THESIS ON INFORMATICS AND SYSTEM ENGINEERING C70

System Modeling for Processor-Centric Test Automation

ANTON TŠERTOV

TUT PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Engineering

This dissertation was accepted for the defense of the degree of Doctor of Philosophy in Computer and Systems Engineering on December 22, 2011

Supervisors:	Dr. Artur Jutman,
	Prof. Raimund-Johannes Ubar
	Department of Computer Engineering, TUT
Advisor:	Dr. Sergei Devadze
	Department of Computer Engineering, TUT
Opponents:	Prof. Matteo Sonza Reorda
	Politecnico di Torino, Italy
	Mr. Gunnar Carlsson; position: Expert DFT and Test Strategies
	Ericsson AB, Stockholm, Sweden
	Dr. Eduard Petlenkov
	Tallinn University of Technology, Estonia

Defence of the thesis**:** February 9, 2012

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any academic degree.

/Anton Tšertov/

European Union
European Social Fund

Investing in your future

Copyright: Anton Tšertov, 2012
ISSN 1406-4731
ISBN 978-9949-23-232-1 (publication)
ISBN 978-9949-23-233-8 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C70

Süsteemide modelleerimine protsessorikesksete testprogrammide sünteesi automatiseerimiseks

ANTON TŠERTOV

TTÜ
KIRJASTUS

To my family

Abstract

The thesis addresses topics related to the manufacturing board level testing. The restricted capabilities of widely adopted board level test methods with respect to such modern challenges as dynamic (timing-accurate), at-speed and high-speed testing as well as in-system diagnosis of functional failures reveal the demand in methodology that could address these problems altogether. The industry is aware of such methodology, but its applicability is restricted by the test program development cost. This board level testing methodology has many names, but in this thesis it is referred as processor-centric board testing (PCBT).

Despite of various existing implementations of PCBT solutions the general drawback of PCBT is the cost for manual development of the necessary test access and test application functionality (test path) for particular board under test. In comparison to the widely adopted test methods the PCBT is for the most part handcrafted test solution. In many cases, the gap between development effort of PCBT program and the acceptable/planned cost place the limitation on the applicability of PCBT.

The main contribution of this research is the methodology that reduces the cost of PCBT solution. The thesis presents approach to automate the development of a PCBT program. The automatic PCBT program synthesis is based on solving the task of test data transportation through the test path. The test path is modeled with the partial functional and structural modeling of printed circuit board assembly and its components. The proposed approach reduces the effort and time for development of high quality PCBT program using developed automation framework.

Kokkuvõte

Väitekirjas käsitletakse tulemusi, mis kuuluvad elektroonika trükkplaatide testimise valdkonda. Tänapäeval laialt levinud trükkplaatide testimise meetodid ei rahulda nõudmisi, mille seavad üles moodsad tehnoloogiad, kus üha oluliseks on saamas süsteemide kiiruslik testimine ja süsteemisisese funktsionaalse diagnoosi vajadus, ja millest seetõttu tuleneb vajadus uue metodoloogilise lähenemise järele, kus mõlemad probleemid oleksid adresseeritud koos. Tööstuses on selline metodoloogia arendamisel, kuid kitsaskohaks on testprogrammide koostamise suur maksumus. Käesolevas töös on leidnud nimetatud metodoloogia uudse lahenduse protsessorikesksete elektroonika trükkplaatide testimise (PETT) nime all.

Vaatamata mitmesuguste PETT lahenduste olemasolule, on siin üldiseks puuduseks komplitseeritud käsitsitöö kõrge maksumus, mis on seotud trükkplaadi komponentide keeruka funktsionaalsuse analüüsi ning arvestamisega testimisteede aktiveerimisel ja testprotseduuride koostamisel. Suurem osa trükkplaatide testide programmeerimise tööst tehakse tänapäeval käsitsi. Enamikel juhtudel ületab testide programmeerimistöö reaalne maht aktsepteeritavad kulud sedavõrd, et osutub paratamatuks teha järeleandmisi testimise kvaliteedi osas.

Käesoleva töö põhitulemuseks on uus metodoloogia, mis viib PETT lahenduste kulutused sedavõrd alla, et need oleksid vastuvõetavad tööstuses. Väitekirjas on esitatud meetod PETT programmide sünteesi automatiseerimiseks. Sünteesi formaliseerimise aluseks on uudne lahendus, kuidas üle kanda ja arvutada testandmeid mööda testimisteed trükkplaadil. Seejuures modelleeritakse testimisteed sellel asuvate komponentide ja nendevaheliste protokollide funktsionaalsete ning struktuursete mudelite abil. Väljatöötatud automatiseerimiskeskkonna kasutamine vähendab testide projekteerimise töömahtu ja testide sünteesiks kuluvat aega, tagades samal ajal kõrge kvaliteediga PETT programmide sünteesi.

Acknowledgements

I would like to show appreciation to everybody who helped me with advice and support during my PhD studies.

In particular, I would like to express deep gratitude to my supervisor Dr. Artur Jutman for helping me to do the first steps in the field of digital test and for encouraging me to finish this thesis. It was a big pleasure to continue my research after master studies under his intelligent guidance. I am very much thankful to my other supervisor Prof. Raimund-Johannes Ubar for his wise advice and support during the research work that is behind this thesis.

I would like to thank Dr. Sergei Devadze for his invaluable contribution and unselfish help. He was so much involved in this research that he deserves to be mentioned as one of the supervisors.

Special thanks to Dr. Margus Kruus, the head of department of Computer Engineering for his warm attitude to the young researches and for outstanding environment for productive work and study.

Furthermore, I would also like to express my appreciation to my good colleagues Dr. Jaan Raik, Dr. Maksim Jenihhin and to the group of young researches: Igor Aleksejev, Sergei Kostin, Konstantin Shibin and Anton Tsepurov.

I would also like to express my sincere gratitude to Thomas Wenzel (CEO, GÖPEL electronic) for the opportunity to participate in the development of board-level test solutions. The valuable experience and inspiration for the research was received during the fruitful cooperation with GÖPEL electronic.

I would like to acknowledge the following organizations that have supported my PhD studies: Tallinn University of Technology, Testonica Lab, Enterprise Estonia funded ELIKO Development Centre, National Graduate School in Information and Communication Technologies (IKTDK), and Estonian Information Technology Foundation (EITSA).

Finally, I would like to thank my family for all the patience and care. In particular, I would like to mention my parents Oleg and Larissa and my beloved fiancée Anna. Thank you!

List of Publications

Board and system level test optimization

- A. Tsertov, A. Jutman, S. Devadze, R. Ubar, Automatic SoC Level Test Path Synthesis Based on Partial Functional Models – *Proc. of 20th Asian Test Symposium*, New Delhi, India, 2011, pp. 532-538.

- A. Tsertov, A. Jutman, S. Devadze, R. Ubar, SoC and Board Modeling for Processor-Centric Board Testing – *Proc. of 14th EUROMICRO Conference on Digital System Design - DSD 2011*, Oulu, Finland, 2011, pp. 575- 582

- A. Tsertov, A. Jutman, S. Devadze, Testing Beyond the SoCs in a Lego Style – *Proc. of IEEE East-West Design & Test Symposium*, St. Petersburg, Russia, 2010, pp. 334-338.

- A. Tsertov, A. Jutman, S. Devadze, Automation of Testing Beyond the SoCs. – *Proc. Of 4th IKTDK Conference*, Essu Mois, Estonia, 2010, pp. 29-32

- S. Devadze, A. Jutman, A. Tsertov, R. Ubar, Microprocessor Modeling for Board Level Test Access Automation – *Proc. of 10th IEEE Workshop on RTL and High Level Testing*, Hong Kong, China, 2009, pp. 154-159.

- S. Devadze, A. Jutman, A. Tsertov, M. Instenberg, R. Ubar, Microprocessor-based System Test using Debug Interface – *Proc. of 26th IEEE NORCHIP Conference*, Tallinn, Estonia, 2008, pp. 98-101.

BIST optimization

- A. Jutman, A. Tsertov, R. Ubar, Calculation of LFSR Seed and Polynomial Pair for BIST Applications – *Proc. of 11th IEEE Workshop on Design and Diagnostics of Electronic Systems*, Bratislava, Slovakia, 2008, pp. 275-278

HW-SW co-design

- U. Reinsalu, S. Devadze, A. Jutman, A. Tsertov, P. Ellervee, "Hardware/Software co-design in practice: MEMOCODE'08 contents experience", *Proc. of 3rd IKTDK Conference*, Voore, Estonia, 2008, pp. 55-58.
-

Laboratory environment for education and research of design and test

- R. Ubar, A. Jutman, J. Raik, S. Devadze, I. Aleksejev, A. Chepurov, A. Tsertov, S. Kostin, E. Orasson, H.-D. Wuttke, E-Learning Environment for WEB-Based Study of Testing – *Proc. of the 8th European Workshop on Microelectronics Education,* Darmstadt, Germany, 2010, pp. 47-52.

- A. Jutman, A. Tsertov, A. Tsepurov, I. Aleksejev, R. Ubar, H.-D. Wuttke, Teaching Digital Test with BIST Analyzer – *Proc. of 19th EAEEIE Annual Conference*, Tallinn, Estonia, 2008, pp. 123-128.

- A. Jutman, A. Tsertov, A. Tsepurov, I. Aleksejev, R. Ubar, H.-D. Wuttke, BIST Analyzer: a Training Platform for SoC Testing – *Proc. of 37th Annual Frontiers in Education Conference*, Milwaukee, USA, 2007, pp. 1534-1539.

- A. Jutman, A. Tsertov, R. Ubar, A tool for advanced learning of LFSR-based testing principles – *Proc. of Baltic Electronics Conference*, Tallinn, Estonia, 2006, pp. 175-178.

- A. Jutman, A. Tsertov, R. Ubar, A Tool for Teaching Pseudo-Random TPG Principles – *Proc. of 17th EAEEIE Conf. on Innovation in Education for Electrical and Information Engineering*, Craiova, Romania, 2006, pp. 182-187.

List of Abbreviations

AC	Alternating current
AOI	Automated optical inspection
API	Application programming interface
ATE	Automatic or Automated test equipment
ATPG	Automatic test pattern generator
AXI	Automated X-ray inspection
BDD	Binary decision diagram
BGA	Ball grid array
BIST	Built-in self-test
BS	Boundary-scan
CAD	Computer-aided design
CS	Constraint solver
CSP	Constraint satisfaction problem
DC	Direct current
DDR	Double data rate memory
DfT	Design for test
DRAM	Dynamic random-access memory
EMF	Eclipse modeling framework
FD	Finite domain
FDV	Finite domain variable
FICT	Fixtureless in-circuit test
GDDR	Graphic double data rate memory

GPR	General purpose register
GUI	Graphical user interface
HDL	Hardware Description Language
HLDD	High-level decision diagram
IC	Integrated circuit
ICT	In-circuit test
IDE	Integrated development environment
IP	Intellectual property
ISA	Instruction set architecture
ISP	In-system programming
JaCoP	Java constraint programming
MDA	Manufacturing defect analysis
NTF	No trouble found
PCB	Printed circuit board
PCBA	Printed circuit board assembly
PCBT	Processor-centric board test
PRPG	Pseudo random pattern generator
RT-Level	Register-transfer level
SIMD	Single instruction multiple data
SMT	Surface mount technology
SoC	System on chip
SRAM	Static random-access memory
SUT	System under test
SVF	Serial vector format
TAP	Test access port
TCK	Test clock
TDI	Test data in
TDO	Test data out
THT	Through-hole technology
TLM	Transaction level modeling

TMS	Test mode state
TRST	Test reset
UML	Unified modeling language
UUT	Unit under test
VHDL	VHSIC hardware description language
VHSIC	Very-high-speed integrated circuits
XML	Extensible markup language
μC	Microcontroller
μP	Microprocessor

Contents

Chapter 1
INTRODUCTION

This introductory chapter gives an overview of the research area addressed by current thesis. The motivation for the work is presented as opening words that are followed by the formulation of the problem and the outline of main contributions. In the last part of the chapter the organization of the thesis is described.

1.1 Motivation

Almost every aspect of modern life depends on the correct functioning of the digital devices. Hence, today the dependability is concerned not only in limited applications in power, medical and aerospace industries, but also in less critical applications like mobile devices and household equipment. The dependability is an important property of particular microelectronic device and it is reflected in its cost. The manufacturer cannot afford the mobile device to be at a cost of communication equipment of the space shuttle due to the high dependability rate. Hence, reasonable testing of microelectronic products and components is required to guarantee an acceptable level of product reliability and a competitive cost. There are constant drive for cost reduction and shorter time-to-market for new products from the hot growth markets such as portable computer products, portable medical equipment, and automotive products.

One of the top manufacturing research priorities reported by International Electronics Manufacturing Initiative (iNEMI) [1] is advanced test solutions for high density boards. The driving forces for constant demand in development of efficient test solutions are miniaturization that is influenced by the rapid development of portable and handheld products, higher performance levels of high-end systems and material evolution.

The printed circuit board (PCB) design has been constantly evolving due to previously mentioned manufacturing trends. Today, the ordinary PCBs may contain more than a dozen of intermediate layers for conducting paths and is populated with components that have packages (e.g. Ball Grid Array (BGA)) with hard-to-access pins. Despite a significant progress in semiconductor technologies, testing of assembled PCBs is often performed using yesteryears means, i.e. accordingly to the Boundary-Scan (BS) standard [2] developed in 1990. As a result, testing of specific manufacturing defect classes becomes economically inefficient for most applications.

The current thesis is focused on the processor-centric board test methodology that combines benefits of structural and functional test strategies. The major contribution of the thesis is the modeling methodology with the goal of automatic test program synthesis for a processor-centric board. Application of the proposed methodology closes the gap between acceptable system reliability rate and the cost of the system test solution.

1.2 Open issues in printed circuit assembly test

Today PCB assemblies (PCBAs) are used in most of commercially produced electronic devices. Electronic components on the PCB are connected using conductive pathways. Contemporary PCBs have multiple layers of separately etched thin boards. Complex PCBs may be stuffed with 50 or more layers. The surface layers are populated with electronic components while most of the interconnections are hidden into internal layers (trace layers). The usage of internal layers in PCB design reduces the dimensions of the board. However, structures on the internal layers are inaccessible for *Flying probe* or a *Bed of nails* tester. Thus additional design-for-testability (DfT) structures are implemented to test the interconnections on the internal layers.

The widely adopted DfT structures described in IEEE 1149.1 standard [2] provides means to test interconnects, clusters of logic, memories etc. without touching a board with physical test probe. The Boundary-Scan (BS) architecture and Test Access Port (TAP) described in this standard are also used for debugging purposes such as watching the pin states, voltage measuring or providing access to internal debug module of a programmable device. Despite of ubiquitous presence of BS structures in modern electronic systems and components, the application of BS is limited due to the low signal frequency. Typically test clock (TCK) frequency is in range from 1 MHz to 40 MHz, whereas actual test application frequency is much lower due to the long shift that precedes each test pattern. Thus, BS-based tests have at least two restrictions. First, there are device classes such as high speed memories that do not support communication at low frequencies. In [3] is reported that even SRAM/DRAM interconnects are causing problems when tested with BS (58% respondents occasionally encountered problems and 28% said they frequently did).

Second, timing-related defects (e.g. transition faults, crosstalk or switching noise) manifest themselves only at high signal frequencies.

Functional test [4] and interconnect Built-in Self-Test (BIST) [5] could potentially overcome drawbacks of BS. However, interconnect BIST requires implementation of additional DfT structures, which is not acceptable in many cases. On the other hand, functional test does not require any modifications to the PCBA, but on its own does not produce measurable coverage of structural faults.

The BS-based tests and other tests that target structural faults belong to the structural type of test. Structural test is based on the *fault models* and it checks only for failures that can be represented by the used fault models, such as stuck-at-fault model. The effectiveness of the test is measured by the *fault coverage* [6]. The fault coverage is a percentage of the detected fault by the test set. The complete set obtains 100% fault coverage. This is desirable fault coverage, but in practice, rarely achievable in most systems under tests (SUTs). Hence, some faults remain undetected when the fault coverage is less than 100%. Moreover, even 100% fault coverage does not guarantee that all possible faults are detectable and the SUT is fault-free. The faults that are not detected by the test set (fault coverage is less than 100%) or not detectable by used fault models are called *test escapes* [7].

Test escape may manifest itself during the functional test or in the normal operation of the device. In recent years, the number of reports of failed system level functional tests at a client side was constantly growing [8]. The reason for functional test failures when the structural tests pass lies in the test escapes. The scenario when system or component refuses to fail on retest (structural test) after it was returned from the field or system customer as having failed is known as No Trouble Found (NTF) scenario [8], [9]. After the NTF is solved, the structural test may be complemented to escape the NTF scenario with the same symptoms in the future. However, more intelligent approach is needed to address NTF problem rather than hide-and-seek with the test escapes.

Test methodology that could potentially address problems of structural test (BS) using benefits of functional test and providing measurable fault coverage is processor-centric board test (PCBT) [10]. This test technology uses functionality of microprocessor (µP) or microcontroller (µC) devices to deliver test patterns to PCB peripheral components outside the programmable devices themselves. The attractiveness of µP or µC-centric solution is very high due to the usage of existing on-board DfT structures without any modifications.

In commercial board level test systems the processor-centric approach is widely adopted [11], [10]. General disadvantage of its implementations is that processor-centric board model (set of functions) is prepared by hand. As a result the cost for development of the test program is much higher than for traditional BS tests that are mostly automated.

1.3 Problem formulation

The processor-centric board launches the execution of the boot routine on the μP (depends on the board configuration) each time the board is switched on. The boot routine is the program that setups μP for operation with particular PCBA peripherals and loads runtime environment or operating system. In addition, the boot program has the board self-test routine that starts the set of tests to determine the integrity of the board. One of the examples of such boot programs is well known Basic Input/Output Systems (BIOS), which is equipped with Power-On Self-Test (POST) functionality [12]. For the normal startup of the PCBA the boot routine should be preloaded into memory that μP uses to boot from.

The problem is that PCBA after manufacturing is not yet loaded with such boot program to run the POST. Typically, the boot program is loaded after the board is tested for manufacturing defects. However, the at-speed testing of PCBA requires the board to be preconfigured. The outstanding property of the PCBT is that this method is capable to test PCBA at-speed just after it is manufactured. The execution of PCBT program starts with functions that setup μP and controllers of μP to communicate with DUTs as in normal operational mode. The development of these functions requires plenty of time of highly skilled test engineer, which is the main barrier to reaching the same "popularity" for PCBT as for BS.

The functionality of PCBT is used not only to test the board, but also to program the boot programs and operating systems into on-board memory using the same test setup. The challenge in programming on-board memories via serial test access port (JTAG TAP) is the programming time. To meet the timing requirements the PCBT program is developed in low level programming languages. The optimized code reduces the data transfers through the JTAG TAP and increases the performance of the program. Obviously, the low level programming extends the development time of the PCBT solution.

In this thesis the problems of test program development for assembled PCB is addressed. In particular, the labor effort to develop the PCBT program is concerned and the methodology for test time and programming time estimation is proposed. The presented work attempts to automate the development of μP or μC-centric board level test programs using adaptive modeling methodology. The proposed modeling methodology implies automatic creation of structural models of PCBA and its components. The proposed methodology also describes the approach for creating partial behavioral models of PCBA components.

1.4 Thesis contribution

The main contribution of the current thesis is a novel approach to automate the development of test program for processor-centric boards. The workflow for automated development of PCBT programs is introduced for the first time in literature.

The sub-contributions that have been made in frames of the research work on this thesis are outlined below:

- Formulae for test application time calculation - The simulation-free calculation of the test application time is useful for fast cost estimation of the manufacturing board test solution. Moreover, these formulae are helpful for comparison of different test application strategies for a given test case.
- The metamodel and its implementation for structural models of PCBA, SoCs and other PCBA components
- The metamodel and its implementation for behavioral models of ICs
- A novel methodology for test data path modeling - Structural models are augmented by behavioral models to assemble the uniform model of the PCBA that is used in test development automation.
- A novel approach and implementation of automated synthesis of PCBT program in SVF
- A new approach for automated synthesis of the VHDL test bench
- The toolchain of developed programs - This toolchain is a platform with broad research capabilities in the field of board level test. It also provides integration possibilities with third-party tools for test and debug of assembled PCBs.

1.5 Thesis structure

The presented thesis consists of 7 chapters. The rest of it is organized as follows.

Chapter 2 forms a background on the researched topic and reviews the state-of-the-art in the field of manufacturing board level test. It reveals the problem areas and presents motivation for the given research work.

Chapter 3 gives a description for the PCBT program. This chapter starts with presenting the typical functionality of the test program. Then the test access and test application parts of the program are discussed in details. The automation of the development of the presented PCBT program is a general topic for the following chapters.

Chapter 4 is dedicated to board and electronic components modeling. The basic knowledge of modeling techniques is given in the beginning of the chapter. Then the proposing modeling methodology for board and electronic components is presented.

Chapter 5 describes the proposed approach for automated test program synthesis on the basis of the proposed modeling methodology. Firstly, the details of PCBT program synthesis are explored and a typical development flow is examined to present the automated flow. The feasibility of proposed methodology is studied on the experiments with ITC99 benchmarks.

Chapter 6 presents the developed toolchain for board and electronic components modeling and automated test program synthesis based on the structural and behavioral models. The chapter is concluded with the description of integration potential into boundary-scan test systems and open source computer-aided design (CAD) software.

Chapter 7 draws conclusions for the thesis and discusses directions for future work.

Chapter 2
BACKGROUND

This chapter presents background knowledge for the topics related to current research. The introduction to manufacturing board test is given in the beginning of the chapter. The introduction is followed by the description of the in-circuit test technology. The notion of IEEE 1149.1 standard is described in conjunction with derived standards to draw the comprehensive picture of boundary scan test technology flavors. Finally, the definition and application of the processor-centric board test is presented to complete the set of the available test solutions in manufacturing board test

2.1 State of the art in manufacturing board test

High-density printed circuit board assembly (PCBA) requires special methods for test access. The test method that provides physical access to component leads, test pads and vias was a viable solution until multi-layered PCBs entered the mass production. Today, in-circuit test (ICT) methods cannot provide solely sufficient test coverage to meet stringent standard quality requirement. BS together with fixtureless ICT (FICT) such as automated optical (AOI) and X-ray inspection (AXI) is complementing ICT to provide test for static faults (opens and shorts) on even most densely populated PCBs [13], [14].

During the last decade the test requirements in PCBA mass productions has changed its focus from finding component failures towards finding manufacturing process faults due to the continuous improvement in overall component quality. In the process of PCB population, industry has moved from through-hole technology (THT) to surface-mount technology (SMT). That has made changes to the fault spectrum. In THT the most probabilistic fault type was shorted connections.

However, with SMT the major problem lies in open connections, but solder shorts are still a noteworthy problem. The other significant fault types are misaligned, missing or wrong components.

The test strategy for manufacturing defects is initially based on the assumption that component supplier is shipping only good parts. Hence, all manufacturing test methods should be capable to detect fault that are caused by soldering related problems and accurate placement of the correct components on the PCB.

In-Circuit Test (ICT) [15] uses a bed of nails fixture for mechanic access to electrical nets of the PCB. Each individual nail in the fixture has a wire connection with the external tester. Nails in the fixture are thoroughly allocated to simultaneously create a steady physical connection to the test points, non-masked vias and soldered leads. By the means of standard complex-impedance measurements, these manufacturing defect analysis tests can be run without powering the PCB.

Power-off testing eliminates the risk to damage a misplaced component by applying a power to its leads. Hence, manufacturing defect analysis (MDA) [16] test should be run before the power-on tests. In-circuit MDA test systems provide an identification of a failing component when a fault is detected. Since components are tested in isolation. The failing component is identified by designating its part reference number. Besides opens and shorts all the typical faults for analog components are testable by MDA techniques. The list of testable analog components consists of resistors, capacitors, diodes, inductors, transformers, transistors.

The general drawback of the ICT is the necessity to have a physical contact with the PCB. In-practice, access to all the electrical nets is hardly possible due to the lack of space for test probes on the surface of contemporary PCB. Probing the SMT lead is not a solution, as probe may introduce extensive pressure to the lead, causing a bad connection to appear to be good. The problems of ICT technology were solved by integrating the tester's probes into the chips and to control them via a simple serial bus. In 1990 this technology became an IEEE 1149.1 standard [2] (Boundary-Scan).

BS is only applicable where IEEE 1149.1 standard is supported on the device level. On the one hand BS is characterized by low speed and limited coverage of dynamic faults. On the other hand BS provides very good [17] diagnostic capabilities, low-cost equipment and it is applicable to a non-functioning system. Due to that BS is heavily used in PCBA structural test and PCBA debug.

On the contrary to BS the functional tests are executed at full speed of the board. Thus, the dynamic faults that escaped the BS tests are detectable with functional test. In general, "functional testing verifies board performance mimicking its behavior in the target system" [18]. However, there are several major problems of functional test. The first is that "functional test is traditionally the most expensive technique"

[18] and the second is that the diagnosis of the cause of functional test failure may take hours.

The low speed nature of BS affects badly the test application time. As a result, the BS test technique has to be complemented by a solution that supports high-speed or at-speed test application mechanisms [5],[19]. This problem has been understood by major BS-tools providers and some early PCBT solutions have been developed. The good examples of state of the art PCBT solutions are Goepel Electronic's VarioTAP® technology [11] and Processor-Controlled Test (PCT) [10] for board level test purposes from ASSET InterTech.

Table 2-1 *Application of board level test methods*

Application domain	**Solder Paste Inspection**	**AOI pre reflow**	**AOI post reflow**	**AXI post reflow**	**ICT/MDA**	**Boundary Scan**	**Board level functional**	**Dependa-bility score**
Automotive								
Entertainment	4	3	3	2	5	2	4	23
Safety	4	4	4	2	5	3	3	25
Portables								
Mobiles	4	3	3	2	0	3	5	20
Netcom								
Consumer	3	3	3	1	3	4	3	20
Enterprise	3	3	3	2	5	5	3	24
Service providers	4	3	3	3	5	5	5	28
Office Systems								
Desktop	2	3	3	2	5	3	5	23
Mobile	4	3	5	1	5	3	5	26
Servers/High end	4	3	5	1	5	5	5	28
Medical								
Imaging	1	2	5	3	4	4	4	23
Monitoring	1	2	5	3	4	4	4	23
Implantables	5	4	4	2	4	4	5	28
Total Method Applicability	39	36	46	24	50	45	51	

Goepel Electronic's VarioTAP® technology is a simplified model of MPU SoC that contains a set of mixed test access/application/configuration functions developed in an ad-hoc manner for a given board. Despite the fact, that VarioTAP® reuses some components (device libraries, debugger, test coverage analysis) from the existing BS test projects, the library of μP/μC models has to be prepared manually by an experienced software engineer using standard algorithmic language (e.g. C/C++).

Processor-Controlled Test (PCT) for board level test purposes that comes from ASSET InterTech, besides the drawback of VarioTAP® (handcrafted solution) also has a weak integration with BS and very limited test automation.

These PCBT technologies (VarioTAP® and PCT) are considered as BS-complementary solutions and are located in between BS and functional test. In addition, the solution from Kozio® [20] is seen as complementary solution to traditional functional test. Kozio® suggests to boot the board with their operating system for board(system)-level functional testing. The location of this solution is in between PCBT and functional test, as it provides capabilities for diagnosis similar to PCBT approaches, but does not focus on testing of structural defects.

Various solutions exist for system-level testing for manufacturing defects of complex electronic boards, but all of them have certain limitations. As a result, the selection of appropriate test strategy (set of tests and test methods) is a not a trivial task. The application of various test methods in different product domains is shown in Table 2-1. Table 2-1 is a modified representation of the initial table from [21]. The textual score for each method in the initial table was substituted by equivalent number (0 - Never, 1 - Audit only, 2 - Rarely, 3 - Sometimes, 4 – Mostly, 5 - Always). The initial table was also supplemented with additional row that summarizes the applicability of every test method for listed application. The dominating manufacturing test technologies are functional test and ICT. Thus according to iNEMI [21] Boundary scan is an important, but not an indispensable test technique for many applications. In addition, the dependability score was calculated for each application to highlight the range products that are most heavily tested for manufacturing defects according to the iNEMI data [21].

2.2 Boundary-Scan and its flavors

BS popularity was continuously growing since 1990 and still growing because of ease of adapting with new PCBA component and board level technologies. These adaptations later developed into the IEEE 1149.x standards. The need in 1149.x standards was also caused by inability of the 1149.1 to address the following problems in manufacturing board level test:

- Dynamic defects (delay, crosstalk)
- Interconnect test for analog, digital and mixed-signal and discrete components
- Communication with high-speed memories (DDR3, GDDR5)
- Parallel busses with timing-critical (accurate) protocols
- High speed serial busses
- Fast In-System Programming (ISP)

In this chapter the key points of several IEEE 1149.x standards are briefly discussed to complete the picture of the available board level test solutions and trends.

IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture defines a solution to the problem of testing assembled PCBs and other products based on highly complex digital integrated circuits and high-density surface-mounting assembly techniques [2]. It also provides means for accessing and controlling DfT features built into the digital integrated circuits themselves. This standard defines the TAP port used for communication between external tester and BS infrastructure inside the PCBA components. The list of TAP port pins consists of test clock (TCK), test mode state (TMS), test data in (TDI), test data out (TDO) and optional test reset (TRST). BS infrastructure encloses TAP pins, boundary-scan register

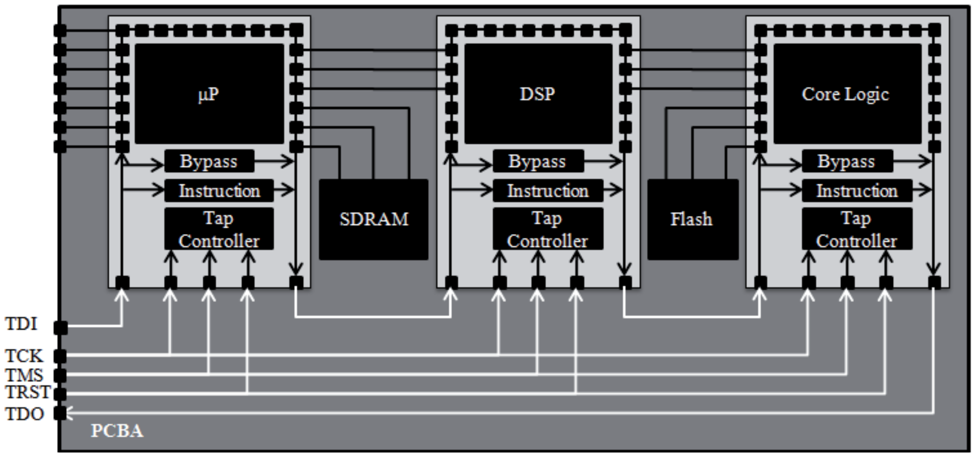


Figure 2-1 Boundary-Scan architecture on the example of simplified PCBA

bypass register and instruction register, and TAP controller state machine (Figure 2-1). The TAP controller state machine (Figure 2-2) selects which type of register (data or instruction) is in use. The purpose of instruction register is to select one of the data registers in accordance with predefined instruction codes. There are four mandatory instructions defined in the standard: EXTEST selects Boundary scan register, BYPASS – Bypass register, SAMPLE – Boundary scan register and PRELOAD – Boundary scan register. The standard also specifies a set of optional instructions: INTEST – Boundary scan register, IDCODE – Identification register, USERCODE – Identification register (for PLDs), RUNBIST – Result register, CLAMP – Bypass (Boundary scan register value is on the device output pins), and HIGHZ – Bypass (device output pins are in high-z state). These ten instructions are known as public instructions, but chip vendor is free to add private instructions that meet the particular testing needs.

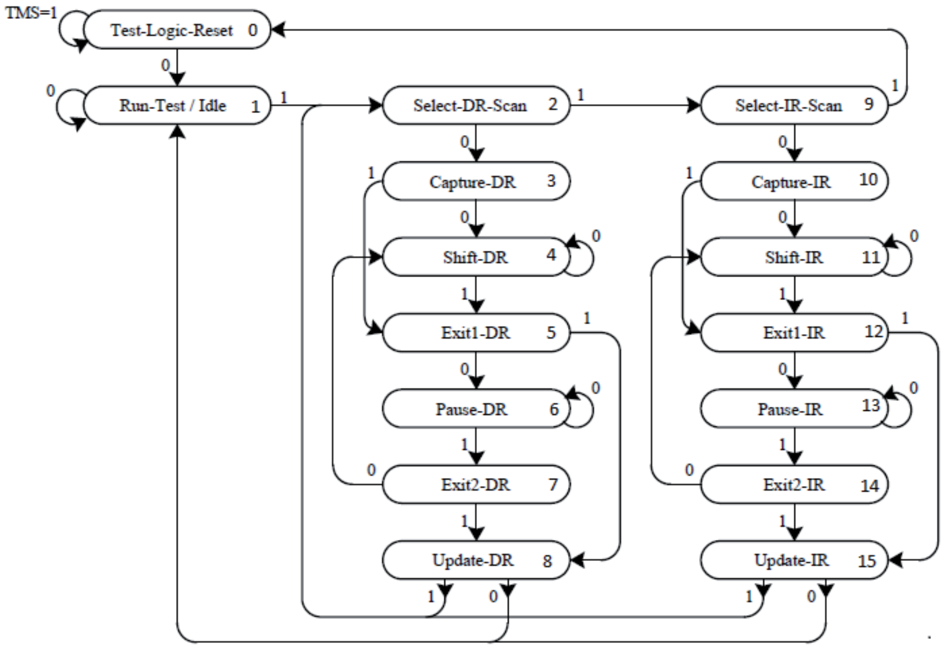


Figure 2-2 TAP Controller state machine

The next standard to be mentioned is IEEE 1149.4 standard that describes mixed signal test bus. The testability structure for digital circuits described in IEEE Std. 1149.1 has been extended to provide similar facilities for mixed-signal circuits [22]. IEEE-1149.4 defines a two-wire analog bus consisting of an analog drive and analog sense pin. By including circuitry within the IC to connect these pins to different analog nodes, the JTAG port (IEEE 1149.1 Std. TAP) can be used to perform analog and mixed signal measurements. First objective of this standard is to provide interconnect test capability for a PCBA populated with analog, digital and mixed-signal and discrete components. Second objective is “extended” interconnect which includes analog measurements to compute parameters of discrete component [22] (e.g. capacitor, resistor) (Figure 2-3). A decade after release, the IEEE-1149.4 standard has not seen wide adoption.

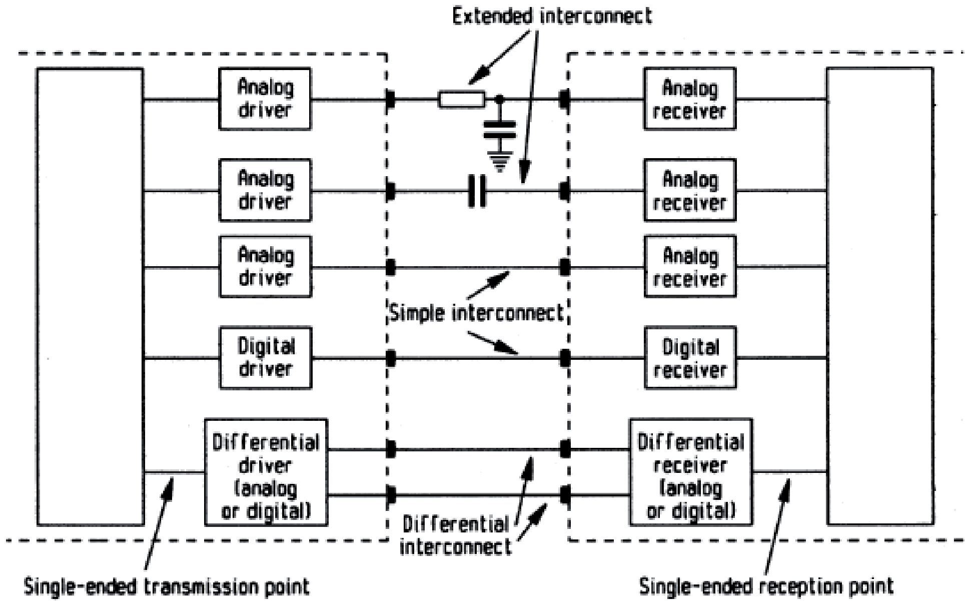

Extended interconnect
Analog driver
Analog driver
Analog driver
Digital driver
Differential driver (analog or digital)
Simple interconnect
Differential interconnect
Analog receiver
Analog receiver
Analog receiver
Digital receiver
Differential receiver (analog or digital)
Single-ended transmission point
Single-ended reception point

Figure 2-3 Simple, extended and differential interconnect [8]

The solution for BS-based testing of high-speed digital networks is proposed in IEEE 1149.6 standard. The IEEE-1149.6 standard, released in 2003, expands on the 1149.1 standard by adding additional capabilities for transmitting and receiving test signals over advanced (high speed) digital networks. “The additional testability elements added by this standard to integrated circuits (ICs) allow interconnect testing with enhanced coverage, to be conducted on differential signal pathways and and/or where AC-coupling (which blocks normal DC Test Signals) has been used on signal paths between ICs” [23]. The objective of this standard is to define design rules for proposed testability circuitry in addition to design for testability structures specified by IEEE 1149.1. IEEE 1149.6 requires design paradigm change and considerable investments from component vendors. Currently, IEEE 1149.6 standard is by far not as popular in practical applications as IEEE 1149.1, although, it was approved in 2003.

The miniaturization of digital devices and new chip design methodologies like System-in-Package (SiP [24]) and Package-on-Package (PoP [25]) questioned the applicability of IEEE 1149.1 standard. The key issues were the number of pins and the appropriate accommodation of multiple on-chip embedded TAP Controllers. The solution that helps to maintain compliance to IEEE 1149.1 in multi-TAP SoCs, solves the above mentioned problems and extends the chip level debug capabilities is IEEE 1149.7 [26].

The high-speed differential links is an open issue for 1149.1 which is addressed in pending standard IEEE 1149.8.1 [27]. In 1149.8.1 is proposed that BS-enabled device only sends the test stimulus to the link and on the other end the capacitive sensing plate on top of the receiving device obtains the test response.

The common challenge for 1149.x standards is that they need to be adopted in design flow before they can be useful. Typically PCBA still contains components that are not compliant to IEEE 1149.1 standard, although, most of the manufactured PCBAs are equipped with BS DfT structures. As a result, the BS-enabled devices are often used as a doorway (from the point of view of external tester) to access non-BS PCBA components for testing purposes. A good example of such doorway is a µP/µC device. The test/debug access in µP/µC is usually based on a JTAG port [28]. As a result, it becomes possible to utilize the processor as a test access mechanism in the board-level structural test (e.g. as an extension of the Boundary Scan (BS) technique [28]). In case when µP/µC is used as onboard tester, BS plays the role of a communication channel for test application mechanisms inside µP/µC rather than a test application method. The test application by mechanisms inside µP/µC (e.g. peripheral controller) solves most of the speed related problems of BS that were mentioned in the beginning of Section 2.2.

The idea of using an embedded µP/µC cores to execute the test program is not new and is widely used for SoC testing and debugging purposes. This testing paradigm was initially known as software-based self-test (SBST) [29]. Many researchers have proposed ideas for testing SoC components where an embedded µP/µC core [30], [31], [32] controls the test data traffic between the test controller and the SoC. Moreover, there are study proposing specially developed IP core [33] to be implemented in the SoC for solving the test, diagnosis and silicon debug issues. The researchers are very active in the field of SBST of SoCs, nevertheless, there is a clear lack of recent academic research on testing beyond the SoCs.

2.3 Processor-centric board test

Back in 1980s industry already was facing a problem of at-speed or high-speed testing of dynamic faults. The functional testers were capable to apply successive patterns at high speed to the UUT (Unit Under Test), but the true rate was actually lower due to the need of measure the responses and react in real time. Then the industry has responded by the evolution of a test technique that “seeks to ‘*emulate*’ the operation of the microprocessor while the remainder of PCB continues to run at its dynamic speed but under the local control of the test system” [34].

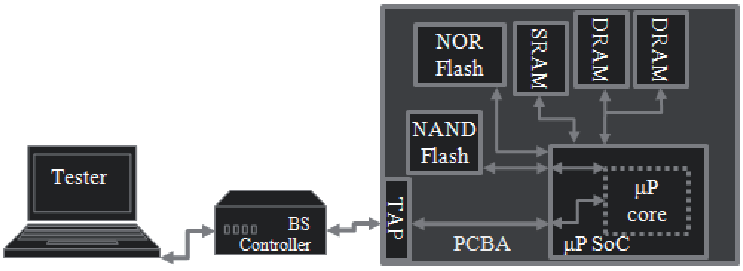


Figure 2-4 Test setup for PCBT

This test technique has been employed in following application. The test system is considered as a 'bus addresser' that communicates with the PCBA components through the interconnection lines of the PCB. The access to the µP pins is gained through the multi-way clip or a substitute mating plug for the µP on the PCBA under test. The PCB tester with the ability to test at dynamic speed through the pins of emulated µP writes and reads words of test data on the UUT interconnects in order to test the PCB peripheral components and associated circuitry in exactly the same way as the µP would do in normal operation.

However, emulation-based test did not gain much attention back in 1980s due to its cost. One of the factors that influenced the cost of emulation-based solution was the absence of standardized way to access µP pins on the PCB. For every new design a custom solution was invented. This issue has been partially solved by the IEEE 1149.1 – 1990 standard. Then the boundary scan register was used to drive and test the nets connected to the µP pins. However, the communication frequencies were continuously growing and the testing of the dynamic faults became limited with BS-based tests. Later in the XXI century industry addresses this problem by processor-centric board test (PCBT) technology. Sometimes, PCBT is still called as processor emulation based test.

The PCBT technology is based on the same principles as µP emulation based solution with the difference that µP internal functionality is not isolated, but µP is considered as an on-board tester. The µP is controlled from external tester via the BS infrastructure on the board. BS standard (IEEE 1149.1) only targets the path from PCB connector to the JTAG TAP port of the µP. The implementation of the internal debug logic of the µP is specific for every System-on-Chip (SoC) vendor.

Generally, PCBT approach uses system's µP to run test routines. The actual control over the µP is performed by external tester. The idea is to apply tests at the actual operation speed of the SUT. A µP is playing the role of an internal (in-system) tester which has an access to PCBA components and interfaces (Figure 2-4). The test software that emulates system normal operation from the point of view of UUTs is executed on a µP. This test software has two possible execution modes:

online and *offline* modes; that are described in Section 2.3.2. Both execution modes are controlled from external tester. Typically the test software consists of the test access (Section 2.3.1) and the test application routines (Section 2.3.2).

PCBT has a good potential to reach high fault coverage, because of the architecture of many electronic systems. The processor usually has to interact with the other PCBA components and thus has good access to them (e.g. through a communication bus). Thus, PCBT can achieve high fault coverage without relying on hardware design modifications or external test equipment.

Nowadays, PCBT is a test strategy that is used not only to test and diagnose static and dynamic faults, but also to perform in-system programming of on-board memories and µP internal memory, to contribute to functional test coverage and to verify the component placement on the PCB. Most of these tasks can be completed by other test methods, but none of them is capable to cover all. Hence, PCBT is a very efficient solution but still costly and effort-hungry as most of the test program functionality is created manually. The comparison of the test methods is given in Table 2-2. The points of comparison were selected to show the main drawbacks of every method. As it is seen from Table 2-2 the only weak side of the PCBT is the "Test automation" and as a result the "Test implementation cost" is high.

Table 2-2 *Comparison of different test methods*

Point of Comparison	**ICT**	**1149.1**	**Functional Test**	**PCBT**
DUT access	Fixed nails	Scan cells	µP	µP
Test implementation cost	High	Low	High	High
Structural Fault Coverage	High	High	Uncountable	High
Dynamic Fault Coverage	No	No	Uncountable	High
Functional Fault Coverage	No	No	High	High
Test access	Low	High	High	High
ISP	No	Slow	Limited	Yes
Test automation	High	High	Low	Low

2.3.1 Test access

JTAG TAP does not provide solely the full access to µP SoC resources (register map, internal memory, external memory controllers, etc.). Moreover, a test engineer needs at least a basic set of debug tool functions like processor halting, breakpoint and watchpoint support, traceability, data flow information and performance measuring. One of the helpful solutions is proposed in NEXUS 5001 Standard. The Nexus standard defines an extensible Auxiliary Port (AUX) that may either be used together with JTAG port or as a stand-alone development port. The Nexus standard defines the auxiliary pin functions, transfer protocols, and standard development features [35], [36]. A set of recommendations (such as additional registers and number of pins) to follow in the debug oriented processor design is also defined.

The alternative solution is provided by the Mobile Industry Processor Interface (MIPI) Test and Debug Working group [37]. This group is exploring a maintenance port, called NIDnT-Port [38], [36] (Narrow Interface for Debug and Test: Speak NIDENT). NIDnT is based on IEEE P1149.7 [26] and the System Trace Module, which includes the MIPI System Trace Protocol (STP) and uses the MIPI Parallel Trace Interface (PTI) for data export.

Traditionally, JTAG TAP remains one of the alternatives for physical connection to µP internal debug interface. Thus, the first part of the test access consists of standard BS infrastructure, which implies JTAG connector on the PCB where the cable from external tester is plugged in, and the scan chain of BS-enabled devices. This part of the test access of the PCBT is the same is in BS-based test. In case if scan chain contains not only µP that is used as internal on-board tester, all the other devices in this scan chain are typically switched to *BYPASS* mode, which is equivalent to the shortest configuration. The shortest configuration of the scan chain provides the fastest communication from external tester to the µP. This configuration of the scan chain is typically needed to efficiently shift the debug instructions through the µP TAP to the debug interface of the µP. Through the debug interface the tester gains an access to the internal buses and components of the µP SoC.

Let us consider the next level of test access as the virtual link between µP and UUT. This link starts at debug interface of the µP and reaches the UUT pins. In general, the establishment of this link implies tuning of the respective controller in the SoC to communicate with the specific UUT. The tuning sequence of commands for peripheral controller inside the µP is transferred through the debug interface. Then the test data is transferred via dedicated UUT controller to the UUT as in the normal system operation. Hence, every signal line between µP and the UUT is exercised in same manner as while executing the system's domain application.

2.3.2 Test application

The test application consists of a test program that controls a microprocessor. The test program may be designed accordingly to *online*[1] or *offline* testing [39] modes.

The *offline* (or autonomous) testing is realized in the following way. The complete test program (test vectors and expected values) is translated into the set of

[1] Online mode of test application is not the same as on-line testing.

microinstructions and loaded as an ordinary micro-program into memory inside the µP. The program execution inside the µP is started by the external tester through debug interface. The test program is constructed in such a way, that the result of execution (PASS or FAIL for complete test) will be stored in one or more general-purpose registers of the µP. After test program execution is finished, the contents of these registers and the result of test execution is retrieved through the debug interface and reported to the external tester for further evaluation and diagnosis.

The *offline* mode is fully independent and does not suppose continuous interaction with an external tester. This autonomous mode requires plenty of memory space in order to store all test vectors as a set of microinstructions. However, another possibility is to implement a special algorithm inside the test program (e.g. walking one, counting sequence, PRPG, etc.) so it will generate driving and expected values on the fly.

The main difference of *online* mode in comparison to *offline mode* is that each test step (i.e. test vector) is executed separately under the control of external tester. Before the test is executed in *online* mode the specially prepared *interpreter program* is loaded into internal memory of the µP during test setup phase. The goal of *interpreter program* is to receive and execute separate microinstructions that will be passed via the debug interface. The test program (that is split into number of test vectors) is synthesized into set of microinstructions and compiled into a sequence of machine code.

On each step, the external tester writes the piece of machine code (that corresponds to the test vector) into certain registers of µP. The *interpreter program* constantly checks the content of these registers for detection of arrival of new microinstructions and executes them. The result of test execution (measured value) is stored in the registers that are accessible by the external tester. Finally, the measured value is compared with the expected one and test execution continues.

There are µP architectures that support instruction injection via debug interface [40], [41]. The injection facilitates instruction execution from the dedicated debug register. This may be used to inject the *interpreter program* command by command instead of loading it into internal memory. However it might extend the test application time. In some cases, the loading of *interpreter program* is time consuming or is not possible (not enough internal memory, internal memory is not accessible or internal memory is occupied by other application), then the instruction injection mechanism becomes extremely useful. The general idea is to load the test vectors to the general purpose registers of the µP. Then, the commands of *interpreter program* that perform test application of previously loaded test vectors are injected. The test results could be obtained in the similar manner by injecting the respective load and store commands to retrieve the tests result signatures to the scan register of TAP for scanning them out to the external tester.

The main drawback of *online* testing in comparison to *offline* is the speed of test pattern application. In *online mode* it is considerably slower due to the overhead of uninterrupted communication via the debug interface. However, the single test pattern is applied at the operational speed in both modes, because the test data is transferred to the UUT through the dedicated peripheral controller that handles the signal timings and the communication protocol. Nevertheless, the fault types, such as delay faults, that require fast subsequent application of test vectors, can still be detected in *online* mode if µP supports multiple store and multiple load instructions (e.g. Single Instruction Multiple Data - SIMD) [42]. These instructions with multiple data sources can be used to emulate the at-speed application of limited number of test vectors to the subsequent addresses.

2.4 Similar works

The academia is not very active in the field of PCBT, however, there are several recent publications on design and implementation of test processors for board-level testing [43], [44]. In [44] authors propose to implement the test processor in FPGA, which is an adaptation of the PCBT technique to FPGA-centric boards. This work was done in cooperation with the company Goepel electronic, hence could be seen as an extension of VarioTAP® technology to the FPGA-centric boards.

In [43] the group from IHP has presented a concept for performing functional tests of asynchronous designs using a specific test processor. The proposed test processor is supposed to be added as a core to the µP SoC or to be implemented as a standalone device on the same board. The automatic synthesis of PCBT program is a part of the proposed approach. However, the implementation details of the test program synthesis are not mentioned in the paper and the status of the tool for the test program synthesis is reported as “under conception” [43]. Meanwhile, at the same conference the methodology for automatic synthesis of the PCBT program and the experimental results were presented in [45] by the author of current thesis.

In the light of these PCBT approaches the PCBT flow described in current thesis is seen as more general and less restricted. The proposed methodology for automatic synthesis of the PCBT program can be applied to any PCBA configuration including the test processor implemented in FPGA or as a co-processor.

2.5 Chapter summary

The purpose of this chapter is to provide a reader with the background information needed to understand the basic principles of the board level manufacturing test. The evolution of the methods and general description of test techniques are presented in the comparative manner. The overview of the recent research activities in the field of PCBT is closing this chapter.

The underlying idea of this chapter is to introduce the drawbacks of the ICT and BS test technologies that can be addressed by the PCBT. The presented arguments lead to the conclusion that PCBT is promising test technology in the manufacturing board testing that guaranties high test quality. However, the cost of PCBT-based solution is high in comparison to the BS solution mainly due to the efforts spend on the manual PCBT test program development.

Chapter 3

PROCESSOR-CENTRIC BOARD TEST FLOW

This chapter gives the description of the typical PCBT flow. In the beginning of the chapter an overview of the PCBT functionality is presented. The standard steps for test path initialization and configuration are explained in the section dedicated to test access. The detailed study of test application steps of the PCBT flow is continued by the analysis of online and offline test application modes. The formulae for test application time estimation are summarized in the concluding section.

3.1 PCBT functionality

The functionality of the PCBT program depends on the test requirements and on the SUT configuration. Test requirements are the set of UUTs including related interconnections that have to be verified, tested or programmed. The UUT verification checks identification code (IDCODE) of the mounted component with the PCBA documentation. Then, interconnect test checks the connections between µP and UUT for static and dynamic faults. If the UUT is a flash memory the interconnect tests are typically substituted by ISP. Besides the interconnect test, it is often needed to do a functional and structural in-situ tests of the UUT. These tests also belong to the PCBT functionality.

Nowadays the programming of the flash memory is often required after the memory is soldered to the PCB. In most cases, it is considered beneficial to program the flash memories with the same tester hardware that is used for verification and test of other components on the PCBA under test. The problem is that ISP has become a very time consuming process because modern flash memories can store much larger images than before. PCBT may reduce the programming of flash memory from hours, as in case with BS, to minutes. The actual ISP time heavily depends on the architecture of the debug interface of the µP, on the instruction set of the µP and on the performance of the external flash memory controller inside the µP SoC.

The structure of the PCBT program for the specific test requirements relies on the standard functional blocks that are adapted to the SUT architecture. The typical configuration is discussed in the following paragraphs. In this chapter the PCBT functionality is described in the statements similar to assembly language to illustrate the overall PCBT program functionality and structure without adapting it to any specific SUT. The two different testing modes (*online* and *offline*) are considered to observe the influence of different testing approaches on the structure of the PCBT program. The complexity of the program is estimated on the example of the most common test requirements.

In the following examples the selected SUT consists of the µP that has the support of instruction injection (this simplifies the description of online testing mode), flash memory with parallel interface and SDRAM. The exact properties and characteristics of the SUT components are irrelevant in the following program examples, because the device specific implementation details are omitted for the sake of simplicity and ease of understanding of the general idea behind the PCBT program.

Let SDRAM be the first to test. The interconnections with µP are tested for static and dynamic faults. The test patterns and algorithms for testing and diagnosis can be reused from interconnect test generated by BS test system. Thus, in this case the same test patterns as for BS-based test are applied at high-speed.

The flash memory is validated against the documented version and capacity by reading the status register. Then, according to the test requirements the memory is programmed with the specified image file. The successful verification of programmed data is sufficient in most cases to conclude that chip is aligned and soldered correctly as well as to claim that the interconnects to µP are functionally tested. Hence, the ISP partially substitutes a functional test for external flash.

Next section presents a study on how previously described test procedures can be implemented in PCBT program.

3.2 Test access

3.2.1 Test path initialization

As initial step, the PCB has to be powered up and checked for consistency. For this task the following sequence of steps is used:

- *Check scan chain* – shift out the IDCODEs of the devices in the scan chain. This allows to identify the order of the devices and to ascertain that BS infrastructure is correct.
- *Initialize the scan chain* – set everything but µP into BYPASS mode. This sets the shortest configuration of the scan chain. The µP should be the only device that listens to the data from external tester. All the devices but µP should bypass the data shifted in and ignore it.
- *Obtain Debug Interface Information* – scan out version and status of the µP debug interface. This secures that communication between BS infrastructure and debug interface is functioning.
- *Stop/Halt µP execution* – stop any program execution in the µP by injecting appropriate instructions via debug interface. Backup the register file and pointer of return address to be able to resume the execution later.
- *Read Status register of µP* – read the status register and other configuration registers to evaluate the state of the µP and its modules. This information is used in the next steps in configuring the µP.

This sequence of five steps is obligatory to include in the beginning of the initialization of the PCB and the µP. Any PCB or µP specific tasks may interleave these steps, but the order of the initial sequence should remain unchanged.

3.2.2 Test path configuration

After the PCB initialization the PCBT program proceeds to the configuration of the µP busses and peripheral controllers. Only those µP modules are configured that are active during communication in native application mode between µP and the PCBA component, which is specified as UUT in the test requirements.

Test path configuration:

- *Set Mode/Privileges* – check if the current mode of instruction execution has enough rights to control and configure the µP controllers. Typically, the debug mode has all the necessary rights.
- *Enable and configure µP controllers* – power up the necessary µP modules. Set up configuration registers of the phase-locked loop (PLL) controller to clock the peripheral controllers. Configure peripheral controller registers with various UUT-dependant settings (signals latencies, address/data bus width, etc.).

- *Verify UUT status* – read the status data of the UUT to check if the UUT is ready to communicate with the μP. This ensures the physical link between μP and the UUT.

These configuration steps are not mandatory, but advisory. The actual need in every step depends on the architecture of the particular μP.

3.3 Test application

Test path initialization and *configuration* belong to the *test access* functionality of the PCBT program. The rest of the PCBT program functionality is the test application, which may be developed in accordance to online or offline test application modes.

It should be stressed explicitly that any test data exchange between the external tester and the μP in the PCBT is going via the test access path. The steps of the test application part of the PCBT program are given in pseudo-assembly statements. These statements are used later for evaluation of the *online* and *offline* testing modes. The key-words of pseudo-assembly instructions are outlined in bold. Every instruction and operand (outlined in italic) is shifted through the test access path. The recipient of the data is denoted by Shift in and Shift out statements. The instructions (**Load, Store, Jump,** etc.) are only shifted in, thus, the recipient notion (Shift in) is redundant in this case.

The overall test application time (t_{TA}) could be calculated by counting the number of shifts. The actual time is a multiplication of number of shifts to the length (in bits) of the test path and divided by test clock frequency. The test clock frequency and the test path length are constant values. Hence, the test application time is in linear dependency with the number of shifts (the delays between shifts that are caused by the test hardware are neglected).

3.3.1 Online mode

The *online* test application of the single test pattern:

1. Shift in *test address* to data exchange register of the debug interface.
2. **Load** *test address* to general purpose register (GPR) from debug data exchange register.
3. Shift in *test pattern* to data exchange register of the debug interface.
4. **Load** *test pattern* to general purpose register (GPR) from debug data exchange register.
5. **Store** *test pattern* from GPR to *test address* (UUT is mapped to the common address space of the μP).

The *test address* is the address of the location inside the UUT that is mapped to the address space of the µP.

The *test response* obtaining sequence is the following:

1. Shift in *test address* to data exchange register of the debug interface.
2. **Load** *test address* to general purpose register (GPR) from debug data exchange register.
3. **Load** *test response* to GPR from *test address*.
4. **Store** *test response* from GPR to data exchange register.
5. Shift out *test response* to the external tester.

The *test response* is evaluated in external tester. If the *test response* does not match with the *test pattern* the further diagnosis is performed to locate the fault.

The formula for test application time calculation is:

(I) $t_{TA} = wm + rm$

Where m is number of test patterns, w is number of shifts to write test patterns and r is a number of shifts for test response obtainment. Hence, for current board under test formula (I) can be reduced to: $t_{TA} = 5m + 5m = 10m$

If more than one test pattern has to be applied, these steps should be repeated for every test pattern. In case if the write and read operations are subsequent and the data is read from the same *test address*, the first two steps may be skipped in the *test response* obtaining sequence. This optimization requires the additional study of the test application algorithm.

Another option for optimization is possible if the instruction set architecture (ISA) supports store and load instruction with multiple data sources (SIMD). For the use of **store multiple** instruction the steps 3 and 4 of the test application sequence have to be iterated *f*-times (for each test pattern), where *f* is a maximum number of source operands. The fifth step is then substituted by:

> **Store multiple** *test patterns* from GPRs to UUT address (each next test pattern is stored to the subsequent memory location in the UUT).

Then the wm summand in formula (I) changes to: $\left(2\frac{m}{f} + m + m + \frac{m}{f}\right)$

For the use of **load multiple** instruction the steps 4 and 5 of the test response obtaining sequence have to be iterated *f*-times (for each test pattern), where *f* is a maximum number of source operands. The third step is then substituted by:

> **Load multiple** *test patterns* from UUT subsequent addresses to GPRs (each next test response is loaded to the subsequent GPR).

The rm summand in formula (I) takes the following view: $\left(2\frac{m}{f} + \frac{m}{f} + m + m\right)$

The limitation for the instruction with multiple operands is that only consecutive memory locations could be written or read with one instruction. Thus, these instructions are not useful in accesses to arbitrary addresses. However, instructions with multiple operands become extremely useful in programming custom application to embedded memory of µP through the debug interface.

