Digital Test in WEB-Based Environment

EERO IVASK

Faculty of Information Technology
Department of Computer Engineering
Chair of Computer Engineering and Diagnostics
TALLINN UNIVERSITY OF TECHNOLOGY

Dissertation was accepted for the defence of the degree of Doctor of Philosophy in Engineering at Tallinn University of Technology on June 19, 2006.

The commencement of the thesis will take place on July 5, 2006 in Tallinn University of Technology, Ehitajate tee 5, Tallinn, Estonia

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 before for any degree or examination at any other university.

ISSN 06
ISBN 5 5 6 6 6

To my family

Abstract

Current thesis presents an Internet based collaborative framework for digital testing using genetic algorithms for test generation software modules. Genetic algorithms are proposed in order to overcome complexity of the test generation problem for modern digital integrated circuits. Issues of hierarchical fault simulation and defect oriented fault simulation for test quality analysis are discussed as simulation is critical issue in genetic test generation. Digital test design flow begins with behavioral level VHDL description. Suitable flow chart like input format is extracted from source VHDL and fed into academical high-level synthesis tool xTractor. Subsequently generation of decision diagram models for test generation tools follows.

Current thesis also addresses issues of collaborative design and test. Universal state-of-the-art collaborative platform MOSCITO is described and possibilities of its use for digital design and test flow are analyzed and suitable strategies for workflow integration with existing test tools are proposed. In addition, necessary enhancements are proposed in order to use the MOSCITO system in firewall-protected environments. Finally, based on earlier studies and experience, the new completely http protocol based environment for remote tool usage is proposed. New platform has three-tier architecture using mostly Java applets as front-end, servlets on Tomcat as middleware and MySql as physical back end database server.

Kokkuvõte

Doktoritöö raames on välja pakutud mitu geneetilisel algoritmil põhinevat testigenereerimise meetodit kombinatsioon- ja järjestikskeemide testimiseks. Vastavad realiseeritud tarkvaramoodulid on mõeldud kasutatamiseks ka allpool mainitud internetipõhises keskkonnas. Geneetilised algoritmid on testigenereerimise probleemi lahendamisel valitud selleks, et ületada skeemide testimisel tekkivat NP keerukat lahendi otsimise probleemi, et vähendada otsinguruumi. Käsitletud on ka hierarhilise ja defekt-orienteeritud rikete simuleerimise meetodeid, kuna geneetilise testigenereerimise algoritmi juures mängib rikete simuleerimine olulist rolli- testigeneraatori poolt pakutud testikomplekte tuleb adekvaatselt hinnata igal iteratiivsel sammul. Digitaalseadme testimine võib alata n. käitumusliku taseme VHDL keelsest kirjeldusest. Doktoritöö raames koostatud kompilaatori abiga ekstraheeritakse VHDL lähtekoodist oluline informatsioon ja teisendatakse voodiagrammi sarnasele kujule, mis edasi leiab kasutust akadeemilises kõrgtasemesünteesi süsteemis xTractor. Viimase väljundist on võimalik genereerida omakorda otsustusdiagramide mudelid kasutamiseks n. eelpoolmainitud testitarkvaraga. Testimine võib alata muidugi ka madalamalt, nn. loogikalülide tasemelt, kui vastav skeem olemas on.

Käesolevas doktoritöös on välja töötatud ka internetipõhine virtuaalne keskkond kasutamiseks digitaalseadmete testimise valdkonnas. Keskkond on vajalik selleks, et olemasolevaid, seni ainult lokaalselt installeeritavaid tarkvaraprogramme üle interneti kasutada. Esmalt sai VILAB projekti raames välja töötatud moodus lokaalsete programmide distantskasutamiseks rakendades tarkvasasüsteemi MOSCITO võimalusi. Välja sai pakutud lahendus süsteemi kasutamiseks interneti tulemüüride olemasolu tingimustes ning sai teostatud hulgaliselt testeksperimente, sealhulgas ka tööstusliku disainiga. Kuna MOSCITO süsteemi võimalused on siiski piiratud, sai lõpuks välja pakutud uus, täiuslikum http protokolli põhine e. veebibrauseriga kasutatav kaugtöö keskond digitaalseadmete testi jaoks. Uuel süsteemil on kolmekihiline arhitektuur. Graafilise kasutajaliidesena kasutatakse Java applette, keskmises kihis kasutatakse Tomcat tarkvara ja servlette, andmebaasi serverina on kasutatud vabatarkvara MySQL.

Acknowledgements

First I would like to thank my supervisor Prof. Raimund Ubar for drawing my attention to testing field, to science, for possibility to do interesting work in his research group, which generally has opened new horizons to me. Thanks for support and valuable guidance in decisive moments.

I would like to thank Margus Kruus, our director of Computer Engineering department for his general administrative support and valuable encouragement towards this thesis.

Many thanks to Prof. Emeritus Leo Võhandu for introducing me to evolutionary algorithms and other innovative techniques in his seminars.

Thanks a lot for inspiring atmosphere and good cooperation and remarks to all colleagues in our research lab, especially to Jaan, Marina, Artur, Elmet, Marek and Peeter.

I am truly grateful to all EAS IIS colleagues and staff in Dresden for wonderful time I have spent there – thanks for all support and hospitality, thanks for providing excellent research conditions and everything. My special thanks to Andre Schneider for his kind attention and fruitful cooperation.

Last, but not least, I would like to thank my family- without you this work would not make sense at all.

Table of Contents

List of publications

1. E.Ivask, J.Raik, R.Ubar. Comparison of Genetic and Random Techniques for Test Pattern Generation. *Proc. Of the 6th Baltic Electronics Conference,* Oct. 7-9, 1998, Tallinn, pp. 163-166.

2. G.Elst, K-H.Diener, E.Ivask, J.Raik, R.Ubar. FPGA Design Flow with Automated Test Generation. *Proc. Of German 11th Workshop on Test Technology and Reliability of Circuits and Systems.* Potsdam, 1999, pp. 120-123.

3. E.Ivask, J.Raik, R.Ubar. Fault Oriented Test Pattern Generation for Sequential Circuits Using Genetic Algorithms. IEEE European Test Workshop, Cascais, Portugal, Mai 23-26, 2000, pp. 319-320.

4. E.Ivask, J.Raik, R.Ubar. Fault Oriented Test Pattern Generator for Sequential Circuits Using Genetic Algorithms. 7th Baltic Electronics Conference, Tallinn, October 8-11, 2000, pp.129-132.

5. K.-H.Diener, G.Elst, E.Ivask, G.Jervan, Z.Peng, J.Raik, R.Ubar. Digital Design Flow with Test Activities. VILAB User Forum, Smolenice, April 8, 2000, 11 p.

6. E.Ivask, R.Ubar, J.Raik, A.Schneider. Internet Based Test Generation and Fault Simulation. Design and Diagnostics of Electronic Circuits and Systems – DDECS‘2001, Györ, Hungary, April 18-20, 2001, pp.57-60.

7. A.Schneider, E.Ivask, J.Raik, P.Miklos, K.H. Diener, R.Ubar, W.Kuzmicz, W. Pleskacz, E. Gramatova. VILAB Test Generation Tools Running Under the MOSCITO System. VILAB User Forum Györ, Hungary, April 18-20, 2001, 12 p.

8. R.Ubar, J.Raik, E.Ivask, M.Brik. Hierarchical Fault Simulation in Digital Systems. Proceedings of Int. Symp. On Signals, Circuits and Systems SCS’2001, Iasi, Romania, July 10-11, 2001, pp.181-184.

9. A.Schneider, E.Ivask, P.Mikloš, J.Raik, K.H.Diener, R.Ubar, T.Cibáková, E.Gramatová. Internet-based Collaborative Test Generation with MOSCITO. IEEE Proc. Of Design Automation and Test in Europe – DATE’02. Paris, March 4-8, 2002, pp. 221-226.

10. A.Schneider, K.-H.Diener, E.Ivask, R.Ubar, E.Gramatova, T.Hollstein, W.Pleskacz, W.Kuzmicz, Z.Peng. Integrated Design and Test Generation Under Internet Based Environment MOSCITO. EUROMICRO Conference, September 3-6, 2002, pp. 187-194.

11. A.Schneider, K.-H.Diener, E.Ivask, R.Ubar, E.Gramatova, M.Fisherova, W.Pleskacz, W.Kuzmicz. Defect-Oriented Test Generation and Fault Simulation in the Environment of MOSCITO. Proceedings, BEC-2002, Tallinn, October 6-9, 2002, pp.303-306.

12. A.Schneider, K.-H.Diener, G.Elst, E.Ivask, J.Raik, R.Ubar. Internet-Based Testability-Driven Test Generation in the Virtual Environment MOSCITO. Proc. IFIP Conference on IP Based SOC Design, Grenoble, France, October 30-31, 2002, pp.357-362.

13. R.Ubar, J.Raik, E.Ivask, M.Brik. Multi-Level Fault Simulation of Digital Systems on Decision Diagrams. IEEE Workshop on Electronic Design, Test and Applications – DELTA'02, Christchurch, New Zealand, 29-31 January 2002, pp.86-91.

14. R.Ubar, J.Raik, E.Ivask, M.Brik. Mixed-Level Defect Simulation in Data-Paths of Digital Systems. 23rd Int. Conf. On Microelectronics. Nis, Yugoslavia, May 12-15 2002, Vol.2, pp.617-620.

15. R.Ubar, J.Raik, E.Ivask, M.Brik. Defect-Oriented Mixed-Level Fault Simulation in Digital Systems. Facta Universitatis (Nis), Ser.: Elec. Energ. Vol.15, No.1, April 2002, pp.123-136.

16. R.Ubar, J.Raik, E.Ivask, M.Brik. Test Cover Calculation in Digital Systems with Word-Level Decision Diagrams. Proc. Of the International Conference on Computer Dependability, Tomsk, Russia, September 10-13, 2002, pp.315-319. Invited paper.

17. A.Schneider, K.-H.Diener, G.Elst, R.Ubar, E.Ivask, J.Raik. Integration of Digital Test Tools to the Internet-Based Environment MOSCITO. Proc. Of 7th World Multiconference on Systemics, Cybernetics and Informatics – SCI 2003. Orlando, USA, July 27-30, 2003, pp.136-141.

18. M.Aarna, E.Ivask, A.Jutman, E.Orasson, J.Raik, R.Ubar, V.Vislogubov, H.D.Wuttke. Turbo Tester – Diagnostic Package for Research and Training. J. Of Radioelectronics and Informatics, No3 (24), July – September, 2003, pp. 69-73.

19. M.Brik, J.Raik, R.Ubar, E.Ivask. GA-based Test Generation for Sequential Circuits. 2nd East-West Design & Test Workshop EWDTW-2004, Alushta 23-26, 2004, pp.30-34.

20. M.Brik, E.Ivask, J.Raik, R.Ubar. On Using Genetic Algorithm for Test Generation. Proc. Of the 9th Biennial Baltic Electronics Conference, Oct. 3-6, 2004, Tallinn, pp.233-236.

21. E.Ivask, P.Ellervee. VHDL Front-End for High-Level Synthesis Tool xTractor. Proc. Of the 9th Biennial Baltic Electronics Conference, Oct. 3-6, 2004, Tallinn, pp.111-114.

22. E. Ivask, J. Raik, R. Ubar, A. Schneider. WEB-Based Environment: Remote Use of Digital Electronics Test Tools. In "Virtual Enterprises and Collaborative Networks", Kluwer Academic Publishers, 2004, pp. 435-442.

List of Figures

List of Tables

1 Introduction

The increasing complexity of VLSI circuits and transition to Systems-on-Chip (SoC) or even Networks-on-Chip (NoC) paradigm has made test generation one of the most complicated and time-consuming problems in the domain of digital design. The more complex are getting electronics systems, the more important become problems of test and design for testability, as costs of verification and testing are getting the major component of design and manufacturing costs of a new product. This fact makes the research in the area of testing and diagnosis of integrated circuits (IC) a very important topic for both the industry and the academy.

Commercial CAD systems for VLSI design and test are both costly and do not provide a good variety of competing or complementary approaches to a given particular problem. They usually have a stiff workflow of standard integrated tools bound together and should be executed accordingly to a certain scenario. It may be not good for a designers and researcher whose goal is the search for new efficient solutions. In order to come up with innovative electronic systems in time and with competitive cost, lot of EDA problems should be solved: HW/SW co-design, high-level synthesis, testability evaluation, test pattern generation. During the last decade, many different low-cost tools running on PCs have been developed to fill this gap. They usually include the major basic tools needed for IC design: schematics capture, layout editors, simulators, and place and route tools. However, low-cost systems for solving a large class of tasks from the dependability and diagnostics area: test synthesis and analysis, fault diagnosis, testability analysis, built-in self-test (BIST), especially for research and educational purposes, are still missing. For this reason, a diagnostic software Turbo Tester [54] is being developed in Tallinn University of Technology. Genetic test tools presented in this thesis and belonging to Turbo Tester toolset contribute to solving digital test problems.

Turbo Tester toolset was long time just a set of command line tools. There was strong need for user friendly graphical interface and even more – the need to make the tools available over the internet to avoid the installation overhead, simplify the maintenance and finally, to offer the computational power of fast application servers to end users. Another issue generally is that usually all the needed design and test tools are not available at persons working site. Therefore, current thesis partially addresses the issue of remote use of existing work tools.

At the same time recent development trends show increasing use of various hardware description languages (HDL) among hardware designers because of the advantages they offer over traditional schematic techniques. The main advantage is the possibility to use the same description both to model the behavior and as a starting point for schematic synthesis. Another important feature of most of the HDL-s (e.g. VHDL) is the possibility to describe an algorithm at higher abstraction levels thus hiding target technology dependent hardware implementation details. The introduction of high-level synthesis (HLS), also known as behavioral synthesis, promised to automate the transformation of a design from system/behavioral level to register-transfer level as efficiently as the introduction of logic synthesis automated transformation from logic to physical level. Most of the HLS tools (methodologies) make use of HDL-s as the language of input and output data. Also fast and efficient hierarchical test generators need both high level (RTL) and low level (gate level) descriptions in order to work. Extracting high level information from VHDL for hierarchical test generator DECIDER and generating input for synthesis system xTractor has great importance. Current thesis addresses the issue of extracting the suitable information from behavioral level VHDL description for synthesis tool xTractor and decision diagram synthesis.

In current thesis chapter 2 gives overview of previous work in the field of test generation and Internet based collaborative design and test. Chapter 3 concentrates on test generation using genetic algorithms. Chapter 4 describes simulation algorithms. In chapter 5 is explained how data is extracted from VHDL for high-level synthesis and test generation. Complete digital design and test flow is presented. In chapter 6 state-of-the-art collaboration platform MOSCITO is described and possibilities of its use for digital design and test flow and suitable workflow integration strategies for existing test tools are discussed. Finally, enhanced web based system using HTTP protocol and MySQL database is proposed.

2 Review of State-of-the-Art

2.1 Test of digital systems

Sequential circuit test generation using deterministic algorithms is highly complex and time consuming. Classical deterministic (topological) test generators like HITEC [1,2] attain high fault coverage and are able to find test sequences for “hard” faults. Each target fault must be activated and then propagated to a primary output. The activation state must be justified using reverse time processing. During test generation, the large number of backtracking must be handled. In order to decrease the number of faults to be modeled, fault simulation (fault dropping) is used. Nevertheless, test sets can be long. The major drawback of such generators is that they are too time consuming while working through search space.

Functional level test generators like [3,4] use functional description of a circuit and exploit functional fault model. Test generation time is reduced. However, estimating fault coverage at the functional level is difficult because the accuracy of functional fault models is unproven and gate level models better characterize physical faults. Functional test based on FSM is used in [5]. The limitation of such functional method is also that a fault does not increase the number of states of machine and therefore does not help distinguish correct and faulty behavior.

As a solution, hierarchical approaches [6,7] have been proposed which take advantage of high-level information during generating tests for gate level faults. While hierarchical test pattern generation still remains the fastest method for solving the problem, it is not applicable for designs that do not have an appropriate modularity or where the higher-level information is not known.

On the other hand, simulation-based techniques have proven their efficiency, being able to obtain high fault coverage using less time than classical topological Automatic Test Pattern Generation (ATPGs). Fault simulation based test generators have the advantage that they can be adapted to new fault models or different circuit descriptions with minimal effort by using a fault simulator suitable for the new fault model. The drawback of simulation-based test generators generally consists in that they do not identify undetectable faults due to absence of any deterministic test generation procedure. In addition, simulation based approaches are fast only for smaller circuits and become ineffective when number of primary inputs and sequential depth of the circuit increase. The first simulation-based test generator was proposed by Seshu and Freeman [8]. Since then, several simulation-based test generators have been developed ([9], etc.).

Simulation-based technique is also used by Genetic Algorithm (GA) based test generators. There are several GA based test generators now. Fitness functions were used to guide the GA in finding a test vector or sequence that maximizes given objectives for a single fault or group of faults. In GATEST [10] the fitness function is biased toward maximizing the number of faults detected and the number of fault-effects propagated to flip-flops; increasing the circuit activity is a major objective in CRIS [11] and GATTO [12]. Maximizing propagation of fault effects to flip-flops and increasing circuit activity were shown to increase the probability of detecting faults at the primary outputs. Activity means the change of the logic values in different circuit points. It is assumed that the more the input vector causes the circuit activity, the more likely the fault effect is carried to primary outputs. Although the fault-detection probability improves, activating a hard fault and propagating fault-effects from flip-flops to a primary output remain difficult problems. Increasing circuit activity may be ineffective in activating a hard fault or propagating fault effects. DIGATE [13] tackles the problem of fault-effect propagation by intelligent use of distinguishing sequences. However, also here faults must be activated in order to apply that technique effectively.

2.2 Web-based digital design

Over the last years, with advancements in the networking technology, several solutions are worked out in order to ease the collaboration in digital design field, to share the software tools, reuse the IP blocks, etc. Subsequently most interesting ones are described, classified by use case.

