here, but have some extensions for modal-μ-calculus or use different definitions (e.g. labeled transition systems with actions) of these, but those can be brought into the form of a Kripke structure again.

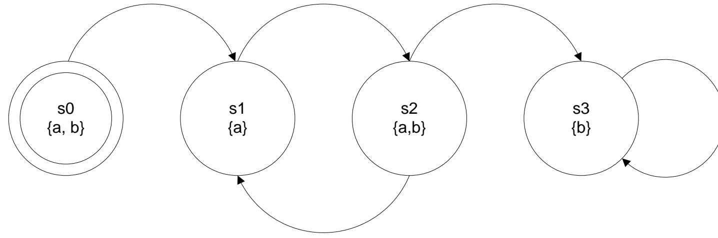

Figure 2.10: Representation of a Kripke structure with $\mathcal{I}$ = s0, $|\mathcal{S}| = 4$

Temporal Logics

Temporal logics are needed to describe the behavior of reactive systems, as their correct behavior also depends on their states and state transitions over time and in certain conditions. There are basically two kinds of temporal logics: linear ones (one future) and branching ones (many futures). The ideas of one and many futures originate from philosophical theories, which go back to Aristotle and the modern interpretation of them was founded by Arthur Norman Prior ([38] and they prove to be very useful for describing certain aspects of system behavior. The two main operators working on both the linear and branching times are the *eventually* ($\Diamond$) and the *always* ($\Box$) operator. *Eventually* describes that at some point in the future the given formula must be true [1]. *Always* describes that the given formula must be true from now on, forever and on all paths.

Linear Temporal Logic

Linear temporal logic is a propositional temporal logic, where for each moment in time only a single possible following state exists. This is a linear representation of time (only one possible future) [7, pg.229f]. LTL uses atomic propositions, basically the same as the state labels (see also 2.4.2), the standard boolean functions (and, or, not, ...) and additionally the two timing operators *next* ($\bigcirc$) and *until* ($\bigcup$). *Next* holds, if the formula it is used on holds in the next step, *until* uses two formulas as input and holds, if the first one holds until the second one holds. With the until timing operator the *always* and *eventually* operators can be created(where φ denotes an LTL-formula) [7, pg. 232]: $\Diamond\varphi = true \bigcup \varphi$ and $\Box\varphi = \neg\Diamond\neg\varphi$ (where $\neg$ denotes the negation)

The first one uses true as formula until φ is active, which will hold (be true) until the occurrence of φ, which means that eventually φ will be true. The second one states that the inversion of φ will be never seen in the future, which means φ must hold from now on until forever, which is the definition of always.

[1] Remark for German native speakers, as this is a common false friend: this word translates to “irgendwann” rather than to “eventuell”

Computation Tree Logic

In contrary to LTL Computation Tree Logic (CTL) is a branching time logic, which means that for every point in time several possible successor states can exist [7, pg. 313ff]. The operators used to express CTL formulae are the two timing operators *next* ($\bigcirc$) and *until* ($\bigcup$), just like in the LTL, and additionally $\exists$ and $\forall$, which have a similar meaning as in normal mathematical functions ("a path exists" and "for all paths") and boolean operators ($\vee$, $\wedge$ etc.). The *always* and *eventually* operators can be derived similarly to LTL [7, pg. 317f]. In other literature (as e.g. in [15]) the operators are defined as **A** ("for all paths"), **E** ("a path exists"), **G** ("globally"), **F** ("eventually"), **U** ("until") and **X** ("in the next tick"), which can obviously be mapped onto the ones mentioned already.

If certain initial conditions for the starting state are given a search through the tree following this state can be performed to prove a condition specified by these operators. With this typical system behavior requirements can be specified and proven with suitable tree search algorithms, for example **AG**¬(read ∧ write) specifies that read and write should never occur at the same time or **AG**(read → **X**(bus_read_ack)) specifies that one cycle after a read a bus acknowledge signal has to be set.

2.4.3 Assertion Based Verification

Assertion based verification has been used for a long time already. About every software developer has used the *assert(false)* statement already to force a program to stop on entering unwanted states. The same assert statement is available in all common hardware description languages. The statement itself does nothing else than checking the boolean expression inside the parenthesis and terminating the program or simulation if it evaluates to false. Usually a message can also be attached to tell what happened. Forcing a program/simulation to stop or at least show an error message that can not be overlooked while executing a normal run of the software or RTL code helps finding errors even if the software or simulation test was not intended to show this. In software as well as in hardware these assertions are not just used in the *assert(false)* style, but for real verification purposes a term gets asserted that checks for example the size of a buffer that should never overflow: *assert(buffer != full)*. Of course buffer and full would have to be defined in a meaningful way to check against them, full would be the maximum capacity of the buffer and buffer the current size. This assertion would be placed in the code just before the additional item is inserted. This is just a simple way to do it. In my experience in software I have always seen assertions as part of the code, so they are placed directly inside the functions. For hardware it is also possible to just insert an assertion statement somewhere inside a process. As an assertion is obviously not synthesizable it will be removed upon synthesis and is only present in RTL. The approach used in hardware is typically to

create at least a second process (surrounded by pragmas to hide it from the synthesizer) which reads the signals of the real hardware process and raises assertions accordingly. For designs nowadays, where the use of Intellectual Property (IP) blocks, which are often even purchased from external companies like ARM, is increasing and the assertions inside a module or entity are shipped with the module and can help to prohibit wrong usage of a design block. Looking into such a design it can be very confusing to see mixtures of code and assertions, so it is another approach to separate the RTL code from the assertions. In the following subsection about SystemVerilog assertions this approach will be shown. The assertion style shown below is really the simplest way to do it. Focusing on hardware a lot of additional assertion functionality has been developed to satisfy the needs to express time dependent behavior, so called sequences. As the style of modeling the special hardware functionality depends on the used assertion language (e.g. SystemVerilog assertions, PSL) this will also be covered in the next subsection.
Assertions have two main fields of application: as assertions running with the simulation in a directed or constrained random test (functional verification) and as the specification language for formal verification.

SystemVerilog Assertions

SystemVerilog is an extension of the Verilog hardware description language. Because of the big amount of new functionalities that it adds it might already be considered to be a new language. The improvements are in the hardware description itself, which are for example the always block definitions for combinatorial, flip flop and latch processes (always_comb, always_ff, always_latch) which allow additional checks on these and implicit sensitivity list generation, interfaces to connect modules, removal of restrictions on port signal types (Verilog could not handle complex types as module ports) and several others. The by far biggest extensions in SystemVerilog are for verification purposes. It adds a whole object-oriented (non-synthisizable) part used for example in the OVM and UVM methodologies. It also adds a native way to express assertions in even very complex ways. It also provides a mechanism to separate the RTL code from the assertions using a so called bind file. For the following subsections the background information was mostly taken from [34], which is also recommended for further reading. Some of the knowledge about this topic was acquired in training courses, but should be mentioned in the book as well.

Separation of Code and Assertions

As it can be confusing or at least cumbersome to have a mixture of code and assertions inside the same module and as the RTL code shipped by IP vendors can be write protected a way exists to separate them. For the automatic generation approach of this thesis it is also easier (and looks nicer) to create a full stand alone file (although it would be possible

to mix manual and generated code). On a side note: For VHDL users that want to use SystemVerilog assertions for their code this solution is also the way to combine these two. The idea behind the bind file is to add the assertions to the code (into the module) without having it really visibly combined for the user. For this approach, three files are needed: a module definition (the RTL code), an assertion module definition (the sequences, properties and assertions) and a bind file. To illustrate this, the SPI-multiplier DUT already shown in the UVM chapter is used. The DUT looks like this:

```
1 module d_spi_multiplier(input reset_n, input clk, input csn, input sck,
    input mosi, output miso);
```

Listing 2.6: SPI multiplier DUT

The SVA file for this, be aware that this module has only got inputs! The assertions watch and check the design, but they must not drive any signal.

```
1 module prop_d_spi_multiplier(input reset_n, input clk, input csn, input sck,
     input mosi, input miso);
```

Listing 2.7: SPI multiplier SVA module

The bind file, which uses automatic name connections of the ports (wildcard .* connection) to bind the input ports of both designs together:

```
1 bind d_spi_multiplier prop_d_spi_multiplier prop_d_spi_multiplier_inst(.*);
```

Listing 2.8: SPI multiplier bind file

After creating this, an instance of the SVA module is attached to the DUT as it is depicted in figure 2.11. It is like an instantiation of the assertion module inside the design, but

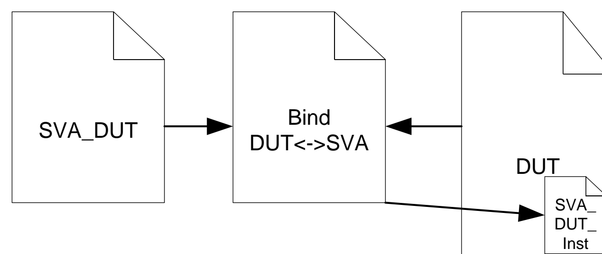

Figure 2.11: Using a bind file assertions can be added to a DUT

without having to touch the design code itself. This also means that internal signals of

the design can be fed into the assertion module using ports (see also [19] for examples on this).

Sequences

Sequences are a way to describe the time dependent behavior of hardware systems. With a sequence definition a pattern is described, which is searched for when using the sequence. It can not pass or fail, only match or not match. As soon as the first expression of the sequence evaluates to true an evaluation attempt is started, waiting for the next sequence expressions to happen. If the first expression stays true or gets true again later another evaluation attempt is started and may be running in parallel to the first one. A typical use would be to describe a handshake protocol: if the signal *REQ* goes high, the signal *ACK* has to be set within 1 to 5 clock cycles and the data transfer will start in the next clock cycle (indicated by *DATA_EN*) and take 3 clock cycles. See figure 2.12 for a graphical representation of this. To describe this, a sequence in SystemVerilog could look like this (leaving out REQ and ACK duration check for ease of reading):

```
1 sequence handshake_seq;
2   REQ ##[1:5] ACK ##1 DATA_EN[*3];
3 endsequence: handshake_seq
```

Listing 2.9: Handshake sequence example for figure 2.12

As one can imagine from this definition and especially the possibility that ACK can be set within a timeframe greater than one clock cycle there are several different ACK-REQ-DATA_EN sequences that match this description. While this is helpful to describe consecutive events that may slightly vary it can also introduce unwanted matches, as one may have not thought about this when writing the sequence. This may in the best case produce a wrong fail of an assertion, which can be checked and found. The worst case is that the design is buggy, but the sequence (or the whole assertion, there are also other possibilities to make mistakes outside of the sequence context) allows this behavior and a real error is overlooked. A little more detail about passes/fails and possible problems with them is given in the assertion subsection.

A look at the figure might raise the question why the signals get their value before the rising clock edges. This is used to avoid reading issues for people seeing sequences the first time. In reality sequence values are sampled one time step before the rising clock edge. This is done to prevent race conditions between the test and design. A flow diagram of the SystemVerilog scheduling is shown in figure 2.13 and the activities for code/assertion evaluation are marked there. The figure is a simplified version of the flow diagram given in [1, pg. 137] with additional annotations for the sampling and evaluation of design information/assertions. For more and in my opinion better explained information on scheduling

Figure 2.12: An example of a typical handshake protocol, two different ways that would both be valid for the defined sequence (there are more valid possibilities), green lines show where the sequence is running, a green arrow where the sequence is successfully finished and red arrows where a sequence checking run is stopped due to a mismatch

and regions in SystemVerilog [18] is recommended for further reading. With regard to the work in this thesis, it is important to understand the fact of the shifted sampling point for being able to read the diagrams in the right way. A signal change at the rising clock edge will be seen at the next rising edge, so a worst-case delay of one clock cycle is possible.

Sequences can also describe time behavior, that can be in an undisclosed future using the expression $. An example sequence looking like this is shown in the following listing.

```
1 sequence open_seq;
2   REQ ##[1:$] ACK ##1 DATA_EN[*3];
3 endsequence: open_seq
```

Listing 2.10: Sequence using an open ended timing

This sequence states: After a request, eventually, but at minimum one clock cycle later, an acknowledge signal is set followed by a data enable that lasts for three cycles. As nice as this might seem at first one problem comes with that: No matter if assertions for simulations or formal verification are used, if a sequence defined this way is used it might not finish until the end of simulation or state space exploration. This will result in a fail

for this test, as the sequence never finished or a pass for this test if it is defined that this sequence should never happen. The main operators for sequences shall be shown here in short (for a complete list refer to [1] or [34]):

Table 2.1: Sequence operators

Sequence Name	Description
##x	wait x clock cycles (can be any number >= 0)
[x:y]	the expression before takes x to y cycles (y >= x)
*x	the expression takes x cycles
[x:$]	the expression takes x to infinite cycles
[=x]	a number of x (possibly nonconsecutive) repetitions has to happen
[->x]	jump to repetition number x (for nonconsecutive events)

The operations mentioned above are for one sequence. There are also operators to combine sequences in different ways like *or* and *and* operations, intersubsections and several other. With those combinations it is possible to specify e.g. different subbranches in a protocol or other complex behaviors of a system. Writing a complex sequence is not a very good idea - it is hard to read and find out which sequence patterns emerge from this and it is hard to debug. It is better to define small, simple sequences and combine them in upper-layer sequences using the special operators or normal wait timings (the wait operation ##0 is used to connect two sequences directly together).

Properties

A property represents a specification of some behavior of a system. It consists of boolean expressions, sequences and other properties. The result of a property is either *true* or *false*, but it is not evaluated on its own. This has to be done by using it in an assertion, assumption or coverage definition (more on these topics in the next subsection). A property can be one of the following seven types [1, pg. 278]:

- Sequence
- Negation
- Disjunction
- Conjunction
- If ... Else

- Implication
- Instantiation

A sequence type property is either an instantiation of a defined sequence or an in-place sequence definition. This property type evaluates to true if there is at least one sequence match, otherwise it is false.

```
1 property prop_sequence;
2   @(posedge clk)
3     REQ ##[1:5] ACK ##1 DATA_EN[*3];
4 endproperty: prop_sequence
```

Listing 2.11: Sequence type property

The negation is the inversion of a property. If a defined property evaluates to true the negated property is false and vice versa. It can be seen as a logical *not* operator.

```
1 property prop_negation;
2   @(posedge clk)
3     not (other_property);
4 endproperty: prop_negation
```

Listing 2.12: Negation type property

A disjunction is an *or*-ing of two properties. It works like a logical or: if at least one sub-property evaluates to true the whole property is true, else it is false.

```
1 property prop_disjunction;
2   @(posedge clk)
3     property1 or property2;
4 endproperty: prop_disjunction
```

Listing 2.13: Disjunction type property

A conjunction is an *and* of two properties. It works like a logical and: if both sub-properties evaluate to true the whole property is true, else it is false.

```
1 property prop_conjunction;
2   @(posedge clk)
3     property1 and property2;
4 endproperty: prop_conjunction
```

Listing 2.14: Conjunction type property

An if ... else property works like the typical if else clause: an expression (property or sequence) is checked and depending on the result the property in the if or else branch is evaluated. The result of the whole property is the result of the evaluated branch. It is also possible to leave out the else, in this case the property evaluates to false if the condition is false (and the nonexistent else path would be taken).

```
property prop_if_else;
  @(posedge clk)
    if (property1)
      property2;
    else
      property3;
endproperty: prop_if_else
```

Listing 2.15: If ... else type property

A property can be used to describe implications and those consist of two main parts: an antecedent and a consequent. These two are connected by an implication operator, which is depicted by /-> or /=>. The difference between these operators is that the first one means the implication has to hold in the same clock cycle, the second one is for the delay of one cycle which would be the same as writing *x /-> ##1 y*. However since it happens quite often in synchronous systems that the reaction to a signal is seen one clock cycle later this abbreviation is quite useful.

```
property prop_impl_immediate;
  @(posedge clk)
    prop_antecedent  |-> prop_consequent;
endproperty: prop_impl_immediate

property prop_impl_delayed;
  @(posedge clk)
    prop_antecedent |=> prop_consequent;
endproperty: prop_impl_delayed
```

Listing 2.16: Implication type property

An instantiation type property is nothing else than instantiating another property inside this one.

```
1 property prop_instantiation;
2   @(posedge clk)
3     instance_of_another_property;
4 endproperty: prop_instantiation
```

Listing 2.17: Instantiation type property

Properties and sequences can be clocked (see above example: @(posedge clk)), but it should rather be left to the uppermost property to define the clock and handle the disabling of the assertion. As a typical property that works at the clock edge has the defined behavior for the active system, most likely the reset state is very different (usually a static value) from this and the assertion would surely fail in such a situation. This is why the construct *disable iff disable_condition* is used do disable assertions under certain conditions. This can be also other states than reset (the design might have additional states like standby-modes, where the design is not active, but keeps the values of the registers that were set when it was sent to standby). The deactivation of assertions can be done in several ways, but for this use case (reset vs. normal operation) it is quite a comfortable and safe way to do it.

```
1 property prop_clocked_disable;
2   @(posedge clk) disable iff (reset)
3     REQ ##[1:5] ACK ##1 DATA_EN[*3];
4 endproperty: prop_clocked_disable
5
6 property prop_reset;
7   $rose(reset) |-> (REQ == 0) && (ACK == 0) && (DATA_EN == 0);
8 endproperty: prop_reset
```

Listing 2.18: Clocked property that can be disabled for reset

The *iff* is not a typo, it stands for "if and only if" (the same as in mathematics and logic equivalence). It means, that the condition before the iff only gets evaluated if the expression after the iff clause is true, otherwise the whole block underlying the expression is not executed. If the right hand side expression is true then the value of the expression before the iff determines the execution of the block.

Assertions, Assumptions, Coverage

As mentioned above, a property itself is passive. To activate it, it needs to be used within an assertion, assumption or coverage block.