The test application time for the sequence with the usage of multiple operand instructions for the selected µP architecture:

$$\text{(II)} \quad t_{TA} = \left(2\frac{m}{f} + m + m + \frac{m}{f}\right) + \left(2\frac{m}{f} + \frac{m}{f} + m + m\right) = 2m\left(\frac{3}{f} + 2\right)$$

Where m is number of test patterns, f is number of source operands in the multiple load or multiple store instructions (defined in the µP Instruction Set Architecture (ISA)). If f is 1, the equation (II) becomes an equation (I). For example, if f equals to 8 (most modern µPs have at least 8 GPRs), then $t_{TA} = \frac{19}{4}m \approx 5m$. As t_{TA} is calculated in number of shifts the final results should be rounded up to integer number.

3.3.2 Monitor-based online mode

Along or instead of the instructions with multiple operands the monitor-based strategy is used to shorten the test application time. The special program is loaded to the program memory of the µP (preferably internal). The monitor reads the test pattern from the debug data exchange register and applies it to the UUT. In this strategy the test application time consists of monitor programming (t_{MP}) and test data transferring (t_{DT}). The $t_{MP} = wp$, where w is a number of shifts to write test pattern to the memory and p is the size of the monitor program in words. The value for w is calculated in the first example (the *online* test application of the single test pattern), which is $w = 5$. The sequence of steps used in test data transferring in monitor-based strategy is given below:

Write *test pattern*:

1. Shift in *command* (write) to data exchange register of the debug interface.
2. Shift in *test address* to data exchange register of the debug interface.
3. Shift in *test pattern* to data exchange register of the debug interface.

Thus, test pattern is written in 3 shifts: $w_t = 3$.

Read *test response*:

1. Shift in *command* (read) to data exchange register of the debug interface.
2. Shift in *test address* to data exchange register of the debug interface.
3. Shift out *test response* from data exchange register of the debug interface.

In other words test response is read in 3 shifts: $r_t = 3$.

According to this sequence the test application time in monitor-based strategy is:

$$\text{(III)}\quad t_{TA} = t_{MP} + t_{DT} = wp + w_t m + r_t m$$

Adopting formula (III) to the selected µP architecture:

$$t_{TA} = 5p + 3m + 3m = 5p + 6m$$

The *gain* estimation in test application time between the default test application strategy (equation (I)) and the monitor-based is:

$$gain = 10m - (5p + 6m) = 4m - 5p$$

Hence, *gain* is positive when $p < \frac{4}{5}m$, which means that the size of the monitor program should be less than the 80% of the test data size for the selected µP architecture.

3.3.3 Offline mode

The *offline* test application requires a test application program with embedded test patterns to be loaded into embedded program memory of the µP. Below is shown the sequence of steps to load one program word.

1. <u>Shift in</u> *target address* to data exchange register of the debug interface.
2. **Load** *target address* to GPR from debug data exchange register.
3. <u>Shift in</u> *program word* to data exchange register of the debug interface.
4. **Load** *program word* to GPR from debug data exchange register.
5. **Store** *program word* from GPR to *target address* in program memory.

These steps are repeated for every program word of the test application program. Then the program is started and the external tester polls the debug data register for the flag that determines that the application of test patterns is finished. Then external tester reads test results that are stored in the GPRs for further evaluation. The steps to complete these actions are given below:

1. **Jump** to the initial address of the test application program (the µP starts program execution)
2. <u>Shift out</u> data from debug data exchange register (this step is repeated until the DONE flag is set by the running test application program).
3. <u>Shift in</u> debug interface instruction that halts the processor and returns control to debug interface (this step is reached after the DONE flag was set in the data exchange register by the test application program).
4. **Store** *test result* (pass or fail) from GPR to data exchange register.

5. Shift out *test result* to the external tester.

In case if test passes, the testing is completed with the last mentioned step. If the test fails, the external tester reads the *test responses* from the memory location that was specified by test application program in a GPR. The external tester compares the expected values with the obtained *test responses* to diagnose the fault.

The test application time for offline mode is:

(IV) $t_{TA} = t_P + t_S + t_{FR}$

Where:

- t_P is time for loading the test application program into program memory $t_P = wp$; w is the number of shifts to program one word and p is the size of the program in words.
- t_S is time for starting the test application program and *test result* obtainment. For further evaluation of the test application time this parameter is assigned with its minimum value (5), which corresponds to the best case scenario, when the polling returns DONE flag after the first attempt. However, the program execution may take time longer than the time of one shift. To simplify our calculations we state that the number of repetitions of step 2 corresponds to the program execution time.
- t_{FR} is time for reading faulty *test response*. These steps are skipped when *test result* contains pass signature. This time is neglected in further calculations, because the typical test application scenario ends with positive *test result*.

The simplified formula (IV) is: $t_{TA} = t_P + t_S + t_{FR} = wp + 5$.

When w is equal to 5 (the default write sequence) then test application time is: $t_{TA} = 5p + 5$. The test application program consists of m test patterns and the instructions themselves. Let a be the number of instructions in words in the test program. Hence, $p = m + a$ and $t_{TA} = wp + 5 = wm + wa + 5 = 5m + 5a + 5$, when w is equal to 5 (as for selected µP ISA).

The time that is used by µP to execute the application is considered to be relatively small due to the much higher (from 10 to 1000 times) clock frequency in comparison to test clock. Thus, the µP execution time is neglected in our calculations.

In the simplified equation (I) (default online mode) the test execution time of the same set of test patterns was: $t_{TA} = 10m$. This leads to the conclusion that the test application time for online and offline modes are equal when $wa = wm - t_S$. In other words, at least the half of the test application program should be test patterns, otherwise the test application time in the offline mode will exceed the time in the online mode (for the same SUT and µP architectures).

In case when μP ISA supports instructions with multiple operands the time for loading the test application is different. In this case t_P is calculated similarly to the test patterns application summand from formula (II), because exactly the same sequence of shifts is used:

$$t_P = 2\frac{p}{f} + p + p + \frac{p}{f} = 3\frac{p}{f} + 2p = p(\frac{3}{f} + 2).$$

Where *f* is a number of source operands in the multiple load or multiple store instructions (defined in the μP ISA). If *f*=8, then $t_P = \frac{19}{8}p = \frac{19}{8}m + \frac{19}{8}a$. By substituting the values of t_P, t_S and t_{FR} into (IV) the following formula for test application time is obtained:

$$\text{(V)} \quad t_{TA} = p\left(\frac{3}{f} + 2\right) + 5 = m\left(\frac{3}{f} + 2\right) + a\left(\frac{3}{f} + 2\right) + 5$$

The difference in test application time between the online (formula (II)) and offline (formula (V)) test application strategy:

$$(II) - (V) = 2m\left(\frac{3}{f} + 2\right) - m\left(\frac{3}{f} + 2\right) - a\left(\frac{3}{f} + 2\right) - 5 =$$

$$= m\left(\frac{3}{f} + 2\right) - a\left(\frac{3}{f} + 2\right) - 5$$

The difference must be positive *(II) – (V) > 0,* in order to justify the efforts spend for development of the test application program that is used in offline mode. Hence, the difference is positive when:

$$m\left(\frac{3}{f} + 2\right) - a\left(\frac{3}{f} + 2\right) - 5 > 0;$$

$$(m - a) > \left(\frac{5}{2 + \frac{3}{f}}\right).$$

If *f = 1* (SIMD is not supported), the previously derived inequality is received:

$$(m - a) > 1.$$

This leads to the conclusion that the size of the program without test patterns should be smaller (independent from the μP ISA architecture) than the set of test patterns, otherwise the test application time in offline mode will exceed the time for the online mode. Another interesting conclusion is that the more operands could be used per one load or store instruction the bigger should be difference between number of test patterns and the size of the test application program not including embedded test patterns. However, even when $f = \infty$, *a* should be smaller than *m* – 3 to satisfy inequality *(II) – (V) > 0*, because:

$$\lim_{f\to\infty}\left(\frac{5}{2+\frac{3}{f}}\right)=2.5$$

As *a* and *m* domain contains only integer numbers all results should be rounded up.

Although, above mentioned calculations are valid only for the selected architecture of the µP, these can be used for the arbitrary ISA when inequality $(m-a)>\left(\frac{5}{2+\frac{3}{f}}\right)$ is transformed to the general form as follows:

$$(m-a)>\left(\frac{t_S}{(w-d)+\frac{d}{f}}\right)$$

Where:

- t_s has the same meaning as for formula (IV).
- w is the number of shifts to write/read one word from/to the external tester.
- d is the number of shifts that is repeated for *1/f* words in case of SIMD instruction.

3.4 Overview of the test application modes

The formulae presented in Table 3-1 are proposed for the test application time estimations and comparison of the listed modes. In case if the SUT and µP architecture used in this chapter do not match the specific test case, the formulae (I), (III) and (IV) should be used to derive the equations suitable for given SUT and µP architecture.

The preferable solution for detecting not only static, but also dynamic faults is offline test application mode. The most of the time in offline mode is spend for loading the test application program (with embedded test patterns) into program memory of the µP. This time may be reduced only by optimization of the size of the test application program. The first direction for optimization is to embed the compression/decompression mechanism for test patterns. The second is to implement the program as short as possible for every SUT. It means that the recompilation of the general test application software is not a solution. The test application program should be developed in the native assembly of the µP to make the binary as short as possible. The latter allows using the complex instructions such as store or load with multiple data sources, which reduce the size of the test application program.

There are test cases when usage of the program memory of the µP is not allowed or not efficient. The program memory may be full, protected, not available or not big enough to store the flash image (in case of ISP). For these cases the online mode can be used instead of the offline mode.

The special case of the online mode is the online mode with monitor software. The monitor software is typically small enough to fit in any memory. The main goal of the monitor software is to reduce the data traffic on the test path. The most of the traffic in the online mode is the instructions to control µP to apply test patterns. These instructions accompany every test pattern, thus the payload in the online mode is relatively low as shown in Table 3-1. The monitor software increases the payload, because it assumes the control over the µP.

Table 3-1 *Evaluation of test application modes for the selected µP ISA*

Test application modes	**Number of shifts* through the test access path (t_{TA})**
Online	$10m$
Online (µP ISA supports instructions with multiple operands)	$2m\left(\frac{3}{f}+2\right)$
Online (monitor software developing and loading)	$5p+6m$
Offline	$5m+\ 5a+5$
Offline (µP ISA supports instructions with multiple operands)	$m\left(\frac{3}{f}+2\right)+\ a\left(\frac{3}{f}+2\right)a+5$

m – Number of test patterns
p – Number of words in the monitor software
a – Number of words in the test application software (not including the embedded test patterns *m*)
* – Number of shifts is integer number, thus, all results should be rounded up.

3.5 Chapter summary

This chapter describes in details the internal structure of the PCBT program. The influence of the test requirements on the functionality of the test program is discussed in the beginning of the chapter. The SUT initialization and configuration steps are described in details in the test access section. In general, these steps prepare the test path for the test application.

In the test application section the online and offline modes are reviewed. Besides the detailed explanation of the possible implementations, the analytical estimations for the time limit of the test application are presented. The analytical estimations are supported by the derived formulae for test application time calculation. The chapter is concluded by the summary of the test application modes and the comparative table with formulae for test application time calculation. The simulation-free calculation of the test application time is useful for fast cost estimation of the manufacturing board test solution.

Chapter 4
BOARD AND ELECTRONIC COMPONENT MODELING

This chapter presents the study of the board and component modeling. The proposed modeling methodology includes modeling of structural and behavioral features of the board and IC components. The selection and development of the underlying metamodels are discussed in details and compared with the existing modeling approaches. The chapter is concluded with the description of the uniform test data path.

4.1 Modeling basics

A model is an abstract representation of an object. The model mimics structure or/and behavior of the real world object and is constructed to reflect certain parts that are essential for the job in hand. The modeling process aims to grasp only relevant properties of the object. Hence, modeling provides complexity reduction in manipulation with the real world objects.

The structure of the modeling instances is defined by the metamodel. The metamodel describes a model. In general, metamodel represents the set of the basic elements of the model, an inner structure of the elements as well as the rules for creating connections between these elements. In other words, a metamodel is the model of a model.

In this thesis the subject for modeling is a printed circuit board populated with electronic components also known as printed circuit board assembly (PCBA). The purpose for the PCBA modeling in this research is to create the representation of the test data path in the PCBA. This test data path is used later for test access and test application program synthesis.

Existing approaches on digital system modeling are based on different standards and languages. Transaction-level model (TLM) [46], IP-XACT (IEEE Std. 1685^{TM}-2009) [47], MARTE [48] are the most noticeable and the most recent ones. These are focused on the digital system design related tasks and suitable to solve only several of the needed subtasks of the test data path modeling (SoC internal structure and implementation). To author's best knowledge, the modeling of the SoC structures together with the structures beyond the SoC on the PCBA is not yet studied by the research community and industry. In order not to reinvent the wheel in modeling the following materials were studied.

The MARTE (Modeling and Analysis of Real-Time Embedded Systems) specification is a language extension to Unified Modeling Language (UML), hence it does not provide any methodology related hints for developing embedded system [49]. The MARTE profile to UML consists of packages that target different modeling aspects (e.g. design, analysis). The necessary instruments for PCBA and component modeling are presented in MARTE, but their usage requires deep knowledge of the model-driven engineering, that is typically uncommon for the test engineer. Thus, this modeling approach was considered too general and heavyweight to fit the cost of the task of test path modeling.

However, the general approach of creating an UML metamodel to describe the structure of the model is one-time effort and a common practice. Therefore it was followed in our methodology. Eclipse modeling framework (EMF) [50] was used to develop the metamodel for structural and behavioral model types. The EMF also facilitates the automatic synthesis of the edit and editor parts of Eclipse plug-in (Chapter 6). This plug-in is used to create the test data path model instance of a particular PCBA following the rules defined in the metamodel.

In transaction-level modeling (TLM), the details of communication among computational components are separated from the structure of computational components [46]. In [46] the number of TLM abstraction models is specified for description of different levels of description of communication time, computation time, communication scheme and processing elements (PE) interface (in this thesis PE is µP SoC). In TLM the design in hand can be described across multiple abstraction levels, which allows hiding of unnecessary details of one module, while providing thorough "implementation level" description of the other. These modeling principles of TLM perfectly match the objectives of the PCBA modeling in this research due to several reasons. The first reason is that the communication time between the SoC components and external on-board devices has to be modeled cycle-wise, but the communication time between the SoC components may be neglected. The second point is the interface modeling between the SoC components. On the one hand, the model has to contain information about exact mapping between

SoC pins and SoC components. On the other hand, the inter-component interfaces inside the SoC do not require pin-accurate modeling. Required SoC model needs an “implementation level” description of the structures communicating with SoC boundary and “component-assembly” level for the rest of the SoC. Although, the TLM methodology fits the task of PCBA modeling, it still misses the point of making the models easy to develop for the test engineer. The TLM implies creation of models by the means of programming language like SystemC [51]. The proposed modeling methodology reuses the basis of the listed useful characteristics of the TLM abstraction levels, but introduces the metamodel-based approach for manual model creation to shorten the expenses on traditional programming.

IP-XACT [47] defines the standardized way to describe those behavioral and structural characteristics of the IP that are relevant to the integration of SoC components. The components, systems, bus interfaces and connections, abstractions of those buses, and details of the components including address maps, register and field descriptions may be described by models supported in IP-XACT. Among the supported descriptions are TLM (SystemC and SystemVerilog), fixed HDL descriptions (Verilog, VHDL) et al. IP-XACT is focused on the integration inside the chip and the board level is not involved, hence, the proposed modeling method cannot fully rely on this standard.

On the current stage of the research the proposed modeling method does not produce IP-XACT compliant models. However, the backward compatibility is supported for IP-XACT compliant VHDL description, which can be automatically parsed into the proposed model. As practice shows VHDL description of the SoC components is typically “closed” information for the third-party tool vendors. Hence, there is a need in recreating the description (model) of SoC component manually.

4.2 Test data path model

The test data path modeling implies the modeling of the structure of PCBA and the structure of electronic components that populate this PCBA. In order to synthesize the test access and test application program the model of the functionality of the PCBA components is also required. Thus, two different kinds of information (structural and behavioral) have to be modeled. The structural part of the board component model contains mostly the component specific settings (e.g. names of pins, addresses of registers and internal modules). On the contrary, the behavioral part models functionality which is typically general to particular component family. Hence, the decision was made to create structural and behavioral models separately for the complexity reduction and wider opportunities for model reuse, but with common interfaces that allow joining these models into uniform test data path model.

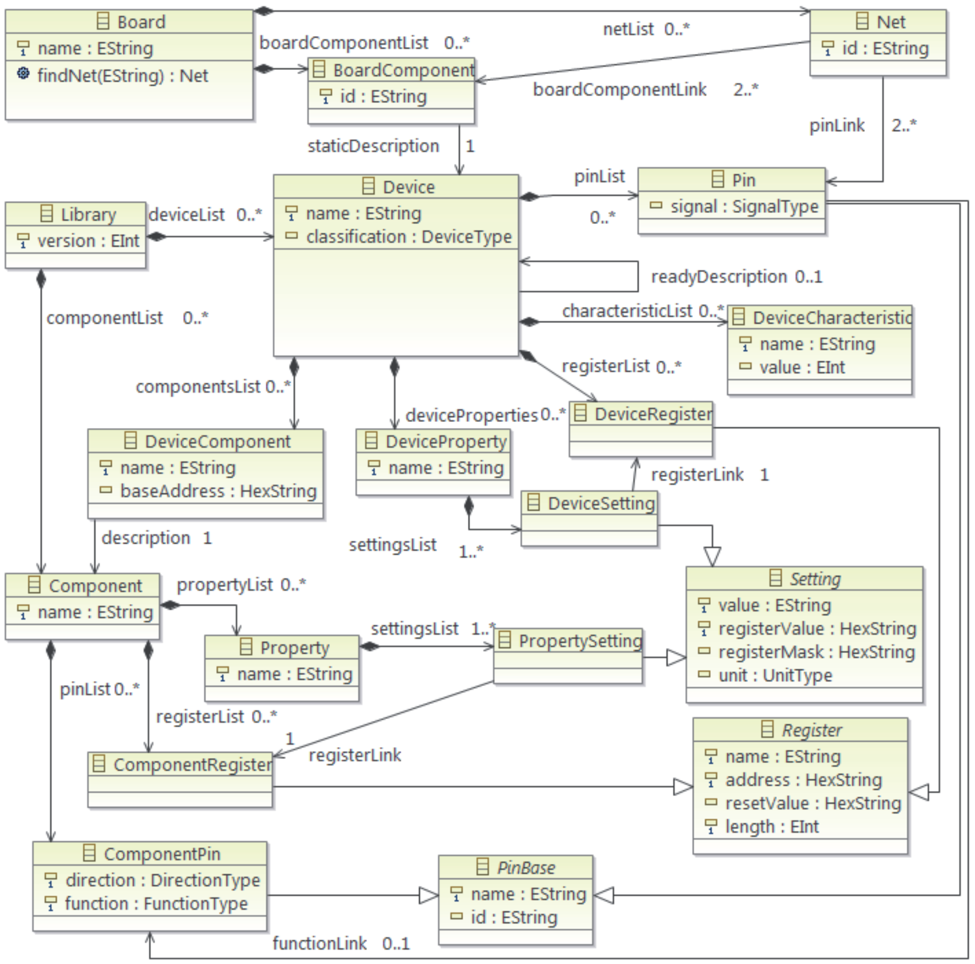


Figure 4-1 Metamodel for structural model of the board and electronic components

4.2.1 Structural Model

The proposed structural model represents a two-level hierarchy. The top level describes the connections between components at the board level and corresponds to the *board structural model*. Bottom level is dedicated to model the internal structure and static properties of the electrical component. This level is further called as *device structural model*. The board and device structural models are united at the level of the metamodel that describes structure of the uniform structural model of the PCBA (see Figure 4-1).

4.2.1.1 Board structural model

The purpose of the board structural model is to represent the interconnections between electronic components. The basic object to be modeled here is the physical link between PCBA components. The important property of every physical link is a list of pins that are connected by the given link. The minimum number of pins in the link is two. Every pin belongs to the electrical component. Physical link between at least two electrical components is modeled as a net. Hence, structural model of the board describes the PCBA by listing the connection between electrical components as nets without including the information about the location of the component on the PCBA.

The board structural model is automatically obtained from PCBA netlist file. There are many formats for describing the PCBA netlist, though, in order to reduce the amount of parsers to implement, we reused the commercial software that is capable to parse most of the formats. This commercial parser translates any supported input PCBA netlist format into simple intermediate format. Hence, to reduce the development efforts, the program that automatically builds the board structural model out of PCBA netlist supports this format. For any other format the commercial parser can be used.

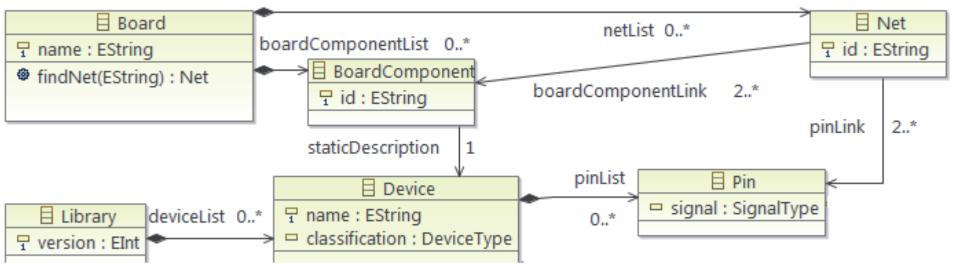


Figure 4-2 Metamodel of board structural model

The part of the metamodel of the board that reflects the board structural model is shown in Figure 4-2. Generally, this metamodel encapsulates the following rules:

- Every object is represented as the titled box (Class object in UML), where title is a general name for all instances of this object. The PCBA is modeled as object named *Board.* The properties of an object are mirrored as fields in the Class. For example, *name* in the class *Board* is a name of the PCBA.
- The containment link shows that one object can enclose the other object. The containing object is denoted with the bold diamond and the contained object is pointed with the arrow. The notation after the name of the containment link shows minimum and maximum number of objects to contain (* stands for unlimited). *Board* has a containment link to *Nets (netList)* and to *BoardComponents (boardComponentList)*.

- The simple reference is depicted as a simple arrow. *Net* has a link to at least two *BoardComponents*. It means that *Net* must have reference to at least two pins. In practice this means that knowing the *Net* one can find the pins of the *Device(*s) that is connected by this *Net*.
- *BoardComponent* should be linked by a *staticDescription* link to the *Device* model that is stored in the *Library* of devices and device components. This abstraction allows storing only one model for the identical PCBA components (e.g. multiple identical memory chips). In other words, instances of the same device have one description in the library.
- *Device* has a containment list of *Pins* that correspond to the physical pins of the *BoardComponent*.
- In order to distinguish identical devices in the model, *Net* has besides the reference to the *Pins* also the *boardComponentLink.* This link creates a reference to particular *BoardComponents* whose pin(s) are connected in the *Net*.

After the *board structural model* is obtained the *device structural model* should be assigned to those *Devices* (*BoardComponents*) that are participating in the test data propagation path. The rest of the PCBA components are unimportant and may be omitted from the uniform PCBA model. This model could be used by the automated test pattern generator (ATPG) to get the set of test patterns to test faults on the interconnections between the board components.

4.2.1.2 Device structural model

The complete board structural model is a template for the further development of the device structural model. This template contains the list of board components with links to the predefined devices that are added to the library. Every board component may be associated with the static device description that contains device internal structural model. This model is created obeying the rules exposed by the metamodel that is shown in Figure 4-3.

Practically, there are two ways to assign a structural model to the device. The first way is to reuse the existing suitable model from the library (*readyDescription* link). This requires the presence of the correct model in the library. The second way is to develop the structural model. Every developed model is stored in the library for further reuse.

The following Lego-style modeling concept was proposed in [52]. In order to reduce the complexity of the structural model the certain parts of the device are modeled separately as device components (*Component*). For example, memory controller, external bus interface or debug interface are modeled independently from µP itself. The independent *Component* model becomes a part of a particular *Device* after it is added via *description* link to the appropriate *DeviceComponent* (Figure 4-3). The splitting of the device model into models of components also contributes to the reusability of the models in the library. The major IP vendors (e.g. ARM) develop the processor-based SoC components that are compliant with various versions of other SoC components. Hence, the reusability is maintained at an IP vendor level. Though, the same strategy to preserve the reusability of the models is followed in the proposed modeling approach.

The central item of the device structural model is a *Device*. It has a number of relations as it is shown in Figure 4-3.

- The device might have characteristics that are possible to express in this model as an object (*DeviceCharacteristic).* The presence of chracteristics is modeled as a containment link (*characteristicList)* to the *DeviceCharacteristic.* The most typical device characteristics describe the timings of the control signals for the DDRx memory model (e.g. CAS latency (CL), clock cycle time (tCK), row cycle time (tRC), refresh row cycle time (tRFC), row active time (tRAS)).
- The properties of the device are expressed as *DeviceProperty*. The property might have several settings (*DeviceSetting*). Every setting is linked to the device register (*DeviceRegister*), which description should be included into *registerList*. For example, the watchdog of the µP can be modeled as a property. The possible watchdog settings are time periods or watchdog state (e.g. disable, enable).
- The device could have a list of components (*DeviceComponent*). This architecture is typical for the SoC with processor core(s) and number of peripheral IP cores. The *DeviceComponent* describes the name and the base address of this component inside the SoC.
- The SoC component that is defined in the model as *DeviceComponent* is intended to have a standalone description (*Component*) in the library.
- The *Component* has containment list (*pinList*) of pins (*ComponentPin*) that are used by this component. The device pin (*Pin*) and the component pin (*ComponentPin*) is physically the same pin of the particular SoC and their relation is modeled with *functionLink.*The *functionLink* exposes the connection between the SoC pins and the SoC component that drives and senses these pins.
- The registers that belong to the SoC component are modeled as *ComponentRegister.* The physical address inside the µP SoC of the *ComponentRegister* inside the SoC is later composed in software by adding the register address inside the component to the base address (*baseAddress*) of the component.
- The *Property* that resides in the *propertyList* of the *Component* is for modeling the various possible configurations of the component. For example, the SDRAM controller has a list of parameters (e.g. CL, tCK, tRC, tRFC, tRAS) that help to setup the proper signal timings for

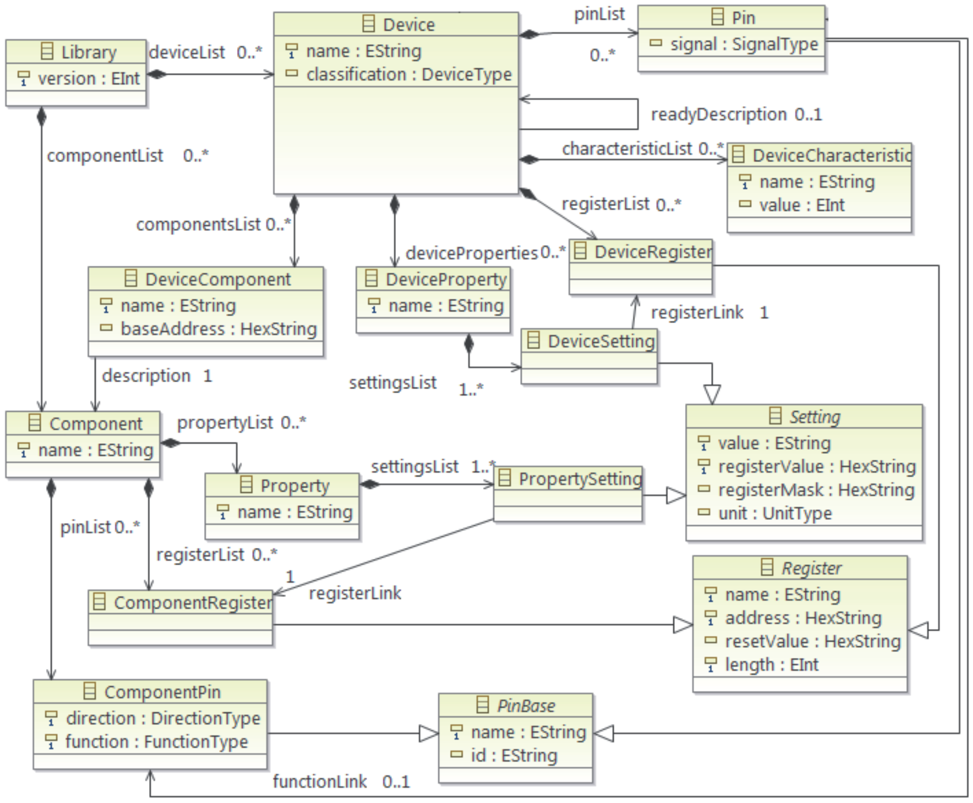


Figure 4-3 Metamodel of device structural model

communication with particular memory. Every *Property* has a list of possible values that are modeled as *PropertySettings*. Any *PropertySettings* has a *registerLink* which specifies the mapping between the property setting and the corresponding value for one of the registers inside the SoC component.

- The *PropertySetting* and *DeviceSetting* are derived from the general *Setting* class. It has fields for defining setting *value* and *type* of this value. The field *registerValue* represents the actual value to be stored in the register for the given setting *value*. The *registerMask* specifies the location of the *registerValue* in the register. The register mask is needed when the register is dedicated to contain the settings of more than one property.

The particular settings of the processor-based SoC are obtained on the basis of this structural model. These settings enable communication between processor-based SoC and the UUT. The value of the setting is found by matching the UUT characteristic name (the *name* field in the *DeviceCharacteristic*) with the property name of the µP (the *name* field in the *DeviceProperty*) or with the component property name (the *name* field in the *Property*). When the matching pair is found the correct *Setting* (*DeviceSetting* or *PropertySetting*) is selected from the settings list (*settingList*) by comparing the *value* field of the *DeviceCharacteristic* and the *value* field of the *Setting*. The obtained settings form the pairs of register and the value that have to be written to this register.