Collaboration support (groupware)

The design of complex hard- and software systems is a collaborative task and is typically solved in a workgroup. Despite of the interest in the field of groupware (computer-supported cooperative work) it is still the case that few systems have been adopted for widespread use. It is especially true for widely dispersed, cross-organizational design teams where problems of heterogeneity in computing hardware and software environments inhibit the deployment of groupware. Research in groupware field focuses on developing new theories and technologies for coordination of groups of people who work together. Key issues are group awareness, multi-user interfaces, concurrency control, communication and coordination within the group, shared information space and support of a heterogeneous, open environment which integrates existing single-user applications.

Feature rich solution for collaboration support is proposed in [15]. Concept of shared workspace for asynchronous cooperation was developed based on the idea of a private workspace. The workspace of single person comprises his personal work context. Workspace is used to store and retrieve documents, which can be stored into different container objects, like folders, also work tools are accessed from this workspace. Additional information about the work process can be stored there, as well the notes and tools needed. Work processes in the group can be organized synchronously or asynchronously. In first case direct response to one’s activities is estimated, in second case response is usually delayed. The shared workspace is the central access point for common data and information of the state of the work process. Support of asynchronous aspects is of greater interest in teleworking of small and big companies. Participants may access and exchange documents at any time and all workgroup members are aware of the overall work progress. Proposed system has autonomously managed shared workspace, which the members of the working group install and use for the organization and coordination of their tasks. It is possible to upload documents from local computer to shared workspace and process documents in the workspace. Different type of objects can be in the workspace: folders, documents, tables, graphics, and links to www pages. There is notification of activities. Ordinary web browser is required only. Standard client-server architectural model is used. CGI (Common Gateway Interface) technique is used in order to extend the web server functionality. Unix and Windows NT are supported. Apache web server and Microsoft IIS web server is supported. There is multilanguage support for user interface. Members can set access rights to control the visibility of the information or operations what can be done by others. HTML pages show the information and are refreshed when activities (buttons pressed etc.) are carried out. System permanently registers events i.e. what is going on, offers some version management. System supports building communities of interest- knowledge co production, support. Communities gather and develop existing information on the web and collaborate in synchronous or asynchronous manner. Developing means structuring and organizing the information. Different roles and different access rights are necessary. Java based user interfaces and role concept, document locking mechanism, moderated workspaces were added to system finally.

Only major drawback of described system is use of old-fashioned CGI technique, which is slow. Interesting is that in final stadium Java based interfaces were introduced to system. This shows the tendency in favor of Java.

IP Reuse Paradigm

In paper [14] a virtual electronic component (Intellectual Property – IP) exchange infrastructure is presented, whose main components are: 1) XML based well structured IP catalog builder responsible for management and e-publishing IPs and 2) XML IP profiler extracting IP files from design directories, transferring files to the used site via IP distribution server. Both applications use Java servlets and have client-server architecture. XML files are used instead of database.

Remote tool usage

Paper [18] presents the concept of a distributed, web-based electronic design framework. System has client-server architecture with multiple tiers. Web server is serving client requests whilst acting as client to the tool servers. In the sample application of the framework, developed in Java, any of the servers can be based on Linux, MS Windows or Sun-SPARC server. The web server has been used to demonstrate the framework for on-line access to VAMS (a VHDL-AMS parser) and Avant!

The multi-user nature of such framework requires the web server to have some 'memory' of what actions users have already been undertaken so that it can prompt for the next stage. HTTP is by definition a stateless protocol and therefore maintains no such 'memory'. Session tracking API provided by Java language is used here. It makes use of the facilities available (allowed) in user's web browser, such as cookies, URL rewriting or hidden fields to maintain this session 'memory'. Simple Java servlet code was written to implement this session tracking. Web server is responsible to maintain a collection of these session objects, which are created while different users make requests allowing to store relevant information and maintain information about current state. Usage of Java session object is sufficient in case of simple system, otherwise real database backend offers much more flexibility and reliability.

The nature of the framework also requires that files (e.g. design netlists) can be uploaded from a user's computer to the web server for processing. The files can be streamed through Java

StreamReader classes in order to write out to the file system. This is job of second servlet. The two servlets form a system, which will uniquely record a user's identification when writing the uploaded files to the web server file system. A system can store data about when files were written and last accessed in order to allow the web server to recover in the event of the crash and delete expired files. The Java RMI (Remote Method Invocation) was used as the communication method between web server and the tool servers. RMI provides a programming method that allows code development as if the remote Java classes are resident locally. This level of transparency is clearly excellent. For example, on the web server side it will look like as if the work tools reside on the web server itself. However, it is necessary to register the remote Java services with a Java RMI registry on the remote machine. When connection is made to a TCP socket, the correct service can be invoked to handle the connection. Java RMI is very elegant solution for distributed network programming indeed, only it has drawback that in presence of firefalls such solution will need opening additional communication ports and configuring firewall rules. Most probably, this will fail in case of big corporate firewalls with strict security policy.

Graphical user interface is here Java based. Java provides a number of built-in classes that enable graphics programming. These are the abstract window toolkit AWT and the swing class that provides a number of reusable graphics components. A simple method of graphing data in Java is available, and indeed this should not be overlooked as a method to visualize numerical data.

Simple name/password authentication provided by the HTTP protocol – called basic authentication was used in order to restrict access to preset domains within a web server. With this technique, the web server maintains a database of usernames and passwords and identifies certain resources as protected. When a user requests access to a protected resource, the server responds with a request for the client's username and password. At this point, the browser usually pops up a dialog box where the user enters the information. Basic authentication is very weak. It provides no confidentiality, no integrity, and only the most basic authentication. The problem is that passwords are transmitted over the network, thinly disguised by a well known and easily reversed Base64 encoding. Great amount of web server configuration is needed by simple authentication. Custom authentication technique, to which up-to-date security techniques may easily be applied, has been chosen for this application. This is a method where a servlet handles the access restriction, governing which servlets may then be accessed.

Paper [17] presents approach for dynamic and secure resource integration and administration, i.e., TRMS (Tool Registration and Management Services). TRMS allows the dynamic discovery of a tool using semantics descriptions of the desired tool behavior. A tool is described by a set of significant properties based on which it can be discovered in the network. Variant of that approach based on SNMP (Simple Network Management Protocol), which is a widely accepted standard for general network administration was presented. The implementation gives the context for an illustrating example in the area of a realistic PCB design flow based on the Zuken Hot-Stage tool suite. Presented approach was intended for intranet only.

Paper [19] uses web services as means to integrate remote tools in a workflow-driven design process for embedded systems. It is of course questionable if "providing a service" may offer a greater market-potential than “supplying a tool”. Problem is that design environments must be highly cooperative.

On the one side, definition and implementation on object level is still the adequate design style for tightly coupled components, on the other side when targeting Internet scenarios the concept of interacting services is now the state-of-the-art specification method. This service-based programming paradigm is backed by new Internet protocols and languages like SOAP [21] and WSDL [22], which serve exactly the purpose of defining and describing services and their intercommunications. New applications or services can rely on services from other service-providers. Since the client of a service is not defined at the time the service is provided the deployment and publication are a most important part during its life cycle. With UDDI [23], a “dictionary” with a standardized access mechanism has been defined to alleviate this problem. This progress affects the traditional tool-centered engineering domains as well, since issues like time-to-market and distributed development and design are common factors in the affected processes. Moreover, the sophistication of tools gains a new level as the design technology for new products evolves rapidly. Therefore, they are valuable assets for a company forming an important part of their intellectual property (IP). Using the service-centered approach, such companies have the chance to offer using their knowledge without having to externalize programs or algorithms. For the user of such services one important question is how to integrate such a service into their work environment. Usually integration focuses on the principle of coupling existing applications or components tightly together to ensure smooth and reliable operation. The resulting (and available) integration environments therefore use proprietary integration mechanisms on top of existing base-technologies like CORBA [20], JAVA [24], and JAVABEANS [25] or similar middleware components. In fact, CORBA-Services for instance constitute conceptually the same idea as web services with WSDL: to enable their use through a common interface by different applications. The important difference of this approach is that web services are build on top of a foundation that is centered on the Internet platform. Which means the common denominator for running such services is standard web server technology, TCP/IP networks and the HTTP protocol accompanied by the flexible XML meta-format.

With web services there is no firewall traversal problem, however there will be problem with consuming these services. Question is about the granularity of the service pieces. Fine grain services are interesting for system integrators mainly. Average chip designer who wants to get a specific task done, will obviously not benefit from that. He needs complete tool not pieces of it. Complete tool must be made available, which falls out bit of the initial meaning of web services.

Remote access of reconfigurable hardware

Paper [16] presented an approach for the integration of reconfigurable hardware and computer applications based on the concept of ubiquitous computing. A set of reconfigurable hardware modules can be plugged in a network and be transparently accessed by client software applications. The client applications must not have any information about the network location or the internal implementation of the reconfigurable modules. The connection between client and reconfigurable hardware is based in a lookup mechanism. The reconfigurable hardware is encapsulated by a service interface, and all the communication with the client is done in the API level, through method calls.

Jini – based technology was used here. This is a relatively new Java-based technology for distributed computing, publicly debuted by Sun Microsystems in January 1999. It claims to address the fundamental difficulties of distributed computing. These difficulties center around the fact that distributed systems are vulnerable to network latency and concurrency problems, complexities in memory management, and inevitable partial failures. Jini technology addresses the difficulties of distributed computing with a simple set of interfaces that are claimed to be well specified and open to implementation by any member of the Java community. Jini technology is built on top of the Java 2 platform. It uses core Java functionality to provide a reliable, portable, distributed computing model. In terms of the Java platform, Jini technology takes advantage of the following:

- The inherent security provided by the Java technology's robust and publicly tested security model
- The portability of Java technology byte code, provided by the widespread availability of Java virtual machines (JVMs)
- The mobility of Java objects, provided by object serialization and remote method invocation (RMI)

Jini technology enables spontaneous networks of devices and software services to assemble themselves into working groups known as federations of clients and services, without the need for intervention by system administrators. The Java language's ability (RMI) to move entire objects, both data and code, allows Jini technology-based systems to deal reliably with partial failures and network issues. Distributed memory management is taken care of using underlying RMI functionality, and concurrency and latency issues become tractable with Jini technology's distributed event support, leasing, and transaction capabilities. Participants in one network can directly access and use the services provided by participants in another network by using objects that move around the network, the Jini architecture makes each service, as well as the entire network of services, adaptable to changes in the network. The Jini architecture specifies a way for clients and services to find each other on the network and to work together to get a task accomplished. Service providers supply clients with portable Java technology-based objects that give the client access to the service. This network interaction can use any type of networking technology such as RMI, CORBA, or SOAP, because the client only sees the Java technology-based object provided by the service and, subsequently, all network communication is confined to that Java object and the service from whence it came. When a service joins a network of Jini technology-enabled services and/or devices, it advertises itself by publishing a Java technology-based object that implements the service API. This object's implementation can work in any way the service chooses. The client finds services by looking for an object that supports the API. When it gets the service's published object, it will download any code it needs in order to talk to the service, thereby learning how to talk to the particular service implementation via the API. The programmer who implements the service chooses how to translate an API request into bits on the wire using RMI, CORBA, XML, or a private protocol.

Jini technology is promising, especially when truly distributed spontaneous and robust systems must be designed. However, firewall traversal can be problematic again – dedicated communication ports are needed. Another issue is the bandwidth required for communication, especially in case of mobile devices.

2.3 Discussion

Considering implementation approaches used above for web-based design activities, it is clear that most popular development platform is Java environment and that's for reason – Java is portable, mature, fast enough nowadays, it is free and what's most important – it is meant for network programming from start.

There exist at least alternatives like Microsoft ASP .NET platform, PHP for web-based software development, but none of them was surprisingly considered in papers found. Development of web applications with .NET could be rapid, actually faster than in case of Java or PHP since GUI development and data access to Microsoft SQL server is well automated, manual coding is much less, web services development is supported as well. However, developer has to stick to Windows platform. Even change of back end database to something else (for example to open source db MySQL) will slow down the development process considerable since data access functions must be mastered by designer itself again. PHP in turn is essentially scripting language engine running as web server extension. PHP language is used much for building web sites. Therefore, it is suitable for development of user interfaces. If information polling strategy can be used then it is obviously possible to use also PHP for simpler remote tool usage system development.

Considering further Java based implementations, RMI based solutions dominated. As it was stated earlier, RMI (Remote Method Invocation) is elegant programming solution for distributed computing were one program can remotely invoke methods physically residing in other machine. However, firewall traversal can be problematic, as dedicated communication ports are needed. Strict security policy might not allow that.

It seems that approaches based on the eXtensible Markup Language (XML) are widely suggested. The W3C [26] defines a set of XML-based languages that are the foundation for the current notion of web services. The Web Service Description Language (WSDL) is used to describe the interfaces of a web service. These interfaces can be accessed using the Simple Object Access Protocol (SOAP). WSDL descriptions can be made available via a central UDDI registration (Universal Description, Discovery, and Integration), e.g., in order to implement resource discovery.

However, currently there are some serious reservations using such web services for industrial applications. The related standards like SOAP are currently evolving fast with the side effect of introducing compatibility issues and general uncertainty. Since these technologies are not stable yet, it is undesirable to apply them today in a true industrial context. Integration technology for WSDL must change and hopefully become as transparent as accessing other content in the Internet in order to be successful in the future. The usage of this technology for embedded system design is of high interest, because it offers on one side more freedom for the user of such systems and on the other side for the tool vendor new business concepts or licensing models.

The ordinary integration technologies still have their merits. Because web services represent a weak coupling and require more dynamic processing for the protocols a certain overhead is generated which slows down the interaction. Especially "popular" web services may have long latencies that reflect the same behavior as visiting heavy-loaded web servers. In the intranet the common integration techniques are to be preferred because the location of tools and applications is known and under direct administration. Additionally this technique of integration allows an efficient adoption to the clients requirements, whereas changing a web service depends on the cooperation of the service provider. Usually there are different clients with contradicting requirements, which are not easy to meet by the provider.

For site-spanning tool integration, one has to consider open and known security problems in order to highly protect the exchanged IP-s. Due to those problems, SNMP-based solutions seem to be less applicable for open networks. Nevertheless, also XML-based alternatives with SOAP servers currently have significant unsolved security problems. We see that network-based solutions for non-secure environments still require significant investigations in authentication and encryption when exchanging control and highly sensitive (i.e., IP-protected) design data [17].

One problem is that XML-based messages are larger and require more processing than those from existing protocols: data is represented inefficiently and binding requires more computation. It has been shown that an RMI service can perform up to an order of magnitude faster than an equivalent Web Service due to the processing required to parse and bind XML data into programmatic objects [27].

We see, there are many possibilities, which makes cooperation harder. For example, WSDL and CORBA are similar approaches, but technically different. This means that it is harder to cooperate between enterprises to come up with virtual laboratories for example. The hope is that interfaces will eventually be standardized allowing seamless operation.

Jini based solution is not attractive alternative either because it is also based on RMI (Remote Methods Invocation). Such approach will not work well in firewall-protected environments. Considerable effort has to be made to configure enterprise firewalls. In some cases, this might not be possible at all.

In conclusion, plain HTTP protocol based solutions seem to be still best choice, since HTTP is well established and such solutions are more flexible in firewall-protected environments. Communication port 80, used by ordinary web browsers is always available on the client side of the system. It is also possible to add SSL (Secure Socket Layer) encryption to communication between two endpoints. This is big advantage when exchanging possibly sensitive design and test information. Choosing Java as development platform gives us portability and rich and widely accepted development environment for network programming. Java has also powerful classes to simplify HTTP communication. Data can be simply sent over the network as objects i.e. data bundles.

As a final remark, all the papers found and discussed above were dealing design issues, there was no evidence of using web-based environment for testing or digital test tools implicitly.

3 Test Generation for Digital Systems with Genetic Algorithms

Several techniques for solving the problem of generating tests for structural faults in sequential circuits have been proposed over the years. On the gate-level, a number of deterministic test generation algorithms have been implemented. However, the execution times are extremely long and for medium and large circuits mostly rather low fault coverage has been achieved. Better performance has been reported of simulation-based approaches. The above approaches are fast for smaller circuits only and become ineffective when number of primary inputs and sequential depth of the circuit increase.

Test generation approaches that rely on functional fault models only do not guarantee satisfactory structural level fault coverage. As a solution, hierarchical approaches have been proposed which take advantage of high-level information during generating tests for gate level faults. While hierarchical test pattern generation still remains the fastest method for solving the problem, it is not applicable for designs that do not have an appropriate modularity or where the higher-level information is not known.

Genetic algorithm based test generation overcomes the difficulties when deterministic generators are too slow and fast approaches, like hierarchical test generation, are not applicable to circuit due to lack of proper information; when fault coverage is too low or amount of generated test vectors is too large. In current chapter three different test generators are presented: for combinational circuit, for circuits represented as FSM and finally fault oriented TPG for gate level sequential circuit.

3.1 Overview of genetic algorithms

John Holland, the founder of the field of genetic algorithms points out in [28] the ability of simple representations (bit strings) to encode complicated structures and the power of simple transformations to improve such structures. Holland showed that with the proper control structure, rapid improvements of bit strings could occur (under certain transformations). Population of bit strings "evolves" as populations of animals do. An important formal result stressed by Holland was that even in large and complicated search spaces, given certain conditions on the problem domain, genetic algorithms would tend to converge on solutions that were globally optimal or nearly so.

In order to solve a problem genetic algorithm must have following components:

1) A chromosomal representation of solution to the problem,
2) A way to create an initial population of solutions,
3) An evaluation function that plays the role of the environment, quality rating for solutions in terms of their "fitness"
4) Genetic operators that alter the structure of "children" during reproduction
5) Values for the parameters that genetic algorithm uses (population size, probabilities of applying genetic operators)

3.1.1 Representation

In genetic framework, one possible solution to the problem is called individual. As we have different persons in society, we also have different solutions to the problem (one is more optimal than other is). All individuals together form population (society).

We use binary representation for individuals (bit strings of 1's and 0's). Bit strings have been shown to be capable of usefully code variety of information, and they have been shown to be effective representations in unexpected domains. The properties of bit string representations for genetic algorithms have been extensively studied, and a good deal is known about genetic operators and parameter values that work well with them [28].