- Assert: check that a property holds
- Assume: assumptions for characteristics of the environment (used for formal verification)

- Cover: check if this condition was seen in a simulation run (functional coverage)

A property with an assertion can look like this:

```
1 property prop_handshake;
2   @(posedge clk) disable iff (reset)
3     REQ |-> ##[1:5] ACK ##1 DATA_EN[*3];
4 endproperty: prop_clocked_disable
5
6 assert property (prop_handshake) $info("Handshake ok") else $error("
    Handshake failed");
```

Listing 2.19: Assertion for a defined property

The informational messages (the action block, can be used for more than just messages, e.g. error counters) can be omitted. The property does not have to be specified beforehand, it can also be done inline:

```
1 assert property ( @(posedge clk) disable iff (reset)
2   REQ |-> ##[1:5] ACK ##1 DATA_EN[*3])
3   $info("Handshake ok") else $error("Handshake failed");
```

Listing 2.20: Assertion using an inline property

The assume statement looks the same, but without the action block at the end, so:

```
4 assume property ( @(posedge clk) disable iff (reset)
5   REQ |-> ##[1:5] ACK ##1 DATA_EN[*3]) ;
```

Listing 2.21: Assume using an inline property

The assume statement could be used to tell the formal solver that the signals *REQ*, *ACK* and *DATA_EN* behave like already specified. This can be used to model restrictions on input combinations, sequences or the like. Typical examples are to specify that two inputs are never (or always) active at the same time or that certain vector inputs are inside a range smaller than their possible bit combinations. This can help the solver rule out special cases and thus speed it up (can shrink the possible state space of the inputs) and it removes cases that would fail because the restrictions are defined outside the tested module. Special care has to be taken not to over-constrain a block, otherwise possible faults might be missed. When moving up one layer in the hierarchy the assumptions of the module inputs become assertions for this upper layer block.

The last standard statement is the cover statement. It is used to monitor design behavior and check for occurrences of special states, signal combinations, settings etc. It has

a possible action statement that can be used to increment a counter or set a flag shows a special setting was seen. This is used to measure functional coverage of a design. As functional coverage, in contrary to code coverage, statement coverage and path coverage, can not be done automatically cover statements are used to measure it. SystemVerilog offers other ways of measuring functional coverage too, which are used in combination with constrained random values. Covergroups can be defined to count occurrences of different values, put them into special bins to aggregate events inside defined ranges etc. The whole topic of functional coverage is quite a big field, actively being researched at the moment and lots of different tools already offer different means of measuring coverage. It would be a far too big topic to cover in this thesis, especially as it is not directly needed in here due to the ways the registers are tested, but nearly every verification book mostly focuses on this nowadays, for example [12] and [8]. Another book by Janick Bergeron [11] features SystemVerilog code examples instead of examples in e and is highly recommended for starting off with these topics.

Assertion Usage in Simulation Versus Formal Verification

Assertions and assumptions can be used in both formal verification and simulation. Assumptions are somewhat different for simulations than for formal analysis. For the analysis it is just a hypothesis to prove the assertions. There is no check if they hold. For simulation it can happen that assumed properties fail. It has to be a checked if they hold and a fail is reported (a tool following the IEEE-1800 standard has to do this), but a success is not needed to be reported (decision left to tool vendor). Assertions can be used the same way in simulation and formal analysis, the difference is how the DUT is stimulated to reach a certain state. For simulation this is some sort of testbench, usually with a testscript that might either be a directed test or a constrained random one like an OVM/UVM test.

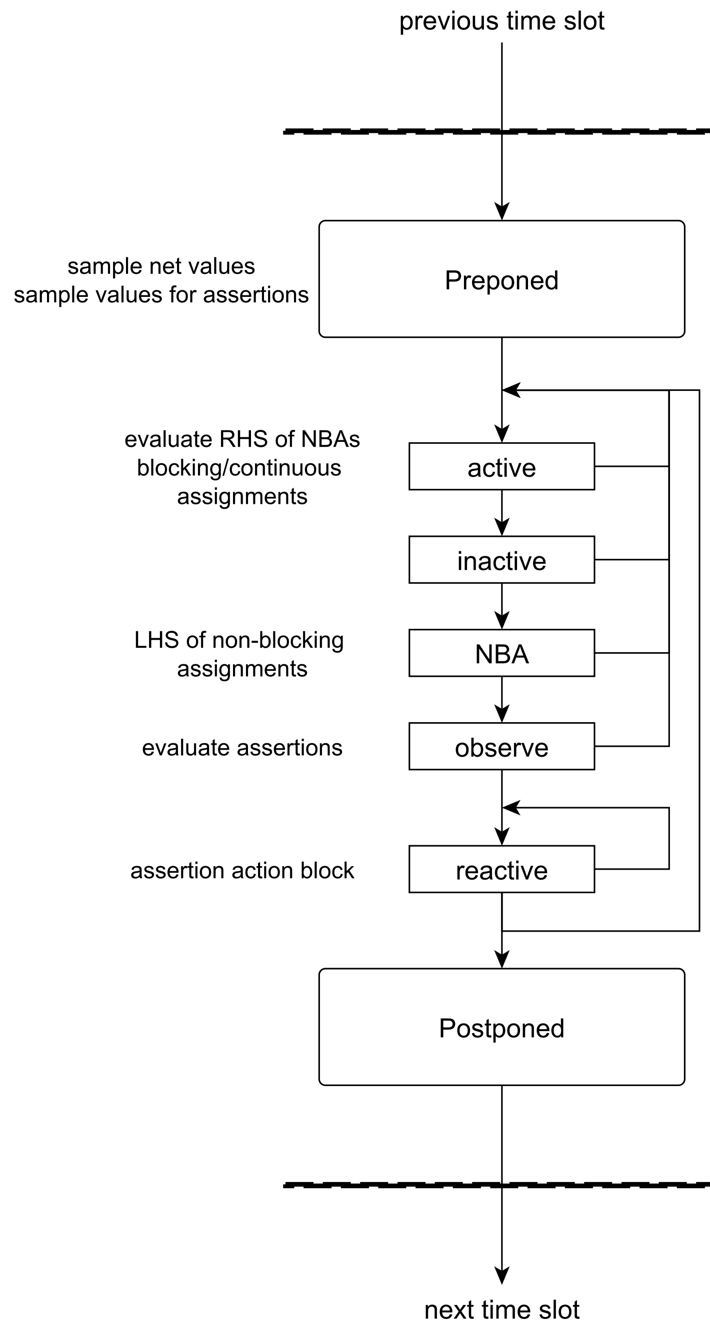

Figure 2.13: The scheduling algorithm for SystemVerilog (with special attention to assertions)

Chapter 3

State of the Art

This chapter gives some insight into related research and literature with the aim of comparing constrained random simulation to formal property verification or with the aim of bundling them into a useful flow with the tests divided into the ones that are the most suitable for a certain verification methodology.

3.1 General

These publications and articles are focusing on a design as a whole, not specifically register accesses, which is a special sub-area in digital integrated circuit design and verification. In figure 2 in [31] a typical functional partitioning is shown, which already gives a hint that register verification can benefit from both functional and formal verification. Still this figure is only an option, as it shows bus protocols as typical simulative use case, but in [9, pg. 822] it is stated, that formal verification proved most useful for verifying the compliance to bus protocols. This already shows a problem in assessing the use cases for the methodologies, as, depending on the field of expertise the verified designs were done in, the results what it can be used for may vary greatly. The best statement to this might already be in [9, pg. 819], where it is stated that verification is “not an exact science”. Also the few papers that actually compare the verification methodologies and not only start from the “well known use cases” or focus on only one of the methodologies in detail were written several years apart, in which time formal property verification as well as constrained random simulation took a huge leap forward, e.g. the SVAs and the UVM were introduced in this time. Both methodologies are currently still under heavy development.

3.2 Detailed Analysis

In the following subsections several papers are presented and analyzed in more detail. In the first two parts papers focusing on one of the verification methodologies are presented (simulative verification including constrained random simulations and formal property verification) and in the third section papers about combined approaches and coverage improvements with these combinations are considered.

3.2.1 Simulation Specific

The following papers focus on simulation based verification like directed tests and constrained random verification.

The first work is [33] and focuses on the software-development intense process of creating a simulation testbench and means to ease the limitations of the reusability of testbenches. In the introduction it is already stated that currently the most widely used verification technique is still simulation with either directed or random stimulus and as the paper is from 2012 this is quite an up-to-date information. This also corresponds to my own experience. They state that in order to perform simulation based verification a testbench and tests have to be created, where the testbench instantiates and stimulates the DUT and also collects the responses. They also state that the biggest amount of time and costs in development are due to verification, which is a problem for more and more complex designs, because this also increases the verification effort dramatically. Possible ways presented as the current solutions for this are the reuse of verification components and raising the level of abstraction. The problem with reuse is that for a long time the components have been written in different description languages, which are often only supported by one tool vendor (however the UVM addresses exactly this problem, but is not mentioned in this paper) and VIP bought from external suppliers needs to be adapted. The alternative, raising the abstraction level, is compared to the development digital integrated circuit design itself has gone through on the way from transistor level to RTL descriptions. On the verification side this means aspect- and object-oriented languages with built-in support of functional coverage collection mechanisms and similar functions. As an additional approach they introduce model driven engineering. In this approach the system is specified as model, which can be described in different levels of abstraction. This abstract modeling is used in combination with code generation and is the same idea as MetaGen, which is the tool used for this thesis. They present a process and tool-set that consists of several meta-models and generators for the e testbench language. The ideas behind it are explained in more detail, which is aspect-oriented programming that can be used in e, as well as an overview of the development of model-driven engineering and the

way this is used. It means starting to create a model, which is done here by using UML (also for a meta-model in MetaGen this is done in UML). The authors use some additional functionality developed before to generate the final e models. The next part is an evaluation of testbench structures and the problems with the rather bad modularization in e testbenches. After this they start the development of their testbench model, which is created in a way that it is able to handle the common testbench structures like agents (see 2.3.1), clocks, signals, scoreboards and many others. The following chapter consists of explanations on how the next steps to generating the code work, which is of less interest for this thesis, as the generation workflow is already defined by MetaGen here. The final outcome in comparison to manually written testbenches showed that the meta-modeling approach can give a huge benefit and sometimes even create better code regarding certain metrics. This experience can be shared with the one using MetaGen, as it drastically improved the effort to benefit ratio for UVM testbenches by using VIPGen, the meta-model for UVM, and this is also the reason to use code generation in this thesis.

The paper by Zhou et al. [42] deals with functional verification with a verification environment created using the object-oriented part of the SystemVerilog language. It uses the approach also used in UVM, OVM and similar methodologies applying constrained random stimuli, layering the verification environment in different abstractions to create reusable verification blocks, using scoreboards to check the resulting DUT responses and the coverage functionality of SystemVerilog to collect functional coverage. The introduction to the paper starts with the current simulation based verification methodologies, where UVM is still missing as the successor of OVM and VMM. In the following chapter the important parts of the test environment methodology are explained a bit further. This is the object-oriented programming approach, the layered abstraction, which facilitates easy maintenance and reusability of the test environment, assertions for dynamic simulations, constrained random stimulus generation and the functional coverage analysis. The verification environment is then set up according to the layered approach, which is similar to the UVM approach. The results satisfy as all the functional coverage points and cross-coverage derived from the requirements are hit several times by the random simulation.

The paper “Functional Coverage Driven Verification for TAU-MVBC” [41] by Aihong Yao et al. also works on a predecessor of UVM, namely VMM. It already includes a lot of the current verification environment ideas and is also based on constrained random stimulus generation and functional coverage measurement. The design verified in this paper is a message sender/receiver and analyzer in a railroad environment. In the beginning they analyze the major possibilities for verification, which is formal property checking and dy-

namic verification. Formal verification is not seen as an option, as the design complexity is too big for it. The constrained random verification based on the VMM is chosen including assertions for dynamic verification. They set up their verification goals by using functional coverage targets derived from the general specification and the detailed design specification. After setting up the test environment, which is layered into test, scenario, functional, command and signal layer, the scoreboard is created to check the DUT responses as well as the assertions, that run alongside the simulation and check signal states, that are hard to describe and check in the scoreboard. For the analysis of the results code coverage and functional coverage is measured. The results show that sometimes the tests have to be adapted to reach good code coverage but it can be achieved as well as the functional coverage, which should not be a problem in most cases, as special tests are written for most of the items.

3.2.2 Formal Verification Specific

The following papers focus on formal property checking and how it can be introduced into the verification flows, as well as what impact this can have and how to measure the coverage in this case.

The first paper is about formal verification used in a practical environment for a networking multiplexer IC [40]. They start off by comparing the currently available verification solutions that could cope with the increasing complexity of designs, which are simulation oriented ones like the coverage driven, constrained random approaches with layered testbenches (as described in the section about simulation based verification), automatic property checkers and improved coverage techniques as well as formal and semi-formal ones like symbolic model checking, simulation-model checking combinations for partial state space exploration (see also [28] which is covered in the next section on combined techniques) and theorem proving at high abstraction levels. The presented way is via model checking with a commercial tool done on a subset of the modules in the design. The reason for using the approach on submodules is the state explosion, which is the main problem for model checking (as mentioned in most of the papers presented here). The design should still be reduced or abstracted to cover bigger subunits and speed up the proofing runtime as well as lower the amount of memory used. The abstraction can be done in a way that if the proof holds on the abstraction it also holds on the original design, if not then it has to be checked where the problem comes from (abstraction or design), see also the next paper for this. The sub-block used for the test is one that is hard to simulate and depending on the abstractions and the property the reachable state space is between 10^{12} and 10^{20}, which already shows a great variation. When they were carrying

out the verification and bugs were found they tried to answer the question, how likely a simulative approach would have reached this. As a directed simulation approach would not hit this without an explicit check on this certain bug a random approach was used for comparison, where even a less hidden bug would take over 24000 input patterns to be found with a probability of 99.99%.

Another paper on this topic, written by Singhal and Aggarwal (the authors of the paper [37], covered in the combined methodology section), is presented in [4]. In contrary to the other paper this one does not have the aim to show how to add some formal verification to the flow and measure the coverage to merge it with simulation coverage, but to replace bigger parts of the simulation with end-to-end formal verification and shift simulation to only toplevel or bigger and more complex modules. Using formal verification for such a task already raises a complexity problem, which is often overcome with some automatically and some manually applied abstractions. It is referred to their previous paper [37] with the information that formal coverage can be gathered very similar to the simulation based coverage and it can also be used to evaluate the usefulness of the abstractions. For the full end-to-end formal verification the most important checkers are the concurrent assertions that are using the inputs and outputs of the module and check them against some abstract reference model. The biggest problem is the complexity due to the cone of influence and the possible state space. This has to be addressed by abstractions. These abstractions should not reduce the design behavior top keep them sound. By proving something correct on this abstraction it is also proven correct on the original design. For a counterexample it has to be checked, whether it is real or due to the abstraction. The example abstraction techniques mentioned are cut-points, where a net is cut open and can take any value, counter abstraction, where big counters are reduced, symmetric data-types, where a reduction of symmetric checks can be done, data independence, where data is only forwarded, stored and used for comparison with other data and so the behavior is independent from it and tagging, where a part of the system is abstracted with respect to a tag. It should also be mentioned here that cut-points were also used in this thesis to reduce the complexity introduced by a communication interface in the real-world design used for testing the methodology. The simulation based coverage metrics can be used in a similar way in formal verification by using the number of cycles of the bound model checker and checking if the code part was executed within these cycles, which has the same meaning as if in a simulation the line of code or statement was executed. The end-to-end verification can now be done in a similar flow as the simulative one by running the proof and checking the coverage afterwards and if the coverage is not over the predefined threshold the checkers have to be improved and another proof run done, else the verification is finished. Their verification of a test design using abstraction

and coverage measurement as described resulted in 15 bugs in RTL, the coverage results are near to one hundred percent. They also tried a run without abstractions but limited to the same run-time as the abstracted and the results were close to zero percent, which means that the abstraction gives a really huge benefit. Without this the use of end-to-end formal verification instead of simulation would not have been possible.

The paper "Strategies for Mainstream Usage of Formal Verification" by Raj S. Mitra discusses the possible ways to implement formal verification within a productive environment [35]. The introduction shows some overview of the current state of formal verification, but less on the technical and algorithm side, but the acceptance in companies and the use in real world projects rather than academic research or test designs. The general problem of formal property verification is a low acceptance of the engineering community even though the tools have progressed by a lot in the last years, which might be due to problems also addressed in some papers here like how to plan formal verification effort and measure it with coverage metrics to use it in a real project schedule. The paper wants to address these challenges in a verification process and propose ways and possibilities for how and where to add model checking in this process. The start is an explanation of a current (simulation based) verification flow and the problems that come with this. A simulative verification, be it directed or constrained random, can never really be complete. The specification points are covered and more depending on the project schedule. The whole verification process is then compared to a productive process, where statistical inspection techniques are used to optimize the process and not to find all defects. Simulative verification has currently another target, which is finding bugs, even though it is just the same as the statistical inspection, which already shows its problem. The solution to this could be formal property checking, but it has acceptance problems also due to some big changes in how verification is done, how the checks are written and how everything can be planned (or not planned). The first problem is that assertions for model checking often have to be different from assertions for simulation, so they do not increase the number of state elements. The limitation to smaller modules is a problem mentioned in nearly all the papers, it also adds the problem of having to add quite some constraints to these smaller modules and a lot of effort has to be put into debugging these in the startup phase of the verification. The limit in complexity can also lead to the problem that a module can just not be verified, which is not known from a simulation based verification (where there is always some way to test its function, at least with a few patterns). Also the missing metrics for progress in formal verification are a problem, but as seen in other papers this is already being addressed. After the drawbacks of formal verification the paper is continued with its benefits. The completeness, which is a benefit always mentioned, is put into perspective here, as due to the real-world tool limitations this is not possible and the design has to

be constrained heavily to make it provable, which in turn makes it incomplete again. The possibility for complete verification only holds for very small designs. The benefit formal verification still has compared to simulation is that, even if it also might not be able to find all bugs, it is able to find them faster and due to not needing a testbench also with less initial effort than simulation and possibly also in early states of the design. The question raised in this paper is also who the formal verification user is - is it the verification engineer or the design engineer? They consider three different use-cases to answer this question. The first use-case is the designer, who writes the white-box assertions to check his module before handing it over. The second one is automated formal verification, where predefined assertions (e.g. for standard interface protocols) and built-in checks of the formal tools (like state reachability, dead code, combinatorial loops etc.) are used already in the early design phases. This can also be done for special checks on system level, e.g. connectivity checks, which can easily be done even on such a high level as it is like a little module with a small amount of state elements that is left if all submodules are blackboxed. Another automated use is bug-hunting with semi-formal technologies. The next use-case is the deep formal verification, where end-to-end assertions are written (see the previous paper), which is also able to find corner-cases, but it is the most difficult method that also needs experts for it. So the benefit of formal verification is not finding more bugs, but doing it faster, which is also what should be pointed out to people that say that simulation could have found that bug too. This is also what the paper continues with: strategies on how to employ formal verification in a team, how to make the transition as easy as possible and how to show people the benefits. These proposals are using assertion based simulation before using real formal verification to get used to the way this is done, using automatic methods that can easily be implemented with nearly no additional effort and already prove helpful, creating reusable assertion packages for standard protocols, checking connectivity, setting up a central expert team, getting designers to write whitebox assertions and using semi-formal methods.