4.2.1.3 Metamodel for structural model

The metamodel for structural model of the PCBA (Figure 4-1) unites the metamodels for board structural model and device structural model. The *Device* (box for *Device* class) is a point of joint of board and device structural models. It acts like a bridge between the PCB-level interconnect structures and the PCBA component internal structures. One of the properties of structural PCBA model is that it includes mapping between SoC components and the board interconnect that is driven by this SoC component. This can be used in the debugging of the created model and for diagnosis of functional failures during the test runtime.

Component model reuse is a very important aspect of the concept as the only part that is not fully automated is the model creation. Hence, reduction in the amount of manually created model components is one of the goals of the proposed methodology. Once the models of the PCBA components are created, they are stored in the library. The next time the known µP SoC, SoC components or any other PCBA device (e.g. flash memory, DDRx) is present on the board we can reuse the respective models from the library. In ideal case, every device of the PCBA under test should have its model in the library except the interconnection information that needs to be processed separately for each new PCBA. However, the latter is a fully automated task.

4.2.2 Behavioral Model

The behavioral part of the uniform test data path model that is proposed in this thesis is composed using the mathematical basis of High-Level Decision Diagram (HLDD). There are research works that study the presentation of the digital circuits at Register-Transfer level (RT-Level) as High-Level Decision Diagram (HLDD) [53]. The HLDDs are graph representations of discrete functions that can be considered as a generalization of Binary Decision Diagrams (BDDs). HLDDs have been proven an efficient model for simulation and fault modeling as they provide for fast evaluation by graph traversal and for easy identification of cause-effect relationships [54].

4.2.2.1 High-Level Decision Diagram theory

Consider a system S as a network of interconnected components (functional blocks, buses, ports) where each component is represented by a function $y = f(X)$ and X is the set of variables (Boolean, Boolean vectors or integers), and $V(x)$ is the set of possible values for $x \epsilon X$ which are finite. Let HLDD G_y with a set of nodes M

represent the component. The terminal nodes $m^T \in M^T$ may be labeled either by variables $x(m^T) \in X$, digital functions $x(m^T) = f_{m^T}(X)$, or constants $a(m^T)$. All remaining nodes $m \in M \backslash M^T$ are labelled by variables $x(m) \in X$, and have $|V(x(m))|$ output edges leading to the successor nodes m^e where $e \in V(x(m))$. The edge (m, m^e) in the HLDD is called activated if $x(m) = e$. A path (m, n) is called activated if all the edges which form the path are activated.

To *activate a path* (m, n) means to assign the node variables along this path with proper values. Let m_0 be the root node of a HLDD G_y. Let X^t be an input vector applied at the moment t on the inputs of the component represented by G_y. We call the vector X^tas the activation solution for the component to satisfy the condition $y = f(X) = x(m^T\ )$ if it activates a *full path* (m_0, m^T) from the root node to a terminal node. The complete test solution implies consistent activation of all the full paths in the HLDDs involved, so that the imposed constraints collected along the activated paths are satisfied. To find such a test solution, a constraint solver can be used.

By activating a full path a symbolic value associated with the terminal node m^T is assigned to the root node m_0. In general, the terminal node may contain constant, operation (arithmetic: $a + b$ or Boolean: $a|b$) or variable. If terminal node contains a variable (or operation of variables), the value of the variable is determined by subsequent activation of a full path in the corresponding graph.

4.2.2.2 Metamodel for High-Level Decision Diagrams

In Figure 4-4 is shown the metamodel that describes the structure of behavioral model part of the test data path.

- *ModelingDomain* is the most top element in this metamodel that is used to collect *ModelingObjects*. The domain (*ModelingDomain*) is typically a PCBA whereas the objects (*ModelingObject*) are PCBA components.
- Any *ModelingObject S* has a number of inputs that are implemented as variables ($Variable\ x$) $x \epsilon X$.
- $Variable\ x$ is defined with the name and the width in bits. The modeling object is represented by the set of *GraphVariables* (Y). The possible values of the *GraphVariable* are modeled as terminal nodes m^T (*Termination*) of the graph G_y that are assigned to this *GraphVariable* $y = x(m^T\ )$.
- *Termination* has link to *Variable* that defines its value. As it is seen from the metamodel the *Variable* is a base class for *Input, GraphVariable, Function* and *Constant* objects. Hence, the value of the *Termination* is one of the objects that are derived from the *Variable* class.
- *Graph* G_y object has containment link to nodes M that belong to this graph. It also may have a direct containment link to terminal nodes M^T.

This link (*TerminalEdge*) is for explicit definition of constants and functions that are referred from the nodes.

- *Node* has a link (*NodeEvaluation* $V(x(m))$) to the variable that contains the possible values of the node. Every edge (*Edge*) (m, m^e), where $e \in V(x(m))$ that goes from the node to the next node corresponds to the one of the possible values of the first node. Same nodes may be connected by more than one edge.
- *Edge* may lead to the next non-terminal node $m^e \in M \backslash M^T$ (*NodeLink*) or to the terminal node $m^e \in M^T$ (*TerminationLink*). The transition value of the edge may also be specified by the *ConstantValue* link to the predefined constant.
- *Function* $f_m(X)$ is an object that defines the operations with variables. The function has a field for selecting an operation from a list of supported functions (*AvailableFunctions*). This list can be easily extended to support any operations (bitwise, logic, etc.). The arguments to the function are specified by the *Arguments* link that select from the list of predefined variables.

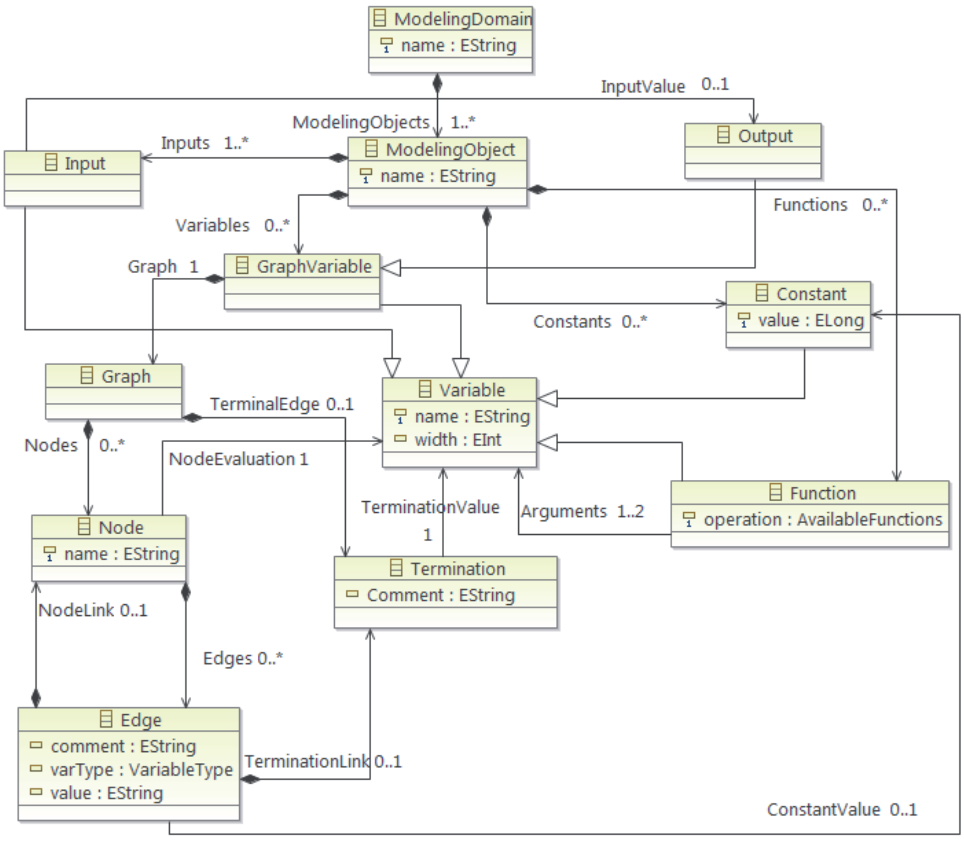


Figure 4-4 Metamodel for HLDD

- *Output* of the modeling object is a graph variable that is explicitly specified as output $y \in Y^O$. The output may be connected to the input of the same or different modelling object inside the same modelling domain by the *InputValue* link $x(y)$.

This metamodel belongs to the contribution of this thesis. HLDD graphs have not been previously described at this level of abstraction. This metamodel is a first method that facilitates the manual HLDD graphs composition. Previously manual HLDD graphs creation was considered as a very inefficient approach to describe digital circuits at RT-Level. The new framework was developed that provides functionality to create, import, edit and export the HLDD graphs. This framework does not require experience in any programming language and allows to create behavioral and structural descriptions of designs at fairly high level of abstraction.

4.2.2.3 High-Level Decision Diagram composition

Although HLDD graphs could be automatically constructed out of HDL description of digital circuit at RT-Level [53], this often is not possible since HDL description is not publicly available. In case, when RT-Level HDL description is not available for the PCBA components the HLDD are supposed to be composed manually. The manual composition of HLDD models relies on the PCBA component documentation.

Let us consider the structure depicted in Figure 4-5 as a part of the test data path to be modeled for test propagation purposes. Figure 4-5 presents a reduced structure of JTAG TAP that consists of TAP controller state machine (Figure 4-2) and scan register that is connected to respective data register. Data is shifted into scan register through serial TDI bus when TAP controller state is "Shift-DR". TAP controller "Controls" output is equal to 4 (Controls = 4) when state is "Shift-DR". The load from scan register into data register is initiated when TAP controller state is "Update-DR" (Controls = 8). The store from data register to scan register is performed when TAP controller state is "Capture-DR" (Controls = 3). TAP controller enters reset state "Test-Logic-Reset" when TRST signal is enabled. In the same state the data register obtains its reset value.

The model of the described structure (Figure 4-5) is shown in Figure 4-6. For ease of understanding, the repetitions of similar parts of the resulting model are omitted. In the model given in Figure 4-6, the data register from structure in Figure 4-5 is represented by "Data Register" variable and scan register corresponds to "Scan Register" variable. The TAP controller state machine is equivalent to the leftmost graph in Figure 4-6.

Let us have a look at leftmost graph in Figure 4-6 for the explanation of the full path activation principles. The shortest full path can be activated by setting "TRST" = 1. In this case "Controls" variable is assigned with the value 0. As "TRST" is a system input one can directly apply any value to it. Hence, the only condition (constraint to be satisfied) we get from this path is that as soon as 1 is applied to “TRST” input, “Controls” will have value 0.

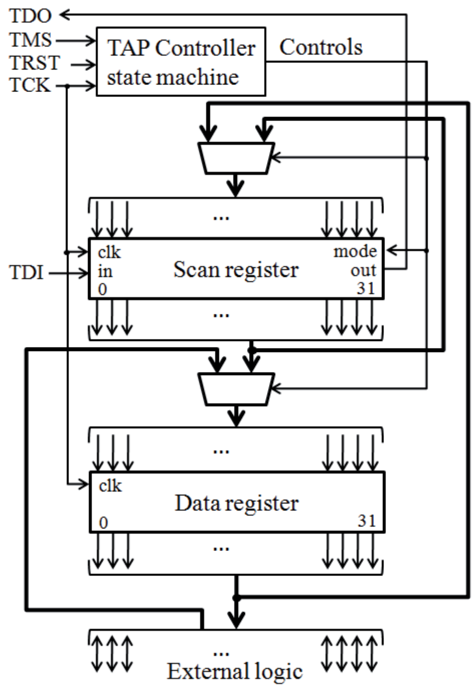


Figure 4-5 Simplified control and data path for JTAG TAP

In other words, the graphs of the model depicted in Figure 4-6 describe the set of constraints in the modeled system. For example, assignment of value 3 to “Controls” exposes the following constraints: “TRST” = 0, “TCK Front” = 1, “Controls′” = 2 and “TMS” = 0 (where “Controls′” is previous value of “Controls”). When these constraints are satisfied the full path to terminal node with value 3 will be activated.

In Figure 4-7 is presented the part of HLDD model of JTAG TAP controller state diagram. This model is another representation of the “Controls” graph in Figure 4-6. The difference is that in Figure 4-6 the HLDD is shown schematically and Figure 4-7 is a screenshot of the HLDD model that was created in the developed framework on the basis of the proposed metamodel (Figure 4-4). The HLDD model has two representations in this framework. The first is shown in Figure 4-7, which is a user-friendly view that facilitates manual interactions with the model. The second is a textual representation that is suitable for the toolchain that operates with this model.

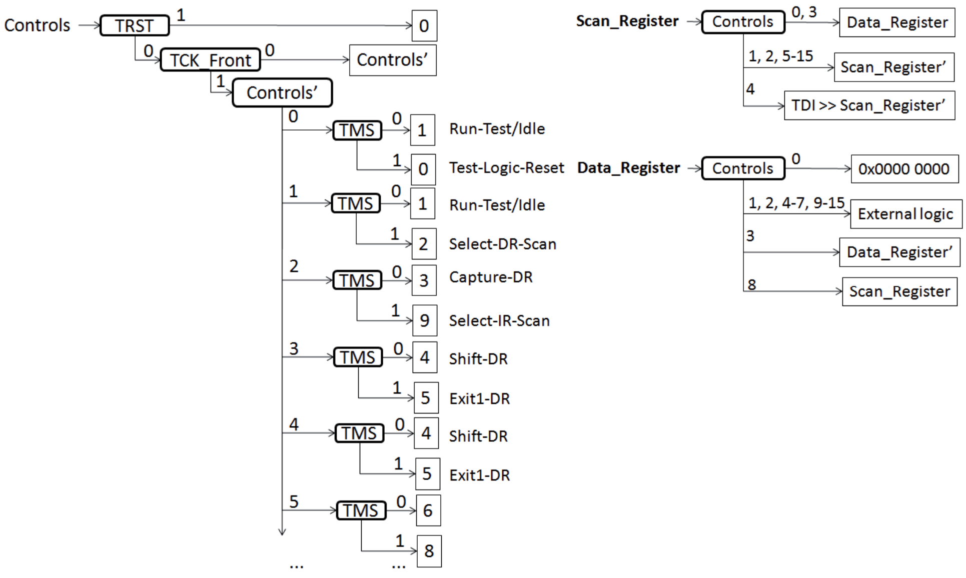
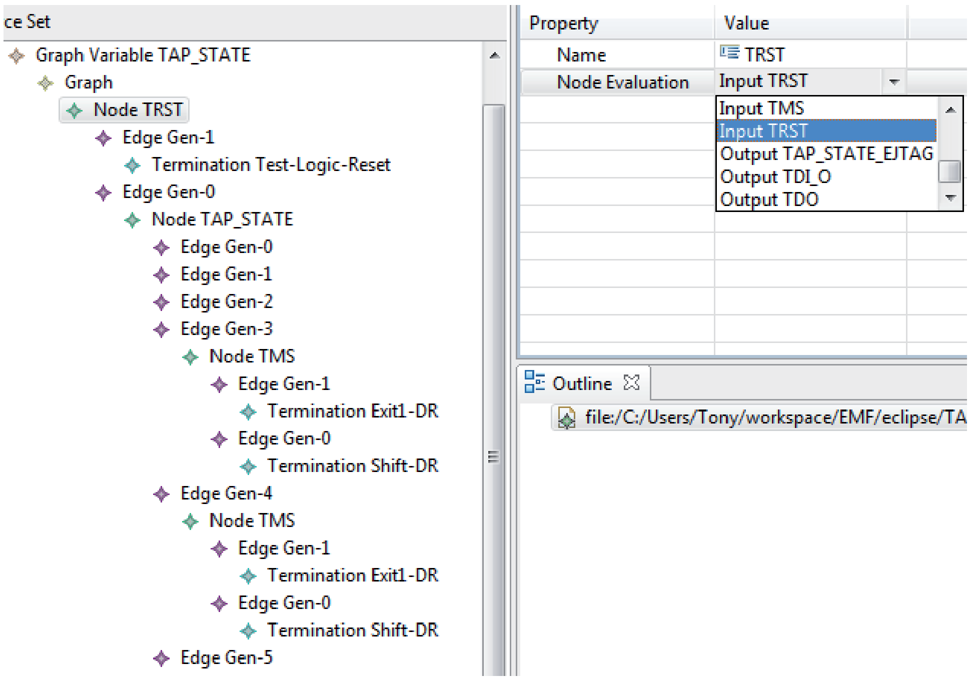


Figure 4-6 Model (HLDDs) of simplified control and data path for JTAG TAP



ce Set

- Graph Variable TAP_STATE
  - Graph
    - Node TRST
      - Edge Gen-1
        - Termination Test-Logic-Reset
      - Edge Gen-0
        - Node TAP_STATE
          - Edge Gen-0
          - Edge Gen-1
          - Edge Gen-2
          - Edge Gen-3
            - Node TMS
              - Edge Gen-1
                - Termination Exit1-DR
              - Edge Gen-0
                - Termination Shift-DR
          - Edge Gen-4
            - Node TMS
              - Edge Gen-1
                - Termination Exit1-DR
              - Edge Gen-0
                - Termination Shift-DR
          - Edge Gen-5

Property	Value
Name	TRST
Node Evaluation	Input TRST

Input TMS
Input TRST
Output TAP_STATE_EJTAG
Output TDI_O
Output TDO

Outline

file:/C:/Users/Tony/workspace/EMF/eclipse/TA

Figure 4-7 HLDD model of JTAG TAP controller state machine

4.2.3 Uniform test data path model

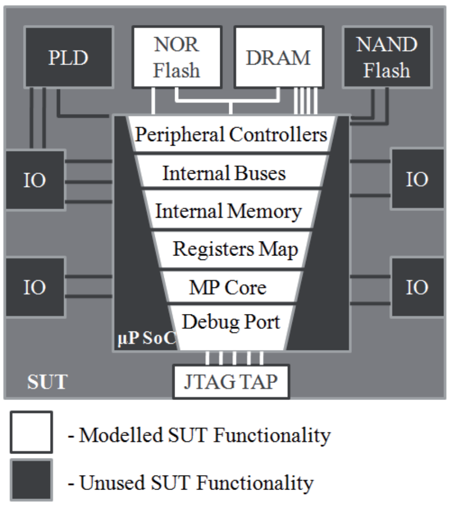


Figure 4-8 Test data path model

The key idea behind the proposed concept is to represent the system as a set of tightly interrelated models. These models are combined together into a uniform model, which represents the continuous test data path (Figure 4-8). The uniform model contains only models of those devices, functional blocks, buses, ports, etc., that need to be tested (interconnect test, functional test, etc.) or activated for the test data propagation during the test application.

The typical components of the uniform model are described in details in the following chapter (Chapter 5). Each component has a structural description. The programmable components (e.g. µP, µC) and other complex devices (e.g. flash memory, DDRx) are presented with the behavioral model as well. The presence of the behavioral model for other components is optional if they are not included in the test data path.

Due to the different metamodels (see sections 4.2.1 and 4.2.2), structural and behavioral models are isolated from each other during their creation phase. This allows reusing the same behavioral and structural model independently for different SUTs. Model reuse is a very important aspect of the concept as the process of creation/import of model itself is the only thing that is not fully automated.

The unification of the structural and behavioral models is automated. The structural model represents the "backbone" where the behavioral models are attached to. The exact place on the "backbone" is found by matching the *name* fields of the certain classes in the structural and behavioral models. The detailed matching parameters are shown in Table 4-1.

Table 4-1 *Fields for models unification*

Structural model		**Behavioral model**	
Class	**field**	**Class**	**field**
Device	name	ModelingDomain	name
Component	name	ModelingObject	name
Pin	name	Input	name
Pin	name	Output	name

4.2.4 Diagnosis of PCBT failure

In the automated approach, test path model is composed in a consecutive manner as shown in Figure 4-8. The advantage of this well structured continuous test path model of the SUT is the possibility to diagnose the root cause of system-level functional test failure. The diagnosis is performed in a top-down manner. First, the blocks that are not modeled are considered as non-relevant to the observed functional failure. Then, the models are removed one by one from the end of the modeled test path. After the model block is removed the PCBT program is re-synthesized and executed. This procedure is repeated until the remaining part of test path reports no failure. That reveals the failing module, which corresponds to the last removed model. Further diagnosis may be applied towards the last removed model where the final resolution depends on the internal structure of this model.

4.3 Chapter summary

The automation of the board test development is based on modeling of PCBA components. Typical PCBA components are microprocessor, flash memory, RAM memory, sensor, controller, display, etc. Every PCBA component has an automatically generated top-level structural model which is a part of the board structural model.

A novel structural model was developed to formalize the description of pin configuration, register map and internal memory organization as well as possible configuration parameters of the component. Generally, any static and descriptive information, such as legitimate values of the configuration register or external bus timing parameters, may be included into the structural model. The metamodel for structural model and detailed explanation are given in section 4.2.1.

Based on the information extracted from the netlist file of the given board a structural model of the board is automatically created. The netlist file conveys connectivity information for the board components and names instances of board components that could be simple components like transistor, resistor, capacitor or more complex ones like integrated circuit.

The structural model of the test data path is complemented by the uniform behavioral model. The behavioral model of the test data path is a unification of behavioral models of the PCBA components. The behavioral model presents the functionality of the component at RT-Level. Typically it includes description of the control path and data path. The mathematical basis for the behavioral model is formed by High-Level Decision Diagrams (HLDDs). The detailed description of HLDD and its metamodel is given in section 4.2.2.1. For the first time the metamodel-based approach is used for efficient manual creation of HLDD graphs

The proposed uniform test data path model is a novel approach to model the PCBA. The novelty of the uniform model is in its ability to combine structural and behavioral descriptions of not only the SoC components, but also of the PCBA components and their interconnections.

Chapter 5
AUTOMATED TEST PROGRAM SYNTHESIS

This chapter describes the proposed approach for automated test program synthesis. Firstly, the field of PCBA test program synthesis is explored and a typical development flow is examined. On the basis of that typical non-automated development flow the automated flow is presented. The comparison of both approaches is given to estimate the development time under different conditions. Secondly, the method for automatic transformation of HLDD model to a constraint satisfaction problem (CSP) is explained. The challenges in solving a CSP for automated test program synthesis are revealed in the following section. The feasibility of the proposed approach is proven by experimental results. The chapter is concluded by the case study that demonstrates the proposed approach on the example of the test pattern transportation through the standard test access port.

5.1 Automated and non-automated test program development

Test program development flow encloses a sequence of steps as shown in the flow chart in Figure 5-1 A). The uppermost step in Figure 5-1 A) is for obtaining information concerning the SUT (here SUT is a PCBA or a system of connected PCBAs). This task is aimed to collect the infrastructural information. For example: the number of devices in scan-chain, the connections between the programmable unit and the unit under test, etc. Second step is for collecting information about UUT. The key moments here are the communication protocol and timing parameters

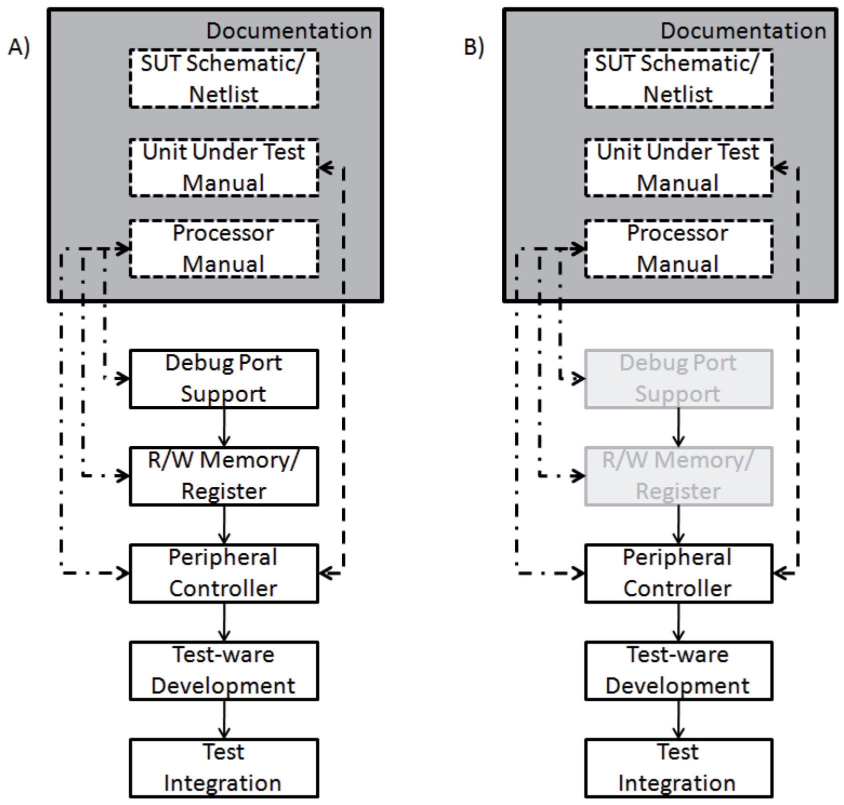


Figure 5-1 PCBT program development flow chart

A) flow without reuse B) flow with reuse

for in/out signals. Next step from the top encloses activities that involve studying the documentation of µP, which plays the role of embedded tester. Typically, important modules of the µP SoC are the debug port, the instruction set architecture, the organization of internal memory and various peripheral controllers. The order of the first three steps is not important as these steps describe the preliminary actions for the following programming steps.

The Fourth step ("Debug Port Support") is for developing the functionality for data passing starting from TAP of the SUT through the debug port. The second goal of this step is to compose sequence of JTAG commands that put processor into debug mode.

In the fifth step ("R/W Memory/Register") the access to the internal memories (registers) is implemented. Normally, it implies recruiting of instruction injection mechanisms of the processor debug port. As soon as the functionality to access the debug port and the internal memory locations of the processor is ready, the registers of the peripheral controllers are configurable from the external tester.

The Sixth step (“Peripheral Controller”) setups the peripheral controllers that provide an interface to the UUTs. Within this step the test access part of the PCBT program is completed.

The test application functionality can be implemented according to online or offline mode (see Section 2.3.2). In the “Test-ware development” step the general micro-code for test pattern application is adapted (compiled) to the instruction set of a particular µP.

Finally, the integration into test system is performed. This step implies the creation of the test project, generation and import of the test patterns and debugging on the SUT.

For general development time estimation, we assume that each step takes approximately the same time to fulfill. In case if the programming device (µP) on the SUT is already familiar to the test engineer (has been studied in previous test projects and part of the source code could be reused) this flow is optimized approximately by 20%. The optimization is possible due to the reuse of functionality for internal memory access and instruction injection. Typically, the micro-code could be also reused with minor changes. In Figure 5-1 B) the boxes for reusable steps are shadowed.

5.1.1 Automated test development flow

In Figure 5-2 the automated test program development flow is shown in comparison to the flow in Figure 5-1. Four additional steps were introduced to the flow chart that form the “Modeling” block.

“Modeling” block consists of steps for creating “Processor Model”, “Unit under test model”, “Peripheral controller model and processor Instruction Set Architecture model (“ISA model”). The “Processor Model” describes the debug port of the processor and an access to the internal memory and registers. The “ISA model” includes the map of the processor instructions and a standard initialization sequences for the µP in the native assembly language of the µP.

The use of these models in the test automation process is described in the following sections. In Figure 5-2 A) the automated PCBT program development flow is presented. Compared to the non-automated approach the flow initially has six manual steps instead of eight. The order of implementation steps in the non-automated approach is important, because every step is based on the previous one. On the other hand, in the automated approach every step in the “modeling” block is independent from others. For example, the peripheral controller model may be created before the processor model itself.

The first step in the automated flow is substituted by the automated import of the system description. Hence, “SUT Schematic/Netlist” step is shadowed to show that no manual effort is needed.

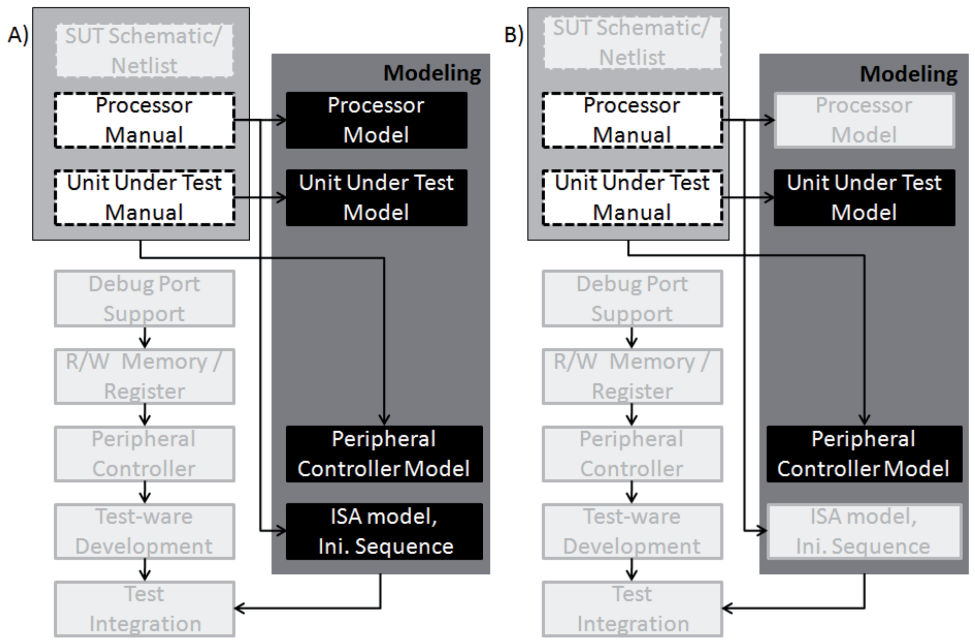


Figure 5-2 Automated PCBT program development flow chart

A) flow without reuse B) flow with reuse

In the automated flow one new step is introduced in comparison to the non-automated creation flow. This step is for UUT model composition. The UUT model is used by the automation framework to obtain the settings of the peripheral controller and to handle communication protocol between the processor and the UUT.