3.1.2 Initialization

For research purpose, random initializing a population is suitable. Moving from a randomly created population to a well-adapted population is a good test of algorithm. Critical features of final solution will have been produced by the search and recombination mechanisms of the algorithm rather than the initialization procedures. To maximize the speed and quality of the final solution it is usually good to use direct methods for initializations. For example, output of another algorithm can be used or human solution to the problem.

3.1.3 Fitness function

It is used to evaluate the fitness of the individuals in population (quality of the solutions). Better solutions will get higher score. Evaluation function directs population towards progress because good solutions (with high score) will be selected during selection process and pour solutions will be rejected.

3.1.4 Fitness scaling

Many properties of evaluation function enhance or degrade a genetic algorithm's performance. One is normalization process used. Normalization is *fitness scaling* (increasing the difference between the fitness values). As a population converges on a definitive solution, the difference between fitness values may become very small. Best solutions can't have significant advantage in reproductive selection. For example, let us have scores 2 and 3. If these scores are used without any change as measures of each individual's fitness for reproduction, it will take some time before the descendants of good individual will gain the majority in the population. Fitness scaling solves this problem by adjusting the fitness values to the advantage of the most-fit solutions. For example, we can use squared values of fitness scores. Then, continuing example above we receive new scores 4 and 9 which are much more different now.

The performance of a genetic algorithm is highly sensitive to normalization technique used. If it stresses improvements to much it will lead to driving out of alternative genetic material in the population, and will promote the rapid dominance of a single strain. When this happens, crossover becomes of little value, and the algorithm ands up intensively searching the solution space in the region of the last good individual found. If the normalization process does not stress good performance, the algorithm may fail to converge on good results in a reasonable time and will more likely to lose the best members of its population.

3.1.5 Fitness function and noise

In some cases, there will be no known fitness function that can accurately assess an individual's fitness, so an approximate (noisy) fitness function must be used. The noise inherent from noisy fitness function causes the selection process for reproduction to also be noisy [29]. We assume that a noisy fitness function returns a fitness score for an individual equal to the sum of real fitness of the individual plus some noise. Noisy information may come from a variety of sources, including noisy data, knowledge uncertainty, etc. To improve run – time performance some genetic algorithms use fast but noisier fitness function instead of more accurate, but slower, fitness function that may also be available. Sampling fitness functions are a good example of this phenomena: fitness function uses smaller (reduced) sampler size to increase run – time speed, at the expense of decreased accuracy of the fitness evaluation.

3.1.6 Reproduction

During reproductive phase of genetic algorithm, individuals are selected from population and recombined, producing child (individual), which will be added into next generation. Parents are selected randomly from the population using a scheme, which favors the more fit individuals. Good individuals will probably be selected several times in a generation, poor ones may not be at all. Having selected two parents, their genes are combined, typically using the mechanisms of crossover and mutation. Next, we take a closer look on genetic operators.

3.1.7 Selection

Selection mechanism finds two (or more) candidates for crossover. The *selection pressure* is the degree to which the better individuals are favored: the higher the selection pressure, the more the better individuals are favored. This selection pressure drives the genetic algorithm to improve the population fitness over succeeding generations. The convergence rate of genetic algorithm is largely determined by the selection pressure. Higher selection pressures result higher convergence rates. Genetic algorithms are able to identify optimal or near optimal solutions under a wide range of selection pressure [30]. However, if the selection rate is too low, the convergence rate will be slow, and the genetic algorithm will unnecessarily take longer to find the optimal solution. If the selection pressure is too high, there is an increased change of genetic algorithm prematurely converging to an incorrect (sub-optimal) solution.

There are several selection strategies possible:

1) Roulette wheel selection

It is a standard gambler's roulette wheel, a spinning circle divided into several equal size sections. The croupier sets the wheel spinning and throws marble into bowl. After the motion of the wheel decreases, marble comes to rest in one of the numbered sections.

In the case of genetic algorithm roulette wheel could be used to select individuals for further reproduction. The wheel corresponds to fitness array and the marble is a random unsigned integer less than the sum of all fitnesses in population. Let us look at Figure 1. There is c-coded fragment of roulette wheel interpretation for genetic algorithm. To find an individual

```
int select_individual()
 {
   int j=0;
   long int rnd_num;

   rnd_num =  floor (Random_btw_0_1() * fit_sum);

   while (rnd_num > fit_arr[j])
     {
       rnd_num -= fit_arr[j];
       j++;
     }
  // returns index of the individual in population
  return (j);
}
```

Figure 1 Pseudo c- code for proportional roulette wheel selection

associated with the marbles landing place, the algorithm iterates through the fitness array. If the marbles value is less than the current fitness element, the corresponding individual becomes a parent. Otherwise, the algorithm subtracts the current fitness value from the marble and then repeats the process with the next element in the fitness array. Thus, the largest fitness values tend to be the most likely resting places for the marble, since they use a larger area of the abstract wheel. That's why strategy described above is called proportional selection scheme. To clarify, let us look small example with a population size five. Table 1 shows the population and its corresponding fitness values.

Individual	**Fitness**
10110110	20
10000000	5
11101110	15
10010011	8
10100010	12

Table 1 Hypothetical population and its fitness

Total fitness of this population is 60. Figure 2 shows pie chart representing relative sizes of pie slices as assigned by fitness. We can also imagine that pie chart as our roulette wheel discussed above. What we actually see is a wheel with slots of different size. Sizes of slices correspond to roulette wheel slot sizes.

We see, that individual 10110110 has 34% chance of being selected as parent, whereas 10000000 has only an 8% chance to be selected. Selecting five parents for example requires simply generating five random numbers, as shown in Table 2. Given a random number we determine a corresponding individual by looking at Figure 3.

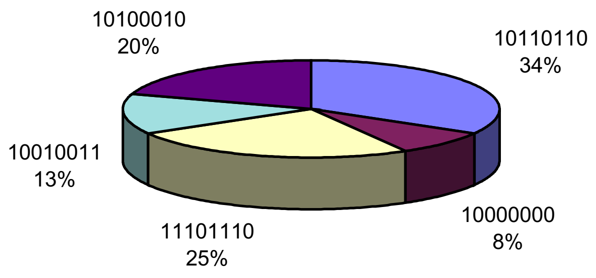


Figure 2 Roulette wheel. Slots are proportional to individual's fitness

Random num.	Individual
44	10010011
5	10110110
49	10100010
18	10110110
22	10000000

Table 2 Randomly selected numbers and corresponding individuals

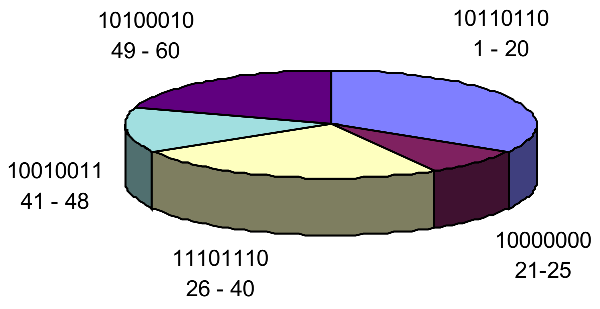


Figure 3 Slot ranges corresponding to certain individuals

We see that individual 10110110 (with the highest fitness) is two times a parent for members of the new population, this is naturally allowable. Even the chromosome with the lowest fitness will be a parent once. However, second-most-fit chromosome did not reproduce- *c'est la vie,* life is unpredictable.

2) <u>Stochastic universal selection</u>
It is a less noisy version of roulette wheel selection in which N markers are placed around the roulette wheel, where N is a number of individuals in the population. N individuals are

selected in a single spin of the roulette wheel, and the number of copies of each individual selected is equal to the number of markers inside the corresponding slot.

3) Tournament selection
s individuals are taken at random, and the better individual is selected from them.
It is possible to adjust its selection pressure by changing tournament size. The winner of the tournament is the individual with the highest fitness of the s tournament competitors, and the winner is inserted into mating pool. The mating pool, being filled with tournament winners, has a higher average fitness than the average population fitness. This fitness difference provides the selection pressure, which drives the genetic algorithm to improve the fitness of each succeeding generation. Tournament selection pressure can be increased (decreased) by increasing (decreasing) the tournament size s, as the winner from larger tournament will, on average, have a higher fitness than the winner of a smaller tournament.

3.1.8 Crossover

Exchanges corresponding genetic material from two parents, allowing useful genes on different parents to be combined in their offspring. Two parents may or may not be replaced in the original population for the next generation, these are different strategies.

Crossover is the key to genetic algorithm's power. Most successful parents reproduce more often. Beneficial properties of two parents combine. Figure 4, Figure 5 and Figure 6 show most common crossover types. White and grey represent different individuals, their genetic material (genes). We can see how genes get mixed in every particular case of crossover.

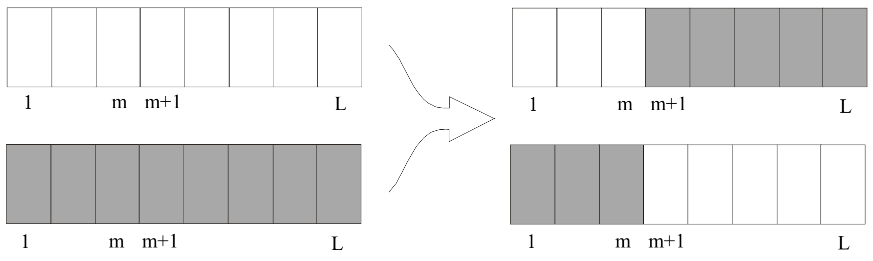


Figure 4 One-point crossover

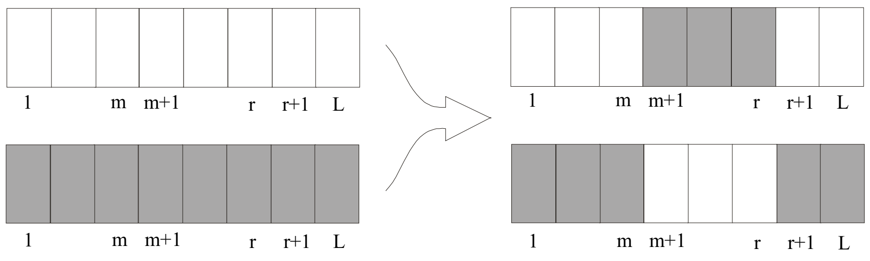


Figure 5 Two-point crossover

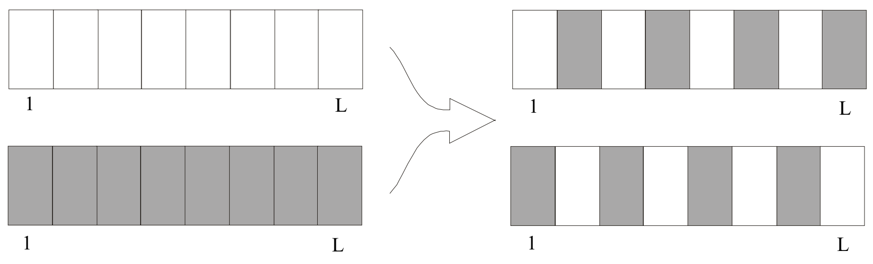


Figure 6 Uniform crossover

3.1.9 Mutation

Random mutation provides background variation and occasionally introduces beneficial material into a species' chromosomes. Without the mutation, all the individuals in population will eventually be the same (because of exchange of genetic material) and there will be no progress anymore. We will be stuck in local maximum as it is used to say. Mutation in case of binary string is just inverting a bit as shown in Figure 7.

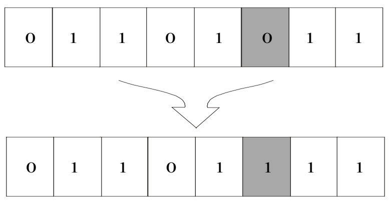


Figure 7 Mutation in binary string

3.1.10 Parameters for genetic algorithm

In order to converge, several parameters of genetic algorithm have to be fine-tuned.
Population size

Of course, considering algorithm's convergence, bigger population is better, because we have more genetic material for selection and reproduction. Changes to build better individuals are bigger. However, there is one important aspect to consider though- aspect of algorithm's speed. Bigger population means more evaluation. Every individual in population has to be measured in terms of fitness. Usually this is computationally most expensive procedure in genetic algorithm. Therefore, population size is kept about 32. Of course, it depends on evaluation complexity.

Mutation rate

It has to be small enough not to loose useful individuals developed so far and big enough to provide population with new genetic material at the same time. Different sources of literature suggest values between 0.1 and 0.01.There exists interesting relation between population size and mutation rate. When we lower population size, then it is useful to increase the mutation rate and vice versa.

Crossover rate

Normally, when two candidates are selected, they always do crossover.

Number of genetic generations

In other words, how many cycles we do before we terminate. It depends on task's complexity. Sometimes hundred generations is enough, another time thousands of generations isn't enough. It is probably wise to stop when no improvement in solution quality is made for certain time. Limit condition can be calculated taking account task's dimensions. For complex task, we may allow more generations.

When it is hard to determine appropriate a value for some parameter, it is good idea to use self-adaptation for that parameter. For example, we can change dynamically mutation rate along the progress of genetic algorithm: allowing mutations more often at the beginning and fewer mutations at the end, when solution needs only fine-tuning. Could be also that we need to use bigger mutation rate when progress has stopped, but we still are too far from plausible solution. Therefore, intelligent self-adaptation would be very promising.

3.2 How genetic algorithms work

In this section, questions like "What is though manipulated by genetic algorithm, and how to know that this manipulation (what ever it could be) leads to optimum or near optimum results for particular problem?" are answered.

3.2.1 Exploration engine: important similarities

Fundamental question has been not answered for long time. If in optimization process actually nothing else is used than values of fitness function, then what information could be extracted from a population of bit-strings and their fitness values? To clarify, let us consider bit-strings and fitness values in Table 3. This corresponds to black box output maximization problem in [31]. Function to optimize is $f(x) = x^2$.

String	Fitness
01101	169
11000	576
01000	64
10011	361

Table 3 Black box output maximization, inputs and fitness.

So, what information could be extracted from this population to guide exploration toward progress? Actually, there is not much: four independent chains of bits and their fitness values. When examining closer, we naturally start evaluate column of bit-strings and we notice certain similarities between strings. Studying these similarities in detail, we can see that certain bits seem to be very correlated with good fitness results. It seems that all strings starting with ‘1’ are within best ones for example. Could that be important element for optimization of the function? Answer is yes. As we know, maximal value of the function f(x) = x^2 is obtained when argument x is maximal. In our 5 bit case x can be 31 in decimal coding which corresponds to ‘11111’ in binary coding.

In conclusion, we must look for similarities between strings in a population. Then, we have to seek for cause – effect relation between similarities and improved performance. That way we have received a new source of information, which helps us guide exploration.

3.2.2 Schema concept

Since important similarities can help to guide exploration, we are interested how one string can be associated with another class of strings containing invariant in a certain positions. A framework defined by *schema concept* will answer more or less these questions.

Schema is a motive of similarity describing sub-set of bits in strings by similarities in certain positions, stated by Holland in [28]. To simplify we use just binary alphabet {0,1}, at the same time not loosing generality of discussion. We can easily introduce schemas by adding a special symbol ‘*’ to this alphabet, which stands for indifference. Now we can create strings based on this ternary alphabet {0,1, *}. Importance of schema comes clear when it is used as tool for (bit) pattern identification:
A schema identifies a string if there is correspondence in all positions in schema and in string under identification. Namely, ‘1’ matches ‘1’ and ‘0’ matches ‘0’ and ‘*’ can match both ‘1’ and ‘0’. Let us take an example. We consider strings and schemas with length 5. The schema *0000 corresponds to two strings {10000, 00000}. Schema 111 describes set of four elements {01110, 01111, 11110, 11111}. As we see, idea of schema lets us easily consider all the similarities between the strings of finite length on finite alphabet. Note that ‘’ is just a meta-symbol, it is never manipulated by genetic algorithm, it is just notation which allows describe all the potential similarities between strings of particular length and on particular alphabet.

Total number of similarities for previous example is 3^5, because each of five positions can take values ‘0’, ‘1’, or ‘*’. Generally, for the alphabet with cardinality of k (number of characters in alphabet) there exist $(k+1)^{l}$ schemas, where l is length of schema. So, what quantity of useful information we have received taking account similarities? The answer lies in number of unique schemas contained in population. In order to count these, we need to know strings in the population. What we can do is to count schemas related to one unique string; such a way we obtain number of upper bound of total schemas in a population.

For example, we consider a string with length of 5 “11111”. 2^5 schemas belong to that string, because each position can take its actual value or indifference symbol ‘*’. Generally expressed, a string contains 2^l schemas. Therefore, a population with size n contains between 2^l and $n*2^l$ schemas, depending on its diversity. As we see, it is much of information to help in guiding of exploration. Taking account similarities is justified. It is also obvious that efficient (parallel) treatment is needed if we want to use all that information in reasonable time.

Well, we know now the number of schemas, but how many of them are effectively treated by genetic algorithm? In order to answer that, we have to consider the effects of genetic operators (selection, crossover, and mutation) on development or disappearing of important schemas from generation to generation. Effect of selection on schema is easy to determine. Strings better adopted are selected with greater probability. Generally, schemas resembling the best ones are selected without loosing their advantage. Meanwhile, selection only is not capable of exploring new points in the space of search. Maybe crossover enters the game now. The crossover leaves schema intact when it is not cutting it, but in the opposite case it can degrade it. For example, let us consider two schemas ‘1***0’ and ‘**11*’. First schema has great changes to be destroyed by crossover (useful bits ‘1’ and ‘0’ will be probably separated). Second schema will probably be preserved (bits are close to each other). Consequently, schemas with short (useful) length are conserved by crossover, and are multiplied in quantity by selection operator. Mutation with normal (low) rate rarely destroys schemas as observations show. Schemas, which are well adopted and with short useful length, (*elementary blocks, building blocks*) are promoted from generation to generation. It is due to exponential growth of number of trials what are given to the best representatives. All this will happen in parallel, not requiring days of computing and much more memory than necessary for describing the population of n strings.

3.2.3 Schema theorem

So far, we have seen that there is great number of similarities to exploit in a population of strings. Intuitively we have seen how genetic algorithm exploits all the similarities contained in elementary blocks (schemas with good performance). Now we are going to look at these observations in a more formal way. We will find precise number of schemas in a population. We will formally study, which schemas will multiply in quantity and which, will decease in a generation. We will conclude with fundamental theorem of genetic algorithm.