3.2.3 Formal Verification - Simulation Combinations

One paper taken into account [9], originating from 2002, is a comparison of directed testing, constrained random simulation, called pseudo-random testing in the paper, and property checking. At the time the paper was published the first constrained random methodologies emerged, which are also the ancestors of the UVM, but most verification at that time was done using directed testing. The three methodologies were not introduced at the same point in time in this project, which might make the comparison a bit harder, but were compared in the end based on several bug types rather than on different metrics as it will be done in this thesis. The results show the biggest amount of bugs were detected by the

random stimulus methodology. Formal property verification worked, as mentioned, best on bus protocols. The drawbacks found for formal verification were an excessive runtime for certain test points and, due to this, also a problem with finding livelock or deadlock bugs. Registers and accesses were not an explicit category, but bus protocol checks could be seen as something very similar to the register access tests. The result on these methodologies is that all of them have their strengths in different areas, only random tests could be translated back to directed tests, which would result in a huge amount of these, or also to properties, but time and resource limitations make this unfeasible.

The next comparison from 2007 already focuses on formal property checking and constrained random simulation, leaving out directed tests as an own method [31]. It starts with listing some benefits of constrained random verification, especially the possibility to do black box testing. For formal verification it is nearly impossible to write a test without knowledge of the design itself, thus making it a whitebox test. The next drawback for model checking is the practical limitation to module level verification. After the general view on some typical system modules a case study is presented, where both methodologies are compared. Unfortunately they were also not introduced at the same time, but random simulation a long time before formal verification, which makes the comparison on caught bugs a bit unfair, but it shows that after reaching the plateau of detected bugs over several weeks (usually this is a criterion for a tape-out) the formal verification was able to uncover additional bugs.

In [28], a paper from 2008, the aim is setting up a test-suite using a special verification plan language that uses both formal property checking and simulation to achieve a high coverage in a small number of simulation cycles. After a practical test on a test design the conclusion is that formal verification only works on a limited cone of influence (e.g. module level test and assumed port behavior), but that it is very good at targeting complex corner cases, which is hard for directed and constrained random verification. The results also show that a combination of formal verification and simulation provides the best results, with formal verification as a tool to also find complex corner-cases, which are hard to hit by just using random simulation.

Another paper focusing on the combined use of formal property verification and simulation to improve coverage considerations is [14]. Already in the introduction the restriction to module level is mentioned. As their methodology is for early design stages this is clear (the top-level is created in a later design phase in a bottom-up design implementation, which is the standard in most design flows), but it is also helpful for restricting the cone of influence. The methodology itself is an automatic way of generating (simple) formal

properties to find out if a hole in the code coverage is due to real unreachability (e.g. default statements or else paths are often unreachable, as they are only safety statements which should never be triggered in a normal operation case) or if it is due to a hole in the verification. The way to do this is automated starting from just the coverage reports rather than having to parse the full code or having the need for additional manual changes to be done to the code and also without having to recompile the code for every item, which are a few of the problems with similar methodologies in related work. Their approach starts by identifying an uncovered part of the code (it works for line, expression and also toggle coverage) using the coverage report of a standard coverage collection tool. From this the condition that would be necessary to reach this code fragment can be found. Using temporal induction it is tried to prove that this part of the code is unreachable. One property for the start of the induction is created and one for the next step (the step from n to n+1) and they are asserted and checked with a normal formal prove engine. If the code is not unreachable the properties will fail and then it has to be checked (manually) why this part of the code was not covered. The experimental results were obtained from the verification of a micro-controller and showed, that this combination of simulation with automatically generated properties improves the possibility of finding real coverage holes and with this bugs in the design or testbench a lot. The number of uncovered statements that were filtered out with this methodology by formally proving their unreachability is over ninety percent for both statement and branch coverage and nearly seventy percent for (focused) expression coverage, which means a huge reduction in the manual work.

The paper by Casaubieilh et al. is about the functional verification of a processor [17]. This is done using techniques to verify the VHDL specification and the circuit level. The design is modeled in different abstraction levels, starting with a C simulator, then a behavioral VHDL model followed by a structural VHDL model and finally the transistor level circuit. This style of modeling in the layers is quite common for processor development, where the C model can also be used as a simulator for the software developers, only the behavioral VHDL model is usually not synthesizable, which it is here. Before their simulation based verification starts, designers check their block (some basic module level functional tests, but not a real verification) and the real verification is then done at the toplevel. The paper features different approaches of doing so, one is a normal VHDL testbench including VHDL models, the other is a hardware emulation on an FPGA. As only the simulative and formal verification are of concern for this thesis the emulation will not be considered any further. The simulative verification is based on generating certain test vectors with a special tool and applying them to the DUT (the C and VHDL model). Code coverage was collected to find missing cases. For proving sequential properties formal property checking was used. For the reduction of complexity (it is mentioned again

here that formal property checking is very limited by the circuit complexity) an additional in-house tool was used. The mentioned findings with formal verification were due to assumptions of the outside of the module not being accurate enough. The conclusion (for the non-emulation part) is that the use of simulations with accompanying formal verification works well, especially as their methodology can be used at early stages in the design already.

The paper by Singhal et al. [37] is the next one to combine simulation and formal verification by means of collecting coverage. It is also mentioned that formal verification is now in a state, where it can complement simulation and also replace it for certain tasks. The introduction shows again some figures that point out that most of the effort in a design is in verification and that this is still mostly simulation based. To have formal verification as a signoff tool for tapeout it is required that formal verification can be planned and the progress can be measured and combined with simulative coverage results. The aim of the work described in this paper is to integrate the sub-block verifications done with formal means (as formal cannot be used on toplevel or complex modules) with the simulative ones. After the following overview of the design and verification flow it is concluded that it is necessary to use a tracking metric to take corrective actions early enough. For simulation this is typically code coverage, especially line coverage. Lines that are not covered need to be checked manually for the reason, i.e. if they are really unreachable or a problem exists (a step into the direction of partially automating this is given, as mentioned, in [14]). Problems and missed bugs in formal verification in a real productive use are, according to this paper, often bad constraints, too many constraints or because bound model checking was used (which is the most common way in current formal engines) and did not go through far enough to uncover the bug. For model checking the same metrics can be applied to judge if it is complete, just with a slight variation in how they are measured to make sense. For bound model checking it can be used if it can be reached in a certain number of cycles. The conclusion of the paper is that the best verification flow is a combination of formal and simulative verification, especially as it can be possible to merge their coverage information.

3.3 Conclusion

The presented papers show an overview of the current state of verification in real-world use and the different approaches and ideas on how to integrate new ideas into todays verification environments. The amount of simulation based papers is not too high, as most of the field has been covered already a long time ago and also it is easy to understand how it works and also the new ideas, at least from the idea of the tests behind it, are easy to

grasp. The big change for simulative environments is the shift to more complex System-on-Chip (SoC) designs and the need to handle the even more complex testbenches and tests and an overwhelming amount of possible interdependencies between modules to achieve the set code coverage and also functional coverage targets. The way to address this is the development of reusable verification IP and the use of constrained random stimulus generation. The need for these led to the development of the layered verification environments present in most flows and methodologies currently used, which provide easier maintainability and sometimes even the reusability of full testbenches due to smart configuration capabilities. It also led to a new way of coverage driven verification, as the use of random stimuli creates a problem with predicting the outcome like in a directed test, where it worked by setting one signal and comparing the outputs to known values. The new way is to define what input signals fulfill a certain required state and which output signals (or also internal signals) show a certain state or function and also the sequence between those shows that a certain functionality has been seen. These functional coverage items must be collected somehow and compared to what they should be, which only works for random signals by having some kind of abstract model of the DUT, which is commonly known as scoreboard in all the methodologies. The benefit of the simulative approaches is that the verification can be planned, as there is nothing that cannot be verified in some way (maybe not in an exhaustive way but at least with a set of patterns) and also tracked by code- and functional coverage metrics. Also the hardware development industry is rather conservative when it comes to the implementation of new and different methodologies and it takes a long time until something new is accepted. The formal verification approach also has some problems still, especially the inability to handle bigger modules and toplevels still makes a simulative approach for parts of the system verification necessary.

The papers on formal property verification show some comparisons to the simulative approach and try to work out the benefits model checking has over simulation and how to approach the apparent drawbacks of this verification methodology. They are also meant as a guide on how to deploy formal verification within a productive environment, which already shows that the acceptance is still low. Formal property verification still has some problems that were either not solved up to now or it still has a bad reputation due to limitations of early tools (which are already solved or eased, but it is not tried again). The most important one and therefore always mentioned is the limited complexity of designs that can be handled with it. This shifts the use away from a full-chip verification towards a module level approach. Certain tricks still provide a possibility to verify at least some things on toplevel, like e.g. connectivity. The problem with the complexity has two direct effects, one is the runtime, as even for not overly complex designs it is often quite high, the other one is the problem that a design or module might just not be verifiable with

property checking. The latter one is a big problem for deploying it in a project, as it makes it impossible to really plan it. Also the runtime problem creates a problem in planning, as it is hard to estimate it upfront, especially if not so much experience has been made with this kind of verification. A way to bypass these two problems or ease the problem is abstraction. Parts of the design are replaced by some model of it or are constrained to have a certain behavior so different properties merge into one. There are automated ways to abstract some parts of the design, but the most effective ones are usually manual abstractions. There are a few common ways of abstraction mentioned in one of the papers, the most important part is that the abstraction should be done in a way that the abstract model does not reduce the design's behavior (i.e. it can add states but not remove them, like e.g. cutting a connection open and allowing it to assume any value instead of only a discreet set of values). This makes sure that a property that holds on the model will hold on the actual design too and only for a failing property it has to be checked if the design fails or if it is a problem in the model. Another problem is a metric that is needed to trace the verification status. For simulation code coverage is a widely accepted way to do so and also functional coverage is used more and more. For formal property checking no widely used standard exists. Some of the papers provide ideas on how to trace a similar kind of coverage to measure it, which is now also starting to be used in various tools. Another problem of model checking is the unclear situation on who should be responsible for the verification. For certain properties to be specified in a way that all corner cases can be found knowledge of the design is needed, which makes it a whitebox verification. This should typically be handled by a designer rather than a verification engineer. Writing the properties in a way that can be handled by the prover tool with the least possible effort requires quite some knowledge about this very special kind of verification and often also the tool behind it, which might better be done by a central expert group. The real benefit of formal property checking is not finding all the bugs, as simulation will also uncover these given enough time. The real benefit is that it is able to find bugs faster and with less effort for the verification environment, which makes it possible to use it already in the beginning of the design phase.

The third section of papers are approaches on how to combine simulative verification methodologies with formal property checking to create a verification suite that combines the best parts of the particular approach and tries to minimize the problems of weaknesses of one methodology through the help of the other one. The combination of the verification methodologies is also a good way to start using formal property checking within a productive environment, as it can be introduced step by step by increasing portions of verification done using model checking, as soon as the verification and design engineers get more familiar with it. For the simulations constrained random should be used in

combination with functional coverage for toplevel and complex modules, smaller modules should be done using formal methods. The papers also feature comparisons of directed tests, constrained random simulations and formal verification on different designs with an amount of bugs and what kind of bugs these were (grouped into rough categories). The problem with these comparisons is that for all of them the introduction was done at different times in the project and thus not directly compared on the same bugs. One result was that after reaching a situation where no bugs were found with simulations for a longer time the introduction of formal property verification uncovered additional bugs. These were special corner cases that would not necessarily have been seen in the application use, but still would have slipped through unrecognized. The papers also offer some ideas on how to combine the methodologies, one idea is for example a testplan language capable of combining them in a way that minimizes the number of simulation cycles. Another one is to use formal property checking to automate the manual check of code coverage holes, where normally every statement that is not covered has to be checked if it is really unreachable or has just not been hit. The approach extracts these holes from the coverage reports and creates properties that check the reachability, which reduces a big amount of holes and just a few are left for manual inspection. If formal verification is used in addition to simulation a way has to be found to specify the coverage of formal verification in a similar manner as the code coverage for simulation. Some proposals are made on how to do it in a very similar way that has the benefit that the results can be merged easily. Another benefit of the combination is that formal property checking can be done without testbenches, which allows an early use in the project before the complex random testbenches are set up and also some of the checkers can be reused along simulation.

Chapter 4

Flow Design

In the following sections an overview of the currently existing flow is given, followed by the additions and changes to it and finally the concept of the UVM based flow and the model checking based flow.

4.1 Current Design and Verification Flow

The current design flow starts with the design entry and setup, in which settings for the modules and dependency analysis are prepared and the design itself is created (by writing HDL code). After this the RTL design is simulated. The standard environment is mainly set up for directed tests. As this flow was already set up a longer time ago constrained random simulation is not directly integrated. If the RTL design is verified it is also the input to the formal verification, which is only a logic equivalence check (see 7.1.3) for comparison with the synthesized and the routed netlist in the original flow. If the design is not verified correctly an iteration back to the design entry is done and the bugs are fixed. If the full design is verified it is synthesized and after this placed and routed. The simplified flow overview is given in figure 4.1.

4.2 Flow Extension

As MetaGen is not part of the current flow an extension was needed for its integration. It would normally be part of the design entry phase, but as it is currently an unofficial addition it has to be seen as extra part. It gets inputs from and has outputs to the design entry, as the information about ports, registers and the system structure are now entered into the (filled) meta-model, passed to MetaGen and in there to predefined generators to generate code which would normally be written manually in the design entry phase. This generated code often features the possibility to add or extend it with manually written

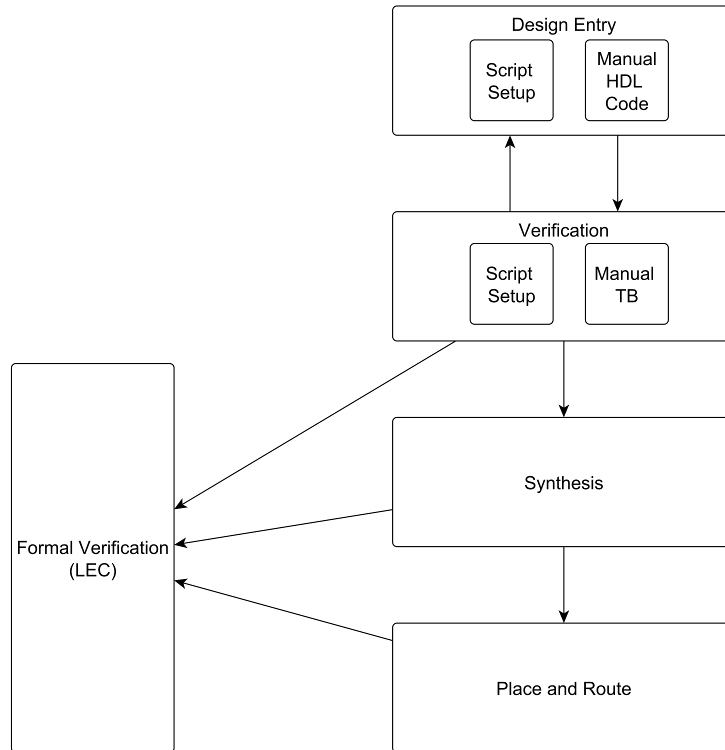

Figure 4.1: A simplified overview of the currently used design flow

code, so this has a connection to the manual coding and is not just stand-alone as e.g. the scripts. The information stored in the meta-model is now used to create the UVM testbench using VIPGen (semi-automated process), as well as the register model, which is mostly an automated process, and the property file and the bind-file for model checking. Manual extensions to assertions and also new assertions can be added into the generated property file easily. As the formal verification in the flow is up to now only the automated equivalence check, model checking has to be added as a new substate of the flow. In figure 4.2 the new flow can be seen. It features new stand-alone states as well as additions to the current states. Model checking has a transition back to the design entry just like verification, as an uncovered bug will have to be fixed in the RTL code.