Similarly to Figure 5-1, Figure 5-2 also shows the PCBT program development flow with reuse (Figure 5-2 B). This flow is based on the reuse of the µP model and the ISA model. Given flow (Figure 5-2 B) contains only 4 steps instead of 6 (Figure 5-2 A)), which is approximately 33% less and compared to the flow with reuse in the non-automated approach (Figure 5-1 B)) it contains 2 steps less, which also stands for 33% time reduction.

5.1.2 Benefits of the automated approach

Different levels of experience with SUT components suppose usage of the various test program development flows (as described in Table 5-1). As practice shows, for every level of experience (shown as Conditions in Table I) the non-automated flow for test program development has more steps than in the automated flow.

According to Table 5-1, the automated test creation flow has the smallest estimated gain compared to the non-automated approach (25%) when totally unknown SUT is met. The other corner case shows that nearly no manual steps required when the unknown SUT contains known processor and UUT. All models in automated approach are checked for consistency as described in the following sections. The consistency check validates the presence of all used variables such as inputs, outputs, constants, variables, memory elements and functions.

Table 5-1 *Comparison in number of steps for automated and non-automated flows*

Conditions	**Non-automated**	**Automated**
Unknown SUT	SUT Schematic/Netlist UUT manual - Processor manual Debug port support R/W memory/register Peripheral controller Test-ware development Test integration	- UUT manual UUT model Processor manual Processor model Processor model Peripheral controller model ISA model, Ini. Sequence -
	8 Steps	6 Steps
Unknown SUT with known processor	SUT Schematic/Netlist UUT manual - Processor manual Peripheral controller Test-ware development Test integration	- UUT manual UUT model Processor manual Peripheral controller model - -
	6 Steps	4 Steps
Unknown SUT with known UUT	SUT Schematic/Netlist UUT manual - Processor manual Debug port support R/W memory/register Peripheral controller Test-ware development Test integration	- UUT manual - Processor manual Processor model Processor model Peripheral controller model - -
	8 Steps	4 Steps
Unknown SUT with known UUT and processor	SUT Schematic/Netlist Debug port support Test integration	- - -
	3 Steps	No steps

In case if any variable is missing or described incorrectly or cyclic dependency is found the engineer will get a notification in an automated approach.

The process of manual creation of the behavior model is iterative. The general idea behind the iterative approach is that the model should not describe the functionality that is not needed for test data path modeling. Otherwise, manual creation of the behavior model of complex components like μP would not be feasible. The iterative approach implies the addition of the new functions to the model as required, typically without any changes to the rest of the model. This approach helps to maintain the complexity of the model.

The behavior model is developed in self-contained iterations. At the end of iteration the test program is synthesized. If the synthesis fails the last iteration should be revised to eliminate the cause of fail. After every iteration, the test program is simulated or executed on the test setup (in case if the SUT is available) to find if the synthesized test program meets the test requirements. In case of inconsistencies with the test requirements the next iteration in the model development is undertaken for adding the functionality that helps the synthesized test program fulfill the requirements (e.g. ISP time limit).

The traditional non-automated development of the test program is not so flexible in adding or changing the functionality of the test program as automated model-based approach. When manually developed test program has to be modified to meet the test requirements it typically implies the deep refactoring of the program or even rewriting the whole program. Thus, it has to be fully verified and tested again. That makes the traditional development flow to be time consuming and the produced test program is hardly reusable in other test projects that have stricter test requirements.

5.2 Test data path model as a constraint satisfaction problem

In PCBT the test program is controlling the processor on the PCBA. The test program is executed on the external test hardware, which translates the program into the sequences of TAP signals. These sequences are applied to the TAP of the PCBA. Let us name these sequences of TAP signals as the “raw” test program. The “raw” test program can be translated into format that particular tester is capable to interpret. Hence, the goal of automated test program synthesis is to obtain the “raw” test program. This makes proposed approach independent from particular test system or test setup. Moreover, the “raw” test program is easily adaptable to the arbitrary boundary-scan test system.

Figure 5-3 depicts the workflow stages for obtaining the “raw” test program from the partial functional model of the test data path.

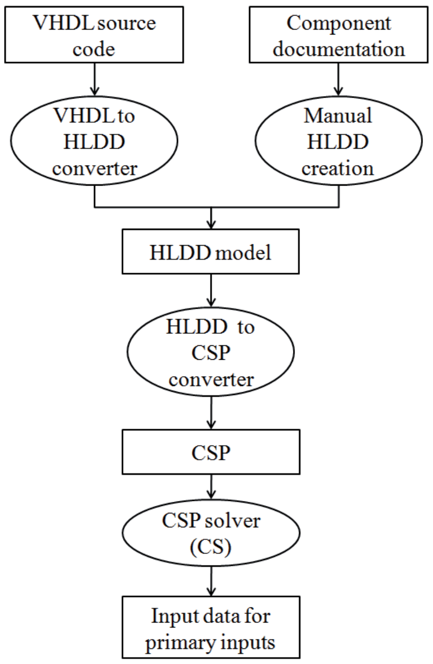


Figure 5-3 Partial functional model to raw test program transformation flow

The HLDDs may be considered as the collection of rules that have to be obeyed in order to justify the test path, apply test pattern, sense the response and propagate it to the external tester. In order to synthesize automatically the “raw” test program, which does previously mentioned test tasks, the test data path model is converted into the constraint satisfaction problem (CSP). The CSP is solved by the constraint solver (CS). As a solution CS reports the values for the variables that represent the TAP pins. In other words, the CS produces the “raw” test program, which is the goal of automated test program synthesis.

Constraint satisfaction, in its basic form, involves finding a value for each one of problem variables. The constraints specify the subsets of values that cannot be used together. The main algorithmic techniques that solve CSPs are local search and backtracking search. The backtracking search traverses the search-tree using a depth-first strategy. The branches that leave the node represent alternative choices that need to be examined to find a solution. The constraints are used for pruning sub-trees that do not lead to the solutions. Backtracking search algorithm guarantees that a solution will be found if it exists. If CSP does not have a solution the backtracking search can be used to prove that and it also finds a provably optimal solution. There are many techniques for improving the backtracking search algorithm. This issue will be discussed in details in the following sections when discussing the backtracking search implemented in JaCoP [55].

A fundamental challenge in constraint programming is to understand the computational complexity of problems involving constraints. In their most general form, CSPs are NP-Hard [56]. The complexity that corresponds to the CSP of the test program synthesis, which is based on the test data path model, is formed by the complexity of CSP that reflects the behavioral model (HLDD graphs). The structural model is traversed in linear time because its metamodel is basically a map of structural properties. Hence, only behavioral part of the test data path model is solved as a CSP and operations with the structural part are considered as programming tasks that does not require a CS to be solved.

5.2.1 Formulation of Constraint Satisfaction Problem

In this section, the concepts used in the remaining sections of this chapter are defined. The definitions are taken from "Handbook of Constraint Programming" [56].

"A constraint satisfaction problem (CSP) is a triple $\langle X, D, C\rangle$ where: X is a set of *variables*, $\{x_1, \dots, x_n\}$; D is a set of *domains* $D_1, .., D_n$ associated with $x_1, \dots, x_n$ respectively; and C is a set of *constraints*. Each constraint $c \in C$ is a pair $c = \langle \sigma, \rho\rangle$ where σ, the constraint *scope*, is a list of variables, and ρ, the constraint *relation*, is a subset of the Cartesian product of their domains."

"The domain of a variable is a set of possible values that can be assigned to it. In board and electronic component modeling task it is assumed that the domain of a variable is a finite set. An assignment is a pair(x_i, a), which means that variable $x_i \in X$ is assigned the value $a \in D_i$. A *compound assignment* is a set of assignments to distinct variables in X. A *complete assignment* is a compound assignment to all variables in X."

"The relation of a constraint $c = \langle \sigma_c, \rho_c\rangle$ specifies the acceptable assignments to the variables in its scope. That is, if the constraint scope σ_c is $\{x_{i_1}, x_{i_2}, \dots x_{i_k}\}$ and $\langle a_1, a_2, \dots, a_k\rangle \in \rho_c$, the compound assignment assigning a_i to x_{i_k}, $1 \le i \le k$, is an acceptable assignment, in other word the assignment satisfies the constraint c. A solution to the CSP instance $\langle X, D, C\rangle$ is a complete assignment such that for every constraint $c \in C$, the restrictions of the assignment to the scope σ_c satisfies the constraint."

"A binary constraint is *arc consistent* if for every value in the domain of either variable, there exists a value in the domain of the other such that the pair of values satisfies the constraint. A non-binary constraint is *generalized arc consistent* or *hyper-arc consistent* iff for any value for a variable in its scope, there exists a value for every other variable in the scope such that the tuple satisfies the constraint. Domain propagation on a constraint removes unsupported values (i.e. values which cannot be extended to a pair of tuple of values satisfying the constraints) from the domains of the variables in its scope until the constraint is (generalized) arc consistent."

“A constraint c on variables with ordered domains (such as integers) is bounds consistent if for every variable x in its scope, there exists a value d_j for every other variable x_j $(1 \leq j \leq k)$ in the scope of c, with $min_{D_j} \leq d_j \leq max_{D_j}$, such that the compound assignment $\{(x, l), (x_1, d_1), \dots, (x_k, d_k)\}$ satisfies c, where l is the minimum of the domain of x, *and* similarly, values d'_j can be found with $min_{D_j} \leq d'_j \leq max_{D_j}$, such that $\{(x, u), (x_1, d'_1), \dots, (x_k, d'_k)\}$, satisfies c, where u is the maximum of the domain of x. *Bounds propagation* on an arithmetic constraint reduces the bounds of the variables until the constraint is bounds consistent.”

5.2.1.1 Representing a problem

The precise definition does not exist for the representation of the particular problem P as a CSP. A possible definition is that CSP $M = \langle X, D, C \rangle$ represents a problem P, or M is a *model* of P, if every solution of C corresponds to a solution of P and every solution of P can be derived from at least one solution of C.

The above given definition does not require the one-to-one correspondence between the solutions of P and M. The reason for that is the possible symmetry of the solutions to M. In other words, multiple solutions of M may correspond to the same solution to P. The symmetry is often introduced modeling a problem as CSP, by representing indistinguishable entities of P by distinct variables or values in M.

If the symmetry is present both in P and M the additional constraints may be added to M to eliminate all but one solution in every symmetry equivalence class. These constraints are called *symmetry-breaking* constraints and obviously they exist only in M and not in P. The symmetry breaking constraints may cause the situation when one solution to M corresponds to multiple symmetrically-equivalent solutions to P. This leads to the conclusion that correspondence between the solutions to P and solutions to M can be many-to-many. The last statement says that finding the true solutions to P by solving M causes the uncertainty and additional complications. Hence, this might be avoided by agreeing that symmetry-breaking constraints can be ignored in considering whether the M is a model of P.

In this thesis in modeling the test data path as a CSP the variables and values are chosen to represent the entities in P and the constraints are written on these variables to represent the rules and restrictions defining the solutions to P. The exact details of modeling are presented in the following sections. Here it is worth to stress that any solution to the proposed CSP model M yields exactly one solution to P, and any solution to P corresponds to a solution to M or is symmetrically equivalent to such solution. Moreover, if M has no solutions, this is because P itself has no solutions.

5.2.2 Java Constraint Programming framework

The problem of constraint satisfaction in the automated test program synthesis reduces to representing the behavioral model (HLDD graphs) as a CSP and imposing additional constraints that are extracted from the structural model. The way of modeling a behavioral part as a CSP can have a dramatic effect on how easy it is to find a solution, or indeed whether it can realistically be solved at all. A complicating factor in modeling is the interaction between the model, the search algorithm and the search heuristics. To reduce this complexity factor the decision was made to use the Java Constraint Programming (JaCoP) framework. JaCoP provides the backtracking engine implemented inside the depth-first search algorithm together with a number of search heuristics and a wide range of various constraints.

JaCoP library provides constraint programming paradigm implemented in Java. It provides primitives to define a triple $\langle X, D, C\rangle$: finite domain (FD) variables for defining X and D, and constraints (C), as well as a number of search methods.

JaCoP supports finite domain variables (FDV) with continuous domains e.g. ($\{0..100\}$) and domains that contain holes e.g. ($\{0..10\} \cup \{12..100\}$), in this domain the value 11 is missing. In this work FDVs are used to model the HLDD variables. One special variable class is a Boolean variable. It has been added to JaCoP as it can be handled more efficiently than FDVs with multiple elements in their domain. Boolean variable can be used as any other variable.

JaCoP library provides most commonly used primitive constraints, such as equality, inequality as well as logical, reified and conditional constraints. It contains also number of global constraints and Boolean constraints.

In this thesis, there are four major types of constraints that have been used in the CSP formulation:

- Primitive constraints
- Logical constraints
- Conditional constraints
- Global constraints

5.2.2.1 Primitive constraints

A set of primitive constraints that are offered in JaCoP include basic arithmetic operations $(+,-,\times,\div)$ as well as basic relations $(=,\neq,<,\leq,>,\geq)$. The specification and the description of available primitive constraints is given in Table 5-2. The subtraction and division are not implemented explicitly, but since constraints define relations between variables, they are provided using addition and multiplication.

Primitive constraints can be used as arguments in logical, conditional and global constrains, and in primitive constraints itself. In the task of HLDDs representation as CSP the primitive constraints are used to define the operation of the function $f_m(x)$ (see Table 5-2). However, not all operations are defined with primitive constraints, namely, logical and bitwise operations are defined using logical constraints and global constraints.

Table 5-2 *List of primitive constraints (*Const - constant)*

Description	**JaCoP Specification**
$X = \text{Const}$	`XeqC(X, Const)`
$X = Y$	`XeqY(X, Y)`
$X \neq \text{Const}$	`XneqC(X, Const)`
$X \neq Y$	`XneqY(X, Y)`
$X > \text{Const}$	`XgtC(X, Const)`
$X > Y$	`XgtY(X, Y)`
$X \geq \text{Const}$	`XgteqC(X, Const)`
$X \geq Y$	`XgteqY(X, Y)`
$X < \text{Const}$	`XltC(X, Const)`
$X < Y$	`XltY(X, Y)`
$X \leq \text{Const}$	`XlteqC(X, Const)`
$X \leq Y$	`XlteqY(X, Y)`
$X \times \text{Const} = Z$	`XmulCeqZ(X, Const, Z)`
$X \times Y = Z$	`XmulYeqZ(X, Y, Z)`
$X \div Y = Z$	`XdivYeqZ(X, Y, Z)`
$X \bmod Y = Z$	`XmodYeqZ(X, Y, Z)`
$X + \text{Const} = Z$	`XplusCeqZ(X, Const, Z)`
$X + Y = Z$	`XplusYeqZ(X, Y, Z)`
$X + \text{Const} \leq Z$	`XplusClteqZ(X, Const, Z)`
$X + Y \leq Z$	`XplusYlteqZ(X, Y, Z)`
$X + Y > \text{Const}$	`XplusYgtC(X, Y, Const)`
$X^Y = Z$	`XexpYeqZ(X, Y, Z)`

5.2.2.2 Logical and conditional constraints

Logical and conditional constraints use primitive constraints as arguments. For detailed description and specification of these constraints see Table 5-3 and Table 5-4.

Table 5-3 *List of conditional constraints*

Description	**JaCoP Specification**
if c1 then c2	`IfThen(c1, c2)`
if c1 then c2 else c3	`IfThenElse(c1, c2, c3)`

Table 5-4 *List of logical constraints*

Description	**JaCoP Specification**
$\bar{c}$	`Not(c)`

$c_1 \cap c_2 \cap \ldots \cap c_n$	`PrimitiveConstraint[] c = {c1, c2, ...,cn};`
	`And(c);`
	or
	`ArrayList<PrimitiveConstraint> c =`
	`    new ArrayList<PrimitiveConstraint>();`
	`c.add(c1); c.add(c2); ... c.add(cn);`
	`And(c);`

$c_1 \cup c_2 \cup \ldots \cup c_n$	`PrimitiveConstraint[] c = {c1, c2, ...cn};`
	`Or(c);`
	or
	`ArrayList<PrimitiveConstraint> c =`
	`    new ArrayList<PrimitiveConstraint>();`
	`c.add(c1); c.add(c2); ... c.add(cn);`
	`Or(c);`

5.2.2.3 Global constraints

The constraint on the first row in Table 5-5 enforce that a sum of elements of FDVs' vector is equal to a given FDV sum. The second row in Table 5-5 explains the weighted sum constraint. The latter is extremely useful when the FDV variable participates in bitwise operations (e.g. |, &). The weighted sum builds a bridge between integer and bitwise representation of FDV.

Table 5-5 *List of global constraints*

Description	**JaCoP Specification**

$x_1 + x_2 + \cdots + x_n = sum$	`IntVar[] x = {x1, x2, ..., xn};`
	`IntVar sum = new IntVar(...)`
	`Sum(x, sum);`
	or
	`ArrayList<IntVar> x =`
	`      new ArrayList<IntVar>();`
	`x.add(x1); x.add(x2); ... x.add(xn);`
	`IntVar sum = new IntVar(...)`
	`Sum(x, sum);`

$w_1x_1 + w_2x_2 + \cdots$
$\cdots + w_nx_n = sum$
	`IntVar[] x = {x1, x2, ..., xn};`
	`IntVar sum = new IntVar(...)`
	`int[] w = {w1, w2, ..., wn};`
	`SumWeight(x, w, sum);`
	or
	`ArrayList<IntVar> x = new ArrayList<IntVar>();`
	`x.add(x1); x.add(x2); ... x.add(xn);`
	`IntVar sum = new IntVar(...)`
	`ArrayList<Integer> w=new ArrayList<Integer>();`
	`w.add(w1); w.add(w1); ... w.add(wn);`
	`SumWeight(x, w, sum);`

5.2.3 Representing HLDDs as a CSP using JaCoP framework

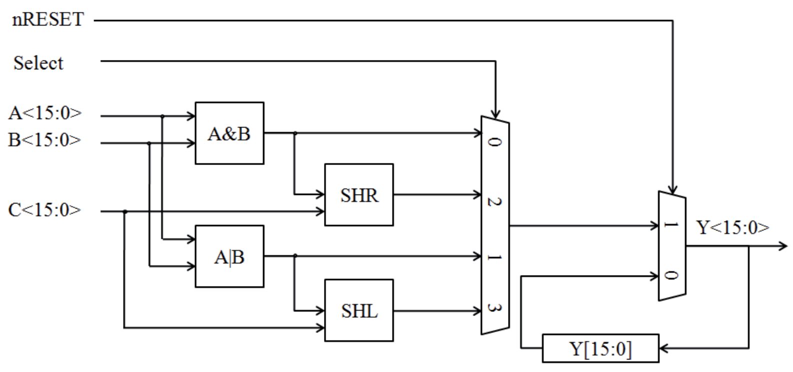


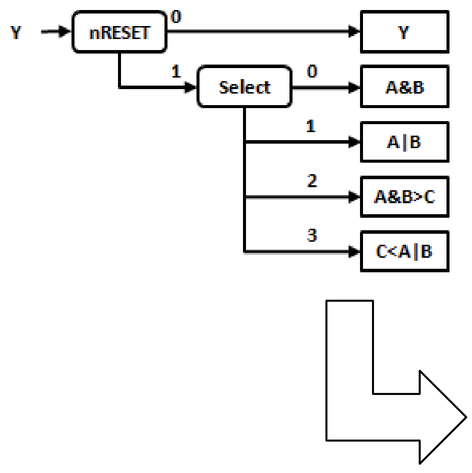
Figure 5-4 Part of a processor data path

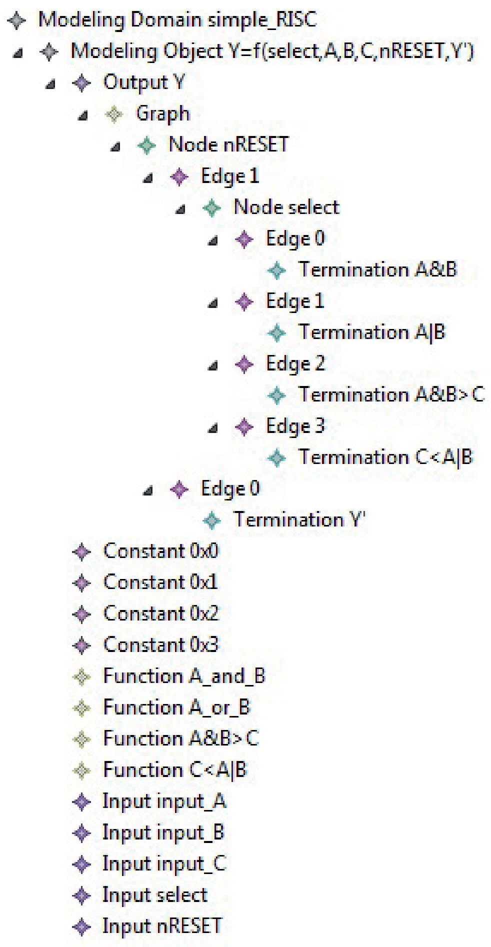
The representation of HLDDs as a CSP using JaCoP framework is explained using the part of the processor data path with control signals as an example, which is shown in Figure 5-4. Briefly, the functionality of the example circuit is the following:

- When *nRESET* signal is low (logic 0) the output of the circuit (let us call it *Y*) is equal to the previous value of Y that is stored in the register *Y[15:0]* (later referenced as *Y'd,* which is delayed value of *Y*).

- When *nRESET* signal is high (logic 1) the output of the circuit depends on the output of the multiplexer that is controlled by the *Select* signal.

  The *Select* signal selects one of the following operations:
  - When *Select* is 0, the result of the AND operation between A and B is propagated to the output of the multiplexer.
  - When *Select* is 1, the result of the OR operation between A and B is propagated to the output of the multiplexer.
  - The SHR operation is selected when *Select* is 2. Then the propagated value is a product of the AND operation between A and B that is shifted into C by one bit from the left.
  - The SHL operation is selected when *Select* is 3. Then the propagated value is a product of OR operation between A and B that is shifted into C by one bit from the right.





- Modeling Domain simple_RISC
  - Modeling Object Y=f(select,A,B,C,nRESET,Y')
    - Output Y
      - Graph
        - Node nRESET
          - Edge 1
            - Node select
              - Edge 0
                - Termination A&B
              - Edge 1
                - Termination A|B
              - Edge 2
                - Termination A&B>C
              - Edge 3
                - Termination C<A|B
          - Edge 0
            - Termination Y'
    - Constant 0x0
    - Constant 0x1
    - Constant 0x2
    - Constant 0x3
    - Function A_and_B
    - Function A_or_B
    - Function A&B>C
    - Function C<A|B
    - Input input_A
    - Input input_B
    - Input input_C
    - Input select
    - Input nRESET

Figure 5-5 HLDD model of the part of the processor data path (HLDD Graph view and equivalent model in the framework view)

The resulting behavior model (HLDD) of the structure shown in Figure 5-4 is presented in Figure 5-5. The underlying textual representation of this model is shown in Figure 5-6. The detailed description of this format (AGM) is given in Appendix A. The most valuable property of this format is that variables ($x \in X$) and nodes ($m \in M$) inside the graph (G_y) definition are ordered. Any variable that is referenced inside the graph should be declared before this graph. The exception is delayed variable. This variable ordering is possible due to the acyclic nature of the HLDD graphs. The strict order of variables in this format is extremely fast and easy to handle in transformation of HLDD model into CSP model. Every declared variable has its index inside the HLDD model. Let us denote the order of variables in the model as "natural HLDD order" relying on the indices in AGM format.

```
VAR#   0:   (i_________) "input_A"   <15:0>
VAR#   1:   (i_________) "input_B"   <15:0>
VAR#   2:   (i_________) "input_C"   <15:0>
VAR#   3:   (i_________) "select"   <1:0>
VAR#   4:   (i_________) "nRESET"   <0:0>
VAR#   5:   (__c_______) "0x0"   <1:0>   VAL = 0
VAR#   6:   (__c_______) "0x1"   <1:0>   VAL = 1
VAR#   7:   (__c_______) "0x2"   <1:0>   VAL = 2
VAR#   8:   (__c_______) "0x3"   <1:0>   VAL = 3

VAR#   9:   (____f_____) "A_and_B"   <15:0>
FUN#   AND  (A1<=0<15:0>, A2<=1<15:0>)
VAR#   10:   (____f_____) "A_or_B"   <15:0>
FUN#   OR  (A1<=0<15:0>, A2<=1<15:0>)
VAR#   11:   (____f_____) "A&B>C"   <15:0>
FUN#   SHIFT_RIGHT  (A1<=2<15:0>, A2<=9<15:0>)
VAR#   12:   (____f_____) "C<A|B"   <15:0>
FUN#   SHIFT_LEFT  (A1<=2<15:0>, A2<=10<15:0>)

VAR#   13:   (_o________) "Y"   <15:0>
GRP#   0:   BEG = 0,  LEN = 7  -----
0  0: (n___)  (  1=>1  0=>2)    V = 4    "nRESET"  <0:0>
1  1: (n___)  (  0=>3  1=>4  2=>5  3=>6)V = 3  "select"  <1:0>
2  2: (____)  (  0   0)    V = 13    "Y"  <15:0>
3  3: (____)  (  0   0)    V = 9    "A_and_B"  <15:0>
4  4: (____)  (  0   0)    V = 10    "A_or_B"  <15:0>
5  5: (____)  (  0   0)    V = 11    "A&B>C"  <15:0>
6  6: (____)  (  0   0)    V = 12    "C<A|B"  <15:0>
```

Figure 5-6 Textual representation of HLDD model (AGM format)

The first step of CSP modeling is to define variables and their domains. These are the first two elements in a triple $\langle X, D, C\rangle$. The variables and respective domains that are extracted from this HLDD model are shown in Table 5-6. The second step is to model arithmetical and logical functions defined in this model. This corresponds to adding constraints like $x(m) = f(x_j, .., x_k)$; $0 \le j, k \le n$ to the set C (where n is index of the last variable in AGM format). The example given in Figure 5-4 was selected to show that not only trivial expression can be modeled using predefined constraints from JaCoP framework. The constraints that were listed in section 5.2.2 can be reused to define new constraints that are needed for particular problem in hand.

Table 5-6 *List of variables*

Variable x ($x \in X$)	Domain d ($d \in D$)	Variable x ($x \in X$)	Domain d ($d \in D$)
Select	{0..3}	Constant_0x1	{1}
Input_A	{0..65535}	Constant_0x2	{2}
Input_B	{0..65535}	Constant_0x3	{3}
Input_C	{0..65535}	A_and_B	{0..65535}
Output_Y	{0..65535}	A_or_B	{0..65535}
nReset	{0..1}	A_and_B_SHR_C	{0..65535}
Constant_0x0	{0}	A_or_B_SHL_C	{0..65535}

The nice property of JaCoP framework is that it could be extended to meet the requirements of the particular task. There are functions in our example that cannot be modeled with those "off-the-shelf" constraints. The bitwise shift-left and shift-right operations could be indeed modeled as division or multiplication, but logic *OR* and logic *AND* operations for variables that are not Boolean variables are missing in the list.

For modeling binary shift, AND and OR operations the variables *Input_A, Input_B* and *Input_C* were presented in a binary view using the *SumWeigth* constraint and Boolean variables for each bit in the binary representations of these integer variables. The results are shown below:

$Input\_A = a_0 \times 2^0 + a_1 \times 2^1 + \cdots + a_{15} \times 2^{15}$;

$Input\_B = b_0 \times 2^0 + b_1 \times 2^1 + \cdots + b_{15} \times 2^{15}$;

$Input\_C = c_0 \times 2^0 + c_1 \times 2^1 + \cdots + c_{15} \times 2^{15}$,

where variables $a_0, a_1, \dots, a_{15}, b_0, b_1, \dots, b_{15}, c_0, c_1, \dots, c_{15}$ are Boolean variables.

Thus the dual representation (binary and integer) of the variable is achieved. This allows building the constraints for the above listed functions of the model:

$A\_and\_B = And(a_0, b_0) \times 2^0 + And(a_1, b_1) \times 2^1 + \cdots + And(a_{15}, b_{15}) \times 2^{15}$;

$A\_or\_B = Or(a_0, b_0) \times 2^0 + Or(a_1, b_1) \times 2^1 + \cdots + Or(a_{15}, b_{15}) \times 2^{15}$;

$A\_and\_B\_SHR\_C = And(a_{15}, b_{15}) \times 2^0 + c_1 \times 2^1 + \cdots + c_{15} \times 2^{15}$;

$A\_or\_B\_SHL\_C = c_0 \times 2^0 + \cdots + c_{14} \times 2^{14} + Or(a_0, b_0) \times 2^{15}$.

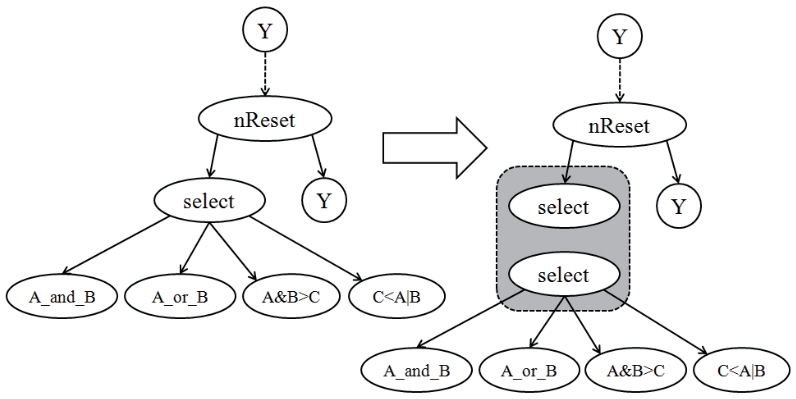


Figure 5-7 Transformation of irregular graph into linked regular graphs

The next step is to add constraints for the transitions in the HLDD graph. The list of the transitions (tuples) in the graph is obtained automatically from the *Regular Graph* data structure. The *Regular Graph* is a graph where every path from the root node to the leaf node is of the same length. The transformation of the HLDD graph structures into linked *Regular Graphs* is a simple programming task and its implementation details are not relevant to this research. The general idea of this procedure is shown in Figure 5-7.