To analyze development and degrading of great number of schemas contained in a population formal notations are needed. We are going to consider the effects of selection, crossover and mutation on schemas in a population.

We assume (without loosing generality) that we have binary strings on alphabet V = {0,1,*}. Where ‘*’ stands for indifference (0 or 1) as we saw in 3.2.2. Let us define strings by capital letters and positions in string with small letters. For example, string of seven bits A = 0111000 can be referred as:

$$A = a_1\, a_2\, a_3\, a_4\, a_5\, a_6\, a_7$$

Bit positions can be reordered, for example:

$$A' = a_3\, a_2\, a_1\, a_4\, a_5\, a_7\, a_6$$

Set of strings $A_{j,}\, j = 1,2\ldots n$, form a population A (t) in a time moment (or in generation) t.

Let us recall that there was 3^l schemas possible (similarities) based on binary string length *l*. Generally expressing, there was $(k+1)^l$ schemas for alphabet of cardinality of *k*. In addition, there was $n*2^l$ schemas in a population with size n, because each string itself contains 2^l schemas. This quantity shows importance of information treated by genetic algorithms. At the same time, in order to isolate elementary building blocks (for further investigation) in that vast quantity of information, we have to be able to make distinction among different type of schemas.

Not all the schemas are created to be equal. Some of them are more specific than others are. For example, schema '011*1**' tells more about of it's important similarities than schema '0******'. Even more, some schemas cover bigger part of the whole length of the string than others. For example, '1****1*' covers bigger part of string than '1*1****'. In order to quantify these notations, we are going to define two characteristics of schema: it's *rank* and *useful length.*

Rank of the schema H, o(H) is just number of the positions instantiated (0 or 1 in case of binary alphabet) in a sample. For example, for a '011*1**' rank is 4 (o (011*1**) = 4), but for '0******' rank is 1.

Useful length of schema H, δ(H) is the distance between first and last instantiated positions in the string.

For example, for schema '011*1**' useful length is 4, because last position instantiated is 5 and first position is 1, and the distance what separates them is δ(H) = 5-1 = 4. For another schema '0******', it is particularly simply to calculate. Because there is only one position instantiated, useful length will be 0.

The schemas and properties defined so far, are instruments, which allow formally study and classify similarities between strings. Moreover, it is possible to analyze effects of genetic operators on elementary building blocks contained in population. What we are going to do now is to consider the effects of selection, crossover and mutation (first separately and then combined) on schemas contained in population.

Effect of the selection on the expected number of schemas in a population is relatively easy to determine. Let us suppose that in given time moment t there is m exemplars of schema H in the population. We describe that m = m (H,t). Number of schemas can vary in different time moments. In case of selection, a string is copied into next generation according to its fitness function (degree of adoption). More precisely, a string A_i is selected with probability

$$p_i = \frac{f_i}{\sum f_i}$$

Having population with size *n,* we expect to have

$$m(H,t+1)=m(H,t)*n\frac{f(H)}{\sum f_i} \qquad (1)$$

representatives of schema in a population in a time moment t + 1, where f(H) is average fitness of the string represented by schema H in a time moment t.

As average fitness of the population can be expressed as following:

$$f=\frac{\sum f_i}{n},$$

then we can rewrite expression (1) as following:

$$m(H,t+1)=m(H,t)\frac{f(H)}{f} \qquad (2)$$

Clearly saying, any schema develops (reproduces) with the speed, which is equal (proportionate) to the ratio of schema's average fitness and population's average fitness. In other words, schema, which has its fitness value greater than population's average, will receive more copies in next generation. The same time, schemas which fitness values are less than population's average, receive smaller number of copies in next generation. It is interesting to observe that this phenomena occurs in parallel for each schema H contained in population. In other words, all the schemas in a population multiply or decease according to their fitness thanks selection operator only. This is important.

Effect of the selection on number of schemas is clear now from qualitative point of view. Maybe it is possible to estimate quantitative aspect to. Let us suppose that we have a schema H which has quantity of *cf* copies, what is more than average; *c* is a constant. Now it is possible to rewrite the equation of schema development (2) as following:

$$m(H,t+1)=m(H,t)*\frac{(f+cf)}{f}=(1+c)*m(H,t) \qquad (3)$$

When starting at time moment t = 0 and assuming that c a constant we obtain equation:

$$m(H,t) = m(H,0)*(1+c)^t \qquad (4)$$

We can recognize (4) as geometric progression, discrete analogue of exponential growth. Effect of selection is now clear quantitatively as well: it exponentially allocates space in a population to schemas which are better than average. Next, we are going to study in witch way crossover and mutations affect this allocation of space.

Although, selection is surprisingly powerful, actually by itself it does not promote exploring new regions in search space. Therefore, no new points are found. If we are just copying old structures without changing them, how can we obtain something new? That is the reason why crossover enters the game. The crossover is an exchange of information between the schemas, which are structured into random parts, consequently. Crossover creates new structures avoiding as much as possible disturbing the allocation strategy dictated by selection only.

Therefore, it conserves the proportions of schemas exponentially increasing (or decreasing) in population.

In order to see which schemas are affected by crossover and which are not, let us consider a string of the length $l = 7$ and two schemas represented by that string:

$$
\begin{aligned}
A &= 0\ 1\ 1\ 1\ 0\ 0\ 0\\
H_1 &= *\ 1\ *\ *\ *\ *\ 0\\
H_2 &= *\ *\ *\ 1\ 0\ *\ *
\end{aligned}
$$

It is evident that schemas H_1 and H_2 are represented in the string A. Recall, that simple (one – point) crossover is carried out by randomly selecting another string, randomly selecting a crossover point, and then exchanging sub-parts between two strings started from crossover point until the end of the strings (see Crossover). Let us suppose that string A was selected for crossover. There are six positions in that string for possible crossover point. Let us suppose, randomly selected crossing point is between positions 3 and 4. Effect of crossover is visible in next example, where crossover point is marked with separator:

$$
\begin{aligned}
A &= 0\ 1\ 1\ |\ 1\ 0\ 0\ 0\\
H_1 &= *\ 1\ *\ |\ *\ *\ *\ 0\\
H_2 &= *\ *\ *\ |\ 1\ 0\ *\ *
\end{aligned}
$$

We can see that schema H_1 will be destroyed, because '1' in the second position and '0' in last position will be separated. It is also obvious that in case of the same crossover point (between positions 3 and 4), the schema H_2 will survive. It is because '1' in fourth position and '0' in fifth position will be kept together in the same descendant (child). It is clear that schema H_1 has less change to survive than schema H_2, because the point of cut has more change to fall between extreme positions. To quantify that observation, we remark that schema H_1 has useful length of 5. If the point of crossover is chosen uniformly or randomly among the l-1 = 7 – 1 = 6 positions possible, then its evident that the schema H_1 has following possibility to be destroyed:

$$p_d = \frac{\delta(H)}{(l-1)} = \frac{5}{6}$$

(There is probability to survive $p_s = 1 - p_d = 1/6$). The same way, the schema H_2 has a useful length of $\delta(H_2) = 1$, and it is destroyed only in case of unique event among of 6 possible, when cutting point occurs between positions 4 and 5. So $p_d = 1/6$ and probability to survive is $p_s = 1 - p_d = 5/6$.

More generally, we can find a lower bound of the probability p_s for any schema. Schema survives when crossover point falls exterior of its useful length, probability to survive in case of simple crossover is

$$p_s = 1 - \frac{\delta(H)}{(l-1)}$$

at the same time, schema has great chances to loose when cutting point (chosen within l – 1 possible points) drops into its useful length. If crossover takes place with probability p_c then for any pairing, the probability to survive is can be expressed as following:

$$p_s \geq 1 - p_c * \frac{\delta(H)}{(l-1)}$$

The expression remains actually same because we have crossover probability $p_c = 1$.

Now we can consider combined effect of the selection and crossover. Supposing that selection and crossover operations are independent, we obtain following estimation:

$$m(H,t+1) \geq m(H,t) * \frac{f(H)}{f}\left[1 - p_c \frac{\delta(H)}{l-1}\right] \quad (5)$$

Let us compare expression (5) to expression (3), obtained earlier for selection only. We see that number of schemas expected to be in next generation is multiplied by surviving probability p_s during crossover. Combination of selection and crossover increases number of schemas. Again, effect of operators has revealed- the schema H develops or degrades in function of certain factor. Considering selection and crossover together, this factor depends on two things:

1) is the schema below or over the of population's average
2) is the useful length of schema short or long

Evidently, the schemas with fitness over average and with short useful length are handled with exponential speed.

The last operator to take account is mutation. Mutation modifies randomly (with probability p_m) one position in a given string. In order to schema H could survive, all its instantiated positions have to survive. In addition, probability that instantiated position remains unchanged (survives) is $1 - p_m$, and as every mutation is statistically independent from others, schema survives only if every position of its o (H) insatiate positions survives. Multiplying the surviving probability $1 - p_m$ with it self o (H) times, we obtain *surviving probability against mutation*:

$$(1 - p_m)^{o(H)}$$

As p_m is small ($p_m << 1$), then we can approximate last expression as $1 - o(H)*p_m$

In conclusion, we can say that *expected number of copies* a schema H receives due selection, crossover and mutation can be given by following equation:

$$m(H,t+1) \geq m(H,t) * \frac{f(H)}{f}\left[1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m\right]$$

Added mutation affects little our previous conclusions.

Short schemas with small rank and with fitness greater than average are objects of under manipulation in genetic algorithm, they receive an exponentially increasing number of trials in successive generations. This conclusion is so important that there is given a special name to it: *Schema Theorem* [28], [31].

3.2.4 Implicit parallelism

We have seen so far that the number of schemas processed in a population is between 2^l and $n*2^l$, where l is length of strings and n is population size. At the same time, we know now that certain schemas are not processed with such a high probability, because crossover destroys those schemas with relatively big useful length. It is showed in [31] that the number of schemas, which are effectively being processed in each generation, is of the order n^3, where n is the population size. This property is one of the explanations for the good performance of genetic algorithms.

3.2.5 Building block hypothesis

The power of genetic algorithm comes much more clear from viewpoint of schema concept. Short schemas with low rank and with good fitness are selected, recombined and re-selected again leading to formation of individuals with even more improved performance. Such schemas are so important that they are called elementary blocks or *building blocks*.

Working with building blocks reduces search space. Instead of constructing efficient strings by trying all the combinations possible, only very best strings are constructed based on parts of best solutions tested before. Idea, that building blocks combine in order to form better individuals, seems quite logical. However, how to be sure it is right? There are many empirical confirmations starting with Bagley and Rosenberg 1967, Greffenstette 1985, 1987; building block hypothesis is confirmed in many domains. Regular, unimodal, multimodal, combinatorial optimization problems are successfully solved with similar type of selection, crossover and mutation operators as considered here so far. Bethke (1981) has researched this subject using Walsh functions and string transformations. His analytic method enables to determine if algorithm reaches optimum (or near optimum) result for given fitness function and coding (are the building blocks combining properly). Holland (1987b) extended that work by analyze of the average of the schema when population is not uniformly distributed.

Practical aspects

From practical point of view, it is important to follow the imperative, that simple genetic algorithm depends on recombination of building blocks in order to find the best solutions. If the building blocks are fuzzy because (improper) coding or because of (bad) evaluation function itself, algorithm may take long times for achieving optimal solutions.

Successful coding scheme is such which encourages the formation of building blocks by ensuring that:

1) related genes (positions) in individual (string) are close together (short useful length),
2) there is little interaction between genes

Interaction, (often referred as *epistasis*) between genes means that the contribution of the gene to the fitness depends on the value of other genes in the individual. In fact, there is always some interaction between genes in multimodal fitness functions (most real functions of interest).

Unfortunately, conditions (1) and (2) are not always easy to meet. Genes may be related in a ways which do not allow all closely related ones to be placed together in a one – dimensional string (if they are related hierarchically for example). In many cases, the exact nature of the relationships is not known to the programmer, so even there are simple relationships, it is still impossible to arrange the coding to reflect this.

3.2.6 Conclusions

In this section we have seen why genetic algorithm works, why it has such a performance. We have used schema conception (model of similarities) for more formal analyze. Most important results conclude in fundamental theorem of genetic algorithms (*Schema Theorem* [28],[31]), which states that short schemas with small rank and with fitness greater than average are objects under manipulation in genetic algorithm, they receive an exponentially increasing number of trials in successive generations. This happens because selection allocates greater number of copies to the best schemas, and because simple crossover does not change short schemas too often. Mutation, at the same time, occurs relatively rarely so it has minor effect on important schemas.

Manipulating similarities (schemas), genetic algorithm reduces complexity of any given problem. Short schemas with low rank and with good fitness are selected, recombined and re-selected again leading to formation of individuals with even more improved performance. Such schemas are so important that they are called elementary blocks or *building blocks*. New solutions are constructed based on parts of great number of combinations of best solutions tested before. *Building block hypothesis* says implicitly that elementary building blocks surely lead to improved performance.

It is important that constructive building blocks are contained in the string, which can be processed by the genetic algorithm. Accordingly, the combination of the fitness function, encoding of the problem parameters (e.g., real or binary), and type of crossover (e.g., uniform or n-point) must ensure the presence of these building blocks.

Genetic algorithms are suitable for solving a test generation problem from building blocks imperative's point of view. In that case individual will be test vector, which in essence is binary string. Test set will be a population. Everything we have discussed above (good performance, etc.) applies to test generation task as far as we can find appropriate fitness function.

3.3 Genetic algorithm for combinational circuit testing

In this section, the first aim was to compare genetic test generator with random generator, because it is known that in essence, genetic algorithm uses much of random numbers. Therefore, the algorithm is designed so that it allows direct comparison with random method. Some basics of gate level test development are recalled also. Genetic operators are revisited in terms of test generation. Example of crossover on hypothetical circuit is presented. Finally, results of experiments are presented.

3.3.1 Introduction

During recent years, great effort is put to overcome test generation complexity problem. Artificial intelligence methods are therefore gained much of attention. One among these techniques is evolutionary algorithms or often referred as genetic algorithms.

Earlier genetic approach for combinational circuits was represented in [11]. The key feature there was the method of monitoring circuit activity. Namely, information about the activity of internal nodes during fault simulation was collected, and points in the circuit where fault propagation was blocked where identified. Based on that information fitness values for test vectors were calculated. Modified crossover operator, which swaps useful parts (identified before) of two individuals was used.

3.3.2 Fault model

Stuck-at fault model is used here. There are two kinds of faults possible: stuck-at-1, what we will identify as *≡1*, and stuck-at-0, identified as *≡0*. In first case, due to some reason (fabrication defect for ex.), logic level of signal remains constantly "high". In second case, logic level of signal remains constantly "low". Stuck-at faults of AND gate are considered in Figure 9 and stuck-at faults of OR gate are represented in Figure 8.

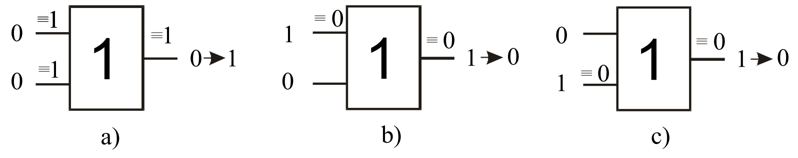


Figure 8 Stuck-at-faults detected for OR gate

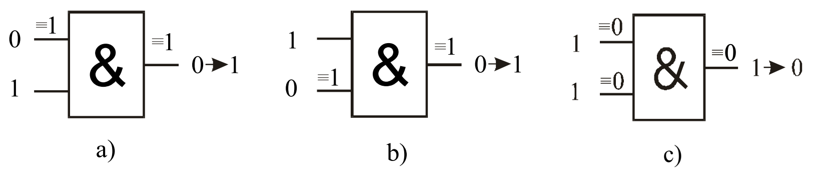


Figure 9 Stuck-at- faults detected for AND gate

We see that, in order to detect ≡1 in input of AND gate (Figure 9, a)), we have to apply *vector* ‘0 1’. Normal response to such input vector should be ‘0’ in gate’s output. But due to fault, both inputs of gate are ‘1’ now, which in turn gives ‘1’ in output. In other words, output value has changed due to presence of fault. *Fault effect* on the component has revealed. We say, that *fault is detected* when fault effect can be observed. However, there is a problem to consider.

We can’t measure directly output of the gate under test, as fabricated integrated circuit is actually black box for us in that sense that we can manipulate its inputs and we can observe its outputs only. We cannot measure its inner points. Therefore, in order to be observable, fault effect must be *propagated* into circuit output- logical values of inner points of circuit have to have certain values.

In conclusion, in order to detect a fault in a circuit, we apply test vector (test impulse) to circuit inputs and observe circuit’s outputs. We know advance what should be output values in case of good circuit (by logical simulation). Therefore, if circuit output values are different, it is said that fault in the circuit is detected by this vector. Good test vector contains two properties at same time: 1) applies appropriate values to component under test, 2) propagates fault effect to circuit values

3.3.3 Representation

In a genetic framework, one possible solution to the problem is called an *individual*. As we have different persons in society, we also have different solutions to the problem (one is more optimal than the other). All individuals together form a *population* (society).

In context of test generation, test vector (test pattern) will be the individual and the set of test vectors will correspond to population.

3.3.4 Initialization

Initially, a random set of test vectors is generated. This set is subsequently given to a simulator tool for evaluation. For research purpose, random initializing a population is good. Moving from a randomly created population to a well adapted population is a good test of algorithm, since the critical features of final solution will have been produced by the search and recombination mechanisms of the algorithm rather than the initialization procedures. To maximize the speed and quality of the final solution it is usually good to use more direct methods for initializations. For example, output of another algorithm can be used or human solution to the problem.

3.3.5 Evaluation of test vectors

Evaluation is used to measure the fitness of the individuals, i.e. the quality of solutions, in a population. Better solutions will get higher score. Evaluation function directs population towards progress because good solutions (with high score) will be selected during selection process and poor solutions will be rejected.