4.3 UVM

This section describes the prerequisites to create a register model according to the UVM standard and the ideas on how to solve problems arising due to missing information in the Essence meta-model and how the result should be used within a UVM testbench generated

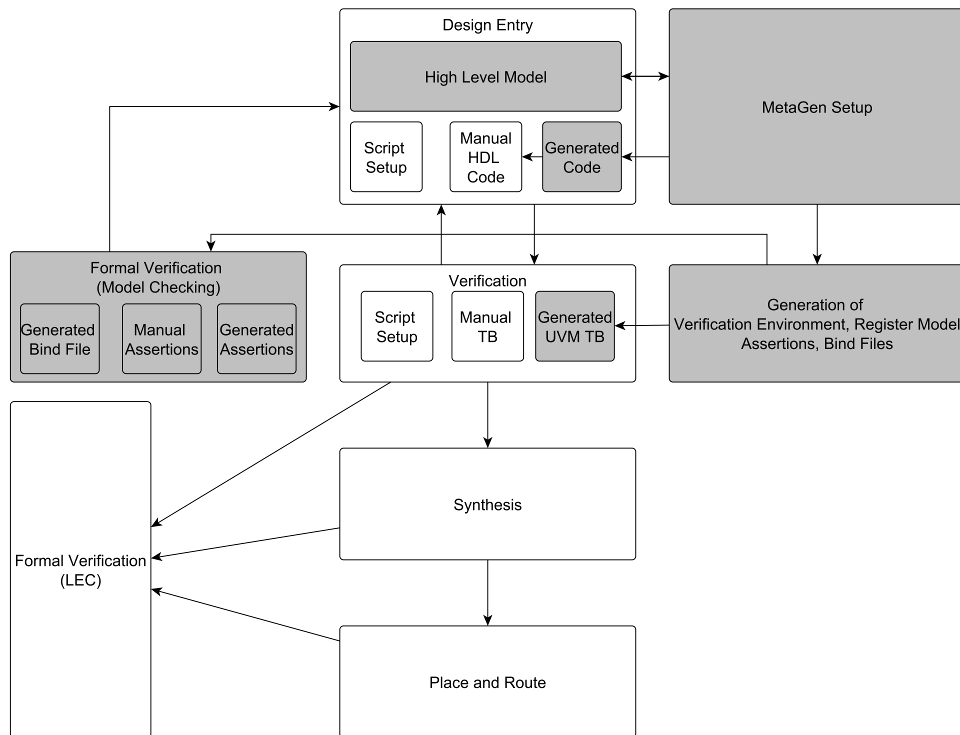

Figure 4.2: A simplified overview of the new design flow using meta modeling and code generation

using VIPGen.

4.3.1 Automatic Generation of the Model and Tests

To enable the creation of a push-button solution for register tests the model has to be generated automatically from a high level description. The framework behind the generation is MetaGen, as described in chapter 2.2. The standard tests can be done using the predefined sequences for reset value verification, bit-bash test and also frontdoor/backdoor access tests. This reduces the needed work to the generation of the register model according to the UVM standard. Additional custom register sequences can be generated using VIPGen, as they are just standard sequences using the register model to access the registers of the DUT.

Parameters Needed

Analyzing the register model as defined by the UVM standard results in several parameters that are needed to express the organization and behavior of the DUT registers. Of course

the names of the registers and bitfields should be known, especially for the registers, as those are used directly in the register sequences. The lengths of the bitfields and their position in a register are also needed, as well as the access policy and the register reset values. For automatically including the mappings some information about the interface(s) is also needed, like the bytewidth and offset and the register addresses for the interface and which ones are to be mapped to which interface. The Essence data model is capable of storing and providing the information on the register structure as well as the interface to register mapping. For the register model with additional custom register classes used in this thesis to provide easy-to-use multi-view registers (i.e. registers that change their content or the representation of their content depending on other bits or signals in the system) some information is needed that describes the currently selected view and also a way to store the selection and representation. The multi-view register is a register type that is already available in the Essence model, but there is no way to store the selector - view pairs. As a result an extension to the Essence meta-model is needed. The extension is an additional meta-model that is referenced from a class in another meta-model. This is possible for every class in the Essence meta-model and in this special case every object of type multi-view register has a link to an object structure of type multi-view extension. The multi-view extension needs to hold the information on the selected view and the selection signals. A typical situation is that a register has two views, which are selected by one bit in another register (e.g. a lock bit that switches a register between read-only and read-write or a test-mode bit that makes a register accessible if it is set). But this restriction is not always applicable, in principle there can be many views selected by a combination of several signals. There are different ways to address this topic, the way chosen for this thesis is the following: A multi-view register has a number of views. If a view is not really existing in-between it has to be modeled as an empty register still. As a selection value a number of bitfields have to be passed and their combination (the first one is used as least significant bit or bits for multi-bit fields and so on) is a new number that defines the index. The result of $2^{numBits} - 1$ has to be greater than or equal to the number of views. The model is shown in figure 4.3.

Register Model Structure

The register model can be structured in various ways. The way chosen for this is to have a register block as the topmost container. Within this container the registers could be directly present. It is an easy way to represent the model in case all registers are present in one module. As soon as the design contains several modules, which all can fulfill independent tasks and thus have registers with independent functions, it is more modular and also readable if the registers for the certain module are structured as an own

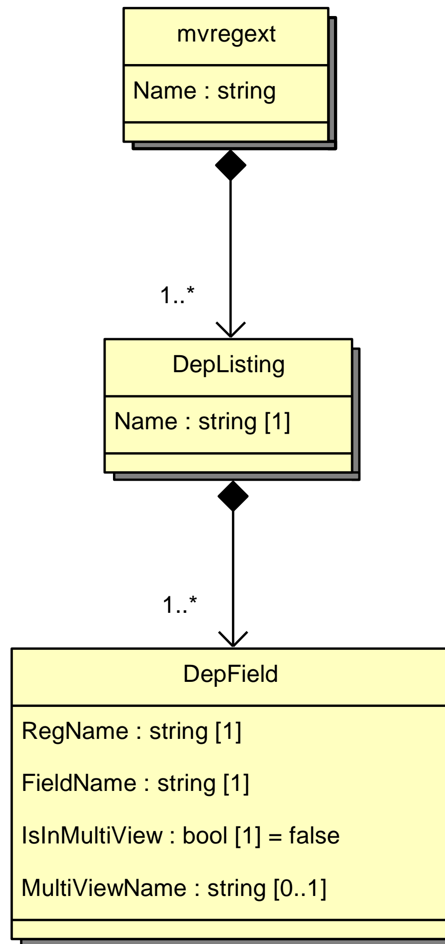

Figure 4.3: Multi-view register extension

register block and bound into the parent block. An overview of the register model file is given in 4.4.

The multi-view register has to be a special register class. An include file is needed to provide classes for registers with special functions. The multi-view register is just one possibility, others might be added there as well, like registers with dependencies between each other (e.g. start calculation bit and result register). The multi-view register consists of an array of register references as well as of a list of index selection bitfield references. The bitfields are part of other registers, so these other registers are instantiated and the addresses of the bitfields passed to the selection array in the build phase. The registers representing the views of the multi-view register also have to be instantiated and the addresses passed to the view array.

Another problem of the Essence model is the missing possibility to express all access

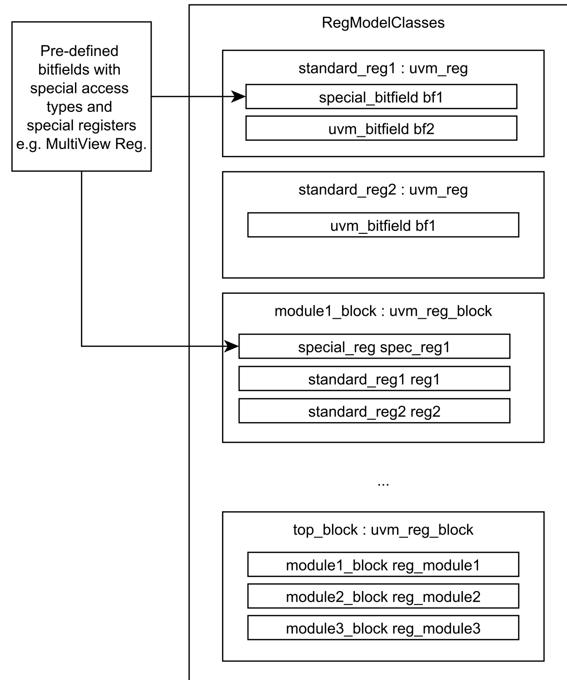

Figure 4.4: Register model file structure

types specified by the UVM. The solution to this is an extension that holds a string representing the access type, as this is an easy, but nonetheless efficient way to do it and also easily expandable. The meta-model is shown in 4.5.

4.4 Assertion Generation

The second practical task for this thesis was to develop a possibility to automatically generate assertions for formal verification from a high level specification. The assertions should be able to check the behavior of a register bitfield with a predefined access type when accessing it. The meta-models for the multi-view register type extension and the access type are the same as for the UVM so that the files for both methodologies can be created from the same (filled) meta-model. As the formal analysis works best on smaller designs it is recommended to do it on module level rather than on system level. Depending on the complexity of the access interface it might also take long to explore the whole state space, which can result in long run-times. This is why it is better to use formal verification or using another method to verify the interface itself and do the formal verification with the internal bus as register access medium. Usually the interface to the outside world is a

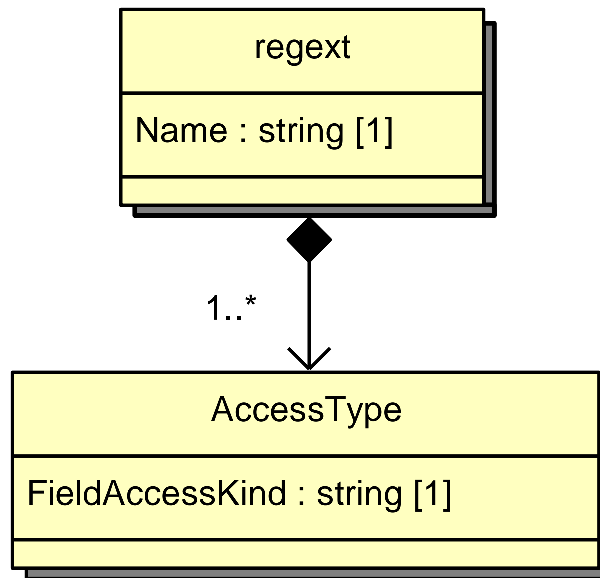

Figure 4.5: Register access type extension

more complex, less pin using and thus serialized one, whereas the internal one is typically a parallel bus with a data width of the size of the full bit width of the registers or a multiple of one byte.

If the assertion file is not bound into the module containing the registers (e.g. because it is used on top level), the HDL path to these is required. For every bitfield a definition is created, which has to be filled out by the user. The problem of finding this path is that a system model with known (and standardized) generated signal names would be needed to trace it through the hierarchy. This way, and this is a possible extension for the future, it could be generated, but only for Verilog/SystemVerilog. As soon as a language boundary is encountered (e.g. crossing to VHDL) there is no standard on how to access the signals. Every tool vendor has a solution for this, but everyone of these is different. If the design is tested on module level the problem does not exist, as internal signals can be fed into the assertion block (see [19]).

As the formal proof sees all possible ways a signal can be set at any time another problem emerges. Usually registers like ADC measurement values are called read-only fields. In reality they are not totally read-only, but can be written internally. So the read-only access type is only valid as the view from outside. For a register test with the UVM this is not necessarily a problem, as it can be decided when to look at the field. There are two possibilities to solve this: declare read-only as usable only for fields that stay with their reset values forever and define new bitfield access types for those that are

internally updated but externally read-only or define read-only registers in a way that a write access on them has no effect (in SVA this means to check if *$stable(register)* is true after a write access). As the term read-only and the use of it also for measurement values and similar bitfields is very common and it is hard to get every system-level engineer to use the read-only access type in a rather unintuitive way for him/her the second option was selected.

4.4.1 Property File

The property file is divided into several subsections. The general partitioning in this file is that one property module for every design module with accessible registers is created. This module is bound into the design module using the bind file, which is explained in further detail in the next section. The property module definition starts with all ports, which means all input and output ports of the original design are defined as inputs (a property module is always passive) and additional inputs are created for every bitfield. Then the definition part is present, which is needed to assign the reset, clock and bus signals to the macros used in the properties. This also includes access sequence definitions to distinguish between read and write accesses. This part has to be completed manually as the generators and models do not know about these signals and sequences. Afterwards the bitfield size, position and bus address are defined, which can be done automatically, as all the needed information is present in the meta-model. After this the assertions for every bitfield are placed, starting with the reset value assertion and followed by all the assertions needed for a certain access type of a bitfield. These assertions rely on the predefined macros (filled out manually as mentioned before) and are fully generated. The only assertions that need manual adaptations are default read and write assertions for unknown access types. A graphical overview is given in 4.6.

4.4.2 Bind File

The problem with the toplevel verification of VHDL or mixed-language designs has to be solved by an approach using one property module per design module with user-accessible registers to allow access to the register signals. All of the inputs and outputs of the design module are fed into the property module to have all signals possibly needed available for the interface protocol, reset and clock definition and also for the manually added module assertions. The connections of the design module ports to the property module ports is automatically generated for the bind-file. For every bitfield a way to connect the data used in the assertion to the real design signal must be present and it is not possible to do this automatically for now. It can be done if a generator is used to create the register definitions in the design and the signal names follow a certain rule, which was not the case

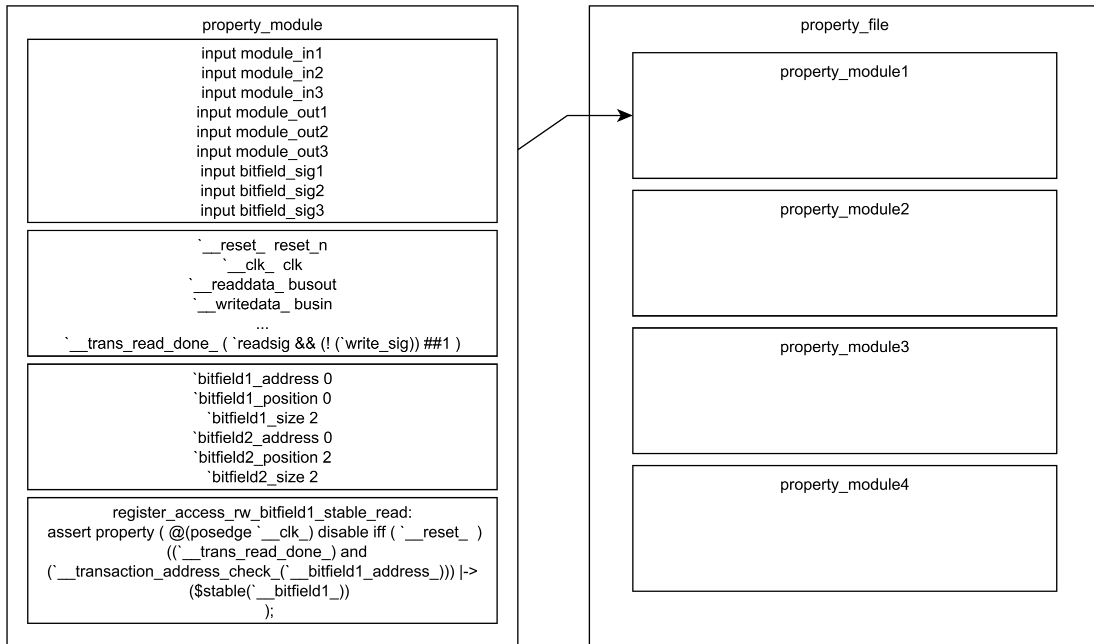

Figure 4.6: The structure of the property files

when the thesis was done. As a solution to make the manual connection as simple and fast as possible (and also easily expendable for the automatic connections if the names are known) an input port is created for every bitfield. This way full signals, parts of vectors, sub-elements of structs or records and so on can be passed into the assertion module.

Chapter 5

Implementation

5.1 UVM Implementation

This section contains a step-by-step explanation of the implementation by showing the workflow on a sample design. The simple example used here is the same shown in the formal verification part and is the least complex methodology evaluation design. It is a multiplier with an SPI interface and six registers: a control register with a start bit and a finished bit, two registers for the operands with 28 bits and two result registers for the high and low part of the result with 28 bits each (the result of a multiplication of two n-bit values can take up to 2*n bits). See figure 5.1 for a graphical representation of the registers. It also features an additional testmode register, which has no real functionality in this design, but is meant to show the functionality of the custom multi-view register type. There are five different access modes: "read-write", "read-write, cleared by design", "read-only, set by design" and "read-only, set by design, cleared on read" and "read only".

5.1.1 Starting in Excel

The flow for the UVM register model generation starts at the high level description of the registers in Excel typically done by the concept engineer. An excerpt can be seen in figure 5.2. The needed parameters (bitwidths, access types, reset values etc.) can be set in this standardized sheet. The register bitwidths in this sheet are greater than the ones really used in the design. This is due to a restriction to several predefined bitwidths of the sheet, which will be changed in a later revision of the Excel workbook. Looking at the figure it should be rather self explanatory. The workbook and importer for it were not created by me for this thesis, but were already available from another department. Due to certain restrictions and slow performance of the workbook due to macros this step was skipped for most of our projects and the data model was filled directly via the Essence component builder graphical user interface (a special GUI for the Essence data model).

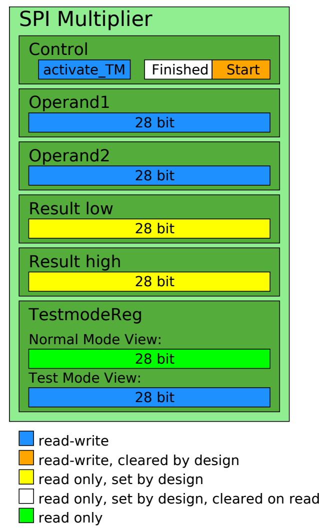

Figure 5.1: The registers of the SPI multiplier

5.1.2 Filling the Data Model

This Excel sheet is now read in and fed into the data model, like in the example in 2.2.1, by using an importer written in Python and the Essence data model. See figure 5.3 for a screenshot of the Essence component builder that shows the filled model.

5.1.3 Generation of Register Model With Mako Templates

As the register descriptions are now in the Essence data model from which the register model is generated, it is robust against structural changes in the Excel sheet, so only the importer needs to be adapted. The templates for the register model are split into several sub-templates to keep the templates more readable and reusable. One template is the main template calling the main routines of the sub-templates in the order specified in the UVM design chapter. In the case of this simple example no system description is available (one-module-design), so only one main register block is present and one for the multi-view registers (as the multi-view registers consist of possibly several sub-registers and an index register it is combined into a register block for the ease of use). The main block template (which utilizes sub-templates) is shown as an example in the code listing below. This template is also called from a higher level template, which calls the templates for register instantiations, multi-view register instantiations as well as this one.