On the basis of the *Regular graphs* the conditional constraints are constructed to model the transitions in the HLDD graphs. The primary target is to impose *full path* (m_0, m^T) *activation* constraints. The solution to these constraints is the vector X^T that satisfies one of the full path activation constraints. The formal view of the conditional constraint that models the *full path* in the HLDD graph is:

IfThen (

And (

XeqC ($x(m_1)$, a where $a \in V(x(m_1))$),

… ,

XeqC ($x(m_i)$, a where $a \in V(x(m_i))$)

),

XeqY ($y, x(m^T)$)

)

The constraints for the transitions in HLDD graph from Figure 5-5 are modeled as follows:

IfThen (*And* (*XeqC* (nRESET, Constant_0x1), *XeqC* (select, Constant_0x0)), *XeqY* (Output_Y, A_and_B));

IfThen (*And* (*XeqC* (nRESET, Constant_0x1), *XeqC* (select, Constant_0x1)),

XeqY (Output _Y, A_or_B));

IfThen (*And* (*XeqC* (nRESET, Constant_0x1), *XeqC* (select, Constant_0x2)),

XeqY (Output_Y, A_and_B_SHR_C));

IfThen (*And* (*XeqC* (nRESET, Constant_0x1), *XeqC* (select, Constant_0x3)),

XeqY (Output_Y, A_or_B_SHL_C));

IfThen (*And* (*XeqC* (nRESET, Constant_0x0), *XeqY* (select, select)),

XeqY (Output_Y, Output_Y'd)),

Output_Y'd is a previous value (delayed value) of Output_Y, that corresponds to the register "Y[15:0]" shown in the initial scheme in Figure 5-4.

It should be stressed that CSP model creation is fully automated in the proposed approach. The algorithm for HLDD transformation into CSP model that was developed to support this research is the following:

```
-----------------------------------------------------------
for each variable in HLDD model do
    if variable has constant flag
          define FDV with single value domain
          add function FDV to functionList
    else
          define FDV with full range domain
    end if
    if variable is a graph root
          regularGraphsList = buildRegularGraphs(graph)
          for each regularGraph in regularGraphsList do
                tuplesList = Get tuples from regularGraph
          end for
          fullPathTuples = joinTuples(tuplesList)
          for each tuple in fullPathTuples
                impose conditional constraint
          end for
    end if
end for
for each function FDV in functionList do
    impose arithmetical/logical constraint
end for
-----------------------------------------------------------
```

Even big HLDD models are transformed in a reasonable time due to the linear complexity of the algorithm. The HLDD graphs are traversed only once to obtain the full list of variables, functions and transitions. The transitions compose the full paths from the graph root node to the graph leafs. These full paths are tuples that are used in conditional constraints for modeling the transitions in the HLDD graphs as was shown before. The constraints that model operations of functions are imposed in the end, when all the variables of the HLDD model are defined. This is done to escape redefinition of variables, which is inevitable in case if functions are modeled before the definition of variables that participate in this function.

At this point the CSP model is constructed and it should be first checked for the consistency. The consistency check is an embedded feature of the JaCoP framework. The consistency check returns *false* if model is inconsistent and no solution could be found, while *true* indicates only that the model is consistent and in order to find the solution the CS should be executed. However, the solution may not exist even if the CSP model is consistent.

5.2.4 Solving the CSP model

Previous section (section 5.2.3) explained in details the modeling of the HLDD graphs as a CSP. In case if CS is executed on the CSP model of the circuit depicted in Figure 5-4, it returns all possible solutions. Whereas, all possible solutions are a lot of data when there are variables with big domains. Typically solving the unconstrained CSP model is not needed. Normally the CSP models are used to obtain the inputs or/and outputs that bring the system into the target state. This target state is modeled as a set of constraints that should be added to the CSP before solving it.

In this work the notion of the state of the system is defined as in Mealy machine definition [57] in the theory of computation. Briefly, the next state output of the system depends on the previous state and on the inputs of the system. Hence, the output will change as soon as the inputs are propagated to the logic. In comparison, the output change according to Moore, appears on the next clock cycle, since the change is caused only by the state. Thus, with Moore theory synchronous designs are described more naturally, whereas Mealy theory may lead to metastability of the outputs. However, Mealy machine definition typically requires fewer states and is more efficient to describe asynchronous systems. As soon as we are not interested in the precise modeling of output timings and clock may often be skipped in the modeling of system behavior the Mealy definition was chosen. However, that does not lead to the inability of modeling synchronous systems as it is shown later.

5.2.4.1 Solving the CSP for single-cycle

The single-cycle solution to the CSP in the proposed approach is defined as one time assignments to the unconstrained inputs and delayed variables. The unconstrained variables are those that do not participate in the additional constraints that specify the target state of the system. In other words, the values of the constrained variables are known in the target state. Unconstrained variables are the ones whose values are unknown in the target state and the CS is executed to obtain their values.

Let us explain the single-cycle solving on the example. First, the target state of the system needs to be defined. Typical situation that is faced in the automated test program synthesis is that engineer knows the values of the certain outputs of the system that characterize the target state, but the values of the inputs and the rest of the outputs are unknown. We mimic this with the following: test engineer knows that the output of the system shown in Figure 5-4 have to be equal to 2 and the inputs *A* and *B* must be equal to 1. To model this knowledge the following constraints are imposed (this type of constraint is referenced as *abridge* constraint):

XeqC(Input_A, Constant_0x1) // Input_A = 1
XeqC(Input_B, Constant_0x1) // Input_B = 1
XeqC(Output_Y, Constant_0x2) // Output_Y = 2

The CS returns the following two solutions for the CSP containing these abridge constraints (denoted as *Sol.1* and *Sol.2*):

Sol. 1:

Input_A = 0x1
Input_B = 0x1
Input_C = 0x1
Output_Y = 0x2
nReset = 0x1
Select = 0x3
Y'd = $\{0\ldots(2^{16}-1)\}$

This is the most obvious solution and the one that is probably expected. However, as soon as design contains a memory element (register) and solution *Sol.1* is not relying on its value, the CS will also produce a number of symmetrical solutions which is equivalent to the range of this memory element. In our case it is 2^{16} -1 symmetrical solutions (*Y'd* is 16 bit register).

The second solution (*Sol.2*) has also a huge number of symmetrical solutions, since the *nReset* is selecting the register (*Y'd*) to be propagated to the output and data path from the circuit inputs to the output is masked. Hence, the *Input_C* and *Select* variables can take any value in their domain.

Sol. 2:

Input_A = 0x1
Input_B = 0x1
Input_C = $\{0\ldots(2^{16}-1)\}$
Output_Y = 0x2
nReset = 0x0
Select = $\{0\ldots3\}$
Y'd = 2

Thus, the total number of solutions (including symmetrical solutions) is $2^{16} + 2^{18}$ and only two of them have practical value. Let us call these two solutions as *diverse* solutions. The goal of solving a CSP is to find all possible diverse solutions while skipping the symmetrical ones. For that several techniques of guiding a CS can be used.

The JaCoP provide possibility to specify the number of solutions to find. The major question that appears is in which order the solutions will be found. The ideal case would be to set the solution limit to two and get the CS to produce these two diverse solutions described above. Evidentially, the order of the solutions is defined by the order in which variables are assigned, called as "variable selection strategy". The second parameter is the "value assignment strategy" that tells the CS which value should be considered next from the domain of the variable. The most common value assignment strategies are: smaller value first, bigger value first, middle value first (selects a middle value from the current domain of FDV and then left and right values) and random value. The value assignment strategy influences the time of finding a solution.

As search method the "depth-first search" algorithm is used. This algorithm searches for a possible solution by organizing the search space as a search tree. In every node of this tree a value is assigned to the variable and a decision whether the node will be extended or the search will be cut in this node is made. The search is cut if the assignment to the selected variable does not meet all constraints. Since assignment of a value to a variable triggers the constraint propagation, the decision can be made to continue or to cut the search at this node of the search tree.

In Table 5-7 the details of solving the CSP with different variable selection and assignment strategies are presented. For the given abridge constraints (*Input_A*=1, *Input_B*=1, *Output_Y*=2) the most efficient strategy according to the experimental results shown in Table 5-7 is reversed HLDD variable order and smaller value first assignment. These results also show that efficiency of different strategies heavily depends on the CSP itself and on the abridge constraints due to the huge difference in diverse solution indices and time. Hence, if the *Output_Y* is constraint to the value close to the domain maximum, then "bigger value first" will be more preferable assignment strategy. The presented variable order selection strategies correspond to the natural and reversed list of variables that is defined in the textual representation in the HLDD graphs (Figure 5-6).

Table 5-7 *Solution details with constraints Input_A=1, Input_B=1, Output_Y=2*

Variable selection strategy	Assignment strategy	Diverse solution index	Time (ms)
Natural HLDD order	smaller value first	1, 5	79
Reversed HLDD order	smaller value first	1, 3	78
Natural HLDD order	bigger value first	1, 131070	>1000
Reversed HLDD order	bigger value first	1, 65534	>1000

The order of variables influences the search space as the search tree is based on it. According to the experiments the natural HLDD order requires more solutions to be traversed for obtaining all the diverse solutions in comparison to the reversed HLDD order. The benefit from using the reversed order depends on the particular HLDD graph structure. As practice shows the natural order require less backtracking than the reversed, but the overall search time may not vary as much as one would expect. Although, there is a slight difference between reviewed variable selection strategies, the reversed HLDD order is used for variables selection strategy in the next experiments. Generally, the reversed HLDD order allows obtaining all diverse solutions in a shorter time while producing less symmetrical solutions.

5.2.4.2 Solving CSP for multiple cycles

Even for relatively simple models with registers or other memory elements sometimes it is not sufficient to find a solution within one cycle. In case if the solution is relying on the value in the memory element it should be proved that this value is valid and could be assigned to this element in a deterministic way. Hence, it is often necessary to know the initial state of the system in a number of states in the past. In other words, the sequence of states to assign that value to the memory element should be found in order to prove that the solution is valid.

The initial state is a state that is reachable by applying for example a reset signal or when values of the memory elements are known to be valid (e.g. reset values). Hence, the initial state is a point in time, which is provably reachable and which is used as a starting point for bringing the system to the target state. The target state of the model should not be defined loosely. Otherwise the number of possible solutions will grow vastly. Thus, in solving for multiple cycles it becomes crucial to define the target state of the model as precise as possible.

Let us discuss CSP solving for multiple cycles on the example used in previous section (section 5.2.4.1). Typically, the goal of CSP solving is to find the shortest sequence of states that leads to the desired state. Thus, if the solution after single cycle solving is not relying on the value in the memory element (see Sol. 1 in section 5.2.4.1) then the shortest sequence is found and there is no need in CSP solving for multiple cycles. That means that the suitable combinatorial path through the circuit that brings the system into target state is found. The combinatorial path is masking the values in memory elements in the way that the output(s) of the system are not influenced by them on the given cycle. However, in many cases this combinatorial path does not exist and the solution that brings the system from deterministic initial state to the desired (target) state is required.

In the example from previous section the solving for multiple cycles is required if the first solution is made invalid by introducing additional abridge constraint for *Input_C* (e.g. *XeqC*(Input_C, Constant_0x0)). Then the second solution (*Sol.2*) becomes the only diverse solution, because in the *Sol.1 Input_C* equals to 1. The solution *Sol.2* depends on the memory element (*Y'd* = 2), hence, it is not a deterministic one. Thus, it is required to find the state in which *Y[15:0]* is assigned with 2 (Y'd := 2). In case, if that state is not a deterministic one (not a reset state) or relying again on the memory element value, another state should be found that leads to that state, and so on, until the repetitive state is met or one of the above mentioned conditions is fulfilled.

In the developed CSP solving framework it is possible to limit the number of states in the sequence as well as to limit the number of solutions to search in parallel. The reset signal should be explicitly defined if it exists and the active value of the reset signal should be declared.

The details of the developed algorithm for solving CSP for multiple cycles are described below:

```
--------------------------------------------------------------
set Parallelism Limit(pLimit)
set Cycles Limit(cLimit)
set reset signal name (resetName)
set reset signal active level (resetLevel)
for each abridgeConstraint do
    impose abridgeConstraint
end for
fdvValueMap = solve CSP
currentStatesList = create states (fdvValueMap)
for each state in currentStatesList do
    if state does not depend on delayed FDV
          solutionsList add state
    else if resetLevel equals (state get value(resetName))
          solutionsList add state
    end if
end for
if solutionsList is not empty
    diverseSolutions =find diverse solutions(solutionsList)
    return diverseSolutions
end if
currentStatesList = remove duplicates in currentStatesList
diverseSolutions = solve state backward (currentStatesList)
return diverseSolutions
--------------------------------------------------------------

solve state backward (List nextStatesList)
--------------------------------------------------------------
for each nextState in nextStatesList
    abridgeConstrs = get abridge constraint from nextState
    for each abridgeConstraint from abridgeConstrs do
          impose abridgeConstraint
    end for
    fdvValueMap = solve CSP
    currentStatesList = create states (fdvValueMap)
```

```
        for each state in currentStatesList do
              if state is repetitive state
                    continue with next state
              end if
              state set next state (nextState)
              if state does not depend on delayed FDV
                    solutionsList add state
              else   if   resetLevel   equals   (state   get
        value(resetName))
                    solutionsList add state
              else  stateListToSolve add state
              end if
        end for
    end for

    if solutionsList is not empty
        diverseSolutions =find diverse solutions(solutionsList)
        return diverseSolutions
    end if
    stateListToSolve = remove duplicates in stateListToSolve
    return solve state backward (stateListToSolve)
    -----------------------------------------------------------
```

This algorithm consists of two parts. The first part is the same as for single cycle solving with the only difference: if no deterministic solution is found the states (non-deterministic solutions) are passed to the function that continues solving procedure. The `solve state backward` function is the second part of the algorithm. It considers each of the supplied state as a new target state and imposes the new set of abridge constraints. The set of previously imposed abridge constraints is excluded from CSP before imposing a new set. Then the CS is executed to get the predecessor states (`currentStatesList`) for each of the new target states (`nextState`).

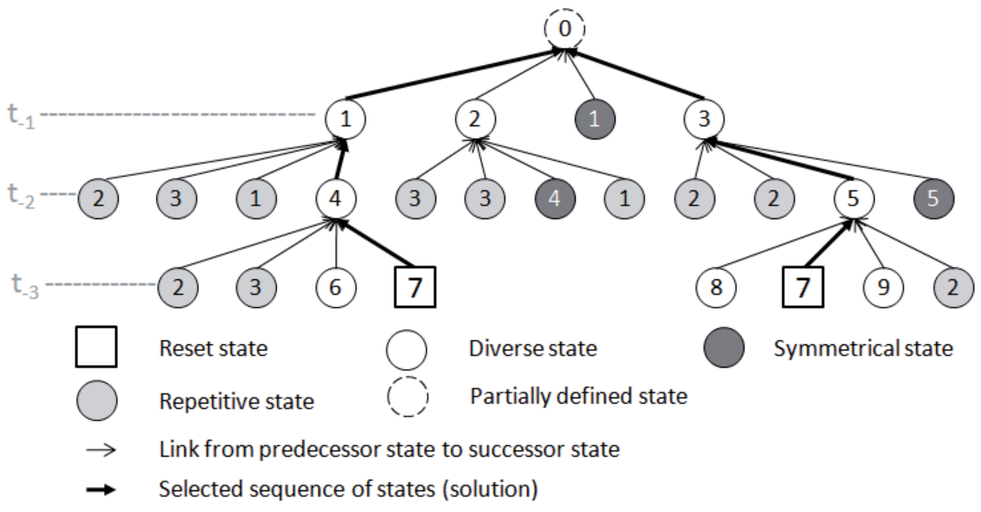


Figure 5-8 Solution searching tree

Every new predecessor state is checked for being repetitive and, in case it is repetitive, state is eliminated from the set of the next target states (`stateListToSolve`). The important property of each state is that it knows its successor state. Hence, in case if the predecessor state is a deterministic state (added to the `solutionsList`) the sequence of states that compose the complete solution can be easily restored.

In Figure 5-8 is shown a typical solution searching tree that pictures multiple cycles solving algorithm. The number inside the circuit represents the index of the state. All symmetrical solutions are represented by the same number and the one, which is found first, is considered to be a diverse state and is highlighted in white color. The symmetrical and repetitive states are shadowed (grey color). The zero state (0) is the first target state and its time label is t_0 (current time). The states that are placed on the level marked as t_{-1} are obtained in the first part of the algorithm. The other levels (t_{-2}, t_{-3}, etc.) are filled with states found by the "solve state backward" function. In the example in Figure 5-8 the solution is found when the reset state is met (state with index 7).

In order to conclude the presented approach the same CSP example from section 5.2.4.1 is used to show the multiple cycles CSP solving methodology in details. Let us consider that the following abridge constraints describe the target state of the system:

XeqC(Input_A, Constant_0x1) // Input_A = 1
XeqC(Input_B, Constant_0x1) // Input_B = 1
XeqC(Input_C, Constant_0x0) // Input_C = 0
XeqC(Output_Y, Constant_0x2) // Output_Y = 2

Due to the fact that the actual reset signal functionality is not present in this design, the only possibility for CS to complete the search with the proper solution is to find the state that is not relying on the value in the register (Y'd). Note that the *nReset* signal is just a control input of the multiplexer (see Figure 5-4). Thus, the reset signal is not defined as well as its active level.

The CS can find only one diverse state on time level t_{-1}:

Input_A = 0x1
Input_B = 0x1
Input_C = 0x0
Output_Y = 0x2
nReset = 0x0
Select = {0…3}
Y'd = 2

This state is then given as argument to the "solve state backward" function. Then the following abridge constraints are extracted for defining the new initial state:

XeqC(Output_Y, Constant_0x2) // Output_Y = 2

Table 5-8 *Solution states for level t_{-2}*

FDV	**State 1**	**State 2**	**State 3**	**State 4**	**State 5**
Input_A	0..0xFFFF	$a \cap b = 2$, $a \in A$, $b \in B$	2	$a \cap b \gg c = 2$ $a \in A, b \in B$	$c \ll a \cup b = 2$ $a \in A, b \in B$
Input_B	0..0xFFFF		2		
Input_C	0..0xFFFF	0..0xFFFF	0..0xFFFF	4, 5	1, 0x8001
Select	0..3	0	1	2	3
nReset	0	1	1	1	1
Output_Y	2	2	2	2	2
Y'd	2	0..0xFFFF	0..0xFFFF	0..0xFFFF	0..0xFFFF

This abridge constraint is derived from the delayed value of the output Y which is Y'd =2. Thus, if on level t_{-1} delayed value of Y is equal to 2, then on level t_{-2} the *Y* itself should be equal to 2 also, where level t_{-2} is the next level to be solved after the t_{-1}. Inputs remain unconstrained because the previous constraints are valid only for the level t_{-1} and there are no requirements for input values of the states at level t_{-2}.

State 1 on level t_{-2} is a repetitive state to the only state on the level t_{-1}. State 1 has the same index as state on the level t_{-1} to underline the repetition of states. Both are relying on the value of the delayed *Y (Y'd)* and in both cases *nReset* signal is masking the values of other inputs, thus FDVs for inputs *A, B, C* and *Select* can take any values from their domains. Because state 1 on level t_{-2} is a repetitive state to state on level t_{-1} it is not included in the solution states list.

In state 2 *nReset* is selecting the combinatorial path from inputs to the outputs, thus *Y* delayed FDV can take any value from 0 to 2^{16}-1. *Select* signal is selecting the result of the logic-AND operation between inputs *A* and *B* to be propagated to the output. The input C is free to take any value from its domain. As a result the first solution is found. It consists from states state 2 (level t_{-2}) and state 1(level t_{-1}).

State 3 leads to another solution, as it is also the deterministic state. The difference between state 2 and 3 is that *Select* signal is selecting in the latter case the result of the logic-OR operation between inputs *A* and *B* to be propagated to the output. In this case the domains of inputs *A* and *B* are narrowed down to value 2 and FDV of input *C* is in boundaries of its initial domain. The second solution consists of state 3 (level t_{-2}) and state 1(level t_{-1}).

State 4 and state 5 also make solutions together with state 1(level t_{-1}). The difference from previously described solutions, besides the value of the *Select* FDV, is that domain of FDV for input *C* is narrowed down to two values. This is because in states 4 and 5 the combinatorial path contains functional blocks that shift left or right value of input *C*.

The states from Table 5-8 make a complete solution tree of the described CSP on the level t_{-2}. However, even in this simple example the number of total states on the t_{-2} level is blasting. Thus, the policy of combining symmetrical states on the same level into a single diverse state (as shown in Table 5-8) is extremely useful also in terms of memory saving. The diverse state contains all the values of symmetrical states for every FDV. Thus, no effort is needed to collect the symmetrical states into a single diverse state. However, at the moment of presenting the results the choice has to be made which value from the FDV domain to use.

The important question is: are we able to find all the possible solutions using the developed framework? The answer is yes and no. In theory, there are no objections to find all solutions using developed framework, however, in practice this requires a lot of resources when a CSP models a real world designs and obviously gets bigger than in these examples. To predict the behavior of the CS the resources are constrained by setting the parallelism and cycles limit. The parallelism limit sets boundaries on the number of states to search for every set of abridge constraints (this includes symmetrical states). When the parallelism limit is set too low, valuable diverse solution may not be discovered. On the other hand, when the limit is too big the resources are spent to find unnecessary symmetrical solutions. The order of the diverse solutions in the search space is selected by the variable selection and variable assignment strategies as was shown previously in Table 5-7. There is no universal strategy to set these parameters for finding all diverse solutions with minimum resources. Every CSP model requires deep study to find the optimal CS settings. Moreover, different abridge constraints to the same CSP influence the time spend by CS to find the same number of solutions.

What is left untouched in this discussion is the size of the CSP. It has a direct influence on the CS runtime and the memory requirements. Moreover, the CSP size (number of constraints, number of FDVs) has relations to the size of the design it models. Thus, modeling the initial design as a set of smaller CSP models rather than a single one may have a positive influence on the resource requirements.

5.2.4.3 Experimental results

The experiments were run on the machine with the following specifications: CPU Intel® Core™ 2 Duo P8700 2,53GHz, RAM 2GB, MS Windows 7 32-bit Operating System. The goal is not to achieve the fast run-times of the CS, but to study the influence of various CS settings and variable selection strategies on the results of solving. The ITC99 benchmarks (b00 [58] and b01 – b10 [59]) are selected due to the availability of the VHDL source code of these designs at RT- Level. Currently, only the subset of the designs from ITC99 benchmarks can be translated to the HLDD graphs. The tool that is used for VHDL to HLDD transformation supports limited subset of VHDL constructs. The list of the designs that are successfully translated is shown in the first column of Table 5-9.

Table 5-9 *Characteristics of ITC99 benchmarks and solving times*

Design	**FFs**	**Nodes in Graphs**	**Variables**	**Memory Variables**	**Decisions made/wrong**		**Solving time for one state (ms)**	
					Natural variable order	Reversed variable order	Natural variable order	Reversed variable order
b00	18	46	70	2	9/0	23/2	10.5	13.1
b01	5	49	27	1	5/0	5/0	1.0	1.0
b02	4	26	14	1	5/1	4/0	0.7	0.4
b03	30	214	49	15	22/1	33/0	2.4	2.6
b04	66	71	70	9	18/1	33/0	2.3	3.5
b06	9	114	25	1	6/1	8/0	0.7	0.5
b09	28	69	77	20	9/1	16/0	1.2	1.1
b10	17	285	123	16	24/2	38/0	2.7	2.1

Table 5-9 presents the characteristics of the designs used in the experiments and the CS run-times for solving these designs. In this table the number of memory cells (FFs), number of nodes in the HLDD graphs, number of variables and number of memory variables are reported. The CS run-times of producing the result for one state (cycles limit and parallelism limit are set to 1) are given in the two last columns. The column next to the last one presents the run-time results when "variable selection order" setting uses the order of variables in the textual representation of the HLDD graphs. This order is called as "natural variable order" in the HLDD graphs. The last column shows the run-time results for the alternative variable selection order, which is the reversed natural order. The columns that correspond to the heading "Decisions made/wrong decisions" outline the number of decision made by CS while traversing the search space tree to find the first met solution. Again, there are two columns under this heading that correspond to the natural variable order and to the reversed natural order.

The figures shown in the last columns in Table 5-9 are denote the time spend for the actions described below. The first action is the CSP model construction from HLDD graphs description, which is the functionality developed in frames of this research. The second action is solving the CSP model by CS. The latter action belongs to the functionality of the JaCoP framework. The time for CSP model construction is negligibly small in comparison to the time taken by the CS.

The figures in Table 5-9 show that none of the variable selection strategies outperforms the other in finding the single state solution. However, it is clearly seen from Table 5-9 that the CS makes less decisions when variables are selected as they appear in HLDD graphs, but in the same case it also makes more wrong decisions, which is costly in terms of time. The exceptional results are received with the design *b00*. The solving time reported for this design is multiple times bigger than for other designs due to the presence of 16-bit variable. This is the only design that has variable with the domain of this size. The other reason is that this variable is the variable with delayed value and, hence, has a dual representation in the CSP (current state and previous state or delayed value) which multiplies the number of nodes in the search tree.

In Table 5-10 and Table 5-11 the results of solving the CSP when the parallel limit is set to 100 states and the cycle limit is 1 are presented. These experiments are aimed to show that proposed modeling methodology constructs models that can be efficiently solved by CS. The CS run-time figures do not grow linearly if the parallel limit is changed from 1 to 100, and in most cases the dependency between found solutions and time is changing. The more solutions CS has to find the less time is spend for finding every solution.

Table 5-10 presents the details of solving the selected ITC99 benchmarks (first column) with parallel limit set to 100 and CS is using natural order of variables for variable selection strategy. The second column outlines the number of found solutions. As the parallel limit is set to 100 and the cycle limit is 1 the maximum number of found solutions is 100. If the number in the second column is less than 100 it means that less than 100 solutions exists for that design. The third column shows the number of diverse solution among found solutions.

Table 5-10 *Details of solving ITC99 benchmarks (Parallel limit =100, variable selection strategy = natural variable order)*

Design	**Solutions found**	**Diverse Solutions**	**Visited nodes in the search**	**Decisions**	**Wrong decisions**	**Max. search depth**	**Time (ms)**
b00	100	100	2097451	1048778	1048673	32803	58591
b01	64	18	83	73	10	11	18
b02	28	9	35	31	4	10	16
b03	100	100	122	121	1	38	16
b04	100	100	118	117	1	117	18
b06	56	35	71	63	8	9	18
b09	100	100	108	107	1	58	24
b10	100	100	174	146	28	32	30

The rest of the columns in Table 5-10 describe the details of process of solving. The column named “Visited nodes in the search tree” tells how many nodes the CS has visited to find the number of solutions shown in the second column. The next columns contain figures of how many decisions are made by CS and how many wrong decisions are taken. The “Max. search depth” column shows the maximum depth reached by CS in the search tree.

In Table 5-10 the correlation between number of found solutions, run-time of CS and other details of solving process are explained. The number of visited nodes in the search tree is a sum of decisions and wrong decisions.

The important conclusion is that the time needed to find the certain number of solutions for particular CSP is hard to predict. As a result the number of visited nodes in the search tree is also unpredictable. However, it is worth to compare these figures with the details shown in Table 5-11. This comparison will help to study the influence of the variable selection strategy on the results of CSP solving.

In Table 5-11 all columns have exactly the same order and meaning as columns in Table 5-10. The difference lies in figures that correspond to the details of solving the CSP models when the variable selection strategy is set to the reversed natural order of variables. The number of found solutions for every listed design equals to the data in Table 5-10. However, the number of diverse solutions varies for designs b00, b03, b04, b09, b10 that is due to the different variables selection order that changes the order of nodes in the search tree. Hence, if not all solutions have been discovered (b00, b03, b04, b09 and b10), then, obviously, among the found solution there are those solutions that are not discovered with other variable selection strategy and vice versa.

Table 5-11 *Details of solving ITC99 benchmarks (Parallel limit =100, variable selection strategy = reversed variable order)*

Design	**Solutions found**	**Diverse solutions**	**Visited nodes in the search tree**	**Decisions**	**Wrong decisions**	**Max. search depth**	**Time (ms)**
b00	100	1	127	125	2	123	27
b01	64	18	117	90	27	14	28
b02	28	9	45	36	9	10	16
b03	100	4	166	148	18	36	19
b04	100	7	130	130	0	39	22
b06	56	35	1871	963	908	16	55
b09	100	52	3145890	1573002	1572888	533	27158
b10	100	1	135	135	0	41	31

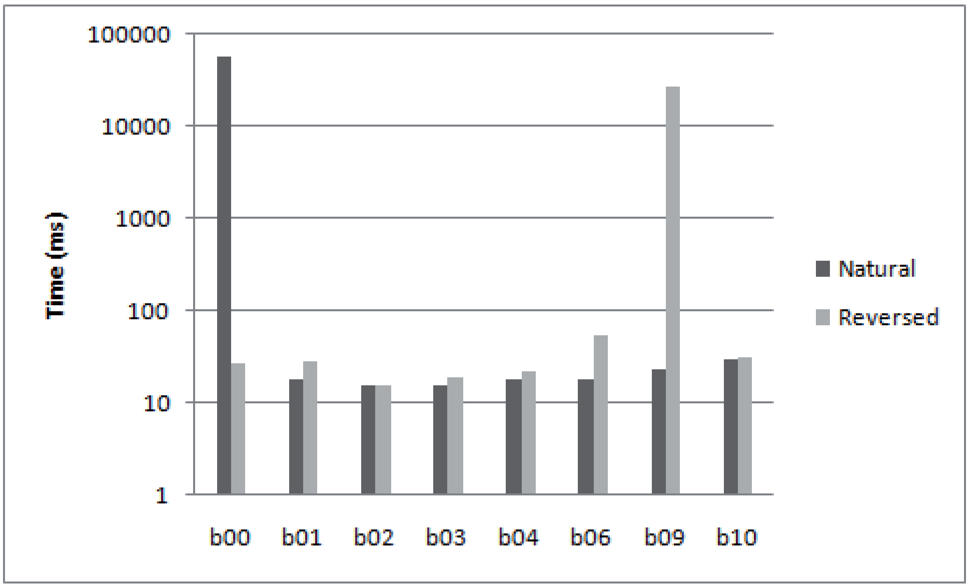


Figure 5-9 CS run-time comparison for different variable selection orders

With natural variable selection order in designs b00, b03, b04, b09 and b10 every found solution is a diverse solution, the situation is different in case of reversed natural variable selection order. Moreover, in the first case less wrong decision are made that explains why the maximum search depth is bigger (every wrong decision is causing a backtracking) and why the number of nodes visited in the search tree is smaller. Hence, time that is spent to find 100 solutions is shorter for almost all designs in case of natural variable selection order (Figure 5-9).