We use fault simulation with fault dropping in order to evaluate the test vectors. The best vector in the population is determined and added to the selected *test vector depository*. The depository consists of test vectors that form the final set of test vectors. By adding only one best vector to the depository, we assure that the final test set will be close to minimal.

3.3.6 Fitness scaling

As a population converges on a definitive solution, the difference between *fitness* values may become very small. Best solutions cannot have significant advantage in reproductive selection. We use square values for test vector's fitness values in order to differentiate good and bad test vectors.

3.3.7 Selection of candidate vectors

Selection is needed for finding two (or more) candidates for crossover. Based on quality measures (weights), better test vectors in a test set are selected. Roulette wheel selection mechanism is used here. Number of slots on the roulette wheel will be equal to population size. Size of the roulette wheel slots is proportional to the fitness value of the test vectors. That means that better test vectors have a greater possibility to be selected. If our population size is N and N is an even number, we have N/2 pairs for reproduction. Candidates in pair will be determined by running roulette wheel twice. One run will determine one candidate. With such a selection scheme, it can happen that same candidate is selected two times. Reproduction with itself does not interfere. This means the selected vector is a good test vector and it carries its good genetic potential into new generation.

3.3.8 Crossover

From pair of candidate vectors selected by roulette wheel mechanism, two new test vectors are produced by one-point crossover as following (see Figure 10):

1) we determine a random position *m* in a test vector by generating a random number between 1 and *L*, assuming that *L* is the length of the test vector
2) first *m* bits from the first candidate vector are copied to the first new vector
3) first *m* bits from second candidate vector are copied to the second new vector
4) bits *m + 1 … L* from first candidate vector are copied to second new vector (into bits *m + 1...L*)
5) bits *m + 1 … L* from the second candidate vector are copied to the first new vector (into bits *m + 1...L*)

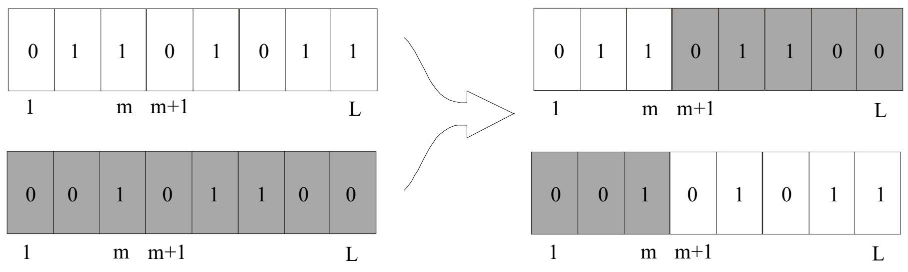


Figure 10 One – point crossover

C language implementation of one-point crossover is given in Figure 11:

1) First two candidate vectors in population are selected.
2) Then, crossover point (bit position) for them is determined.
3) Next, exchange of bits (genetic material) between carries out. Leftmost bits from first vector up to cutting point are replaced by leftmost bits from second vector.
4) Then bit exchange continues in positions after crossing point.

Steps 1 to 4 are carried out over the population with increment value 2, see *for* cycle. This means, two parents always produce two children (new vectors). Note that therefore population size has to be even number, too. New vectors are put into new population. Finally, current population is replaced entirely with new population. There exists actually a strategy where old population is replaced only partially.

```
Void crossover()
  {
    int l,k;
    int cross_point,mate1,mate2;
    char **temp_ptr;

    for (k=0; k < popul_size; k+=2)
    {
      mate1=select_individual();    // select one vector
      mate2=select_individual();    // select other vector

     // determine crossing point of the two vectors

      cross_point = floor (Random_btw_0_1() * InpCount);


     // exchange of bits until cutting-point


      for (l=0; l< cross_point; l++)
        {
          new_popul[k][l]  = popul[mate1][l];
          new_popul[k+1][l]  = popul[mate2][l];
        }

     // exchange of bits after cutting-point
```

```
          for (l=cut_point; l < InpCount; l++)
            {
              new_popul[k][l] = popul[mate2][l];
              new_popul[k+1][l] = popul[mate1][l];
            }
        }

        // swapp current and new population,
        // new built pop. Becomes current one

           temp_ptr = popul;
           popul=new_popul;
           new_popul = temp_ptr;

    }
```

Figure 11 C code implementation of one-point crossover

3.3.9 Mutation in test vectors

Random mutation provides background variation and occasionally introduces beneficial material into a species' chromosomes. Without the mutation, all the individuals in population will eventually be the same (because of the exchange of genetic material) and there will be no progress anymore. We will be stuck in a local maximum.

In order to encourage genetic algorithm to explore new regions in space of all possible test vectors, we apply mutation operator to the test vectors produced by crossover. In all the test vectors, every bit is inverted with a certain probability *p.* It is also possible to use a strategy where only predefined number of mutations are made with probability *p*=1 in random bit positions. This should reduce the computational expense. However, experiments showed decrease in fault coverage. Therefore, this method is not used here.

3.3.10 Getting uniform random numbers

Although genetic algorithms are not pure random algorithms, they use extensively random numbers. Genetic algorithm needs many random numbers in order to converge. Unfortunately in C language built-in *rand* function is entirely inadequate in circumstances where thousands- or even millions- of random values need to be generated.

Instead, combination of two random number generators based on L'Ecuyer's algorithm, is used here. Which gives us a period of approximately $2.3 * 10^{18}$. Details of algorithm can be found in [32].

3.3.11 Test generation algorithm

Steps of selection, crossover and mutation are repeated until all the faults from the fault list are detected or a predefined limit of evolutionary generations is exceeded. Test generation terminates also when the number of noncontributing populations exceeds a certain value. The value depends on the circuit size and is equal to *Number of inputs / const,* where const is a constant that can be set by the user. The smaller the value of *const*, the more thoroughly we will search. In current implementation, the test generation works in two stages, with different mutation rates:

1) In the first stage, when there are many undetected faults and fitness of vectors is mostly greater than zero (in each evolutionary generation many faults are detected), a smaller mutation rate is used ($p = 0.1$).
2) In the second stage, when there are only few undetected faults and none of the vectors in population detects these faults, the weights of the vectors will all be zeros. We cannot say which vector is actually better than others. Now the mutation rate is increased ($p = 0.5$) to bring more diversity into population, in order to explore new areas of the search space.

Test generation algorithm is represented in Figure 12.

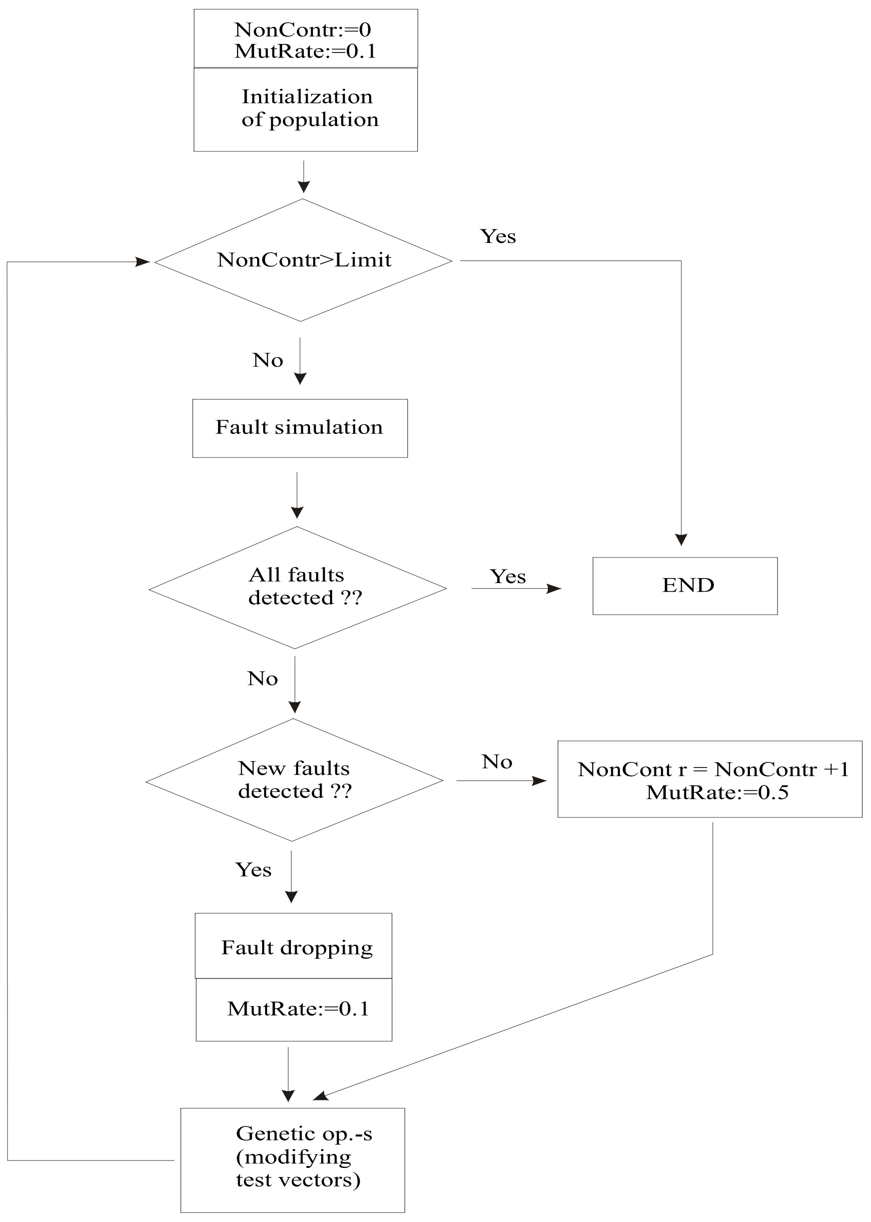


Figure 12 Genetic test generation for combinational circuits

3.3.12 Experimental results

The experiments with the genetic program developed above, were partly aimed at showing how much is the genetic approach better than random. In order to achieve that, same simulation procedures were used for random and genetic test generation. Population size for the genetic test generator was set to 32. It is a tradeoff between speed and fault coverage. In each (evolutionary) generation, or step, one vector from 32 is selected and put into final test set (vector depository). The random test generator performs in a similar way. It generates patterns in packages of 32 vectors. The best vector from the package (based on simulation results) will be selected, if it detects some previously not detected faults. Therefore, we can compare the two methods adequately. Both of the test generation tools belong to the diagnostics software package Turbo Tester [33]. All of the experiments were run on a Sun SparcStation 20 computer.

The experiments were carried out on ISCAS'85 benchmarks [34]. In first experiment, minimum number of test vectors was determined to detect all detectable faults. It comes out that genetic method requires always fewer test vectors (patterns) to yield the same fault coverage than random method. For the 'hard-to-test' circuit c2670, equal number of test vector simulations for both methods was taken and then the fault coverage reached was estimated. Genetic method

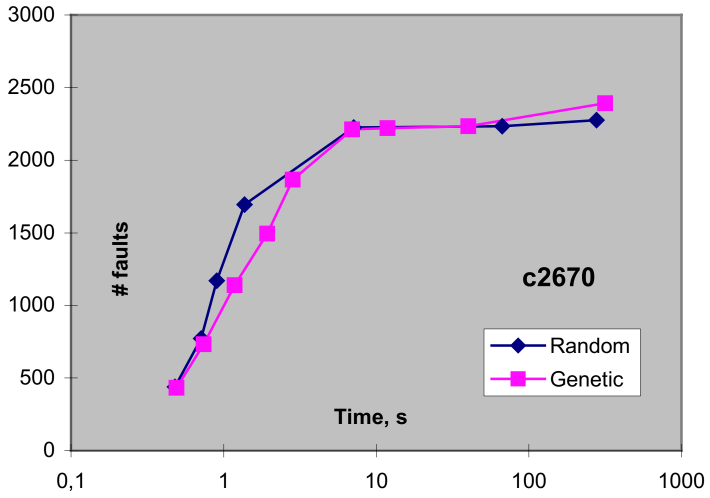


Figure 13 Fault detection in time. Circuit c2670

ISCAS 85 benchmarks			Genetic Test Generator			
circuit	total faults	det.able faults	Det.ed faults	patt. Sim.	Tests	time,s
c432	616	573	573	2048	46	0.93
c499	1202	1194	1149	4096	85	2.74
c880	994	994	994	4096	54	2.09
c1908	1732	1723	1723	8192	126	7.33
c2670	2626	2508	2393	138016	85	316
c3540	3296	3149	3149	19200	149	28.5
c5315	5424	3564	5364	6400	120	21.46
c6288	7744	7693	7693	2048	23	20
c7552	7104	6969	6834	787008	226	3600

Table 4 Results for genetic test pattern generator

ISCAS 85	Random Test Generator			
circuit	det.ed faults	patt. Sim.	Tests	Time, s
c432	573	2240	49	0.84
c499	1149	4800	84	2.69
c880	994	8416	63	3.83
c1908	1723	10880	131	8.8
c2670	2275	138016	84	278
c3540	3149	46400	167	65.48
c5315	5364	33600	132	75
c6288	7693	960	24	23.54
c7552	6801	787008	211	3180

Table 5 Results for random test pattern generator

ISCAS 85 benchmarks		CRIS [11]			Genetic Test Generator		
circuit	det.able faults	patt. Sim.	Det.ed faults	tests	patt. Sim.	Det.ed faults	tests
c432	520	3674	519	72	2048	520	46
c880	942	5309	937	229	4096	942	54
c499	750	3152	749	553	4096	750	85
c1908	1870	4501	1852	253	8192	1870	126
c3540	3291	8000	3277	452	19200	3291	149
c5315	5291	8000	5258	682	6400	5291	120
c6288	7709	2822	7709	131	2048	7709	23

Table 6 Results for genetic test pattern generator and comparison with CRIS

discovers 118 faults more than random in the case of c2670 and 33 faults more in the case of c7552. Execution times for the random method were slightly shorter for smaller circuits like c432 and c499.

Subsequently, fault detection in time for random and genetic generators was investigated. One of the result graphs for bigger circuit is represented in Figure 13. Random generator achieves good fault coverage sooner but genetic generator detects additional faults in the end. Except for the smallest circuits c432 and c499 as we see in Table 4 and Table 5.

Effectiveness of genetic generator comes evident in case of circuits that have a large number of inputs. Results obtained here were compared to the ones achieved in [35], where the key feature was keeping certain inputs together (in order to better propagate fault effects) during reproduction process. The method detected all faults for c7552 and c2670. However, the approach given here uses (up to 2 times) less of test vectors for all circuits. There was not possible to compare execution times, because they were not revealed.

In addition, results here were compared to the genetic approach in [11]. The key feature of the latter method is monitoring circuit activity. Namely, information about the activity of internal nodes during fault simulation is collected, and points in the circuit where fault propagation was blocked are identified. Based on that information fitness values for test vectors are given. The comparison between approach here and [11] is presented in Table 6. It is evident that logic simulation and such a monitoring used in [11] is not effective.

Simple fault simulation based approach given here detects all detectable faults with a smaller time for all circuits and generates 1,6 – 6,5 times less test vectors than [11]. Comparison was not adequate for circuits c2670 and c7552 because in [11] the test generation was terminated too early.

3.3.13 Conclusion

Comparison of random and genetic test generators reveals that test sets of genetic generator are always more compact. During genetic test generation dynamic test vector ‘packing’ occurs because vectors are carefully chosen all the time. This was first remarkable result. Second interesting observation was that genetic generator performs better than random in last stadium in test generation when only hard-to-test faults are left. Shortly, genetic test generator is justified for large circuits. Tracing a program log file revealed that fitness values of individuals too often coincide. Therefore more accurate fitness function should be used. More information from fault simulation should be incorporated into fitness function. Simulation procedure should be modified. New fitness function could take following form:

$$f = C_d \cdot faults\_\det ected + C_a \cdot faults\_activated + C_p \cdot num\_of\_propagation\_steps$$

where C_d , C_d and C_d constants (weights).

Another experiment could be done using strategy that after fault dropping, entirely new population could be created for detecting new faults. Assumption is made here that new faults are ‘situated’ in different region in search space. However, some ‘seed’ vectors could be maintained from old population. This ensures that new search region will not be too far away from previous.
Finally, in a stage of test generation, where fitness comes to zero (new faults were not detected), then uniform crossover could be used instead one-point crossover. It is known that uniform crossover is highly disruptive. It mixes bits in vectors. Other words, it encourages exploration.

3.4 Genetic algorithm for Finite State Machine testing

In this section, first we give short overview of FSM testing. Second, we introduction to finite state machines. Fault model and test sequences are discussed. Difficulties faced in FSM testing are pointed out, third we develop systematically a genetic algorithm for FSM testing, and finally comparative experimental results with other approaches are given and future work is discussed.

3.4.1 Introduction

Finite state machine can be regarded as sequential circuit. Test generation for sequential circuits has been investigated widely and have been recognized as a difficult problem [36,1]. Traditional gate level test generation algorithms use an iterative array model where each time frame is represented by a cell of combinational logic. A single fault in the circuit is treated as a fault in each cell of the iterative array. Many of these algorithms use techniques developed for combinational circuits, which are applicable to this model. The complexity of test generation is very high because of line justification and fault propagation generally require multiple time frames. For large circuits these approaches have been time consuming because of the great number of backtracking.

Several sequential circuits test generation algorithms exploit high-level information about circuits and use the finite state machine (FSM) as a model. Some of these algorithms use concepts of checking experiments. Others use the FSM model to get justification and propagation sequences for a fault during gate level test generation. There are approaches, which obtain a fault-independent sequence using FSM model. Further, this sequence is improved by adding another one to detect faults introduced in the gate-level implementation of FSM and not detected in first phase [37].

Functional approaches based on branch testing in state transition diagrams (STD) [38] are more effective than structural approaches, however the fault coverage of test sequences generated in relation to realistic structural faults remains open.

New hierarchical technique of generating tests for sequential circuits represented by finite state machine (FSM) was proposed in [37]. It is assumed that high-level information in terms of FSM along with a gate-level description is available for circuits. The method is based on using decision diagrams (DD). For describing the function, structure and faults in FSM, three levels are used: functional (state transition diagrams), logic (or signal path) and gate levels. For each level uniform procedures based on DDs were elaborated. Faults from different classes are inserted and activated at different levels by these procedures. The results on synthesis benchmark circuits show that high stuck type fault coverage can be obtained with this technique.