Register name Bit name	Address	Resetvalue	Register Description	Moduls
MultCore_Control	0x0	0x00000000	Control Register	MultCore

	Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25	Bit 24	Bit 23	Bit 22
	n.u.	n.u.	n.u.	n.u.	n.u.	n.u.	n.u.	n.u.	n.u.	n.u.
Resetvalue										

Bit name		
Start 0:0		
	n.u.	
	n.u.	
Finished 1:1		
	n.u.	
	n.u.	

Register name	Address	Resetvalue	Register Description	Moduls
MultCore_Operand1	0x1	0x00000000	Operand1 Register	MultCore

	Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25	Bit 24	Bit 23	Bit 22
	n.u.	n.u.	n.u.	n.u.	OP1	OP1	OP1	OP1	OP1	OP1
					rw	rw	rw	rw	rw	rw
Resetvalue					0	0	0	0	0	0

Bit name		
OP1 27:0		
	n.u.	
	n.u.	

Figure 5.2: Excel description of registers of the SPI multiplier

```
1  <%def name="generateBlock(RegMemSets, Interfaces, dataInput)">\
2  <%
3    MVBlocks = []
4    for RegMemItem in RegMemSets:
5      for Register in RegMemItem.getRegMemElements():
6        if Register._getType() == "Register":
7          if len(Register.getRegisterViews()) > 1:
8            MVBlocks.append(Register)
9
10 %>
11 class ${dataInput.getName()}_top_block extends uvm_reg_block;
12
13   `uvm_object_utils(${dataInput.getName()}_top_block)
14
15 % for RegMemItem in RegMemSets:
16 ${Registers(RegMemItem.getRegMemElements())}\
17 %   endfor
18
19
20 % for ifc in Interfaces:
21   uvm_reg_map ${ifc.getName()}_map;
22 % endfor
23
24   function new (string name = "${dataInput.getName()}_top_block");
25     super.new(name, build_coverage(UVM_CVR_ADDR_MAP));
26   endfunction
27
28   virtual function void build();
```

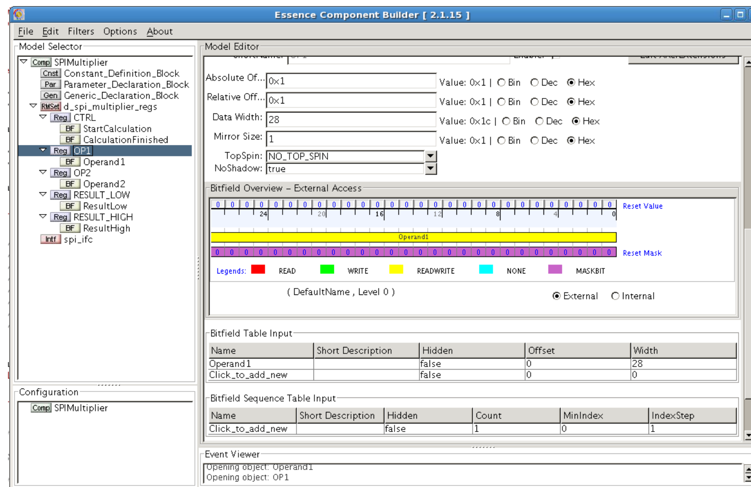

Figure 5.3: The filled data model represented in the Essence component builder

```
29      uvm_reg_field view_select_fields[$];
30
31 % for RegMemItem in RegMemSets:
32 ${RegisterConfigs(RegMemItem.getRegMemElements())}
33 % endfor
34
35 % for Register in MVBlocks:
36 <% BlockClassStr = "d_" + Register.getName().lower() + "_block" %>\
37 <%
38    if len(Register.getExtensions())>0:
39      theDependencyList = Register.getExtensions().pop().getLocationRef().
          getDepFields()
40    else:
41      dataInput.Logger.critical(": No multiview extension set for MV-Reg " +
          Register.getName())
42 %>\
43 %        for depfld in theDependencyList:
44      view_select_fields.push_back(${depfld.getRegName().upper()}_REG.${depfld
          .getFieldName().upper()});
45 %        endfor
46      ${Register.getName().upper()}_BLK = ${BlockClassStr}::type_id::create("$
          {Register.getName().upper()}_BLK", , get_full_name());
47      ${Register.getName().upper()}_BLK.configure(this, "", view_select_fields
          );
48      ${Register.getName().upper()}_BLK.build();
49 % endfor
50
51 % for interface in Interfaces:
```

```
52 ${InterfaceMappings(interface)}\
53 % endfor
54
55 % for Register in MVBlocks:
56     ${Register.getName().upper()}_BLK.lock_model();
57 % endfor
58
59     this.default_map = ${Interfaces[0].getName()}_map;
60     add_hdl_path("dut", "RTL");
61     lock_model();
62   endfunction
63
64 endclass : ${dataInput.getName()}_top_block
65 </%def>\
```

Listing 5.1: Template for the uvm_reg_block class for a module

5.1.4 Overview of the Generated Verification Environment

The verification environment, which includes the testbench, test, environment and agent, is generated using a different template set and data model, which are not part of this thesis. For this simple kind of testbench it would not be much work to create it manually though. As an example the generated class definition of the control register is given in the listing below, followed by an example of the register block, for which the template was given in the section before:

```
1  class d_ctrl_reg extends uvm_reg;
2
3    `uvm_register_cb(d_ctrl_reg, uvm_reg_cbs)
4    `uvm_set_super_type(d_ctrl_reg, uvm_reg)
5    `uvm_object_utils(d_ctrl_reg)
6
7    rand uvm_reg_field STARTCALCULATION;
8    rand uvm_reg_field CALCULATIONFINISHED;
9
10   virtual function void build();
11     STARTCALCULATION = uvm_reg_field::type_id::create("STARTCALCULATION");
12     STARTCALCULATION.configure(this, 1, 0,"RW", 0, 0, 1, 1, 0);
13     CALCULATIONFINISHED = uvm_reg_field::type_id::create("
           CALCULATIONFINISHED");
14     CALCULATIONFINISHED.configure(this, 1, 1,"RO", 0, 0, 1, 1, 0);
15
16   endfunction : build
17
```

```
18   function new(input string name="d_ctrl_reg");
19     super.new(name, 28, build_coverage(UVM_CVR_FIELD_VALS));
20   endfunction : new
21
22 endclass : d_ctrl_reg
```

Listing 5.2: SystemVerilog class of the control register

```
1  class spimultiplier_top_block extends uvm_reg_block;
2
3    `uvm_object_utils(spimultiplier_top_block)
4
5    rand d_ctrl_reg CTRL_REG;
6    rand d_op1_reg OP1_REG;
7    rand d_op2_reg OP2_REG;
8    rand d_result_low_reg RESULT_LOW_REG;
9    rand d_result_high_reg RESULT_HIGH_REG;
10   d_multiview_block MULTIVIEW_BLK;
11
12
13   uvm_reg_map spi_ifc_map;
14
15   function new (string name = "spimultiplier_top_block");
16     super.new(name, build_coverage(UVM_CVR_ADDR_MAP));
17   endfunction
18
19   virtual function void build();
20     uvm_reg_field view_select_fields[$];
21
22     this.CTRL_REG = d_ctrl_reg::type_id::create("CTRL_REG", null,
           get_full_name());
23     this.CTRL_REG.configure(this, null, "d_ctrl_reg");
24     this.CTRL_REG.build();
25     this.OP1_REG = d_op1_reg::type_id::create("OP1_REG", null, get_full_name
           ());
26     this.OP1_REG.configure(this, null, "d_op1_reg");
27     this.OP1_REG.build();
28     this.OP2_REG = d_op2_reg::type_id::create("OP2_REG", null, get_full_name
           ());
29     this.OP2_REG.configure(this, null, "d_op2_reg");
30     this.OP2_REG.build();
31     this.RESULT_LOW_REG = d_result_low_reg::type_id::create("RESULT_LOW_REG"
           , null, get_full_name());
32     this.RESULT_LOW_REG.configure(this, null, "d_result_low_reg");
33     this.RESULT_LOW_REG.build();
34     this.RESULT_HIGH_REG = d_result_high_reg::type_id::create("
           RESULT_HIGH_REG", null, get_full_name());
```

```
35      this.RESULT_HIGH_REG.configure(this, null, "d_result_high_reg");
36      this.RESULT_HIGH_REG.build();
37
38
39      view_select_fields.push_back(CTRL_REG.ACTIVATETM);
40      MULTIVIEW_BLK = d_multiview_block::type_id::create("MULTIVIEW_BLK", ,
            get_full_name());
41      MULTIVIEW_BLK.configure(this, "", view_select_fields);
42      MULTIVIEW_BLK.build();
43
44      this.spi_ifc_map = create_map(.name("spi_ifc_map"), .base_addr(0), .
            n_bytes(4), .endian(UVM_LITTLE_ENDIAN));
45      this.spi_ifc_map.add_reg(CTRL_REG, 0*4, "RW", 0, null);
46      this.spi_ifc_map.add_reg(OP1_REG, 1*4, "RW", 0, null);
47      this.spi_ifc_map.add_reg(OP2_REG, 2*4, "RW", 0, null);
48      this.spi_ifc_map.add_reg(RESULT_LOW_REG, 3*4, "RO", 0, null);
49      this.spi_ifc_map.add_reg(RESULT_HIGH_REG, 4*4, "RO", 0, null);
50      MULTIVIEW_BLK.mappings.push_back(MULTIVIEW_BLK.create_map(.name("
            MULTIVIEW_spi_ifc_map"), .base_addr(0), .n_bytes(4), .endian(
            UVM_LITTLE_ENDIAN)));
51      this.spi_ifc_map.add_submap(MULTIVIEW_BLK.mappings[$], 5*4);
52
53      this.default_map = spi_ifc_map;
54
55      MULTIVIEW_BLK.lock_model();
56
57      add_hdl_path("dut", "RTL");
58      lock_model();
59    endfunction
60
61  endclass : spimultiplier_top_block
```

Listing 5.3: SystemVerilog class of the register block generated using the template shown before

5.2 Formal Verification Implementation

For the example the SPI multiplier is used, just like in the UVM section. The register definition can be found in figure 5.1.

5.2.1 Starting in Excel

The high level specification in Excel is the same as the one for UVM and an example picture can be seen in figure 5.2.

5.2.2 Generating Assertion Files

The way to use the assertion generator is now creating a separated assertion file and using a bind file to connect it to the RTL module, which in turn requires defining a header with the inputs. This is done automatically, if the interface definition of the module is available in the model. For the SPI multiplier it looks like this:

```
1 module prop_d_spimultiplier_wrapper(input reset_n, input clk, input csn,
    input sck, input mosi, input miso);
```

Listing 5.4: Property module definition

The design itself was originally modeled to have a SystemVerilog interface on top level, which can be connected to the virtual interfaces in UVM. For the formal analysis with IFV interfaces are not allowed on top, as well as for synthesis (and some other tools). This is why a simple wrapper was created, which just connects the interface ports to normal I/O ports. The most important signals used in the assertions are then created automatically, but as mentioned before HDL paths have to be entered manually for now.

```
1  /**********************************************************************
2  Define signal or operation
3  **********************************************************************/
4  `define __reset_              ( reset_n == 0 )
5  `define __clk_                ( clk )
6
7  // Transaction fields, enter HDL path
8  `define __transaction_done_          ( d_spi_multiplier_wrapper.dut.spi_done
    )
9  `define __transaction_type_          ( d_spi_multiplier_wrapper.dut.write )
10 `define __transaction_address_       ( d_spi_multiplier_wrapper.dut.address )
11 `define __transaction_write_data_(__upper, __lower)        (
    d_spi_multiplier_wrapper.dut.data[``__upper:``__lower] )
12 `define __transaction_read_data_(__upper, __lower)         (
    d_spi_multiplier_wrapper.dut.bus_data_out[``__upper:``__lower] )
13
14 `define __transaction_read_          ( 0 )
15 `define __transaction_write_         ( 1 )
```

Listing 5.5: Clock reset and main transaction signal definitions first version

Reset and clock are needed for the reset property, the clocking of the assertions and the disable statement. The transaction should be defined in a generic way, so that it fits most applications. It is to be noticed that instead of an HDL path also an expression combining several signals can be used, a simple example is the way the active low reset_n is converted

to an active high signal for the assertions in this code snippet. The field transaction_done should point to a signal or a combination of those that show the completion of the data transfer. The type of the transaction is either a write or a read transaction. There has to be a signal or a combination that show the kind of access. For example an Altera *Avalon*^{TM} bus uses the signals read_n and write_n to distinguish between the requests, SPI frames are often defined to have the MSB or LSB as a read_n/write bit. The whole transaction type has to resolve to some value, that can be compared to the values defined in the transaction read and transaction write fields. Those can be changed if needed. The transaction address field is the representation of the register address accessed by the interface. The two macros for transaction write data and transaction read data are used to point to the input data (write) and the output data (read).
As it turned out when testing more complex designs it makes life a lot easier to change these one-point-in-time checks to sequences. More complex bus protocols can be described much easier then. All assertions have to be changed from using *&&* as logic connector to *and* which is a sequence connector (see [34] for more information on sequence connectors).

```
16 /************************************************************************
17 Define signal or operation
18 ************************************************************************/
19 `define __reset_              ( reset_ni == 0 )
20 `define __clk_                ( clk_i )
21
22 // Transaction fields, enter protocol sequences
23 `define __trans_read_done_          ( !chipselect ##1 chipselect && (read_n
       ==0) && (write_n==1) ##1 !chipselect)
24 `define __trans_write_done_         ( !chipselect ##1 chipselect && (read_n
       ==1) && (write_n==0) ##1 !chipselect)
25
26 `define __transaction_address_      ( ((address)) )
27 `define __transaction_address_check_(__field_addr)        (  ( 1'b1 ##1 (
       `__transaction_address_ == ``__field_addr)) )
28
29 `define __transaction_write_data_(__upper, __lower)       ( (writedata[
       ``__upper:``__lower]) )
30 `define __transaction_read_data_(__upper, __lower)        ( (readdata[
       ``__upper:``__lower] ))
```

Listing 5.6: New definitions which support sequences for protocol decoding

The HDL paths for the bitfields shown in the following code snippet only works for pure Verilog (or SystemVerilog or mixed) designs, but not for VHDL designs or designs that mix VHDL and Verilog/SystemVerilog. As soon as such language borders are present usually a solution adapted to the simulator is needed. In the case of this thesis the Ca-

dence Incisive simulator was used. For this simulator a feature called *ncmirror* is needed. It maps an internal signal to another signal (usually one in the testbench for directed/constrained random tests or the assertion file for formal verification). Similar solutions exist for Mentor Graphics QuestaSim and other simulators. Due to a still unknown problem in the Incisive Enterprise Verifier (IEV) setup in our flow the ncmirror signals do not work in formal verification. The workaround is to bind an assertion module into each module containing (user-accessible) registers and adding the register signals as additional ports in the SVA module (as described in the chapter about the bind mechanism). An additional advantage of this approach is that a test for each module as separate unit can be done easily as well as a top level test by binding all modules into the full design. If the register instantiation within the design has been done by generators also the additional SVA-module ports for the registers can be generated and also automatically connected in the bind file.

```
1 ////////////////////////////////////////////////////////////////////////////////
2 // Fields
3 ////////////////////////////////////////////////////////////////////////////////
4
5 `define __ctrl_startcalculation_          ( d_spi_multiplier_wrapper.dut.
      start_calculation )
6 `define __ctrl_calculationfinished_       ( d_spi_multiplier_wrapper.dut.
      calculation_finished )
7 ...
8 `define __multiview_normalmode_view_some_ro_value_        (
      d_spi_multiplier_wrapper.dut.tm_reg )
9 `define __multiview_testmode_view_some_rw_value_          (
      d_spi_multiplier_wrapper.dut.tm_reg )
```

Listing 5.7: Field definitions - for mixed language designs the paths are replaced by input signals

Subsequently all code is automatically generated, with the only exception that for unknown access types two standard properties for read and write are generated, that have to be filled by the user. To know the position of every field, the register address, bitfield bit position inside the register and size are needed (known from meta-model and thus automatically generated). The position and size values are used to calculate the left and right arguments for the transaction_write_data and transaction_read_data macros.

```
1 ////////////////////////////////////////////////////////////////////////////////
2 // Field addresses and positions
3 ////////////////////////////////////////////////////////////////////////////////
4
```

```
5 `define __ctrl_startcalculation_address_      ( 0 )
6 `define __ctrl_startcalculation_pos_          ( 0 )
7 `define __ctrl_startcalculation_size_         ( 1 )
```

Listing 5.8: Field positioning is automatically generated from the model

The first property is the reset property. The delay at the beginning (*##1*) is needed, because the function *$rose(reset_n)* looks into the past. If it would be the very first cycle the past would be undefined, so waiting one cycle is the safe way to check this. The *$rose(signal)* function checks if there was a transition from 0 to 1 from the last cycle to the current and if so it returns true (respectively 1, as SystemVerilog does not know boolean data types).

```
1  /*------------------------------------------------------------------------
2  Reset
3  Check register values after reset
4  ------------------------------------------------------------------------*/
5
6  d_spi_multiplier_regs_reset_values:
7  assert property ( @(posedge `__clk_)
8  ##1 $rose(`__reset_) |->
9  (`__ctrl_startcalculation_ == 0) &&
10 (`__ctrl_calculationfinished_ == 0) &&
11 (`__ctrl_activatetm_ == 0) &&
12 (`__op1_operand1_ == 0) &&
13 (`__op2_operand2_ == 0) &&
14 (`__result_low_resultlow_ == 0) &&
15 (`__result_high_resulthigh_ == 0) &&
16 (`__multiview_normalmode_view_some_ro_value_ == 0) &&
17 (`__multiview_testmode_view_some_rw_value_ == 0)
18 );
```