Figure 5-9 shows an exceptionally long run-times for b00 and b09. Whereas, design b00 is causing long run-time when variable selection strategy is set to use the natural order and b09 appears to be hard-to-solve for the reversed variable order setting. However, these designs are solved in a reasonable time when the variable selection strategy is changed. The study of the search tree for both cases gives an explanation for these run-time anomalies.

In both cases (b00 and b09) exists a variable whose domain is much smaller than the domain of other variables in the list. These variables are represented as “reconvergent fan-out” in the search tree. Thus, the deep location of this variable in the search tree causes a lot of backtracks to be made before the wrong assignment is found. This is a drawback of the used search heuristics (depth-first search). Hence, when the variables with smaller domains are assigned before the variables with bigger domains, the runtime is shorter, because of smaller number of backtracks to make in case of assignment that causes inconsistency. The sorting of variables by the domain size may help to achieve the shortest possible run-times. However, this optimization does not give any value to the current research as the assumption that runtime is sensitive to the order of variables with various domain sizes can be proved by reversing the natural HLDD variable order. The latter is shown in Figure 5-9 .

Thus, the searching algorithm of the CS is sensitive to the variables selection strategy. For some designs the long CS run-times might be cured by selecting different variable selection strategy. If the selected searching heuristics is not efficient for the CSP in hand the other heuristic should be used.

5.2.4.4 Experimental results of CSP solving for multiple cycles

Typically, the CSP of the electronic design that contains memory elements requires solving for multiple cycles to obtain the useful solutions, unless the single cycle solution is masking memory elements. The proposed modeling methodology uses the CS for solving CSP for multiple cycles as explained in Section 5.2.4.2.

The experiments were carried out on the same ITC99 benchmark circuits as in previous section. The results of solving ITC99 benchmarks for multiple cycles are presented in Table 5-12. The cycle limit is set to 10 and parallelism limit is set to 1000. The selected benchmark circuits are listed in the first column. The rest of the table is split into two parts. The first part corresponds to the natural variable selection strategy setting of the CS and the second part is for reversed natural variable selection strategy setting. The “Cycles” column contains the number of solved cycles for reaching the target state from the initial state. Next column (“Found solutions”) shows the number of found solutions. Every solution differs from the others at least by the initial state. Whereas, intermediate states of the same cycle may match between different solutions. The type of initial state is shown in “Initial state” column. The *Reset* and *Combinational* types mean that initial state is deterministic.

Table 5-12 *Details of ITC99 benchmarks solving for multiple cycles*

Design	Natural variable selection				Reversed natural variable selection			
	Cycles	Found solutions	Initial state	Time (ms)	Cycles	Found solutions	Initial state	Time (ms)
b00	2	1000	Reset	5304376	2	1	Reset	12955
b01	5	4	Reset	61	5	4	Reset	24
b02	5	2	Reset	30	5	2	Reset	3
b03	4	1	Combin.	227641	3	1	Combin.	6150
b04	2	1	Reset	26066	1	8	Reset	8065
b06	6	4	Reset	80	6	4	Reset	84
b09	2	1	Reset	6151	2	2	Reset	2332717
b10	2	2	Looped	281099	9	2	Combin.	222600

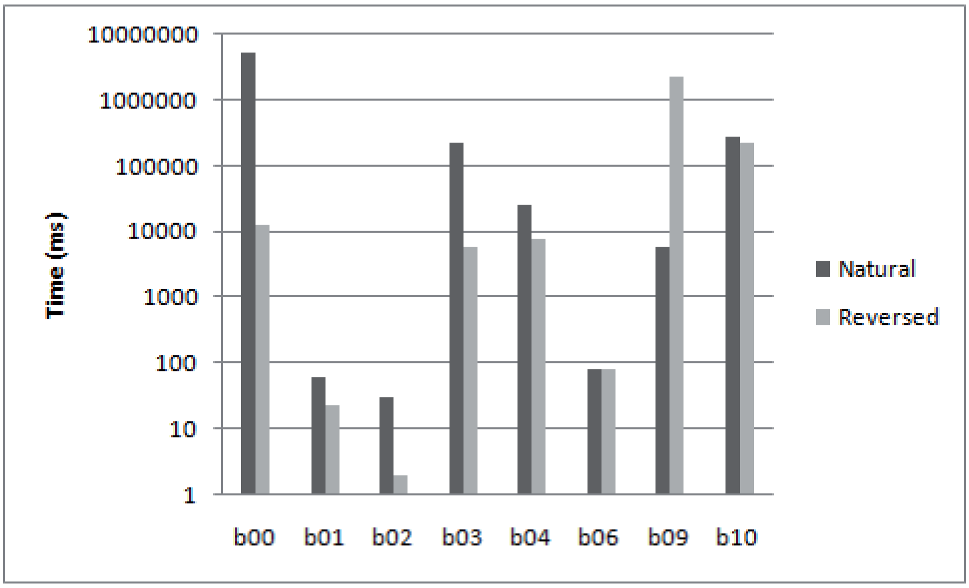


Figure 5-10 CS run-time comparison for different variable selection orders

The *Looped* type means that the search is not able to found the deterministic initial state and is terminated because of finding repetitive states only. Hence, due to the low parallelism limit necessary diverse solutions may not be discovered. This can be solved by increasing the parallelism limit, which also increases the run-time of the CS. Thus, often the trade-off between number of solutions and time should be made. The run-times of CS for producing the results are shown in the *Time* column.

On the basis of the results presented in Table 5-12 the reversed natural variable selection setting outperforms the natural variable selection setting in all cases except the run-time for *b06* and *b09* designs. The run-time comparison is shown in Figure 5-10. The difference in run-times for *b06* design is less than 5%, and as the found solutions are identical, we conclude that the variable selection strategy plays no role in solving this design. On the other hand, the difference in run-times for *b09* design is enormous. The run-time with natural variable selection setting is less than 0.3% of the run-time with reversed natural variable selection setting, whereas the number of found solutions varies only by one.

The results in Figure 5-9 in comparison to the results in Figure 5-10 show that the same designs that are hard-to-solve (*b00, b09*) for certain variable selection strategy remains problematic only for this variable selection strategy in the multi-cycle solving and not for the other one. For all the other designs the reversed natural order of variables becomes a preferable variable selection setting in multi-cycle solving despite the fact that it is worse in single cycle solving (Figure 5-9).

The general approach for selecting appropriate settings for CS is derived on the basis of the presented experiments. The first step of the approach is to find single state solutions using both variable selections strategies. Then the run-times of hard-to-solve cases are compared and the setting that is not causing the extraordinary long run-times is selected. If such hard-to-solve cases are not present then according to Table 5-12 reversed variable selection order is a preferable setting.

5.2.5 Solving of joined CSPs

One of the important properties of the proposed modeling methodology is the ability to combine sub-models of various components into single model. Typically, test data path model is not practical to describe as a single model. The test data path model is divided into smaller models as described in Chapter 4. Thus, the CS must be able to solve the set of joined CSP models.

To study the effectiveness towards that property the system under test is emulated by combining arbitrary ITC99 benchmarks into single model. The results of solving combined ITC99 designs are shown in Table 5-13.

Table 5-13 *Details of solving combined ITC99 benchmarks*

Design links		**Cycles**	**Initial state**	**Time (ms)**
Out	**In**			
b01 [OUTP]	b02 [LINEA]	5	Combin.	190
b02 [U]	b01 [LINE1]	4	Combin.	166
b01 [OUTP]	b06 [EQL]	3	Reset	381
b06 [OVERFLW]	b01 [CONT_EQL]	2	Reset	267
b02 [U]	b06 [EQL]	3	Reset	268
b01 [OUTP] b02 [U]	b02 [LINEA] b06 [EQL]	3	Reset	606
b01 [OUTP] b02 [U]	b06 [CONT_EQL] b06 [EQL]	3	Reset	413
b10 [CTS] b10 [CTR]	b01[LINE1] b01[LINE2]	4	Looped	4091
b10 [CTS] b10 [CTR] b02 [U]	b01[LINE1] b02 [LINEA] b01[LINE2]	3	Reset	2720

The first two columns in Table 5-13 explain the way how designs are connected to each other. For example, the first row shows that output “OUTP” of design b01 is connected to input “LINEA” of b02. Inputs and outputs are selected randomly obeying the rule that the width of the input port should match the width of the output port. Additionally, the random constraints are imposed on the free outputs of the combined designs. The third column explains the reason for stopping the process of constraint solving. The “Reset” means that reset signal is activated, hence the deterministic initial state of the model is found. The “Combin.” means that in the last state variables of memory elements are not constrained, thus this state does not impose any abridge constraints for the next level state, which means that this is a deterministic state. The “Looped” means that only repetitive states are found for the last cycle, consequently initial state is not deterministic and solution is not found.

These ITC99 benchmarks are not supposed to be connected to form the single design. However, the number of solved cycles and solving time shows that even random designs connected together can be solved in reasonable time. If the solution that satisfies imposed constraints is not found, the settings of the CS should be revised as described in previous sections and the initial HLDD model may be supplemented.

5.2.6 Verification of the results

The manual checking of the correctness of the results produced by CS is inefficient even with relatively small benchmarks used in previous experiments. Thus, automatic or semi-automatic way is needed for verification of the results. In case when the description of the modeling object is available in the VHDL the results are checked automatically.

The following approach is proposed to check the experimental results. Typically, a template of the test bench file for the selected design can be generated automatically in the arbitrary CAD system. The sequences of signals produced by CS are automatically converted into VHDL statements and inserted into the VHDL test bench template. Thus, with the help of the proposed approach for test program synthesis the creation of the test bench files can be fully automated.

Let us bring an example of a test bench file for the results from the experiment with *b01* benchmark explained in section 5.2.4.4. These are the abridge constraints that are used in this experiment to describe the target state:

XeqC (OUTP, Constant_0x1);

XeqC (OVERFLW, Constant_0x1);

Table 5-14 *Sequence of signal values reported by CS for design b01*

Input name	t_{-5}	t_{-4}	t_{-3}	t_{-2}	t_{-1}
LINEA1	X	1	1	1	1
LINEA2	X	1	0	0	0
RESET	1	0	0	0	0

The results produced by the CS are shown in Table 5-14 and the remaining details are presented in Table 5-12. The CLOCK signal is not mentioned in Table 5-14 because it is not modeled. Modeling of the CLOCK signal in given case would be redundant, because CS will report two identical sets of values for the period of CLOCK signal (CLOCK = 0 and CLOCK = 1). Thus, in the test bench file the CLOCK variable is triggered in a separate VHDL process with predefined time period (10 nanoseconds in this example).

The values listed in Table 5-14 are converted into a separate VHDL process. Every solving cycle of CS (t_{-1} to t_{-5}) is represented as a set of assignments to the input signals. The time between assignments equals to the period of CLOCK signal. The synthesized part of the testbench file is presented below:

```
-- clock gen process
clock_gen : process
begin
 clock <= '1' after clkhalfper, '0' after 2*clkhalfper;
 wait for 2*clkhalfper;
end process clock_gen;
-- test process
process
begin
      -- t-5
      line1 <= '0';
      line2 <= '0';
      reset <= '1';
      -- t-4
      wait for 2*clkhalfper;
      line1 <= '1';
      line2 <= '1';
      reset <= '0';
      -- t-3
      wait for 2*clkhalfper;
      line1 <= '1';
      line2 <= '0';
      reset <= '0';
      -- t-2
      wait for 2*clkhalfper;
      line1 <= '1';
      line2 <= '0';
      reset <= '0';
      -- t-1
      wait for 2*clkhalfper;
      line1 <= '1';
      line2 <= '0';
      reset <= '0';
      wait;  -- suspend process
end process;
```

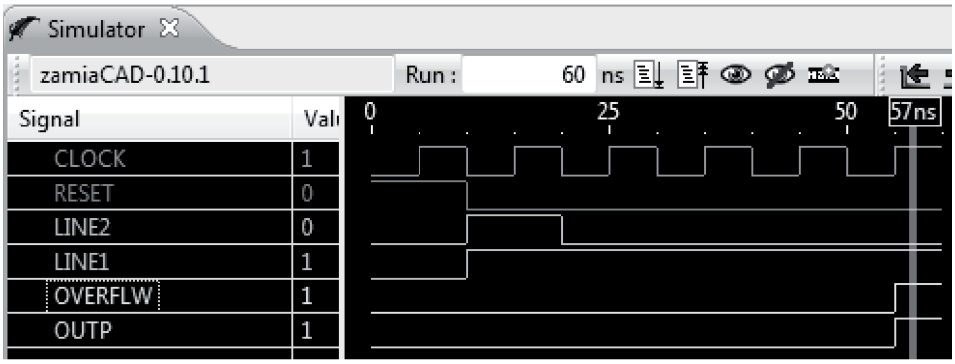


Figure 5-11 ZamiaCAD simulator output for b01 test bench

The results of the simulation of the test bench for b01 design are presented in Figure 5-11. The open source platform for advanced hardware design ZamiaCAD [60] has been used for simulation and for the waveform generation. It can be seen from the waveform that OVERFLW and OUTP signals indeed have the correct values on the 6th clock cycle (time stamp t_0). Hence, the target state of the system is reached.

The synthesized test bench besides checking the results of the CS facilitates the debugging of VHDL sources of the design in hand. Moreover, it is often necessary for debugging purposes to trace the internal values of the design during the virtual test application. The efficiency and depth of the debugging mostly depends on the functionality provided by the CAD software. In ZamiaCAD the *annotate* feature was used to trace the values of internal variables in any moment of simulated time.

5.3 Case study

This section presents the case study that summarizes the presented methodology on modeling the test data path for automated test program synthesis.

The functionality that is necessary to include in every test data path model is a TAP. The typical TAP contains a TAP controller state machine, an instruction register and a set of data registers. Instruction and data registers are scan-registers. Thus, these are the basic components of arbitrary TAP model. In Section 4.2.2 is described the creation of the HLDD model of the simplified TAP control and data path (Figure 4-5). The respective models are shown in Figure 4-6. These models are reused in the current section to describe step by step the test program synthesis methodology starting from the test pattern and finishing on the SVF [61] (Serial Vector Format) instruction.

The test pattern in board level test is typically a pair of address and data to be written to this address. Test address is the value to be applied to the address pins of the board component (e.g. memory). The test data is a value for the data pins of board component under test. The component under test most certainly has control pins that should also be driven during the test data application. Generally, the values for control pins are not included in the test pattern and have to be obtained during the development of the test program. In case of PCBT these control signals are handled by the respective peripheral controller inside the microprocessor SoC. Thus, only the test address, test data, debug instructions and microprocessor instructions have to be passed to the SoC through the TAP. In this case study the SVF instruction for transporting the pattern through the TAP port will be synthesized.

The CSP model of the simplified control and data path for JTAG TAP that is shown in Figure 4-5 has to be complemented by the abridge constraints. The first source of the abridge constraints is the test pattern itself. The second source is the structural description of the model. These constraints define for the CS the target state of the system to reach.

In this case study let the arbitrary test pattern be a pair of test address (0xA0000004) and test data (0x5A5A6B6B). These are the values that have to be sent through TAP *Data_register* to the debug port logic (*External logic*). From the CS point of view the address and data are the two cases to solve. Firstly, the address value should be transmitted, and then the CSP should be solved again to transfer the test data value. Besides that, the value that may come from external logic must be defined. Otherwise, the CS may suggest taking the values of the test pattern from the external logic to pass them back to the external logic, instead of shifting them through the TDI pin. Taking values form *External logic* makes no sense as the task is to do the opposite. For that the structural model is analyzed to impose the additional abridge constraint that restricts the value of external logic to known reset value. Let this reset value be 0x00000000.

These are the abridge constraints for the address transfer:

XeqC (Data_register, Constant_0xA0000004);

XeqC (External_logic, Constant_0x00000000);

The abridge constraints for the test data transfer:

XeqC (Data_register, Constant_0x5A5A6B6B);

XeqC (External_logic, Constant_0x00000000).

These constraints are passed to the CS together with the CSP. For the first set of abridge constraints the CS returns the following sequence of input signals:

The sequence of signals for TAP returned by CS for the set of constraints for address transfer is presented in Table 5-15.

Table 5-15 *Sequence of signals for TAP pins for application of address value*

Signal name	Sequence of signals
TAP state	x012344444444444444444444444444444444458
TMS	x01000000000000000000000000000000000011x
TDI	xxxxx10100000000000000000000000000100xx
TRST	10000000000000000000000000000000000000x

The ‘x’ stands for *don’t care* value, thus it can be substituted by any value from the domain of the variable.

The sequence of signals for TAP returned by CS for the second set is presented in Table 5-16.

Table 5-16 *Sequence of signals for TAP pins for application of test data value*

Signal name	Sequence of signals
TAP state	x012344444444444444444444444444444444458
TMS	x01000000000000000000000000000000000011x
TDI	xxxxx01011010010110100110101101101011xx
TRST	10000000000000000000000000000000000000x

The results presented in Table 5-15 and Table 5-16 for TMS and TRST inputs are the same. The values for TDI vary as the data to be shifted in is different. The respective SVF instructions for the data in Table 5-15:

```
TRST ON;
TRST OFF;
SDR 32 TDI (A0000004);
```

and for the data in Table 5-16:

```
TRST ON;
TRST OFF;
SDR 32 TDI (5A5A6B6B);
```

These SVF instructions show that solving of two consecutive data transfer as independent cases causes redundant SVF instructions (TRST ON; and TRST OFF;). In order to eliminate redundant SVF instructions, the second case should be supplemented by the results of the previous case. The last state of the previous state is defined as the first state for the next case. Thus, the CS will be informed of the initial state of the system and will not search for the deterministic initial state, which is a reset state in the example for the test data transferring. This will modify the sequence of signals for the TAP pins in the following way:

Table 5-17 *Modified sequence of signals for TAP pins for application of test data value*

Signal name	**Sequence of signals**
TAP state	82344444444444444444444444444444458
TMS	1000000000000000000000000000000000011x
TDI	xxx01011010010110100110101101101011xx
TRST	000000000000000000000000000000000000x

The test program for applying single test pattern (address and data) is presented below:

TRST ON;
TRST OFF;
SDR 32 TDI (A0000004);
SDR 32 TDI (5A5A6B6B);

As it can be noticed from this case study one of the purpose of presenting the results as SVF instructions is to make the test program readable. Moreover the SVF test programs are executable by BS test systems. The additional benefit of presenting a test program in SVF is possibility to add constructions like TDO and MASK that enable debugging of the test program and the system itself.

5.4 Chapter summary

The first contribution of this chapter is the analytical study that reveals benefits of automated generation of entire PCBT program. A novelty of this research is the proposed methodology for test development automation based on partial functional SUT model. The proposed method for automatic test program synthesis allows significantly speeding up the development of test program, which is a considerable contribution towards reducing time-to-market in the PCB industry.

The next contribution of this chapter is a novel methodology to present the partial functional model as a constraint satisfaction problem (CSP). The transformation of the model into CSP is automated. The proposed CSP model is based on Java Constraint Programming (JaCoP) framework. The functionality of the constraint solver provided by JaCoP framework (CS core, implementation of search algorithm, backtracking engine) is extended to operate with multiple joined CSP designs and to produce results for reaching the target state over many clock cycles. The former enhances scalability in handling big industrial problems by modeling them as a set of smaller related problems.

On the basis of the synthesized “raw” test program the approach for automatic VHDL test bench synthesis is described. The automatic synthesis of test bench files facilitates the testing of the test program and contributes to the debugging of the VHDL source code of the design under test. The test program translated into the SVF instructions is executable by many available test systems. The synthesized test program may be also used in debugging of the test setup and SUT, besides the test access and test application.

This chapter also reports the details of experiments with ITC99 benchmarks. These experiments prove the feasibility of proposed methodology and are used to study the influence of various CS settings (variables selection strategy and variable assignment strategy) on the CS run-time and synthesized test program. The general strategy is developed for selecting the CS settings in order to obtain the acceptable result in a reasonable time period. The chapter is concluded by the case study that summarizes the proposed methodology on the example of test program synthesis for the test pattern transportation through the JTAG TAP.

Chapter 6
TOOLCHAIN FOR PCBT DEVELOPMENT AUTOMATION

In this chapter the proposed and developed toolchain for PCBT development automation is presented. The goal of this chapter is to draw an overview picture that connects the proposed methodology, developed tools and existing test environment. Firstly, the workflow is discussed and source data relations are explained on the data flow diagram. Then the general view on the developed and reused tools is presented. The chapter is concluded with possible applications of the developed tools and integration with the existing third-party tools and frameworks.

6.1 PCBT development automation workflow

The PCBT development automation workflow proposed in this chapter has many use case scenarios. The data flow diagram in Figure 6-1 combines possible data flows from various scenarios. The fourth layer in Figure 6-1 contains the end-points for all scenarios that determine the target applications for the obtained results. The CAD software may be used for:

- Verification of the obtained input data for primary inputs
- Debugging of the VHDL source code of the PCBA component using the test bench generated from obtained input data
- Simulation of the test application
- Estimation of the test run-time

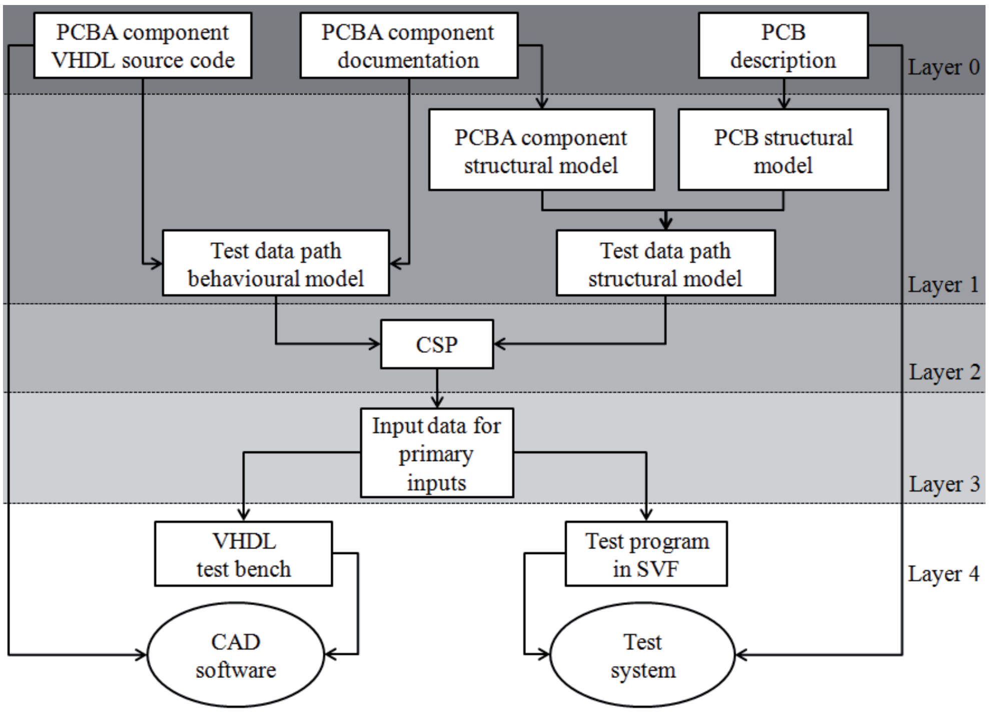

PCBA component VHDL source code
PCBA component documentation
PCB description
Layer 0
PCBA component structural model
PCB structural model
Test data path behavioural model
Test data path structural model
Layer 1
CSP
Layer 2
Input data for primary inputs
Layer 3
VHDL test bench
Test program in SVF
Layer 4
CAD software
Test system

Figure 6-1 Data flow diagram and transformation layers

In case if VHDL source code of the PCBA components is not available the above mentioned tasks cannot be fulfilled. However, if the CSP model does not include the structural model of the test data path, the produced input data can still be used for the CAD software related task.

The other end-point of the data flow diagram is the test system. In the frame of this work the test system is considered as boundary-scan test software (capable to interpret the test program in SVF) and boundary-scan test hardware that is connected to the PCBA under test. The inputs for the test system are PCB description and test program in SVF. The test program properties and objectives have been discussed in details in Chapter 3. The test system may be used for different purposes, depending on the objective of the test program:

- Test interconnections between µP SoC and another PCBA component
- Load program into on-board or on-chip memory (µP SoC internal memory)
- Test PCBA component using µP as on-board tester (e.g. in the field)
- Debug the execution of the application running on the µP
- Debug PCBA (NTF problem)

The layers in Figure 6-1 split the data flow into 5 parts. The arrows that cross the border between layers mean not just data passing, but transformation of data. “Layer 0” contains the data that is coming from third-party sources. The VHDL source code and the documentation of PCBA components are obtained from the vendor of the component. The PCBA description is a documentation of the PCBA that includes a list of components with interconnection information that typically comes as a netlist file.

The first set of transformations is performed when crossing the border between “Layer 0” and “Layer 1”. The latter layer contains the behavioral and structural model of the test data path. The behavioral model can be automatically converted from the VHDL source codes of the PCBA component (VHDL to HLDD converter) or to be created manually with the help of the documentation. The documentation is also needed to create the structural model of the selected PCBA component. The PCBA structural model is created automatically from board description (Board netlist parser).

The behavioral and structural models are transformed into CSP (HLDD to CSP converter and structural constraints extractor), which lies solely on the second layer (Layer 2). The CSP is supposed to be solved by the CS (CSP solver) in order to obtain the input data for primary inputs of the uniform model. The input data for primary inputs corresponds already to the “Layer 3” and is used in converter from raw TAP signals to SVF instructions and for automatic test bench synthesis. The test program and the test bench are located on the last layer and can be used in test system and CAD software respectively.

The important issue that has not been mentioned in Figure 6-1 is the source of the test patterns. The test patterns may come from different sources. First, the test system itself may be able to generate the test patterns. Second, the test patterns may be imported from external automated test pattern generator (ATPG) or taken from the pre-generated library. The test generation is out of scope of given research, thus, for the sake of readability the test patterns source and their origin is not included in the data flow diagram in Figure 6-1.

6.2 Toolchain and integration

The functionality that was developed within the framework of this research is shown in Figure 6-2. Parsers, converters and solver can be run individually or can be accessed via the common plugin for Eclipse Integrated Development Environment (IDE). Eclipse is a universal tool platform, an open extensible IDE for “anything and nothing in particular”.

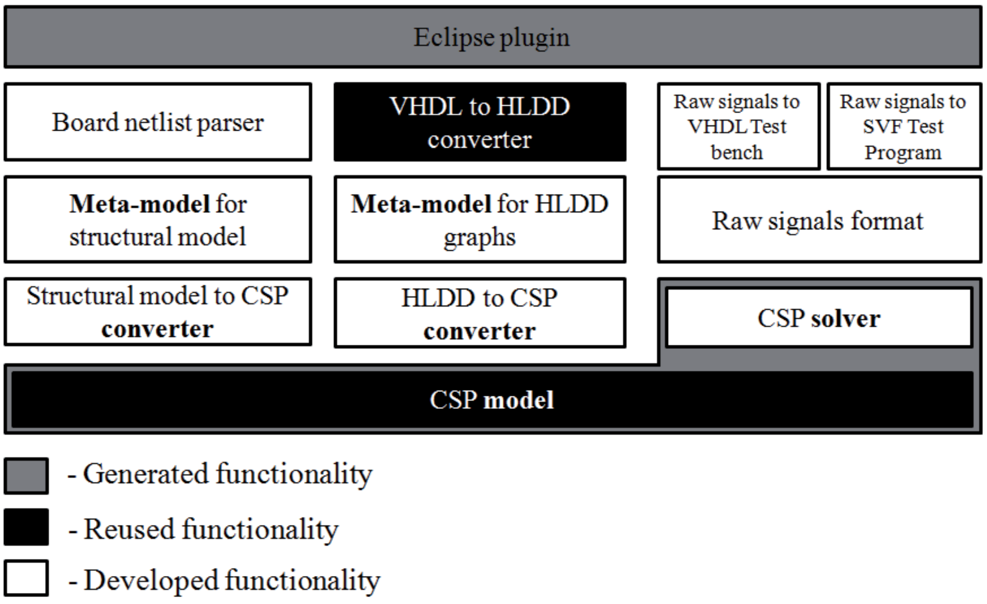

Eclipse plugin
Board netlist parser
VHDL to HLDD converter
Raw signals to VHDL Test bench
Raw signals to SVF Test Program
Meta-model for structural model
Meta-model for HLDD graphs
Raw signals format
Structural model to CSP converter
HLDD to CSP converter
CSP solver
CSP model
- Generated functionality
- Reused functionality
- Developed functionality

Figure 6-2 Plugin modules and abstraction layers

Eclipse has the considerable support of the leading companies and organizations in the technology sector. Eclipse is gaining widespread acceptance in both the commercial and academic arenas [50].

Eclipse plugin facilitates the smooth integration of developed functionality into widely used open source IDE for many applications. The developed toolchain for automation of PCBT program development can be easily integrated with previously mentioned CAD software ZamiaCAD as plugin installation.