In current section of thesis, we are going to integrate that hierarchical technique with genetic algorithms. Genetic algorithm interacts with simulation procedure, which is used to evaluate new test sequences. Genetic program described below can work standalone or together with the test generator introduced in [39]. Evolutionary program tries to detect faults, which had remained undetected.

3.4.2 Definition of the finite machine

A *finite state machine* is defined by the 6-tuple M=<S,I,O,δ,λ,s_0>, where S is a finite set of states, I is a finite set of input values, O is a finite set of output values, δ is a state transition function, λ is an output function, and s_0 is the initial state [37].

FSMs are modelled as either a *Mealy* or *Moore* machine. The next state transition function δ: S×I→S', maps the present state and the input into the next state, where S'=S∪{∅}. ∅ allows representing an unspecified next-state. The output function for Mealy machine is such as λ: S×I→O. It maps the present state and the input into the output. For Moore machine we have λ: S→O.
A FSM is traditionally represented by its state transition diagram (STD) (Figure 14a).

3.4.3 Decision Diagrams for FSM

There are different ways to represent finite state machine:

3) in the structural way by a circuit which can be decomposed into a combinational part and a memory part (a set of flip-flops)
4) in the functional way by STDs.

The output functions and transition functions of the FSM are Boolean, and therefore can be represented by Structurally Synthesized BDDs (SSBDD) [40]. For the second case, we use a general form of decision diagrams [41], which exploit integer variables for representing inputs, outputs and internal states of the FSM. In these graphs, node variables may have, in general, more than two values. There exists a one-to-one correspondence between the values of a node variable and the successors of the node. The number of successors for each node can be more than two (differently from the binary case). Internal nodes of DDs are labelled by state and input variables. Terminal nodes are labelled by constants and their values correspond to internal states or output states of the FSM.

Two extreme cases can be considered when representing FSMs by DDs:

5) the case of an abstract FSM for which, we have two DDs to represent the transition and output functions respectively,

6) and the case where inputs, outputs and internal states of the automata are binary coded, and we can represent it by a set of Boolean output and transition functions. Mixed cases can be placed between these two extremes.

As an example, two representations of the benchmark circuit *dk27* are given in Figure 14: a state transition diagram and its corresponding DD representation. The DD represents the behaviour of the FSM:

$$Q = F (q', xl),$$

where Q is next state variable. By q' we denote the previous state variable. The input of the FSM is structured. Terminal nodes are labeled by constants, which represent the new state of the FSM. The illegal states of the FSM are specified by q = *.

There are two interesting properties of general DDs:

7) similarity in the representation form with SSBDDs, that easily allows to generalize methods developed for the logic level as well to higher functional (state transition) levels;
8) in DDs, only one model in the form of graph is used whereas STDs consist in two models – a graph for representing transitions between states and Boolean expressions to give the branching conditions.

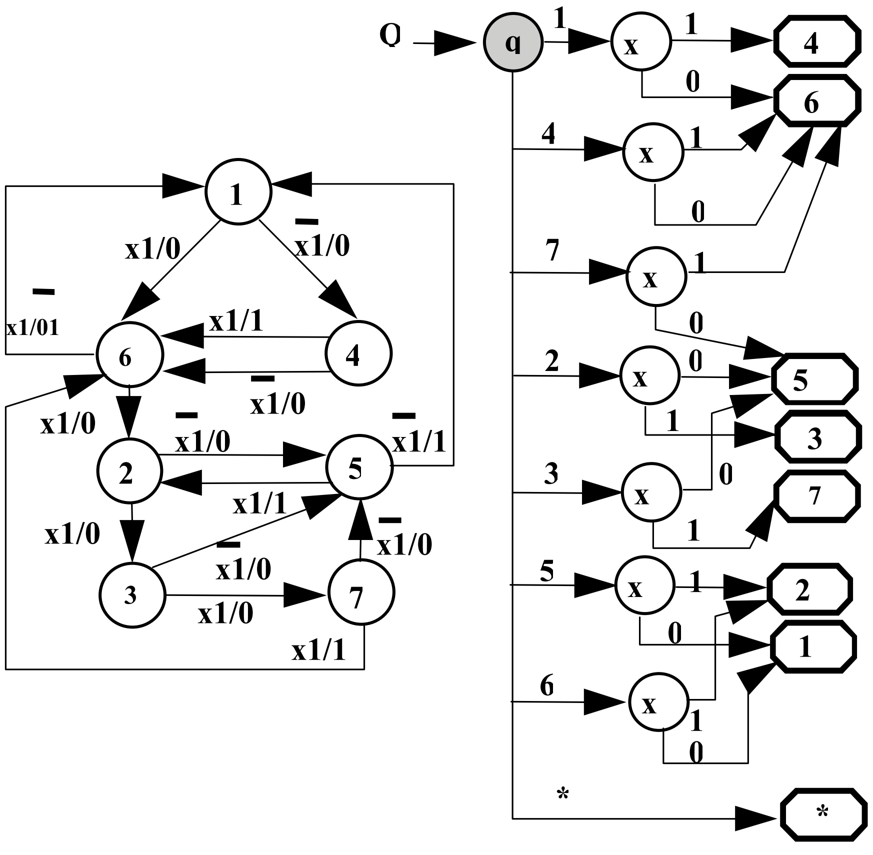


Figure 14 STD and DD for benchmark circuit dk27

3.4.4 Fault classes

Following fault classes are considered here:

a) transition faults that effect on transition conditions
b) input faults that effect on the input
c) state faults that effect on the state

3.4.5 Limitations

In this approach, no observability at the output of the state registers is required. No complete or partial scan path insertion is needed. No design modifications are used. However, the reset state from the FSM is required. In other words, the simulation procedure assumes that the fault-free machine has always a predetermined initial state.

3.4.6 Test sequence

The *test sequence* for a single fault consists of three sub-sequences:

1) *initialization sequence*, which brings the FSM from current state to the state needed for activation the fault
2) *activation sequence*, which contains only one additional input pattern, needed for the fault activation
3) *fault propagation sequence*, which is a ordered set of *state-pairs,* differentiating the good destination state from faulty destination states, and thus, propagates the fault effect to the primary output

All these sub-sequences are created at the functional level (behavioral level) […]. Only fault activation and fault propagation procedures for *transition faults* are carried out at the structural level (corresponds to gate level). However, also for transition faults, after they have been activated at the lower structural level, the results can be easily transformed to the functional level by specifying the input and internal states needed for fault activation.

The *necessary but not sufficient condition to create a test* is traversing a set of paths that contains all branches in the DD. If not all faults are tested yet by this sequence, we have to find new set of branches in order to activate the remaining faults. Thereafter, we have to traverse according set of paths that contain all these branches.

During simulation we try the given test sequences by dropping and activating faults along a set of control flow paths that contains all branches in the DD-model.

3.4.7 Problems in FSM testing

The difficulty in generating tests for finite state machines lies in two aspects:

9) Setting the flip-flops to certain states, in order to activate fault under test
10) Propagating the fault effect to primary outputs

What is needed in both cases is a sequence of several vectors. The greater the length of the shortest input sequence needed is, the more difficult it is to find. Feedback, which we have in FSM, makes these sequences especially long. Here is the point where genetic algorithms can overcome the difficulty. Genetic algorithm reduces dramatically the number of trials needed to find appropriate vector sequence.

3.4.8 Representation for genetic algorithm

In Holland's [28] (and his follower's) work, individuals are bit strings consisting 1's and 0's. Bit strings have been shown capable to represent of usefully variety of information, and they have been shown effective representations in unexpected domains. The properties of bit string representations for genetic algorithms have been extensively studied, and a good deal is known about genetic operators and parameter values that work well with them. According to *Schema theorem* (see 3.2.3), bit string coded implementations do have good performance.

Bit string representation is very natural for representing input vectors for FSM. However, in essence FSM is sequential circuit. It includes feedback. It can take several time frames (cycles) in order to achieve required state. When in case of combinational circuit we needed only one input vector to detect a certain fault, then in case of FSM we normally need several vectors (a sequence) in order to detect a fault.

The question is should we still consider one vector as an individual or should we take a whole sequence as an individual. Let us recall that in the genetic framework, individual corresponds to one possible solution to the problem. In population, we have many individuals, other words many solutions. We have to estimate goodness (fitness) if each of them. Before choosing representation for individual we have to consider, what possibilities we have for estimation of goodness in a case of single vector and sequence of vectors. In order to estimate goodness, there have to be a condition to be fulfilled. Other words there have to be fitness function for individual to measure its fitness.

What could be a fitness function for one single vector? What is a result of applying the vector to FSM's inputs? In the best case, FSM changes to new state and new outputs are available. However, it is quite likely that a single vector does not activate any fault, nor does detect one. Here, the ultimate goal is to detect a fault or several faults in FSM. As only several vectors applied successively can do that, choosing a vector sequence as an individual is justified.

However, the question remains how big should be the length of the sequence. We discussed that there is greater possibility to detect a fault by the longer sequence, but these sequences must be evaluated, too. Simulation, however can be quite time consuming (computationally costly). So, trade off must be made here.

Now, as sequence of vectors corresponds to individual, then *population* will be a set of such sequences in genetic algorithm. Suitable representation for C language implementation using linked lists is shown in Figure 15.

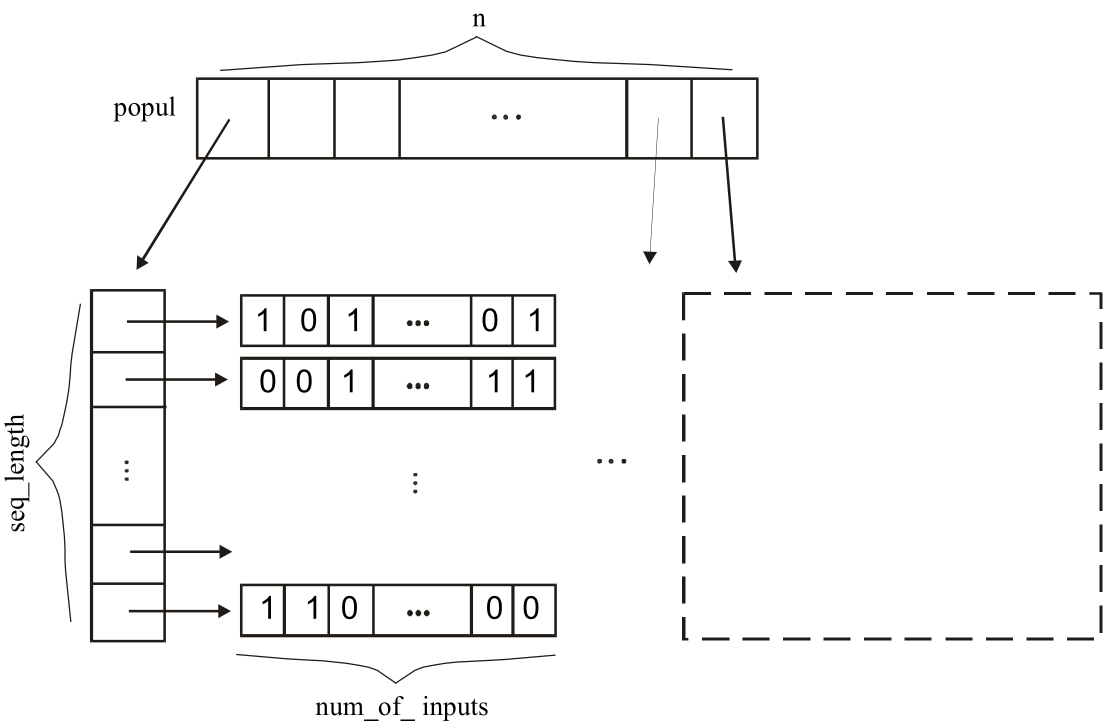


Figure 15 Memory structure of population

3.4.9 Initialization

For research purpose random initializing of population is the best. Moving from a randomly created population to a well adapted population is a good test of algorithm, since final solution will have been produced by the search and recombination mechanisms of algorithm rather than the initialization procedures. Therefore, random initialization for population is used here.

To maximize the run speed of the program and the quality of the fault coverage for FSM, it is promising to use pre-calculated test vectors for initializing purpose. Initial test vectors can be obtained from deterministic FSM test generator. These vectors can also be mutated with certain probability before applying to encourage detecting new faults.

3.4.10 Evaluating test sequences

Evaluation method is needed to estimate the fitness of the test sequences (individuals) in population. It is necessary to measure the quality of the test solutions in order to select best ones. Fault simulation is used for that purpose here. Every test sequence is simulated on FSM model using simulation procedures described in [39,37]. Simulation procedures were suitably modified for genetic algorithms in order to collect information about how many faults are activated, how many are propagated, how many are detected. This is the base information what is taken into account in fitness evaluation for an individual. *Fitness function* itself can be expressed as following:

$$f = C_d \cdot faults\_\det ected + C_a \sum w_i^a \cdot + C_p \sum w_i^p$$

where w^a is a weight for an activation event of certain fault, and w^p is a weight for a propagation event of certain fault. Additional weights C_a and C_b in fitness function may or may not be used in last expression. They give additional possibility manually bias selection.

Constants C_d, C_a, C_p in fitness expression determine how much stress is given to corresponding parameter in expression. All these weights must be well tuned in order to achieve good performance of algorithm. Otherwise, fitness function will include much of noise and selection is less effective.

Weights C_a, C_b and C_d in fitness function could be changed dynamically during run of program. For example, activating some faults gives higher reward to individual than activating other faults. At the same time, when fault is activated its weight is lowered. It is not so ‘attractive’ to individuals anymore. Individuals will concentrate to activating other faults. When certain faults are not activated for certain amount of time, their weights are increased. The same strategy can be applied to promote propagating certain faults.

Better test sequences will get higher score according to this function. Fitness function directs population towards progress, because good test sequences (with high score) will be selected during reproductive selection process and pour ones will be rejected.

When determining values for the weights above, then we must consider which parameters affect the most the expected results (good fault coverage). In the program following default settings are used: $C_{d=1}$, $C_a = 0.8$ and $C_p = 0.6$ It is possible to adjust these parameters from command line of the program, too.

Fitness scaling. When there will be only few faults left undetected for FSM, fitness values became small (not many new faults are detected by sequence). Therefore, squared values of fitness scores are used as fitness measures.

3.4.11 Dynamically increasing test sequence length

1) After certain genetic generations, length of sequence will be increased by one. One new vector will be added to test sequence. This is useful, because for some faults, longer sequences may be required in order to propagate fault effects to primary outputs.

2) Increasing test sequence length when no progress is made for certain generations. Some heuristics can be used for determining how much should test sequence increased.

Increasing test length brings along computational cost, however. Therefore, only new vectors incorporated into test sequence should be simulated again. Namely, before test generator enters to next stage (before sequence length is increased), all the state information about finite state machines associated with individuals in population, are stored in memory. Then, when best individual (test sequence) is known, sequence length for all individuals in the population is increased by certain numbers of vectors (memory is reallocated for population). New part of each individual could be filled with random vectors for example.

Now, when fitness estimation is needed for individuals, only new part of individual is simulated. This is possible because states information of FSM was saved before. Such state information amount can turned out to be quite large; therefore, we copy only best individuals' test sequence into all individuals of newly allocated population. In that case, in future only information about one FSM is needed.

If memory problem for such state information conservation turns out more severe, or just for simplicity, there is another solution for smartly increasing test sequence length. We can just simulate once the old part of individual, store the state information and in the future use this information as starting point for new part of individuals.

Identifying during simulation these flip-flops, which do not have distinguishing sequence and avoiding propagating fault-effects through hard-to–observable states.

3.4.12 Self-adaptive mutation rate

In the beginning of the run, smaller mutation values can be preferred, because there is enough genetic material available, non-of the individuals is dominating. At the end of the run however, more exploration is needed for particular faults, bigger mutation rate could be promising. In addition, the circuits under test are different: for some circuit, less mutation is useful than for others (some circuits do have more 'don't care' inputs than others for ex.).

3.4.13 Fault sampling

Since fault simulation is the most time consuming operation in genetic test generation, only partial set of faults could be used in that process. Sample faults can be selected from fault list randomly or based on faults' equivalence classes. In latter case, in the sample fault list only representatives of similar faults are considered. The assumption is that test sequences developed for sample faults are able to cover also other similar faults.

3.4.14 Selection

Roulette wheel selection is used for determining of candidates for crossover. Change to reproduce is given proportionally to each individual according to its fitness. That means, most effective test sequences are more often used in process of creating new test sequences for next generation (iteration) of genetic program.

3.4.15 Crossover

Crossover is the important feature to genetic algorithm's power. It exchanges genetic material from two parent individuals, allowing useful input vectors from different parents to be combined in new test sequence. However, crossover has to conserve important similarities between test sequences (schema conception). One-point crossover suitable for test sequences is shown in Figure 16 and corresponding C coded implementation is given in Figure 17. First, two individuals for crossover are randomly chosen, then crossover point *c* is randomly selected between 1 and L-1, where L is test sequence length. Parent test sequences are crossed at that point. The first child is identical to the first parent up to crossing point and identical to the second parent after the crossing point.

One – point crossover does not destroy single vectors- all values of inputs are kept together after crossover. This is important of point of view of building block concept, which tells that important elementary blocks must be conserved.