Listing 5.9: The reset property

If a bitfield with an undefined access behavior is encountered then standard assertion stubs are generated. Note that UVM standard accesses are predefined, others have to be added. These stubs consist of standard write and read accesses and have to be completed manually by the user. To prevent forgetting to add the correct access definitions for these fields the assertions are generated in a way that they will always fail (the consequent is always false).

```
1 /*------------------------------------------------------------------------
2  Properties for field StartCalculation
3
```

```
4  ----------------------------------------------------------------------------*/
5  /*----------------------------------------------------------------------
6   assertion stub for field StartCalculation
7   assertions for read/write access
8  ----------------------------------------------------------------------*/
9   register_access_rwcd_ctrl_startcalculation_written:
10  assert property ( @(posedge `__clk_) disable iff ( `__reset_  )
11    ((`__trans_write_done_) and (`__transaction_address_check_(
        `__ctrl_startcalculation_address_))) |->
12    0
13    );
14
15  register_access_rwcd_ctrl_startcalculation_read:
16  assert property ( @(posedge `__clk_) disable iff ( `__reset_  )
17    ((`__trans_read_done_) and (`__transaction_address_check_(
        `__ctrl_startcalculation_address_))) |->
18    0
19    );
```

Listing 5.10: Default properties for undefined access types

The next code example shows a typical read-write field assertion set consisting of a check if the value is written from the bus into the bitfield, a check that the read access to the field does not change it (if a field is updated by the design too this will fail, so a different access type has to be defined for this) and a check that data of the bitfield is actually transferred to the bus. Code examples for all different predefined access types can be found in the appended CD in the assertion file for the SimpleBus Regpack, as this features one field for every access type.

```
1  /*----------------------------------------------------------------------------
2   Properties for read-write field ActivateTM
3
4  ----------------------------------------------------------------------------*/
5  /*----------------------------------------------------------------------
6   bus data is written to rw field ActivateTM after write
7  ----------------------------------------------------------------------*/
8   register_access_rw_ctrl_activatetm_written:
9   assert property ( @(posedge `__clk_) disable iff ( `__reset_  )
10    ((`__trans_write_done_) and (`__transaction_address_check_(
        `__ctrl_activatetm_address_))) |->
11    (`__ctrl_activatetm_ == $past(`__transaction_write_data_((
        `__ctrl_activatetm_size_+`__ctrl_activatetm_pos_)-1,
        `__ctrl_activatetm_pos_)))
12    );
13
```

```
14 /*--------------------------------------------------------------
15  data of read-write register ActivateTM
16  does not change with a read operation on it
17  --------------------------------------------------------------*/
18  register_access_rw_ctrl_activatetm_stable_read:
19  assert property ( @(posedge `__clk_) disable iff ( `__reset_  )
20    ((`__trans_read_done_) and (`__transaction_address_check_(
        `__ctrl_activatetm_address_))) |->
21    ($stable(`__ctrl_activatetm_))
22    );
23
24 /*--------------------------------------------------------------
25  data of read-write register ActivateTM
26  is transferred to data-out register (or bus ifc)
27  --------------------------------------------------------------*/
28  register_access_rw_ctrl_activatetm_read:
29  assert property ( @(posedge `__clk_) disable iff ( `__reset_  )
30    ((`__trans_read_done_) and (`__transaction_address_check_(
        `__ctrl_activatetm_address_))) |->
31    ( `__transaction_read_data_((`__ctrl_activatetm_size_+
        `__ctrl_activatetm_pos_)-1,`__ctrl_activatetm_pos_)  == $past(
        `__ctrl_activatetm_) )
32    );
```

Listing 5.11: Properties for a read-write bitfield

5.2.3 Usage

The generators can generate the bind file with all module bindings and automatic port connections for all inputs and outputs of the module under test to the property module. The ports for the bitfields are prepared in the property module but not connected in the bind file, because this can not be done automatically with the current development flow (future use of a standardized register generator will make this manual step obsolete). The property module is generated too, the bus protocol definition with the read and write sequences, read-data and write-data bitfields etc. have to be specified manually as the bus protocol can be different between designs (otherwise it is only copy and paste anyhow). Another manual task is writing the Tcl file needed to initialize the design, set pin constraints and set it into the desired state before the formal verification starts. This is usually completely different for all designs, so this will not be automated. After this the tool can be invoked supplying the design files, the property file, the Tcl file, the bind file and the desired parameters for the model checking runs (like effort, possible tool specific settings etc.).

5.2.4 Overview of the Generated Verification Environment

In this section a short version of the generated files for the SPI multiplier DUT is presented. The bind file for the SPI multiplier looks like this:

```
1  // ---------------------------------------------------------------------------
2  // Project:  SPIMultiplier
3  // ---------------------------------------------------------------------------
4  // Autogenerated register access assertion bind file
5  // Generator by:
6  //            Michael Langreiter
7  //            IFAT DCGR ATV PTS
8  //            XXXXXXXXXXXXXXXXXX@infineon.com
9  //            +43-XXXXX-XXXX
10 // Copyright Infineon Technologies Austria AG 2012
11 // ---------------------------------------------------------------------------
12 bind d_spimultiplier_wrapper prop_d_spimultiplier_wrapper
      prop_d_spimultiplier_wrapper_inst(
13     //-------- spi_ifc
14     //-------- Sideband
15     .reset_n_i(reset_n_i),
16     .clk_i(clk_i),
17     .csn_i(csn_i),
18     .sck_i(sck_i),
19     .mosi_i(mosi_i),
20     .miso_i(miso_o),
21
22     .ctrl_startcalculation_i(dut.start_calculation),
23     .ctrl_calculationfinished_i(dut.calculation_finished),
24     .ctrl_activatetm_i(dut.activate_tm),
25     .op1_operand1_i(dut.operand1),
26     .op2_operand2_i(dut.operand2),
27     .result_low_resultlow_i(dut.result[27:0]),
28     .result_high_resulthigh_i(dut.result[55:28]),
29     .multiview_some_rw_value_i(dut.tm_reg)
30
31 );
```

Listing 5.12: Generated bind file with manual bitfield connections

The first part (under “Sideband”) is created and connected automatically. The ports below that are created automatically, but have to be connected manually, as the bitfield signal names cannot always be known. It could only be known if the bitfields are generated, which is now already done for some designs.

The property file has already been presented step by step in the example section 5.2.

Chapter 6

Practical Results

This chapter first shows the different VLSI designs used for testing the two methodologies. The first two are very simple designs intended mainly for checking the implementation of the automatically generated (and as needed manually improved) testbenches and assertions. The last one is a real design with much higher complexity and it has additional challenges like mixed languages (VHDL, Verilog, SystemVerilog) and Design for Test (DFT) functionality. The results of the tests on the three designs are then compared regarding to different metrics important in chip development: The effort of setting up the tests and the reusability of these, the coverage in terms of how much of the designs behavior regarding the registers is observed, the time needed to get results from these tests and of course the number and types of bugs found on these designs. One of the two simple designs also includes the possibility to inject faults to see the reaction of the verification environment.

6.1 The Designs Used for Testing

6.1.1 SPI Multiplier

The SPI multiplier test design is a small module with a simple functionality. It features a serial peripheral interface to access the registers and it calculates the product of two register values and stores the result in two additional registers (as the multiplication of two n-bit values needs up to 2*n-bit for the result). A block diagram can be seen in figure 6.1. There are no measures for synchronization taken within this design. Usually, the SPI clock is asynchronous to the internal clock and therefore typically the chipselect is synchronized with a simple synchronizer consisting of two flip-flops to create a signal that triggers the safe transfer of data from the shift register within the SPI clock domain to a register in the system clock domain. In addition there are no DFT implementations (like a scan-shell) added to keep the design as simple as possible for the tests.

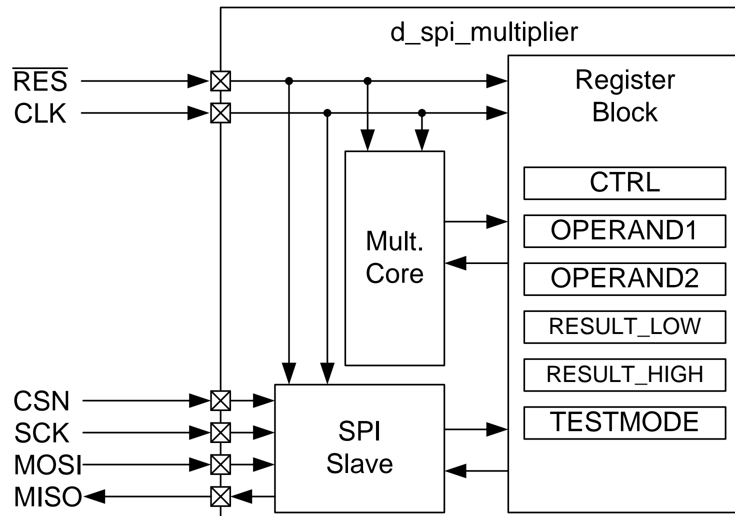

Figure 6.1: Block diagram and register overview of the SPI multiplier test design

6.1.2 SimpleBus Register Pack

This module is designed to incorporate every standard bitfield access as defined by the UVM. Communication is done via a very simple parallel bus, which is similar to several on-chip buses and buses used in FPGA based SoC designs. The bus features a chipselect signal, a write/read-not signal, address and input and output data lines (input and output are separate signals). There is no real functionality in this design other than the register accesses and the additional option to control faults that are injected into certain bitfields from the outside (via the sequence for the UVM test or the Tcl script used for Formal Verification (FV)). This is used to prove that the UVM and the FV based methodology are able to identify all errors in the standard access behaviors. In figure 6.2 a block diagram is given.

6.1.3 Lithium Ion Battery Balancing IC

The last design is a real world design. It is the digital part of a mixed signal IC, which is used to measure and balance lithium ion battery cells. Balancing means evening out differences between the batteries caused by manufacturing and aging by discharging them or redistributing charge between the batteries. One IC is able to handle up to twelve cells, the next twelve are then done by another one. The communication can be done via an SPI interface (direct communication to one IC) or a second interface (called Inter Block Communication Bus (IBCB)), which was designed to meet the special requirements of communication between the ICs along the stack of batteries. This design also includes

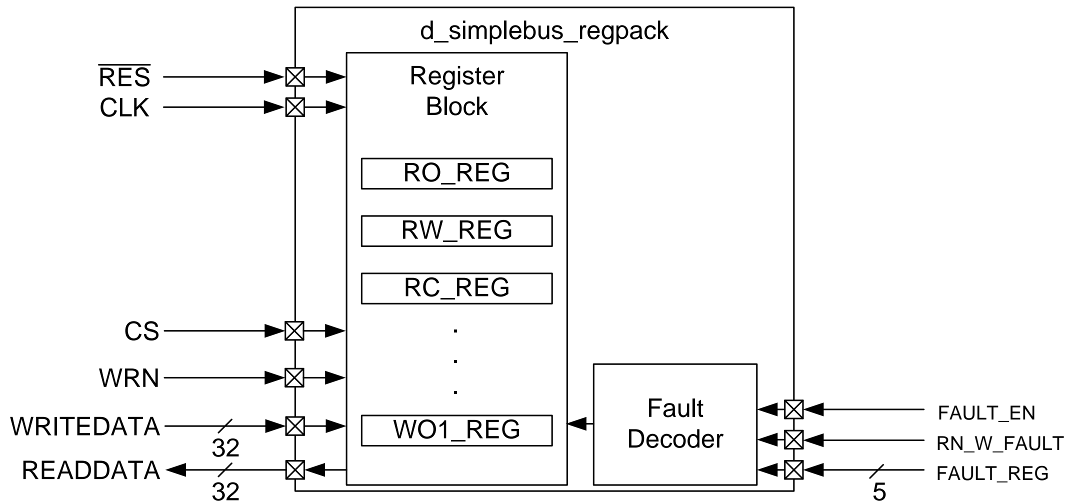

Figure 6.2: Block diagram and register overview of the register pack test design with a simple parallel bus and a fault injection mechanism

synchronization mechanisms, as well as DFT features. The system including connectivity and registers is already modeled according to the Essence data model. As the final tests of the methodologies were done this design was already in production and it is now already available in silicon in our laboratory. This design proved to be quite problematic for the FV to handle it on toplevel, which also lead to a few adaptations to the assertion templates and the overall assertion distribution in the modules. The serial interface, which is additionally secured against transmission errors by a checksum, is too complex for the formal engine to get through to the registers within an acceptable amount of time. This lead to the need to cut open certain connections within the design to directly access the on-chip bus for driving data to the registers. As the design is also written in mixed languages (Verilog, SystemVerilog, but mostly VHDL), it was not possible to bind all assertions into the toplevel and probe into the modules to read the register values. Due to the existing model of the design it was quite easy to change the generators to create one register assertion module per internal module and create a bind file that binds every property file to the correct design file. A simplified block diagram is given in figure 6.3.

6.2 Metrics for Comparison

6.2.1 Effort and Reusability

Effort and reusability are two very important characteristics of a methodology, particularly with regard to its use within a productive environment. If no dedicated verification depart-

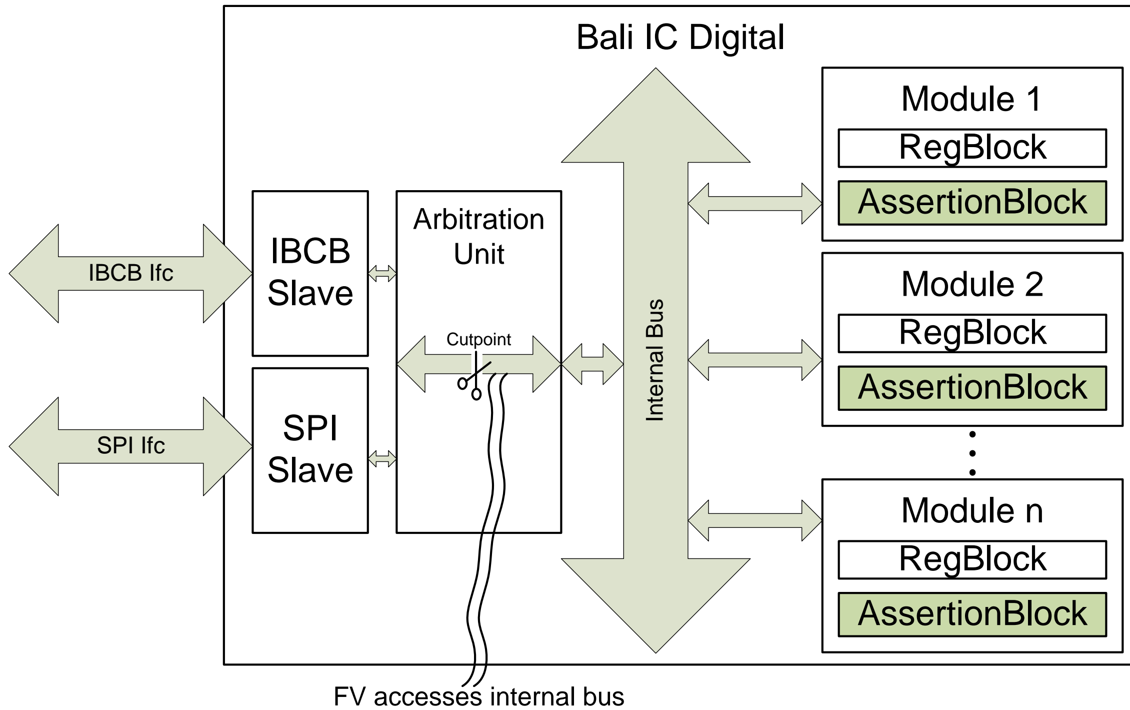

Figure 6.3: Diagram of the blocks important for this thesis including the binding hierarchy for the assertion modules and the cutpoint for access to the internal bus for formal verification

ment, or dedicated verification engineers within the department, is present the designer has to take the additional step of doing the verification alongside the design. According to literature the ratio of the amount of time spent on verifying to the amount spent on doing the RTL design should be at least 60:40, better even higher. As it is hard to reach even this, the methodology should not take away time needed for the design and also writing tests for other topics than the register access verification. To facilitate this as much as possible, automatic code generation was used. But aside from generated code for the special purpose of the register accesses, it is also interesting if these generated verification environments can be directly used for other verification too. The first comparison on effort, if we put aside the possibility to generate the code, is definitly won by the formal verification using SystemVerilog assertions. A simple assertion file can be written, bound into the needed module and activated within just a few minutes. There is no testbench needed, no dedicated stimuli elements and no check elements. The assertion modules created with this thesis are bound into each module and can be immediately extended with additional assertions for various checks, not only ones that are related to register access checking (all input and output ports, as well as the register signals are

fed into the assertion module by default). As the signals representing the bitfields of the registers can not always be found automatically this has to be done manually in the bind file in this thesis. In comparison a UVM testbench needs quite an overwhelming amount of boilerplate code to fulfill just a simple verification task, but as the verification tasks get more complex the amount of code does not rise in such an amount anymore. The code needed for a task like the register access verification alone is still a lot larger than for the formal verification. On the other hand having set up this testbench for the registers it can be used for other verification tasks too. As automatic code generation is available for this thesis setting up such a minimal testbench requires quite little effort. If no automatic code generation is available the configuration mechanisms within the UVM also allow to quickly reuse a testbench for other designs like the verification of a slightly varied chip or a different test setup with just some options, that may even be passed from the command line to the simulator (for more details on the configuration mechanism of the UVM refer to [3], [20] and [26]). For the register verification, what has to be done manually is to create all needed agents for the different interfaces, through which the registers can be accessed and that should be checked for correctness. These agents will anyhow be needed for other functional tests on the system. If standard interfaces are used, it is possible to even buy agents as VIP in case the project timeline does not allow developing an own one or an existing agent could be used, if the IP repository of the company also provides VIP.

6.2.2 Coverage

If a property is proven in formal verification the full logic cone that is influencing the (boolean) value of this property starting from the input pins is checked, but only on its influence on the outcome of the boolean statement under test. If this cone of logic increases due to verifying a big module or even verifying a toplevel design this will lead to excessive runtimes or even make it impossible to prove the assertion within normal time and memory limits. This problem is known as the state space explosion problem (see [34, pg. 569ff]). When trying the assertion based methodology on the full blown design on toplevel this happened. To still get some information out of it, the first reduction was to bypass the real (serial) interface to the chip and directly use the parallel on-chip bus. With this at least some assertions could be proven, some failed (no real design fails, but fails on the specified access type, more on this in the subchapter about the type of bugs) but still a third set did not finish at all. This was due to the fact that th state space was explored within the allowed limit, which means that some of the possible coverage is lost. It is very simple to add an assertion that checks the register contents also outside of the transaction scope, so it can be checked, that there is no side-effect of a read/write to another register or other design influences.