The toolchain depicted in Figure 6-2 consists of blocks of three different types (generated, reused and developed). The block of generated type (Eclipse plugin) corresponds to the graphical user interface (GUI) functionality that was automatically generated in the Eclipse Modeling Framework (EMF) [50]. EMF was also used for developing the metamodels for PCBA structural model and for PCBA behavioral model. The behavioral model is based on mathematical foundations of HLDD graphs. The developed plugin is the first GUI for creating and editing of HLDD graphs. The “VHDL to HLDD converter” was added to the toolchain for automatic behavioral model creation from VHDL source code (RT-Level) of PCBA components. The “Board netlist parser” was developed to automatically obtain the PCBA structural model from board netlist file. This model has to be complemented with structural models of PCBA components created manually in Eclipse GUI.

The above mentioned functionality forms a bridge between GUI level and an underlying level with metamodels. The tools on the next level transform the structural and behavioral models of PCBA to the CSP model. The CSP model solver plays a role of backward link to the metamodel layer by producing the results in raw format of solving the CSP model. The implementation of CSP model belongs to the reused functionality. It is based on the JaCoP framework [55] library. The core of the CSP solver also uses a search engine and a backtracking functionality provided by JaCoP framework library.

The modules in the developed toolchain may be swapped or added without a significant impact on the other modules. For example, the benefits of implementing behavioral model as HLDD graphs may be assessed by adding other implementation of behavioral models (metamodel, CSP converter) for comparison experiment without introducing changes to the rest of toolchain modules. Moreover, the CS itself can be substituted by more efficient one, just by exchanging the CSP model and CSP solver modules. Thus, in both cases only neighboring modules are affected while the rest of the toolchain remains untouched.

The developed functionality and corresponding toolchain combined with CAD software and boundary scan enabled test system in addition to the practical applications also form the research environment for studying the field of board level test and debug.

6.3 Chapter summary

The toolchain for PCBT development automation is presented in this chapter. The various data structures and formats that were described previously in this thesis are shown in a layered data flow diagram (Figure 6-1) to outline their purpose and place in the final toolchain.

The integration possibilities into third party Test system and CAD software are highlighted in this chapter. The main advantage of the developed toolchain is that it can be used as a plugin to Eclipse IDE as well as a standalone Eclipse-based application.

The developed functionality is included into the toolchain and the hierarchy between enclosed tools is depicted in Figure 6-2. The modularity and the correspondence to different abstraction layers facilitates the substitution of modules without changing the rest of the functionality, that leaves the space for further research in the field of board level test automation using the developed toolchain.

Chapter 7
CONCLUSIONS AND FUTURE WORK

The aim of this thesis is to propose a novel methodology to model the test data path with the goal of automatic synthesis of board level processor-centric test program. The proposed modeling approach reuses the theory of high-level decision diagrams as a basis for behavioral modeling of the test data path and presents a new model to describe the structure of the PCBA and its components. The automation of the processor-centric board test program development is based on solving the constrained test data path model of the PCBA under test.

This chapter summarizes the thesis, brings together the contributions of this research and points out promising directions for future work.

7.1 Conclusions

This thesis presents a new approach for automatic synthesis of PCBT program that is executable on ATE and uses on-board µP as the central component of test access and application mechanism. Structural and behavioral models are automatically transformed into Constraint Satisfaction Problem (CSP), which is passed to the search algorithm provided by Java Constraint Programming (JaCoP) [55] framework to solve the task of test pattern transportation from board under test TAP pins to UUT pins.

The feasibility of the proposed methodology was proven by the presented experimental results. ITC99 benchmarks and models of various µP SoC modules have been used in conducted experiments.

7.1.1 Results and contributions

A comprehensive analytical study has been carried out that revealed the benefits of automated generation of processor-centric test programs and motivated the research presented in the thesis.

The new results and contributions of the presented work are summarized as follows:

- Formulae for test application time calculation - The simulation-free calculation of the test application time is useful for fast cost estimation of the manufacturing board test solution. Moreover, these formulae are helpful for comparison of different test application strategies for a given test case.
- The metamodel and its implementation for structural models of PCBA, SoCs and other PCBA components
- The metamodel and its implementation for behavioral models of ICs
- A novel methodology for test data path modeling - Structural models are augmented by behavioral models to assemble the uniform model of the PCBA that is used in test development automation.
- A novel approach and implementation of automated synthesis of PCBT program in SVF
- A new approach for automated synthesis of the VHDL test bench

7.1.2 Advantages

The proposed toolchain of developed and reused programs corresponds to the platform with broad research capabilities in the field of board level test. It also provides reach integration possibilities with third-party tools for test and debug of assembled PCBs.

The modules of the toolchain such as CSP model or underlying metamodels can be substituted or supplemented with minor changes to the rest of the framework. This may help to assess the proposed approach and to speed up the automatic synthesis of PCBT programs in the future.

7.2 Future work

This section outlines the most important issues that require further investigation for improving the proposed techniques.

The research presented in this thesis was primarily targeting the PCBT program development automation. The proof of concept implementation of the toolchain is created to carry out the feasibility study of the proposed approach. Hence, the performance related issues of the used tools are not considered in the first place. Thus, to assess the efficiency of JaCoP framework towards other CS packages the developed toolchain should be extended with several other CS techniques for fair comparison.

The extension of the library of models should be continued. This may assist to achieve the wide adoption of proposed methodology. The models of SoC components from major IP vendors have to be considered first, while on-board memories and interfaces are the next targets.

The model development methodology could be complemented towards supporting the import and export of IP-XACT compliant models. This will conduce to reusing of developed models in other applications. Moreover, the effort for model development could be reduced by importing the IP-XACT compliant model from third party applications.

The integration opportunities of the developed toolchain into existing test systems and CAD software should be investigated further to find more common standpoints. It would reveal the demands and problems that could be solved by integrating the whole toolchain or individual modules into third-party tools.

References

[1] **iNEMI, International Electronics Manufacturing Initiative.** Research Priorities 2011. [Online] 2011. [Cited: 7 12 2011.] http://www.inemi.org/node/2135.

[2] IEEE Standard Test Access Port and Boundary-Scan Architecture. 2001. IEEE Std. 1149.1-2001.

[3] **P. B. Geiger, S. Butkovich.** Boundary-Scan Adoption – An Industry Snapshot with Emphasis on the Semiconductor Industry. – *Proc. of International Test Conference*, Austin, Texas USA, 2009, pp. 1-10.

[4] **P. Maxwell, I. Hartanto, L. Bentz .** Comparing Functional and Structural Tests. – *Proc. of International Test Conference*, Atlantic City, NJ , USA, 2000, pp. 403 - 407.

[5] **A. Jutman.** At-speed on-chip diagnosis of board-level interconnect faults. – *Proc. of 9th IEEE European Test Symposium*, France, 2004, pp. 2-7.

[6] **S. Mourad, Y. Zorian.** *Principles of testing electronic systems.* New York : A Wiley-Interscience Publication, 2000.

[7] **K.M. Butler, J.M. Carulli, J. Saxena.** Modeling Test Escape Rate as a Function of Multiple Coverages. – *Proc. of International Test Conference*, Santa Clara, CA, USA, 2008, pp. 1 - 9.

[8] **S. Davidson.** Towards an understanding of no trouble found devices. – *Proc. of VLSI Test Symposium*, Palm Springs, California, USA, 2005, pp. 147-152.

[9] **S. Davidson.** Understanding NTF component from the field. – *Proc. of International Test Conference*, Austin, TX, USA, 2005, pp. 332-342.

[10] **J. Webster, B. Fenton, D. Stringer, B. Bennetts.** On the synergy of boundary scan and emulation board test: a case study. – *Proc. of Board Test Workshop*, Charlotte, USA, 2003, p. 10.

[11] **T. Wenzel, H. Ehrenberg.** Combining Boundary Scan and JTAG Emulation for Advanced Structural Test and Diagnostics: White Paper. [Online] 2009. p.9. http://tmworld.resourcecenteronline.com.

[12] **M. Daud.** PC Maintenance And Troubleshooting Expert Systems. – *Proc. Of Int. Conf. On Robotics, Vision And Parallel Processing For Automation*, 1999, pp. 528 - 534.

[13] **S. Oresjo.** A New Test Strategy for Complex Printed Circuit. – *Proc. of Nepcon West 1999*, 1999.

[14] **J. Kirschling.** Improved Fault Coverage in a Combined X-ray and In-circuit Test Environment. – *Proc. of EtoniX 2001*, 2001.

[15] **J.H. Shim, H.S. Cho, S. Kim.** A new probing system for the in-circuit test of a PCB. – *Proc. of International Conference on Robotics and Automation* , Minneapolis, MN, USA, 1996, pp. 590 - 595.

[16] **J.K. Berger.** New directions in loaded board testing. – *Proc. of Automatic Testing Conference - AUTOTESTCON*, Philadelphia, PA , USA, 1989, pp. 212 - 216.

[17] **R. Ubar.** Alternative Graphs and Test Generation for Digital Systems. – *Proc. of 2nd Conf. On Fault Tolerant Systems and Diagnostics*, Brno, Czechoslovakia, 1979, pp. 177-184.

[18] **Stephen F. Scheiber.** *Biolding a Successful Board-Test Strategy.* Woburn, MA : Butterworth-Heinemann, 2001. p. 83.

[19] **Nadeau-Dostie.** An embedded technique for at-speed interconnect testing. – *Proc. of Int. Test Conf.*, Atlantic City, USA, 1999, pp. 431 - 438.

[20] **Kozio®.** *One Button Test Strategy for Volume Manufacturing.* Longmont, CO : Kozio, Inc. White Paper.

[21] **M. Reagin, S.Yang.** Test, Inspection and Measurement.
[Online] 21 04 2009. [Cited: 12 12 2011.]
http://thor.inemi.org/webdownload/newsroom/Presentations/SMTA_China_Apr09/TIM.pdf.

[22] IEEE Standard for a Mixed-Signal Test Bus. 1999. IEEE Std 1149.4-1999.

[23] IEEE Standard for Boundary-Scan Testing of Advanced Digital Networks. 2003. IEEE Std 1149.6-2003.

[24] **S.K. Lim.** Physical design for 3D system on package. IEEE Design & Test of Computers, 2005, Vol. 22, pp. 532 - 539.

[25] **F.P. Carson, Young Cheol Kim, In Sang Yoon.** 3-D Stacked Package Technology and Trends. – *Proc. of IEEE*, Vol. 97, 2009, pp. 31 - 42.

[26] IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture. 2010. IEEE Std 1149.7™-2009.

[27] **K. Parker, J. Burgess.** What is IEEE P1149.8.1 and why? – *Proc. of International Test Conference 2009. ITC 2009*, Austin, TX, USA, p. 1.

[28] **K.P. Parker.** The Boundary-Scan Handbook. Boston, MA, USA : Kluwer Academic Publisher, 2003. p. 373.

[29] **D. Gizopoulos, A. Paschalis, Y. Zorian.** *Embedded Processor-Based Self-Test.* Boston : Kluwer Academic Publisher, 2004. pp. 81 -156. Vol. 28.

[30] **A. Apostolakis, M. Psarakis, D. Gizopoulos, A. Paschalis.** Functional Processor-Based Testing of Communication Peripherals in Systems-on-Chip. – *IEEE Trans. on VLSI*, 2007, pp. 971 - 975.

[31] **J.-R. Huang, M. K. Iyer, K.-T. Cheng.** A self-test methodology for IP cores in bus-based programmable SoCs. – *Proc. of IEEE VLSI Test Symposium 2001*, 2001, pp. 198 - 203.

[32] **K. Jayaraman, V. M. Vedula, J. A. Abraham.** Native mode functional self-test generation for systems-on-Chip. – *Proc. of International Symposium for Quality Electronic Design*, 2002, pp. 280 - 285.

[33] **P. Bernardi, M. Grosso, M. Rebaudengo, M. Sonza Reorda.** Exploiting an infrastructure-intellectual property for systems-on-chip test, diagnosis and silicon debug. – *IET Computers & Digital Techniques*, Vol. 4, Issue 2, 2010, pp. 104 - 113.

[34] **D. S. Morris.** In-circuit, functional or emulation - choosing the rigth test solution. June 1986, The IEE Computer-Aided Engineering Journal, Vol. 3, pp. 94 - 101.

[35] **Nexus 5001 Forum.** The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface. 2003. EEE-ISTO 5001™-2003.

[36] **B. Vermeulen, N. Stollon, R. Kuhnis, G. Swoboda, J. Rearick.** Overview of Debug Standardization Activities. IEEE Design & Test of Computers , 2008, Vol. 25.

[37] **MIPI Test and Debug Working Group.** *MIPI Test and Debug Interface Framework.* 2006. v3.2, White Paper.

[38] **MIPI Test and Debug Working Group**. *MIPI Alliance Test and Debug - NIDnT-Port.* 2007. v1.0, White Paper.

[39] **S. Devadze, A. Jutman, A. Tsertov, M. Instenberg, R. Ubar.** Microprocessor-based system test using debug interface. – *Proc. of 26th IEEE NORCHIP Conference*, Tallinn, Estonia, 2008, pp. 98-101.

[40] *MIPS® EJTAG Specification.* MD00047, 1225 Charleston Road, CA, USA : MIPS Technologies Inc, November 2008.

[41] *ARM9EJ-S Technical reference manual.* ARM DDI 0222B, : ARM Limited, 2002.

[42] *ARM Architecture reference manual.* ARM DDI 0100I, : ARM Limited, 2005.

[43] **S. Zeidler, C.Wolf, M. Krsti´c, F. Vater, R. Kraemer.** Design of a Test Processor for Asynchronous Chip Test. – *Proc. of 20th Asian Test Symposium (ATS'11)*, New Delhi, India, 2011, pp. 244 - 250.

[44] **S. Ostendorff, H.-D. Wuttke, J. Sachße, S. Köhler.** A new Approach for Adaptive Failure Diagnostics Based on Emulation Test. – *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010* , Dresden, Germany, 2010, pp. 327 - 330.

[45] **A. Tsertov, R. Ubar, A. Jutman, S. Devadze,.** Automatic SoC Level Test Path Synthesis Based on Partial Functional Models. – *Proc. of 20th Asian Test Symposium (ATS'11)*, New Delhi, India, 2011, pp. 532 - 538.

[46] **D. Gajski, L. Cai.** Transaction Level Modeling: An Overview. – *Hardware/Software Codesign and System Synthesis*, Newport Beach, California, USA, 2003, pp. 19 - 24.

[47] IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows. 2010. IEEE std. 1685-2009.

[48] *UML Profile for MARTE: Modeling and Analysis of Real Time Embedded Systems.* formal/2009-11-02, : http://www.omg.org/spec/MARTE/1.0, Object Management Group (OMG), November 2009.

[49] **S. Demathieu, F. Thomas, C. André, S. Gérard, F. Terrier.** First experiments using the UML profile for MARTE. – *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, Orlando, Florida, USA, 2008.

[50] **E. Gamma, L. Nackman, J. Wiegand.** *EMF: Eclipse Modeling Framework.* Second Edition. Addison-Wesley Professional, 12.2008.

[51] **T. Grötker, S. Liao, G. Martin, S. Swan.** *System Design with SystemC.* New York, USA : Kluwer Academic Publishers, 2002.

[52] **A. Tsertov, A. Jutman, S. Devadze.** Testing Beyond the SoC in a Lego Style. – *Proc. of East-West Design & Test Symposium*, St. Petersburg, Russia, 2010, pp. 334 - 338.

[53] **A. Chepurov, G. D. Guglielmo, F. Fummi, G. Pravadelli, J. Raik, R. Ubar, T. Viilukas.** Automatic generation of EFSMs and HLDDs for functional ATPG. – *Proc. of 11th International Biennial Baltic Electronics Conference*, Tallinn, Estonia, 2008, pp. 143-146.

[54] **R. Ubar, A. Morawiec, J. Raik.** Back-Tracing and Event-Driven Techniques in High-Level Simulation with Decision Diagrams. – *Proc. of the IEEE ISCAS2000 Conference*, Vol. Vol. 1, Geneva , Switzerland, 2000, pp. 208-211.

[55] **K. Kuchcinski, R. Szymanek.** *JaCoP Library User Guide. Version 3.0.* November 2010.

[56] **F. Rossi, P. van Beek and T. Walsh, [ed.].** *Handbook of Constraint Programming.* Elsevier, 2006.

[57] **G. H. Mealy.** A Method to Synthesizing Sequential Circuits. Bell Systems Technical Journal, 1955, pp. 1045 - 1049.

[58] **G. D. Guglielmo, F. Fummi, C. Marconcini, G. Pravadelli.** Improving high-level and gate-level testing with FATE: A functional automatic test pattern generator traversing unstabilised extended FSM. – *Computers & Digital Techniques, IET*, 2007.

[59] **F. Corno, M. S. Reorda, G. Squillero.** RT-level ITC'99 benchmarks and first ATPG results. Design & Test of Computers, IEEE, 2000, Vol. 17, pp. 44 - 53.

[60] ZamiaCAD. [Online] 2011. [Cited: 1 11 2011.] http://zamiacad.sourceforge.net/web/.

[61] Serial vector format specification. [Online] 1999. [Cited: 30 11 2011.] ASSET InterTech, Inc. http://www.assetintertech.com/support/svf.pdf.

[62] **H. Fang, Z. Wang, X. Gu, K. Chakrabarty.** Ranking of suspect faulty blocks using dataflow analysis and dempster-shafer theory for the diagnosis of board-level functional failures. – *Proc. of 16th European Test Symposium*, Trondheim, Norway, 2011, pp. 195-200.

[63] **S. Devadze, A. Jutman, A. Tsertov, R. Ubar.** Microprocessor modeling for board level test access automation. – *Proc. of 10th IEEE Workshop on RTL and High Level Testing*, Hong Kong SAR, China, 2009, pp. 154–159.

[64] **C.H.-P. Wen, L.-C. Wang, Kwang-Ting Cheng, Kai Yang, Wei-Ting Liu, Ji-Jan Chen.** On A Software-Based Self-Test Methodology and Its Application. – *Proc. of VLSI Test Symposium*, 2005, pp. 107 - 113.

Appendix A
AGM FORMAT

AGM format is described in this appendix. This format was proposed in Tallinn University of Technology to describe the design at RT-Level and behavioral abstraction level (HLDD).

This format is not a contribution of this thesis, but rather presented here for explanatory purposes. AGM stands for Alternative Graph Model. The origin of this abbreviation is in the first publications of Prof. Raimund Ubar on topic of decision diagrams, where they were referred to as alternative graphs (e.g. [17]).

AGM format is case sensitive. It is a line-based format where maximum line length can be 256 characters. In the following the BNF syntax of HLDD model format is presented. The meta-syntax used obeys the following rules:

1. Syntactic categories (non-terminals) are printed in italics; literal words, characters and constants are enclosed to 'quotes'.
2. If a construct is enclosed to [square brackets], it is optional.
3. If a construct is enclosed to {curly brackets}, it may be repeated zero or more times.
4. A choice is indicated with a vertical bar |.
5. If a construct is enclosed in <chevrons>, it can occur at most once.

AGM Syntax

AGM :=

statistics

mode

[control_signals]

hldd_description

statistics :=

‘STAT#*’ natural ‘Nods,’ natural ‘Vars,’ natural ‘Grps,’ natural ‘Inps,’ natural ‘Outs,’ natural ‘Cons’ [‘,’ natural ’Funs’] [‘,’ natural ’Mems’] [‘,’ natural‘C_outs’]*

The natural values reflect the number of nodes, variables, graphs, inputs, outputs, constants, functions, memory arrays and control part outputs, respectively. The number of functions and memory arrays are meaningful in the high-level descriptions. The number of control part outputs is used with the RTL descriptions divided into a control part and a datapath only.

control_signals :=

‘COUT#’ *natural {‘,’ natural}*

Shows the variable indexes of control signal variables. Used in RTL descriptions partitioned to datapath and control parts.

mode :=

‘MODE#’ ‘RTL’ | ‘BEHAVIORAL’

Indicates whether an RTL model, or a behavioral model is being described.

hldd_description :=

[{input_definition}]

[{memory_definition}]

[{constant_definition}]

[{function_definition}]

[{control_definition}]

{graph_variable_definition}

The definitions are ranged according to the order shown above. *control_definitions* are used only in the RTL descriptions partitioned into control and datapath parts.

input_definition :=

'VAR#*' var_index ':' '(' variable_flags ')' var_name var_range*

Defines a primary input of the model.

memory_definition :=

'VAR#' *var_index ':' '('variable_flags')' var_name var_range [row_range]*

column_range

memory_row

{memory_row}

Defines a memory array. The optional *row_range* is used with two-dimensional arrays, and it determines the range of row addresses used in memory. In one-dimensional arrays, *row_range* is omitted. In similar way, *column_range* determines the range of column addresses used in the memory variable.

memory_row :=

'{' integer {',' integer} '}'

Defines the contents of a memory variable. The number of integers in memory_row is determined by column_range.

row_range := mem_range

Row_range is used with two-dimensional arrays, and it determines the range of row addresses used in memory. In one-dimensional arrays, row_range is omitted.

column_range := mem_range

Determines the range of column addresses used in the memory variable.

mem_range := '[' integer '-' integer ']'

In *mem_range* the first integer must be less than the second one.

constant_definition :=

'VAR#' *var_index ':' '('variable_flags')' var_name var_range 'VAL' '='integer*

Defines a constant. The integer value shows the value of the constant.

function_definition :=

'VAR#' var_index ':' '('variable_flags')' var_name var_range

'FUN#' function_type arguments_definition

Defines an operation or function.

function_type := identifier

Shows the type of the operation.

arguments_definition :=

'(' [argument] {',' argument} ')'

Defines the arguments (if any) of an operation.

argument :=

'A'argument_index '<=' argument_variable range

The *range* shows the bit-slice of the variable *argument_variable* that is used as afunction argument.

argument_index := natural

Shows the index of the function argument.

argument_variable := natural

Shows the index of the variable used as the function argument.

control_definition :=

'VAR#' var_index ':' '(' variable_flags ')' var_name var_range

Defines a control signal. Used to define control part output signals of the RTL designs partitioned into datapath and control parts.

graph_variable_definition :=

'VAR#' *var_index ':' '('variable_flags')' var_name var_range*

graph_definition

Defines a variable for which a graph corresponds.

graph_definition :=

‘GRP#’ graph*_index ‘:’ ‘*BEG’ *‘=’ natural ‘,’ ‘*LEN’ *‘=’ natural ‘’*

node_definition | *parallel_node_definition*

{node_definition | *parallel_node_definition}*

Defines a graph in the HLDD model. The ‘BEG=’ construct shows the absolute index of the first node in the graph. The ‘LEN=’ construct in turn shows the number of nodes in the graph.

node_definition :=

nod_abs_index nod_index ‘: (‘nod_flags’) (’ successors ‘) V *=’ nod_var*

nod_name nod_range

Defines an HLDD node. *nod_abs_index* and *nod_index* represent the absolute (inside the model) and relative (inside the graph) indexes of the node. Construct successors shows the successor nodes of current node which are chosen with different node values. Index of the variable labelling the node is determined with *nod_var*.

parallel_node_definition :=

nod_abs_index nod_index ‘: (v___)’ ‘(‘ ‘0’ ‘0’ ‘)’ ‘VEC =’ *nod_var_vector*

Defines a terminal node of the FSM graph of RTL description. *nod_abs_index* and *nod_index* represent the absolute (inside the model) and relative (inside the graph) indexes of the node, respectively. Indexes of the variables labelling the node are determined with *nod_var_vector*.

nod_var_vector :=

‘ ”’ state_value {signal_value} ’ ”’

state_value shows the value of the next state. The signal_value constructs show the values of the control signals defined in the *control_signals* construct.

state_value := natural

Shows the value of the next state.

signal_value := natural | *‘X’*

The *signal_value* constructs show the values of the control signals defined in the *control_signals* construct.

nod_var := natural[[‘[‘ ‘V’ ‘=’ row_index ’]’ *]* ‘[‘ ‘V’ ‘=’ column_index ’]’ *]*

Shows the index of the variable labelling the node. Optional constructs *row_index* and *column_index* are used with memory variables labelling the node. These constructs determine the indexes of the variables used for addressing rows and columns, respectively.

nod_name := string

Shows the name of the node.

nod_range := range

nod_range determines the bit-slice of the variable that labels the node. HLDD model format allows slices of variables to be used for labelling a node.

row_index := natural

Determine the indexes of the variables used for addressing rows of the memory variable.

column_index := natural

Determines the index of the variable used for addressing columns of the memory variable.

nod_abs_index := natural

Shows the absolute (inside the model) index of the node.

nod_index := natural

Shows the relative (inside the graph) index of the node inside the graph.

graph_index := natural

Shows the index of the graph.

variable_flags :=

< 'i' | 'm' | 'c' | 'f' | 'o' | 'n' | '_' | 'F' > *{*<'d'> | '_'*}*

The variable flags have the following interpretation:

'i' - input variable

'm' - memory variable

'c' - constant variable

'f' - function variable

‘o’ - output variable

‘d’ - clock cycle delay, e.g. in registers, flipflops.

The following flags are used with RTL descriptions only:

‘n’ - control part output signal

‘F’ - FSM graph variable

‘r’ - reset variable

‘s’ - state variable

nod_flags :=

< ‘i’ | ’_’ > { ‘n’ | ‘v’ | ‘_’}

The node flags have the following interpretation:

‘i’ - inverted node (in gate-level descriptions only)

‘n’ - non-terminal node (RTL, behavioral)

‘v’ - control part terminal node (RTL)

successors :=

nonterminal_successors | *terminal_successor*

Construct successors shows the successor nodes of current node which are chosen with different node values.

nonterminal_successors :=

node_values ‘=>’ successor_index {node_values ‘=>’ successor_index }

This construct shows the indexes of successor nodes which will be selected with corresponding node values.

terminal_successors := ‘0’ ‘0’

Terminal nodes are nodes which have no successor nodes.

node_values := natural { ‘,’ | *‘’ natural}*

Determines the set of node values that activate the corresponding branch. The comma ‘,’ character is used for separating the indexes; the minus sign ‘-‘ is used for index ranges, e.g. ‘3-5’, which can be alternatively written as ‘3,4,5’.

successor_index :=

natural | 'X'

If *successor_index* is a natural number, it shows the index of the successor node. Otherwise, if *successor_index* is 'X', it means that the successor is undetermined.

var_index := natural

Shows the index of the variable.

var_name := string

Shows the name of the variable.

var_range := range

Shows the bit-width of the variable.

range := ['<' natural ':' natural '>']

Range is a construct for describing bit-vectors. The first natural shows the index of the most significant bit and the latter is for the least significant bit, respectively. If range is omitted, it will default to '<0:0>'.

string :=

' " ' {character} ' " '

Character can be any character, except newline and double quote '"'.

integer :=

['-']natural

Any integer number.

natural

Natural can be any non-negative number.

identifier :=

alphabetic_character{alphabetic_character | digit | '_'}

alphabetic_character := 'A'| ...| 'Z' | 'a' | ...| 'z'

digit := '0' | '1' | ...| '9'

Curriculum Vitae

Personal Data

Name	Anton Tšertov
Date of birth	09.09.1984
Place of birth	Estonia
Citizenship	Estonian

Contact Data

Address	Raja 15, Tallinn, 12618
Phone	+372 6202264
E-mail	anton.tsertov@ttu.ee

Education

2007 - …	Ph.D. Student, Department of Computer Engineering, Tallinn University of Technology
2006 – 2007	M.Sc. in Computer Engineering, TUT
2003 – 2006	B.Sc. in Computer Engineering, TUT

Carrier

2010 – …	Researcher, Department of Computer Engineering, TUT
2010 – …	ELIKO Competence Centre in Electronics-, Info- and Communication Technologies, R&D Engineer
2008 – 2011	R&D Engineer, Testonica Lab OÜ
2008	Java Developer, Cybernetica AS

2007 – 2010	Extraordinary Researcher, Department of Computer Engineering, TUT
2005 – 2007	ELIKO Competence Centre in Electronics-, Info- and Communication Technologies, R&D Engineer

Academic Degree

Master of Science in Computer Engineering, TUT,

“BIST Optimization Using LFSR Polynomial Calculation Method”

Awards

2007	Contest on Scientific publications in Tallin University of Technology - 1st place (Master degree category)
2007 – 2010	"Tiger University" scholarship for ICT PhD students (EITF)

Research topics

Optimization of board-level testing, decision diagrams, integrated circuit modeling, BIST

Elulookirjeldus

Isikuandmed

Nimi	Anton Tšertov
Sünniaeg	09.09.1984
Sünnikoht	Eesti
Kodakondsus	Eesti

Kontaktandmed

Aadress	Raja 15, Tallinn, 12618
Telefon	+372 6202264
E-post	anton.tsertov@ttu.ee

Hariduskäik

2007 - …	doktorant, arvutitehnika instituut, Tallinna Tehnikaülikool
2006 – 2007	tehnikateaduste magister, arvuti- ja süsteemitehnika eriala, TTÜ
2003 – 2006	tehnikateaduste bakalaureus, arvuti- ja süsteemitehnika eriala, TTÜ

Teenistuskäik

2010 – …	teadur, arvutitehnika Instituut, TTÜ
2010 – …	arendusinsener, ELIKO OÜ Tehnoloogia Arenduskeskus
2008 – 2011	arendusinsener, Testonica Lab OÜ
2008	programmeerija Java keeles, Cybernetica AS

2007 – 2010	erakorraline teadur, arvutitehnika instituut, TTÜ
2005 – 2007	arendusinsener, ELIKO OÜ Tehnoloogia Arenduskeskus

Teaduskraad

Tehnikateaduste magister, arvuti- ja süsteemitehnika, TTÜ

“BISTi optimeerimine, kasutades nihkeregistri polünoomi arvutamise meetodit”

Teaduspreemiad

2007	TTÜ teadustööde konkurss I koht tehnikateaduste valdkonnas magistri kategoorias
2007 – 2010	Tiigriülikooli stipendiumid IKT doktorantidele Eesti avalik-õiguslikes ülikoolides

Teadustegevus

Trükkplaatide testimise optimeerimine, otsusediagrammid, lõplike automaatide dekompositsioon, integraallülitusest kiibi modelleerimine

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo.** Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.