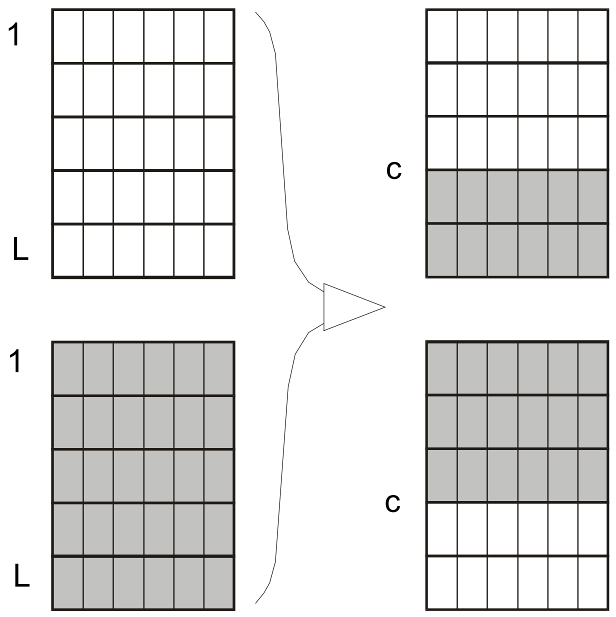


Figure 16 Crossover of vector sequences

```
void crossover()
{
  int k,i,j;
  int cut_point,mate1,mate2;
  char ***temp_ptr;

   for (k=0;k<popul_size;k+=2)
   {
    // selecting a pair for crossover

      mate1=select_individual();
      mate2=select_individual();

    // determining of point of crossover

      cut_point = 1 + rint(rand_num_0_1() * (seq_length-2));

     // exchange of vectors, first stage

     for (i=0;i<cut_point;i++)
     {
       strcpy(new_popul[k][i], popul[mate1][i]); // copying a vector
       strcpy(new_popul[k+1][i], popul[mate2][i]);
     }

     // exchange of vectors, first stage

     for (i=cut_point;i < seq_length;i++)
      {
       strcpy(new_popul[k][i], popul[mate2][i]);
       strcpy(new_popul[k+1][i], popul[mate1][i]);
      }

   }

// new population will be current, as memory addresses do not
// overlap, then data of new population is just overwritten into
// old population memory region (as it is not needed anymore)
// new population gains old population's memory region (will be
// filled with appropriate information during next crossover)


  temp_ptr = popul;
  popul=new_popul;
  new_popul = temp_ptr;

}
```

Figure 17 One-point crossover producing two children.

3.4.16 Mutation

Random mutation provides background variation and occasionally introduces beneficial material into the individuals. Without the mutation, all the individuals in population will eventually be the same (because of exchange of genetic material) and there will be no progress anymore. Program will be stuck in local maximum as it is used to express.

There are several ways to introduce mutations into test sequences:

1) Adding a new randomly generated vector into random position within existing sequence with certain probability. That way, it is possible automatically increase test sequence length. New faults can be detected when longer sequence was needed for certain faults.

2) Removing a randomly selected vector from a sequence with certain probability; if the vector was not essential, the fitness function's value increases. These first two methods of mutation change vector sequence abruptly. It is not possible to introduce small mutations if necessary.

3) Mutation can occur in every bit position in every vector with certain probability. This method was used in genetic algorithm, because it does not change vectors too much at once. It is possible to adjust mutation rate.

4) Mutating randomly selected just one bit or some bits in test vector. It is less computationally costly than previous method of mutation, because it is not necessary to calculate so many random numbers. However, experiments with combinational circuits (results in 3.3.12) have shown a little decrease in final fault coverage.

3.4.17 Description of algorithm

Initially, random population (set of several test sequences) is provided for simulator tool for evaluation. Then, for every vector sequence (individual) a fitness value is calculated (according to its success in FSM state initialization, fault activation, and fault propagation to primary outputs). In addition, all the faults detected by the sequence are dropped from fault list.

Then, reproduction process is carried out. Roulette wheel (proportional) selection method is used for determining pairs of candidates for further crossover. Selection procedure randomly chooses one candidate at time. When a pair is determined that way, crossover point is randomly selected. Then crossover itself is carried out. One - point crossover is used. Two descendants are produced consequently because of the crossover act. Now, next two new candidates are selected and crossed over, and again two new individuals are produced. This takes place N/2 times in cycle, when population size is N (population size has to be therefore an even number).

When crossover in the population is finished, mutation procedure is called out. Mutation operator is applied with given probability over whole population. As mutation rate is small (0.1 … 0.005), then just few bits in each test sequence are inverted. This is should be enough to encourage the genetic program to discover new test sequences, which can lead to activating and propagating new faults and eventually to detecting them. Mutation though, plays secondary role in genetic algorithm. Most important factors are still accurateness of fitness function and crossover mechanism.

Creation on new population is finished at this point. It entirely replaces old population. This is done by swapping of pointers of new and old population. New test vectors are passed to simulator for evaluation again. In turn, simulator returns quality measures, does fault dropping and reproduction process begins again. Subsequently, new vectors are developed again.

Process continues until all faults from fault list are detected or limit of evolutionary generations is exceeded or non-contributing test vectors limit is exceeded.

3.4.18 Experimental results

Results are presented in Table 8. The best results of series of runs are presented. Circuits *lion9* and *dk15* are smaller circuits, *sand* and *planet* are medium size circuits (see Table 7) for which mixed approach (hierarchic + genetic) was used in order to reduce run time. “Hard-to-test faults” faults where left to genetic approach to test.

FSM	States	Inputs	Outputs	Transitions
lion9	9	1	2	25
sand	32	11	9	184
planet	48	7	19	115
dk15	4	3	5	32

Table 7 Characteristics of sample circuits

Combined genetic FSM test generator receives much higher fault coverage in case of large circuits than HITEC, which is often actually considered a relatively good deterministic test generator for sequential circuits. For smaller circuits, test sets are much more compact as well. For larger circuits, mixed approach of test generation was tried in experiments here. After hierarchical test generator had traversed all paths in FSM (which is necessary, but not sufficient condition for detecting all faults), genetic test generator was launched automatically. The latter was able to detect additional faults, which is encouraging; however, run times were relatively long. Obviously longer test sequences were needed for detecting hard-to-test faults.

	FsmGenetic			**HITEC**			**Asyl**[1]		
FSM	Vec	%	T,s	Vec	%	T,s	Vec	%	T,s
lion9	26	100	0.25	38	97,3	8.6	-	-	-
dk15	35	100	0.78	53	100	0.73	44	99.3	-
planet	2414	99.5	3219	91	64.5	917	284	98.8	-
sand	648	98.7	311	52	45.2	1339	308	97	-

Table 8 Experimental results for genetic FSM test generation

[1] Exact comparison is not possible due to different synthesis environment

Comparison with another tool, functional test generator Asyl, current genetic generator had better fault coverage, however difference is not so significant. Asyl is capable to develop shorter test sequences.

In case of *sand* circuit, hierarchical generator alone was able to detect two faults more than combined one. It obviously takes advantage of the knowledge of FSM structure, state table. These are though preliminary results, it is too early to draw far conclusions. More experiments have to be done with different implementation strategies for genetic test generator.

3.4.19 Conclusion

In this section, an approach based on genetic algorithms to generate test vectors for finite state machines (FSM) was presented in order to overcome difficulties with detecting hard-to-test faults. This method includes an evolution program that initially generates random sets of test vectors for FSM, and then these sets are given to a simulator tool for evaluation. Every vector set receives a quality measure (success of FSM initialization, activation of fault, and fault propagation). Based on these quality measures (weights) program builds better vector sets using such genetic operators as selection, crossover and mutation. New test vectors are passed to simulator for evaluation again. Process continues until all faults from fault list are detected or limit of evolutionary generations is exceeded. Program described above works standalone or together with the test generator introduced in [39][37]. Evolutionary program tries to detect faults, which had remained undetected. Preliminary results of experiments are encouraging. Comparison with deterministic test generator HITEC was made. Prototype program developed here received smaller run time and smaller number of vectors for benchmark circuits. In case of *sand* circuit, hierarchical generator alone was able to detect two faults more than combined one. It obviously takes advantage of the knowledge of FSM structure, state table. The future work would be to take only these undetected faults and try fault – oriented approach.

3.5 Fault oriented genetic test generation for sequential circuits

Current section presents a genetic algorithm based approach to test generation for gate level sequential circuits. This approach differs from most of the previous works by specifically targeting single faults, also some structural knowledge about the circuit is used. The priority was to improve the fault coverage, to detect additional faults.

In order to solve the problem, the following components are must in genetic algorithm [31]:

- chromosomal representation of solution to the problem,
- way to create an initial population of solutions,
- an evaluation (fitness) function in order to estimate the quality of the solution
- genetic operators that alter the structure of "children" during reproduction
- fine tuned parameters

Subsequently we will present these issues in detail.

3.5.1 Representation

In context of test generation for sequential circuits, sequence of test vectors will be the individual. Several concurrent sequences form the population.

3.5.2 Initialization

Initially, a random set of test sequences is generated. Such an initial test sequence set is subsequently given to a simulator tool for evaluation. Following steps of algorithm are carried out repeatedly.

3.5.3 Evaluation of test vectors

Evaluation measures fitness of the individuals, i.e. the quality of solutions in a population. Better solutions will get higher score. Evaluation function directs population towards progress because good solutions (with high score) will be selected for crossover and poor solutions will be rejected. We use fault simulation in order to evaluate test sequences. Simulation is carried out only for particular fault under consideration. SSBDD based fault simulator [42][43] was improved to keep track the number of fault effects activated and propagated onto flip-flops and primary outputs.

3.5.4 Fitness

Fitness of the test sequence is calculated as following:

$$C_a * \text{activated} + C_p * \text{propagated},$$

where ‘activated’ is number of clock cycles when particular fault effect was activated in the circuit and ‘propagated’ is the number of clock cycles when fault effect was propagated onto some flip-flop. *Ca* and *Cp* are constants, which show how much stress is given to parameters. We selected 0.1 for *Ca* and 1 for *Cp*.

As a population converges on a definitive solution, the difference between fitness values may become very small. Best solutions cannot have significant advantage in reproductive selection. We use square values for test sequence’ fitness values in order to differentiate good and bad individuals.

3.5.5 Selection of candidate sequences

Selection is needed for finding two candidates for crossover. Based on fault simulation results better test sequences are selected. Roulette wheel selection mechanism was used here. Number of slots on the roulette wheel will be equal to population size. As we see in Figure 18, size of the roulette wheel slots is proportional to the fitness value (denoted as f on the figure) of the test vector sequence.

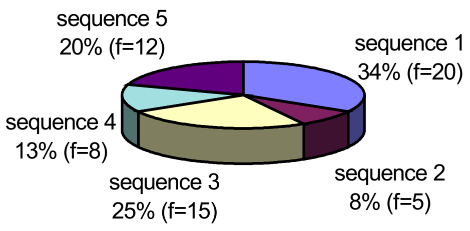


Figure 18 Roulette wheel selection

This means that better sequences have a greater possibility to be selected. If our population size is N, and N is an even number, we have N/2 pairs for reproduction. Candidates in pair will be determined by running roulette wheel twice. One run will determine one candidate. With such a selection scheme, it can happen that same candidate is selected two times. Reproduction with itself does not interfere. This means the selected test sequence is good and it carries its good genetic potential into new generation.

3.5.6 Crossover

Swapping genetic material of the two parents allows useful genes (relevant bits) to be combined in their offspring (new test sequence). Most successful parents reproduce more often. Beneficial properties of two parents combine. Crossover and selection (fitness function) are the keys to genetic algorithm's power. Here, one-point horizontal and one-point vertical crossover (see Figure 19) were implemented.

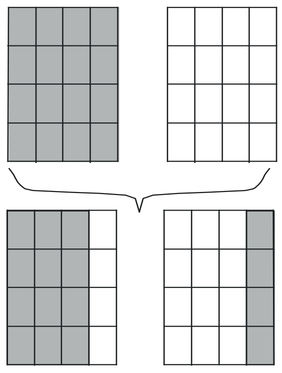


Figure 19. Vertical one point crossover

3.5.7 Mutation

Random mutation provides background variation and occasionally introduces beneficial genetic material [9]. Without the mutation, all the individuals in population will eventually be the same (because of the exchange of genetic material) and there will be no progress anymore. In order to encourage genetic algorithm to explore new regions in space of all possible test sequences, we apply mutation operator (see Figure 20) to the test sequences produced by crossover.

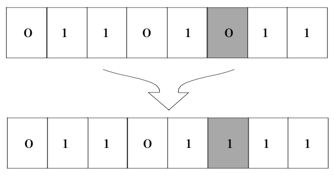


Figure 20. Mutation in test vector.

In all of the test vectors, every bit is inverted with a certain probability p. An alternative would be not to consider all vectors but select some with certain probability- this would be computationally less expensive. However, experiments did not justify the use of that alternative- fault coverage tend to be lower. Important is to point out that, bit position corresponding to reset input is not altered during mutation. Using such a novel knowledge based technique helps to reduce search space.

3.5.8 Working algorithm

GA works in two stages:

In *first stage*, fault activation sequence for the particular fault is generated: at first, short (given by user) random test sequence is simulated with fault simulator. If fault was not activated then test sequence length is automatically doubled and fault simulation is repeated.

This happens until fault is activated or test sequence length limit is exceeded. In latter case fault is aborted and next fault from list is taken. Activation process starts again with short sequence. Fault is considered activated when we could set up the necessary logic value in the particular schematic node.

Important is that in such initialization sequence bit position corresponding to reset signal is filled with zeros. Only in second vector there is ‘one’ i.e. we describe behavior of the reset signal based on a priori knowledge. User can supply reset index with command line option.

Second stage of GA begins if fault activation was successful. At first, activation sequence is distributed into all individuals- the beginning of all test sequences in our population is filled with fault activation vectors (keeping reset ‘0’). The rest of the sequences are filled with random patterns. Sequence length in this stage is twice as long as final fault activation sequence was. With command line option there is possible to select also vector sequence length dynamic increase- it takes into account if fault effect was not propagated onto primary outputs, but still progress was made compared to previous iteration. In such cases sequence length is doubled. This can happen until fault is detected (propagated to primary outputs) or until sequence length limit is reached. Such a technique has proved to be effective in terms of fault coverage increase and shorter test sequences.

Mutation. Random mutation provides background variation and occasionally introduces beneficial genetic material. Without the mutation, all the individuals in population will eventually be the same (because of the exchange of genetic material) and there will be no progress anymore. In order to encourage genetic algorithm to explore new regions in space of all possible test sequences, we apply mutation operator to the test sequences produced by crossover. After the population of test sequences is initialized, GA main repetitive cycle begins. At first, all sequences are evaluated subsequently by simulation procedure again. For each sequence, numeric fitness value is calculated. Then, N/2 times roulette wheel selection routine is invoked in order to select candidates for crossover. N is here number of test sequences in our population. Each time to candidates are selected and crossed over by invoking appropriate sub routine. Crossover type can be selected by user. New population will be filled only with newly constructed sequences, however optionally it is possible to conserve the best individual from the last generation- this is called elitist selection. Finally, before new cycle begins, some mutation is introduced into newly engineered test sequences. Mutation probability is increased dynamically when several subsequent generations did not improve fault propagation. When there was success finally, then mutation rate is lowered again down to initial value. After new population of test sequences is ready, GA main cycle is repeated. This will last until current fault is detected or number of predetermined (by user) number of generations is exceeded. Thereafter new fault is considered if any is left.

3.5.9 Experimental Results

Experiments show that the fault oriented method and usage structural knowledge about the circuit is more effective in terms of achieved fault coverage than any other ATPG it was compared to. In all experiments following parameters for GA were used: population size 16, sequence length limit 200, maximum number of generations 200, crossover type- vertical one point, mutation rate 0.01, elitist selection.

In Table 9 main characteristics of the benchmark circuits are presented. Table 10 shows comparison of test generation results of four ATPG tools. These are, GATEST [10] and HITEC [1], DECIDER [7] and GA (based on the approach described in current section), respectively. GATEST is a genetic algorithm based test generator, HITEC is a deterministic gate-level ATPG, while DECIDER represents a hierarchical approach.

Circuit	**Gates**	**Faults**	**PIs**	**POs**	**Flip-flops**	**FSM states**
gcd	227	844	9	4	15	8
mult8x8	1058	3915	17	16	95	8
diffeq	4195	15,836	81	48	115	6
huffmann	2000	2816	31	30	118	21

Table 9 Characteristics of the benchmark circuits

Circuit	**HITEC [1]**		**GATEST [10]**		**DECIDER [7]**		**GA**	
	Cov., %	**time, s**	**cov., %**	**Time, s**	**cov., %**	**time, s**	**cov., %**	**Time, s**
gcd	89.3	196	92.2	90	91.0	3.4	**93.0**	702
mult8x8	63.5	2487	77.3	3027	79.4	13.6	**80.5**	19886
diffeq	95.1	> 4 h*	96.0	4280	95.9	80	**97.9**	53540
huffmann	12.5	16200	27.6	3553	12.5	8460	**52.8**	> 10 h*

Table 10 Test generation results

The experiments for the first three tools were run on a 300 MHz SUN UltraSPARC 10 computer. Test for GA were generated on a 366 MHz SUN UltraSPARC 60 which is a slightly faster machine. On the other hand, GATEST and HITEC used internally the parallel fault simulator PROOFS, which is in average about 4 times faster than the serial simulation implemented in GA. Thus, the comparison of run times in Table 10 is not exactly correct, however it gives an idea of the speed differences between various algorithms.

In order to perform straight comparison of the test quality, actual stuck-at fault coverages of the test patterns generated by all the four tools were evaluated by a single fault simulator.

As it can be seen from Table 10, GA is capable of achieving the highest fault coverages on all the four circuits. Especially remarkable is its result with the hard-to-test Huffman encoder circuit. The CPU times of GA are by far longer than the ones of DECIDER and HITEC. However, if we take into account the speed differences between the fault simulators used these differences will be quite small. Important is that goal set – improving fault coverage – was fulfilled successfully.

3.6 Discussion

In current chapter, genetic algorithms where used for creating tools for test pattern generation purposes. Genetic method is interesting because it is robust. It is based on mechanisms of natural selection what is also observed in nature. It is based on principles that 1) better adopted species (solutions) will survive throughout generations and 2) changes are made to information pseudo randomly so that with time a species adapt to it's environment, their properties optimize. In each generation, a new set of artificial creatures (sets of bits) is created using the parts of the best elements of the previous generation instead of using totally new elements randomly. Therefore, if used properly, genetic algorithms are not pure random algorithms. They use effectively previously obtained information for exploring new points in search space. The experiments have shown that results also depend of the quality of the pseudorandom numbers used in algorithms- changing the number generators will have considerable effect to results. When random numbers are not truly random then regions of solution space will not be explored.

Three different test generators, all based on genetic algorithm, were implemented here: for combinational circuit, for FSM and for gate level sequential circuit. This shows the portability of genetic algorithms within the application domain. It is possible to reuse core functions with a relatively little of redesign. Of course finding a good fitness function for evaluation of solutions is always crucial. The more precise the evaluation is, the better (faster) the convergence of the algorithm will be.