A UVM testbench also checks the interface, because it is needed for the frontdoor access to the registers and in combination with the backdoor access a possible change of the value on the way to the register can be detected. Even though the whole way from the inputs to the register is used it is not checked in all states. It is only checked in the very few states that actually are present within this simulation run, if there are influences in certain failure states of the design they will only be seen if it is simulated with this state present. As UVM is based on using randomization whenever possible a useful random seed value and a lot of runs make it possible to see this behavior. The more the randomization is used, the more the verification shifts to real coverage driven verification, which demands mechanisms to collect coverage to be implemented and is able to give a feedback about the functional coverage of the verification. A different mindset concerning stimulation and checking is required than with the normal directed simulation approach, which is why lots of designers/verification engineers fear to take the step away from the directed tests.

6.2.3 Runtime

The test designs were chosen in a way, that, if the assumptions are correct, a difference should be visible, because the difference would strongly increase with the complexity of the design. It is not too easy to really specify the complexity of the designs in a number, which is normally used in VLSI designs. The number of Gate Equivalents (GEs) can be misleading, as it shows a size which will increase for a larger design, but this does not automatically mean that there are more interconnects and dependencies in the system. A cryptographic algorithm or some other design with a big datapath can have an equal or even a lot higher number of NAND2 equivalents, but still be a lot simpler than a controller, just because everything exists several hundred times in parallel. But still increasing size of a similar kind of design can at least point the direction of the complexity. The Lines of Code (LoC) can be used as a measure as well, as this is, following a normal coding style, also reducing the impact of the same parallelized modules being instantiated in the design. The number of registers available to the user might also show something about the complexity, as the more settings can be done for a design, the more complex it will most likely be, but still this could be misleading for the same reasons as the normal gate count and the same applies to the lines of code and other metrics. For every kind of verification a problem can be constructed, that will be hard to solve and thus take a long time, so the most common way of expressing a designs complexity, which is the number of gate equivalents, will be used here. The runtime of the verification of a design strongly depends on its complexity. It might not just be the runtime for the single tests that are longer, but also the number of tests, as more and more interdependencies have to be checked. For this special verification task of the register access verification, the runtime of the UVM

verification will be dependent on the interface (amount of time for a frame, one or more bus accesses for a read command, etc.) but mainly on the number of registers and the number of bits in these registers. For formal verification a lot more influence on a bit is seen, as the full cone of logic driving the bitfield is checked. A more complex design will add a lot of additional states to the register's behavior in most cases, which increases the runtime quite extensively. This influence can already be seen for most designs if the formal verification of a module is done and in comparison a verification on the top level of the design.

6.2.4 Number and Type of Bugs

The kinds of bugs this methodology has to find are well known. It is the target to uncover wrong bitfield behavior on a read or write access to it. But the results can also prove, that some additional bugs might be uncovered using either one or the other verification methodology. Sometimes both can produce the same results, but for one requires substantially more effort to set it up. The number of predefined bitfield access types defined by the UVM is the number of behavior sets for which the effect of an access is defined and where a violation has to be found. Parts of these behavior sets are shared between the different bitfield types, e.g. w1s and w1src share the "write one to set" part and rc and w1src share the "read clears all" part. Additionally own access types can be defined by adding these types to the generators and for formal verification also to the templates and for UVM to a register package, that is imported in the register model. The types of bugs, that will be used as a metric here, are all the predefined access behaviors as "must-have" and additional sources of error affecting these bitfields (so not on custom behavior).

6.3 UVM Results

The UVM worked as expected on the SPI multiplier design. The time consumed for the reset and bitbash test is about one minute. As the design is quite simple and was adapted to fit to the needs of verifying this methodology, this was working out quite well as expected. For the testbench an agent was needed for the interface, as well as quite some (for this first design manually written, as the generators for this did not exist at that time) boiler-plate testbench code. The register model was already generated from the Essence description.

The SimpleBus register pack was designed to test all correct and incorrect behaviors for all register access types predefined by the UVM. For a certain set of registers, there are some topics, that have to be kept in mind: Predefined access sequences do not check if a write-only bitfield type (wo, wo1, woc, wos etc.) is readable, so the result of a read operation

on these is undefined - which might not be wanted. Usually these bitfield types are used for strobe signal generation and thus can anyhow not be read back afterwards (as a strobe only lasts for one clock cycle), but if it is meant for something else, e.g. setting a counter value that is then internally decremented and should not be readable, this has to be kept in mind and a user defined access check sequence is needed. If a write/read on one bitfield has an (unwanted) effect on another there will be no error flagged if they do not share the same register address. Formal verification can find such paths by adding an assertion that checks if outside of the bitfield access (transaction done and correct address) the value is stable. This assertion is not automatically generated with the generator framework in this thesis, as in most designs the registers have different internal accesses in addition to the external accesses (via the interface), which are used to change the value outside of the transaction context (e.g. updating an ADC value, setting diagnosis flags and so on).
The tests on this design were done with one fault per test, so a read fault was introduced into register 0, a full bitbash sequence was run, a write fault was introduced in register 0, a full bitbash sequence done and so forth for all registers. This sums up to fifty separate register test runs (25 registers with a separate read and write test). The run was done in graphical simulation mode (batch-mode usually speeds it up a little additionally) and took two minutes. As expected the faults were found and all other registers were found to be acting according to the specification. The testbench for this design was already automatically generated with the Metagen based tool created by the Infineon Design Enabling and Services group. The agent was quite easy to fill out, as this bus protocol is really simple. The register2bus adapter is a very short sequence item transformation and the register model was generated using the generators created for this thesis. This kept the effort really low, the setup of the whole testbench with the model took less time than writing the design itself.
The final test was done using a real life design. It has about 80 register with all together about 400 bitfields in them. The register verification run on the design took about 15 minutes.

6.4 Formal Results

The formal run on the SPI multiplier is not noticeably slower than on the UVM. While developing the methodology and always testing it on the design and vice versa the formal verification already uncovered small design flaws and it verified the register accesses successfully. Setting everything up was done quite quickly, as only a few HDL path mappings needed to be done and the controlling Tcl file was also just a few lines driving the reset and starting the proving run from this point. Even the serial interface used for the design did not create any problems, because it was done in a very basic way.

The SimpleBus register pack was verified in the same way as with the UVM by creating one read or write fault, verifying the whole design, creating another fault and doing it again and so forth. These all in all fifty runs already showed the runtime difference between the formal verification and the directed method. The verification, which was done in batch mode (without any graphical environment starting up), took about twenty three minutes. As the design was created in a very simplistic way, only providing the register accesses and fault injection mechanism and no real functionality, there were no additional bugs uncovered. The effort for setting up the verification was not too high, the startup script was about the same as for the SPI multiplier, the mappings via bind file could be done quickly due to the way the register names were chosen and defining the bus protocol was fairly easy due to the simplicity of the interface.
The verification of the actual design was already expected to be very troublesome and it turned out to really be that way. The first methodology based on Verilog with the possibility of mappings via HDL paths could not be directly used, as the design was partially written in VHDL. As a result the simulator specific command nc_mirror was used, which turned out not to be working, even though it should have according to the documentation. After being able to map all bitfields correctly by rewriting the assertion generator to create one assertion module for every module with a bus connection and a bind file to bind them into the corresponding module the formal verifier was still not able to prove or disprove any given assertion except for the reset assertions. All bus transfer dependent assertions were explored with the highest default setting for proving effort. Explored means that the state space was checked up to a certain limit without finding a counterexample or getting through the whole space to prove the assertion correct. It was apparent that the problem was to access the registers through the serial interface, which also has special commands and a checksum and not just plain address and data fields, that are written to/read from the registers. One possibility would be to use assumptions to model the data transfers. This can be quite challenging and due to the additional logic in the design, where also some arbitration is done for accessing the on-chip bus, the confidence in having success with this to make the checks work was rather small. So instead of modeling the serial interface and the second interface to access the internal bus the design was cut at some point in the arbiter, where the parallel internal bus could be driven. This approach already yielded a useful result, where, of 230 assertions all together, one third already passed and one third failed and only one third was still explored. The assertions were reduced to this number, as for similar fields only one was checked. Still this took more than seven hours. A way to speed up the verification is to do only a module verification instead of verifying all modules combined. This is the recommended way for formal verification, but this means that the real influences on some register contents might not be seen, as they reside within other modules. For these signals assumptions can be made at the module

boundary, but this does not mean that the design will behave this way, as the assumption could be wrong. The failing assertions originate from some simple problems. First of all, some registers were declared as read-only registers. In most cases, like for this design, the registers will only be read-only to the interface, but can still be changing, as they represent ADC values, error flags, diagnosis bits and similar information. The formal verification found these mismatches, whereas the UVM did not see them. Some fields should not be affected by a readout so it is checked, if the value at readout and one cycle later is the same (stable-read assertions). The formal verifier also found the combination where this is not true, as it can happen that the next cycle after readout is exactly where a new ADC value or similar is set in the register. To bypass problems like these a custom bitfield is needed, which has some access type like e.g. “rosd” (read-only set-by-design). These need an extended version of the stable read command, which adds the exception of data transfer from a certain place in the design. As this is totally application specific and there exist virtually infinite possibilities of different combinations, it is hardly possible to predefine such behaviors. If such a type is used more often it is possible to extend the generator with default assertions for this special access type. If only a few special bitfields are used and the results of the verification run are checked these can be seen by looking at the counter examples of the failed assertions. The methodology will in this case flag more fails than really exist, but this is the safer way than to suppress (possibly real) fails.

6.5 Comparison

As both methodologies have been analyzed separately it is time to compare them in respect to the metrics defined at the beginning of the chapter. A first overview and comparison of the designs focusing on the possible measures of complexity and the resulting verification runtimes is given in table 6.1.

Table 6.1: Comparison of the two verification methodologies on designs of different complexity in respect to runtime

Design	LoC	GE	#Bitfields	Runtime UVM	Runtime FV
d_spi_multiplier	212	6.5k	8	1 min	2 min
d_simplebus_regpack (50 runs)	598	10k	25	2 min	23 min
Balancing IC	38.5k	53k	400	15 min	>7 hours

In the next sections all used metrics are compared in more detail. For each section a table with a summary of the results in a simple form is presented. This is a rating system

ranging from +++ to - - -, the base to which the two methodologies are compared to are the typical verification systems with directed tests and simple stimuli mechanisms, which are still used widely nowadays, be it Tcl based or VHDL/Verilog based testbenches with simple control scripts. The meaning of each rating can be seen in table 6.2.

Table 6.2: Comparison system

Rating	Sign
Very good	+++
Good	++
Slightly Better	+
Neutral	o
Slightly Worse	-
Bad	- -
Very Bad	- - -

6.5.1 Comparison of Effort and Reusability

The effort of writing a UVM testbench manually is very high. Even implementing only the really necessary classes (and functions and tasks within them) requires effort and not even a single test sequence is done then. Creating the register model additionally would raise the effort even more if it is done manually, depending on the number of registers and bitfields in the design of course. The generators ease this problems to some extent. The UVM register model was already created in a way that makes it easily generable, in the meantime there are tools available from the different Electronic Design Automation (EDA) vendors, that can generate a UVM register model from widely used register and IP definitions like IP-XACT. For generating the whole UVM testbench a few commercial products already exist, if nothing similar to MetaGen is available at a company this might be an option to speed up this process. As soon as this needed boilerplate code is done it is quite easy and fast to extend this minimal testbench for all different tests and settings, which can then be used for the verification of the whole digital implementation (currently attempts are also made at some EDA vendors and companies to transfer the UVM methodology into the analog/mixed-signal domain, which might then make this usable for the whole pre-silicon verification of the IC). As UVM facilitates reuse and the creation of verification IP, parts of the testbench can be reused in other projects or often also bought from external suppliers. The effort of setting up formal verification is quite low. For module tests setting up

the bind file and a simple property module definition along with a simple startup script usually written in Tcl is enough to start adding assertions for different aspects of the module's behavior. Also reusing existing module assertions is possible if the module is placed within another one, if assumptions were used to specify the behavior of the module input ports these can be turned into assertions in this case (most formal verification tools can do this automatically). Reusing every assertion withing the toplevel of the chip will not be possible for most designs, as the state space for some will be too big. The behavior specified by these assertions might have to be checked by other means of verification. The generated assertions and bind file can directly be used with just a few manual adaptions (connections of the bitfield signals to the assertion bitfield signals) and an application specific startup Tcl script. Other assertions for the module behavior that have nothing to do with the registers, can also be added here with no extra effort.

Table 6.3: Comparison of the two verification methodologies in respect to (initial) effort and reusability

Methodology	Effort	Reusability
UVM	- - -	+++
Formal	++	++

6.5.2 Comparison of Coverage

In the register verification with UVM only the accesses and changes to the registers within the limited time of the access itself is checked. Interdependencies between bitfields in different registers will not be seen. If they are needed, this can be modeled with a custom bitfield type. Unwanted interdependencies (some signal erroneously triggers another one, that sets the bit) might not be found. Of course some additional sequences can be written, that can at least ease this problem by setting every field, checking every other field, resetting the device, setting the next field and so on. This still might not find some special kinds of bugs, for example if it is a dependency of multiple other fields. An access will not only check what happens to a field, it also implicitly checks the interface which is used to communicate with the design. Bugs in this interface could be uncovered this way. Using frontdoor/backdoor access sequences also helps to rule out or find the dependence of a data change within the interface. The complexity of the interface will not have a big impact for most designs.

Formal verification sees every possible path to change the bitfield at any given time. If there is some interdependency that could create an error it will be found, as long as it can be found within the state space exploration limits that were set. Every influence within the cone of logic pointing to the monitored expression will be seen, so it covers all influences at all times. This will often lead to an explosion of the possible states, especially for bigger modules or even toplevel verification. Only verifying it on module level on the other hand will not show all possible influences. In order to be able to check it also on toplevel some trade-offs might be necessary, like bypassing more complex interfaces (as it was needed with the balancing IC design). A formal verification of the interface module alone should then be done separately.

Table 6.4: Comparison of the two verification methodologies in respect to coverage

Methodology	Coverage
UVM	+
Formal	+++

6.5.3 Comparison of Runtime

Running the register access tests, and reset value tests, with UVM takes only a short time, even for a real design this stays in an acceptable time frame. The values for the different designs can be seen in table 6.1. The increase of time needed to run the basic register access test suite is mostly depending on the interface complexity, which is not a big impact on the test designs, as the interfaces on these are quite simple, and the number of bitfields and bits, as for everyone of these a write one/zero and read cycle has to be performed. The complexity of the design itself has a lower impact on the runtime, only certain constructs that slow down the simulator itself will of course add to the needed real time (but those will also affect the formal verification run). For formal verification the time needed for small designs is quite low, this is also true for most modules or submodules of full designs. The coverage of such a verification is already a lot higher than the one of a directed/constrained random approach. As soon as the design becomes quite large the state space gets too big for the formal verification to handle all assertions and for others it takes very long. The full design used in this thesis is an automotive application specific IC and these are typically smaller designs compared to others in different areas of chip design like microcontrollers or signal processing applications. For those applications

a toplevel formal verification will be impossible in most cases, so using a module level formal verification will often be the only possibility for those.

Table 6.5: Comparison of the two verification methodologies in respect to runtime

Methodology	Runtime
UVM	o
Formal	- - -

6.5.4 Comparison of Number and Type of Bugs

UVM is able to find bugs that happen within an access or within two accesses, if the register is read again. There is no permanent update and check of the register contents if it is not done by a manually created sequence. For backdoor access enabled the current state of the registers could be monitored without doing a real access. But this will not prove that a certain situation can never occur. By randomizing accesses to the registers (best create read/write, address and data randomized within some constrained limits) additionally to the predefined linear checking sequence, the possibility to see some state, that was not thought of, can be increased. Problems with the access via the interface, especially interface decoding or arbitration problems in multi-master systems, can be visible in this verification too, as well as problems and differences for register accesses through different interfaces of the design. Simple access violations within the access could be found, as the checks on all register access types with the SimpleBus register pack test design succeeded, which includes one of every bitfield type. Formal verification traces all possible influences on the asserted condition, which is a register access check in this case, and is thus able to find also well hidden bugs that do not have to be related to the access itself. In the tests it was possible to uncover behaviors like a signal change one clock cycle after the read access, which was checked against in the assertion to make sure a read access does not alter data of certain register types. For this certain case it was not a problem of the design, but a too restrictive definition of the access type for this register, as some internal diagnose flag could well be set just in the cycle after the read, but also at any other time. Interface related fails might not be seen in the full design, as the interface might have to be bypassed if it is too complex. Problems with the register accesses through different interfaces will not be uncovered by this kind of verification then. If the interfaces and the bus arbiter are formally verified separately this should be no problem. In case a bug is found it is not always easy to understand how the design reached this state, as it is

not operated in the way it is operated in the system. This is something the verification engineer has to get used to in the beginning, as it is very different from normal verification. On the other hand this is exactly why it often uncovers bugs or problematic states that have not been considered in the specification or design phase. The bad part of this is that the specification of the internal conditions with sequences can be quite complex and hard to understand when reading them, so additional possible sequences could have been defined without wanting it (especially when using more complex sequence combinations), which can lead to unwanted assertion fails (which can be traced back and corrected then), but even worse to passing assertions even though an unwanted state was present.

Table 6.6: Comparison of the two verification methodologies in respect to bugs

Methodology	Number/Types of Bugs
UVM	++
Formal	+++

6.6 Conclusion

UVM is useful to see the design in a real operating condition, where checks on the register access behavior can be done quickly and on different interfaces with very little additional effort and by using randomization of the register transaction items the possibility to uncover bugs not visible with the predefined access sequences increases too. The testbench with all the agents, register model and other UVM classes can be used for all the other verification on the design too, which means that the additional effort of creating it pays off later.

Formal verification has the big advantage that all influences on the asserted expression at all times are analyzed. The correctness of the assertions is the most important part. As more complex assertions using connected sequences can become hard to understand quite quickly it is hard to assure the correctness, especially because the proof is done on the design outside of a normal use-case. Fails due to this might not be a problematic issue, as those fails will anyhow be analyzed further. A test that passes even though the design behaves incorrectly (but it behaves correctly in regards to the assertions!) is a very big problem, as this might then slip through the whole verification if no other means of testing this are included. A testbench for simulating the design in a functional way is often recommended. For the register access types defined by the UVM the assertions automati-

cally generated by the generators written for this thesis have been verified to be working, the critical part in this thesis is the manual definition of the bus access sequence. As the design gets more complex the time needed for the prove that the design acts according to the assertions increases significantly. One way to address this is to use formal verification for overnight and weekend runs or split up the design and do verification on module level. Setting up the formal verification can be done quickly, as the needs are just the bind file (or bind command passed via command line), the assertion module and a small Tcl script for the verification startup and settings. For the register verification the bitfield input ports need to be connected manually, which can be a little bit more work, but all in all it can be set up quickly and adding assertions for other behavior than register accesses is no more effort than just writing the assertions (all module ports are already fed into the property module). Altogether it seems to be the best approach for a real-life project to use both kinds of verification. A testbench for normal simulation will always be needed, so setting up the UVM testbench is no extra effort, especially if some verification IP is already present. This allows to do some quick tests on the design and also see, that for a passing assertion no real functional bug is slipping through. The formal verification for the register accesses, and other parts of the design behavior, can be set up quickly and be run overnights and over weekends for a very thorough analysis.

Chapter 7

Post RTL Verification

7.1 Post Synthesis Verification

The verification methodologies were only described for RTL code up to now. But this is only a high level format of description, the system does not necessarily have to be the same after starting the backend processing. Parts of the system might get optimized, some timings or intermediate states might produce unexpected results and so on. This chapter is focusing on the possibilities to ensure correct functionality of the register access behavior in the final system for tape-out. This includes all steps starting from the synthesized netlist on to the back-annotated netlist extracted from the final digital layout.

7.1.1 Verifying the Synthesized Netlist

UVM

If only frontdoor access is used it is very easy - just replace the instantiation of the RTL design by the instantiation of the netlist. As the access is done via the interface everything has to work just like with the RTL design, otherwise the synthesis result differs from the desired one. If backdoor access is used more effort is required to adapt the testbench. UVM already offers the possibility to add different HDL paths for different design abstractions. These are selected by passing the kind-argument in the set_hdl_path function family. The synthesis will change the names of module instances and depending on the settings even flatten out hierarchies, thus it is not possible to give a general solution for this problem, but knowing the settings and the tool the HDL path names are generated according to a certain scheme, which could be used to determine the paths automatically. The alternative is to search through the netlist and derive the paths from this manually.

Formal Verification

In most cases the formal verification using SystemVerilog assertions is done by binding a property specification into a module, which uses the input and output ports of the design module as inputs, to check the correct behavior of the design. When doing synthesis the ports of a module are typically left with the same names afterwards, so if the mechanism of renaming the modules is known (same as for the UVM backdoor accesses) the bindings could be adapted easily. The problem for the formal verification approach used in this thesis is that register values have to be monitored and they are not always visible on ports of modules, but rather as a bunch of flip-flops or similar within a module. As those are replaced by standard cells every bit turns into a submodule and the bit-widths are usually optimized. This makes it very hard to reconnect everything in an automated or semi-automated way and manual adaption of the connections would be a very painful process. The possibilities to adapt this are thus very dependent on the way the design itself is done.

7.1.2 Verifying the Back-Annotated Netlist

With regard to the back-annotated netlist the main differences to the netlist after synthesis are the clocktree, the additional buffers and the available timing information for minimum, maximum and typical timing. There is no real functional difference, except for fails due to timing violations, which should be found by the Static Timing Analysis (STA), the netlist still consists of the standard cells and the hierarchy of the netlist is about the same as before. This means, that the same problems for the verification methodology exist as for the verification of the netlist after synthesis. For UVM with frontdoor access it is still easy, for backdoor access and for formal verification the problem of the renamed or removed signals exists and the means of solving these problems are the same as the ones for the post-synthesis verification.

7.1.3 Alternative: LEC

As it became clear in the previous sections the verification of the design at some point in the flow after synthesis can be a very problematic if it is needed to observe internal signals of the design. After processing the RTL design, because optimization might combine signals or delete duplicate signals, it is not sure if the names of the signals have changed or if the signals still exist or if the bitwidths are still the same etc. So it would be very helpful to have the possibility to check the design in RTL and then just make sure the design stays the same. This is exactly what the Logic Equivalence Check (LEC) does. Two designs can be compared for equivalence, this can be a comparison of RTL to a netlist, a netlist to another netlist or RTL code in one hardware description language to code in another one. The LEC is a formal method, which abstracts the design similar to the ways

shown in chapter 2.4 to receive a description independent system representation. Certain features of optimization steps can be found and are not seen as nonequivalent points (e.g. inverted equivalent logic). Using the tool is very easy and can be automated, so it is included in most VLSI design flows nowadays. The possibility to check netlists against each other is also used for doing an Engineering Change Order (ECO). An ECO is a change to the design that is done by reconnecting the cells. As this only influences the metal masks, this is cheaper and can be processed quickly, if wafers processed up to this state are available. To fix small bugs often spare cells, a set of different standard cells, are added to the design. If a bug is found in verification after tapeout a fix can be applied to RTL code. This is then synthesized and the original netlist of the design and the new one are compared with the LEC. The nonequivalent points are the changes to the design, this is what needs to be reconnected.

7.2 Silicon Tests

The last part of the thesis was to provide means for aiding the verification of the design in the laboratory. According to the needs of the laboratory staff no automated access tests were created, but rather register representations for use with the already existing systems. In the laboratory mainly two different verification settings are present: The first one is a setup using a verification PCB controlled by a microcontroller or FPGA, which is operated by a Graphical User Interface (GUI) running on a PC. The second one is an automated chip tester, which is configured using an Excel workbook with integrated VBA code.

7.2.1 VBA for Tester

As mentioned earlier, for the automated chip tester VBA is used to define the measurement and evaluation process. The access to the registers of the chip is needed to set the device in to the desired states, activate test modes, read measurement and diagnose values and so on. This access is done via the interface that was typically also used in the pre-silicon verification with UVM for the frontdoor access, like e.g. SPI, LIN, CAN. The definition of the available registers, the bitfields they include, the meaning of the values for certain enumerated states, range checks and the functions merging this information into a command that can be passed to the interface handling class are generated from the Essence model also used for design and pre-silicon verification purposes. The generators have been written in a way to create Visual Basic code similar to the one that was manually written before, to make the transition to the new system easier. The test routines are written already before the chip arrives to the laboratory, normally even before tape-out. Changes

to the register mapping can now be done without having to manually adapt the register control code anymore.

7.2.2 XML for LabView GUI

The verification boards used for the normal laboratory verification are directly controlled by a microcontroller or an FPGA, which is then controlled by a GUI on the PC, which is typically written in C++ or LabView. Already some years ago I defined an XML based format for describing registers, which defined the address, bitfields, decodings (i.e. the meaning of a certain value, e.g. "0 - valve closed" and "1 - valve open") and read and write access. This is used up to now as well in the C++ GUIs as in the LabView GUIs in combination with a tree view to represent it graphically, and has proven to be very helpful. The XML description of these registers was done manually up to now. With the introduction of MetaGen and the modeling of the system it became possible to generate this. The format has been kept similar to the one used now, but if the need for additional information in it is present it is easy to extend it.

Chapter 8

Conclusion

This chapter is meant to provide some information on the current state of the methodology, which parts of the thesis are currently in use and how they are used, as well as some outlook on possible extensions and improvements (especially ones that are only practicable if a larger part of the design is modeled and generated) and on how it might be integrated into the existing flow.

8.1 State

The generators have been finished and are usable with certain manual adaptations, e.g. bitfield connections, backdoor paths. This automation already lowers the effort substantially. The only prerequisite is a suitable modeling of the registers and the interface (I/O ports). For existing projects this may be an usually comparably small additional effort, for new projects this modeling is already standard and thus no extra effort has to be spent for this part in a project. The pre-silicon verification methodologies have been compared with focus on register accesses, but most of the results can also be applied to other functionality too. The results were already anticipated, as some literature and also personal experience pointed to this. As the comparisons of formal verification with model checking and dynamic tests with directed tests and/or constrained random verification found were quite a few years old, it was interesting to see how this has changed. The formal verification tools, and also the hardware they are running on, have improved over the years. Formally verifying a design on toplevel is quite a challenge due to the state explosion problem. Finding an error in the design with formal verification is usually fast, but proving correctness can take a very long time, this is typical for SAT solvers [13].

8.2 Current Usage

While the work on the thesis was still ongoing and the register model generation for UVM was already finished the central design services group created a generator for the register model using mine as a reference, but with certain features kept more general. This included especially the extended register access definitions (the general ones already present in the data model were mapped on predefined ones, but this does not cover all UVM types) and the change of the currently selected view of multi-view registers, which has to be done manually. The defined selection with the multiview extension for the data model in the thesis has a benefit for the assertion generation and for the register model it is adapted to the typical use in our department. In general the official generators should be preferred if possible, as dedicated support and improvements are available. For future projects using UVM testbenches the official generators will be used, but custom bitfield classes created in the thesis can be used for special access behavior.
The formal verification, i.e. property file and bind file generation, will be tested in some future projects to see how helpful it proves, as the property and bind file can be used for all sorts of formal verification and also for assertions for dynamic simulations.
The generators for the XML register representation for the laboratory measurements and the VBA source code for the chip testers are already in use for several projects. These generators prove very useful also outside of the context of automated register access verification, and because lots of settings and test functions can only be accessed via the registers, this was already in practical use long before this thesis was finished. Especially the VBA register classes have saved a lot of time, because without any additional work apart from the design modeling this now generates about seven thousand lines of code for a typical project, which would have to be handwritten otherwise.

8.3 Outlook

With the increasing use of modeling for the designs the direct usability of the generators created in this thesis improves, but even more possibilities to automate the process of the register model and property and bind file generation are present. It has been mentioned in several chapters of the thesis, that more modeling and generator use (especially the use of standardized generators) could make manual adaptations obsolete or at least keep them at a minimum. This thesis and the generators originating from it now only target register access behavior checks, which are only a small part of the system functionality. Verification for other parts of the design that are structured in some regular pattern might also be included in future versions of the property file for formal verification and sequences and scoreboards for the verification using UVM.

Bibliography

[1] IEC Standard for Systemverilog - Unified Hardware Design, Specification, and Verification Language (Adoption of IEEE Std 1800-2005). *IEC 62530:2007 (E)*, 2007.

[2] Accellera. *Universal Verification Methodology (UVM) 1.1 Class Reference.* Accellera, June 2011.

[3] Accellera. Universal Verification Methodology (UVM) 1.1 User's Guide. Technical report, Accellera, 2011.

[4] Prashant Aggarwal, Darrow Chu, Vijay Kadamby, and Vigyan Singhal. Planning for End-to-End Formal Using Simulation-Based Coverage: Invited Tutorial. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 9–16, Austin, TX, 2011. FMCAD Inc.

[5] Thomas Alsop. Accellera's Verification Intellectual Property (VIP) and Universal Verification Methodology (UVM). Website/pdf, 2011. Available online at http://www.accellera.org/home/VIP_TSC_2011_article_031611.pdf; visited on August 25th 2011.

[6] Henrik Reif Andersen. An Introduction to Binary Decision Diagrams. Lecture Notes, 1999. Lecture Notes for Efficient Algorithms and Programs at the IT University of Copenhagen.

[7] C. Baier and J.P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[8] B. Bailey. *The Functional Verification of Electronic Systems: An Overview from Various Points of View.* Design Handbook series. International Engineering Consortium, 2005.

[9] M.G. Bartley, D. Galpin, and T. Blackmore. A comparison of three verification techniques: directed testing, pseudo-random testing and property checking. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 819–823, 2002.

[10] Michael Bayer. Mako Templates for Python. Website. Available online at http://www.makotemplates.org/; visited on August 25th 2011.

[11] J. Bergeron. *Writing Testbenches Using SystemVerilog*. Springer Science+Business Media, 2006.

[12] Janick Bergeron. *Writing testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2003.

[13] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking Without BDDs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[14] Tim Blackmore, David Halliwell, Philip Barker, Kerstin Eder, and Naresh Ramaram. Analysing and Closing Simulation Coverage by Automatic Generation and Verification of Formal Properties from Coverage Reports. In John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne, editors, *Integrated Formal Methods*, volume 7321 of *Lecture Notes in Computer Science*, pages 84–98. Springer Berlin Heidelberg, 2012.

[15] Roderick Bloem. Verification and Testing Lecture Notes. Lecture notes to the lecture Verification & Testing held at the Institute for Applied Information Processing and Communications of the Technical University Graz in the winter term 2010/11, 2010.

[16] B.W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *Software, IEEE*, 1(1):75–88, 1984.

[17] Françoise Casaubieilh, Anthony McIsaac, Mike Benjamin, Mike Bartley, François Pogodalla, Frédéric Rocheteau, Mohamed Belhadj, Jeremy Eggleton, Gérard Mas, Geoff Barrett, and Christian Berthet. Functional Verification Methodology of Chameleon Processor. In *Proceedings of the 33rd annual Design Automation Conference*, DAC '96, pages 421–426, New York, NY, USA, 1996. ACM.

[18] Clifford E. Cummings. SystemVerilog Event Regions,Race Avoidance & Guidelines. In *SNUG Boston 2006*, 2006.

[19] Clifford E. Cummings. SystemVerilog Assertions Design Tricks and SVA Bind Files. In *SNUG San Jose 2009*, 2009.

[20] Doulos Ltd. *UVM Golden Reference Guide*. Doulos, 2011.

[21] Edmund M. Clarke et al. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4), 1996.

[22] Gerwin Klein et al. seL4: Formal Verification of an OS Kernel. Technical report, NICTA, 2009.

[23] Mukul R. Prasad et al. A Survey of Recent Advances in SAT-Based Formal Verification. *INTERNATIONAL JOURNAL ON SOFTWARE TOOLS FOR TECHNOLOGY TRANSFER*, 7(2), 2005.

[24] Thomas Ball et al. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. Technical report, Microsoft Corporation, 2004. Available online at http://research.microsoft.com/pubs/70038/tr-2004-08.pdf.

[25] Wolfgang Ecker et al. Metamodeling and Code Generation - The Infineon Approach. In *MeCoES - Metamodelling and Code Generation for Embedded Systems, Proceedings*, pages 1–4, 2012.

[26] Mentor Graphics. Verification Academy. Website, 2011. Available online at http://verificationacademy.com/; visited on August 25th 2011.

[27] Luis Guerra e Silva, L Miguel Silveira, and Joöa Marques-Silva. Algorithms for Solving Boolean Satisfiability in Combinational Circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe*, page 107. ACM, 1999.

[28] A. Hazra, A. Banerjee, S. Mitra, P. Dasgupta, P.P. Chakrabarti, and C.R. Mohan. Cohesive Coverage Management for Simulation and Formal Property Verification. In *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual*, pages 251–256, 2008.

[29] Verisity Design Inc. vmanager. Website/pdf, 2004. Available online at http://www.verisity.com/products/pdf/vmanager.pdf; visited on August 24th 2011.

[30] Intel. Moore's Law 40th Anniversary. Website. Available online at http://www.intel.com/pressroom/kits/events/moores_law_40th/; visited on April 6th 2013.

[31] Sachin A. Basheer Jentil Jose. A Comparison of Assertion Based Formal Verification with Coverage driven Constrained Random Simulation, Experience on a Legacy IP. Website, 2007. Available online at http://www.design-reuse.com/articles/18353/assertion-based-formal-verification.html; visited on April 17th 2013.

[32] H. Kaeslin. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication.* Cambridge University Press, 2008.

[33] Éamonn Linehan, Eamonn O'Toole, and Siobhán Clarke. Model-Driven Automation for Simulation-Based Functional Verification. *ACM Trans. Des. Autom. Electron. Syst.*, 17(3):31:1–31:25, July 2012.

[34] F.H.J. Michelson. *The Art of Verification with SystemVerilog Assertions*. Verification Central, 2006.

[35] Raj S. Mitra. Strategies for Mainstream Usage of Formal Verification. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 800–805, New York, NY, USA, 2008. ACM.

[36] K. Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Texts in theoretical computer science. Springer, 2004.

[37] Vigyan Singhal and Prashant Aggarwal. Using Coverage to Deploy Formal Verification in a Simulation World. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 44–49. Springer Berlin Heidelberg, 2011.

[38] Moshe Vardi. From Church and Prior to PSL. *25 Years of Model Checking*, pages 150–171, 2008.

[39] Wikipedia. Factory Method Pattern. Website. Available online at http://en.wikipedia.org/wiki/Factory_method_pattern; visited on August 29th 2011.

[40] Y. Xu, E. Cerny, A. Silburt, A. Coady, Y. Liu, and P. Pownall. Practical Application of Formal Verification Techniques on a Frame Mux/Demux Chip from Nortel Semiconductors. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 110–124. Springer Berlin Heidelberg, 1999.

[41] Aihong Yao, Jian Wu, and Zhijun Zhang. Functional Coverage Driven Verification for TAU-MVBC. In *Internet Computing for Science and Engineering (ICICSE), 2010 Fifth International Conference on*, pages 89–92, 2010.

[42] Zhili Zhou, Zheng Xie, Xin'an Wang, and Teng Wang. Development of Verification Envioronment for SPI Master Interface Using SystemVerilog. In *Signal Processing (ICSP), 2012 IEEE 11th International Conference on*, volume 3, pages 2188–2192, 2012.