Although testing of combinational circuit is generally considered solved, it made a good experimental starting point for genetic algorithm study. One aim of implementing such test generator for combinational circuits was to compare genetic test generator with random generator, because it is known that in essence, genetic algorithm uses much of random numbers. Therefore, the algorithm was designed so that its direct comparison with random method was possible. It comes out that genetic method requires always fewer test vectors (patterns) to yield the same fault coverage than random method. Genetic method discovers more faults than random in the case of larger circuits. Random generator achieves good fault coverage sooner but genetic generator detects additional faults in the end. Execution times for the random method were slightly shorter for smaller circuits. It is because just small set of random combinations of inputs values are capable of detecting the all faults when same time genetic algorithm has to make *n* times more simulations, because there is *n individuals (test vectors)* to evaluate "in parallel" in each evolutionary step. Therefore, the effect of the performance increase comes sensible with increase of circuit size, when circuit becomes hardly observable because small amount of inputs available compared to the number of inner points of circuit. Increase of the number of inputs itself raises complexity, too. That is the case genetic algorithm can take advantage. By its selection and crossover operators, it avoids searching through the whole space of test vectors. Instead, a small fraction is searched through.

Results obtained here were compared to the ones achieved in [35], where the key feature was keeping certain inputs together (in order to better propagate fault effects) during reproduction process. The method detected all faults for benchmark circuits c7552 and c2670. However, the approach given here uses (up to 2 times) less of test vectors for all circuits. There was not possible to compare execution times, because they were not revealed.

The comparison between approach here and CRIS [11] revealed that such a monitoring circuit activity used in CRIS is not effective. Experiments show that using fault simulation leads better fault coverage than using logic simulation (CRIS). Simple approach given here detects all detectable faults with a smaller time for all circuits and generates 1,6 – 6,5 times less test vectors. Comparison was not adequate for circuits c2670 and c7552 because in [11] the test generation was terminated too early.

The experience and knowledge gained by implementing and experimenting on prototype program discussed above, was applied in design process of test generator for finite state machine. It was aimed to solve the difficult problem of sequential circuit testing. The testing problem of sequential circuits is even more complicated because of the presence of feedback which causes observability and controllability problems in inner nodes of circuit. FSM model of sequential circuit was used. Difficulty here lies in finding appropriate input vector sequences. This problem is at least of magnitude of order harder than just finding one good vector. Genetic algorithm interactively uses fault simulation procedures of hierarchical test generator in order to evaluate test sequences. Although, such an interaction showed to be capable to improve fault coverage, several improvements can be made to algorithm. Field of genetic algorithms is developing and new advanced techniques like self-adoption concept, niching theory, penalties, multi - objective optimization, multi – parent crossover etc. are raised. More work must be done to evaluate these techniques effectiveness in test pattern generation domain.

The main contribution of the work regarding genetic test generation is that differently from the known genetic algorithms, a fault oriented genetic approach for sequential gate level circuits is developed. Unique feature is the use of knowledge about the circuit under test. For example, input of reset signal is not altered during genetic manipulation because otherwise essential building blocks of test vector set are destroyed and noise is introduced to the algorithm which decreases convergence. Reset signal is made active once and then kept non active in the test sequences. Experiments show that targeting single faults can improve the convergence of a genetic algorithm. In comparison with other GA based generator GATEST, considerably better results were obtained, especially for Huffman encoder circuit. The experiments have shown that targeting single faults however suffers loss in run times in comparison to the other compared approaches. Good news is that better fault coverages are obtained by this technique compared to other solutions. Using more internal knowledge by doing some circuit preprocessing prior to test generation will probably have some potential in order to further limit the search space and improve the convergence of genetic algorithm.

Comparing two last methods, hierarchical FSM based and gate level fault oriented sequential testing, the first one is suitable for use when circuit is described as state machine table (control path of the system) on multiple levels – both, state transition diagram in KISS format and also gate level description are needed. User must provide (synthesize) gate level description from state transition diagram. Second method can be used when the gate level netlist for circuit is available. Fault oriented approach also is targeting faults one by one. Since circuit models are different, there is no comparison information of fault coverage available yet. FSM based testing is faster since it is working on higher level, but other sequential method is more robust in terms of usability.

4 Fault simulation in digital systems

A major part of the genetic test generator is a fault simulator. For this purpose, a research was carried out to improve the existing fault simulation methods.

Simulators are widely used in many areas of electronic design. Test generation, fault diagnosis, test set compaction etc. – these are some examples where fault-free and fault simulation can be used. The quality of Automatic Test Pattern Generation (ATPG) significantly depends on efficient fault simulation, especially in the case of simulation-based test generators [44] and even more in case of Genetic Algorithm-based test tools, because simulation is used to distinguish good and bad solutions. Simulation procedure for genetic test generator has to be fast as possible because very large number of evaluations is carried out during algorithm work.

4.1 Test cover calculation in digital systems with multi-level decision diagrams

Fault simulation has traditionally been performed at the gate-level with the stuck-at fault model. Usually these methods are time consuming. Hierarchical methods allow taking the advantage of high-level information while simulating tests for gate-level faults. Therefore using higher-level information during simulation is promising for speeding up general simulation process.

Binary Decision Diagrams (BDDs) are now commonly used for representing Boolean functions because of their efficiency in terms of time and space [46,47]. They have become the state-of-the–art data structure in many VLSI CAD systems. On the other hand, there have been also several approaches to broaden the use of Decision Diagrams (DD) for representing digital systems at higher-level presentations. Decision Diagrams have been successfully introduced as a uniform mathematical model of the system behavior for the different domains of application in the design process: for system verification by simulation, test generation and fault simulation [48,49,50]. It has been shown that the performance of the simulation of a system represented by DDs is higher than the simulation of the hardware description language model [50].

In this section, decision diagrams are introduced, hierarchical fault simulation process using a uniform DD model for all abstraction levels is presented. Fault detection criterias are outlined, the advantages are discussed and experimental results are given. RT netlist is regarded as the high level whereas gate network is regarded as the low-level representation. Faults are defined at the low-level and fault propagation is carried out at the high level. In such a way, the efficiency of high-level calculation is combined with the accuracy of low-level fault handling.

4.1.1 High Level Decision Diagrams

Consider a digital system as a network $N = (Z, F)$ of components where Z is the set of all variables (Boolean, Boolean vectors or integers) of the system, which represent the connections between components, inputs and outputs of the network. Denote by $X \subset Z$ and $Y \subset Z$, correspondingly, the subsets of input and output variables. $V(z)$ denotes the possible values for $z \in Z$, which are finite.

Let F be the set of digital functions on Z: $z_k = f_k (z_{k,1}, z_{k,2}, \dots , z_{k,p}) = f_k (Z_k)$ where $z_k \in Z$, $f_k \in F$, and $Z_k \subset Z$. Some of the functions $f_k \in F$, for the state variables $z \in Z_{STATE} \subset Z$, are next state functions.

Definition 1

A *decision diagram* (denoted as DD) is a directed acyclic graph $G = (M, \Gamma, z)$ where M is a set of nodes, Γ is a relation in M, and $\Gamma(m) \subset M$ denotes the set of successor nodes of $m \in M$. The nodes $m \in M$ are marked by labels $z(m)$. The labels can be ether variables $z \in Z$, or algebraic expressions of $z \in Z$, or constants.

For non-terminal nodes m, where $\Gamma(m) \neq \emptyset$, an onto function exists between the values of $z(m)$ and the successors $m^e \in \Gamma(m)$ of m. By m^e we denote the successor of m for the value $z(m) = e$. The edge (m, m^e) which connects nodes m and m^e is called *activated* iff there exists an assignment $z(m) = e$. Activated edges, which connect m_i and m_j make up an *activated path* $l(m_i, m_j)$. An activated path $l(m^0, m^T)$ from the initial node m^0 to a terminal node m^T is called *full activated path*.

Definition 2

A decision diagram $G_k = (M, \Gamma, z)$ represents a function $z_k = f_k (z_{k,1}, z_{k,2}, \dots , z_{k,p}) = f_k (Z_k)$ iff for each value $v(Z_k) = v(z_{k,1}) \times v(z_{k,2}) \times \dots \times v(z_{k,p})$, a full path in G_k to a terminal node m^T is activated, where $z(m^T) = z_k$ is valid.

Each function $f_k \in F$ in the system network $N = (Z, F)$ is represented by a decision diagram $z_k = G_k (Z_k)$ [47,48]. Depending on the class of digital system (or level of its representation), we may have various DDs, in which nodes have different interpretations and relationships to the system structure.

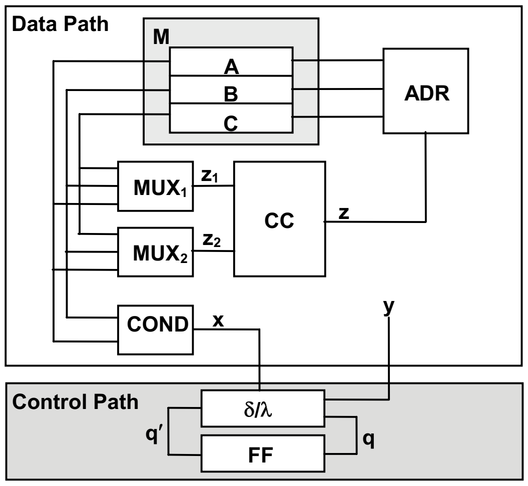
In register transfer level (RTL) descriptions, we usually decompose digital system into control and data parts. State and output variables of the control part serve as addresses and control words, and the variables in the data part serve as data words. High-level data word variables describe RTL component functions in data parts.

Consider a digital system with a behavioral description in Figure 21. The system consists of control and data parts. The FSM of the control part of the system is given by the output function $y = \lambda (q', x)$ and the next-state function $q = \delta (q', x)$, where y is an integer output vector variable, which represents a microinstruction with four control fields $y = (y_M, y_z, y_{z,1}, y_{z,2})$, $x = (x_A, x_C)$ is a Boolean input vector variable, and q is the integer state variable. The value j of the state variable corresponds to the state s_j of the FSM. The apostrophe refers to the value from the previous clock cycle.

The data path consists of the memory block M with three registers A, B, C together with the addressing block ADR, represented by three DDs: $A = G_A (y_M, z)$, $B = G_B (y_M, z)$,

$C = G_C\,(y_M, z)$; of the data manipulation block CC where $z = G_z\,(y_z, z_1, z_2)$; and of two multiplexers $z_1 = G_{z,1}\,(y_{z,1}, M)$ and $z_2 = G_{z,2}\,(y_{z,2}, M)$. The block COND performs the calculation of the condition function $x = G_x\,(A, C)$.

The component level model of the system consists of the following set of DDs: $N_1=\{G_q,G_y,G_A,G_B,G_C,G_z,G_{z,1},G_{z,2},G_x\}$.



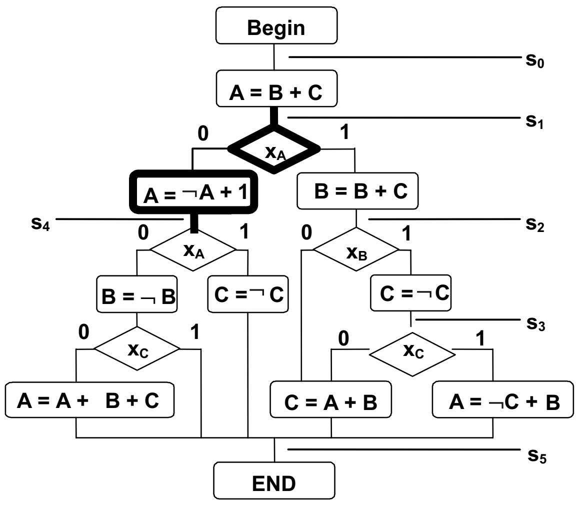


Figure 21 A digital system and its behavior

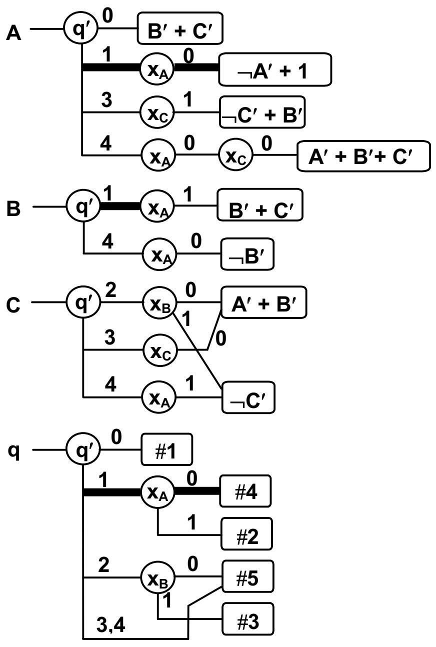


Figure 22 High Level Decision Diagrams for the system in Figure 21

Using now the following chain of superposition of DDs:

$$\begin{aligned} A &= G_A(y_M, z) = G_A(y_M, G_z(y_z, z_1, z_2)) = \\ &= G_A(y_M, G_z(y_z, G_{z,1}(y_{z,1}, M), f_4(y_{z,2}, M))) = \\ &= G_A(y_M, y_z, y_{z,1}, y_{z,2}, M) = G_A(y, M) = \\ &= G_A(G_y(q', x), M) = G'_A(q', A, B, C) \end{aligned}$$

we create a new compact DD model of the system:

$$N_2 = \{G_q, G'_A, G'_B, G'_C\}.$$

The part of the model related to the data path is represented in Figure 22 by three diagrams G'_A, G'_B, G'_C. For simplicity, in these diagrams, the terminals nodes for the cases where the value of the function variable does not change, are omitted.

4.1.2 Low-Level Decision Diagrams

Binary Decision Diagrams (BDD) can be regarded as a special case of high-level DDs described in the previous section where all the variables $z \in Z$ are binary. However, traditional BDDs can be used only for representing functions and not for the faults in gate networks. A special case of BDDs called structurally synthesized BDDs (SSBDD) [51]. Can be used also for directly representing faults. SSBDDs have the following important property: each node m in a G_y which describes a tree-like subnetwork N_y of the gate-level circuit N, represents a signal path L_m in N_y.

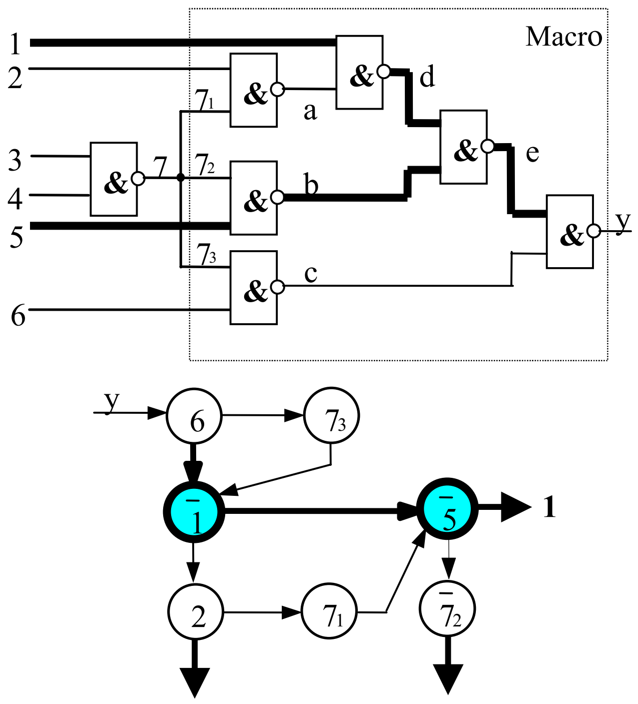


Figure 23 Low-level decision diagram for a circuit

An example of a SSBDD for the subcircuit extracted by dotted lines is represented in Figure 23. For simplicity, the values of variables on edges are omitted (by convention, the right-hand branch corresponds to 1 and the lower-hand branch to 0). Also, terminal nodes with constants 0 and 1 are omitted (leaving the SSBDD to the right corresponds to y = 1, and down to y = 0). The nodes are marked by indexes of the line variables at the beginning of the circuit path represented by a node.

4.1.3 Hierarchical fault simulation

In hierarchical simulation the fault analysis is made block by block at the higher-level network. An example of the approach is illustrated in Figure 24 where the network of the system consists of 3 blocks: A, B and C. The block B is taken as the target for fault analysis, and therefore is represented also at the lower level. Test sequence is simulated as usual, pattern by pattern, starting from the inputs of the system. When the target block B is reached by the first test pattern P, low-level fault analysis is carried out, and the set of all faults R activated by the pattern P is calculated. For each fault $r \in R$, the corresponding faulty output pattern P(r) of the block B is calculated. Activated faults are grouped into subsets $R_i \subseteq R$, so that each $r \in R_i$ for what the same (faulty) output pattern $P_i=P(R_i)$ corresponds will be in the same subset. The fault-free pattern P, and all the faulty patterns $P_1, \ldots, P_k$ where

$R_1 \cup R_2 \cup \ldots \cup R_k = R$ are simulated through other blocks of the network at the higher level. At the output of each observable block the faults in R that can be observed and detected are removed from the further calculation and fixed as detected faults. In general, at the output of each higher level block, a data structure D = $\{P, (P_1,R_1), \ldots, (P_k,R_k)\}$ will be generated where $R_1 \cup R_2 \cup \ldots \cup R_k = R$. When the target block B is reached by D, all the patterns $\{P,P_1, \ldots,P_k\}$ should be fault simulated at low-level. For P, new faults will be determined which are activated by P; and for all the faults in R_i it will be detected if they are propagated at the pattern P_i through B or not. After that, a new data structure D will be created at the output of B.

Fault analysis on DDs is based on *path traversing* procedures. In path traversing, the values of node-variables are given, and we have to move along the path determined by these values.

Consider, at first the SSBDDs and introduce the following notations: l(m) - activated path from the root node up to the node m; l(m,=1) (or l(m,=0)) - activated path from the node m up to the terminal node labeled by the constant 1 (or 0); m^1 (or m^0) - successor of the node m for the value z (m)=1 (or z(m)=0).

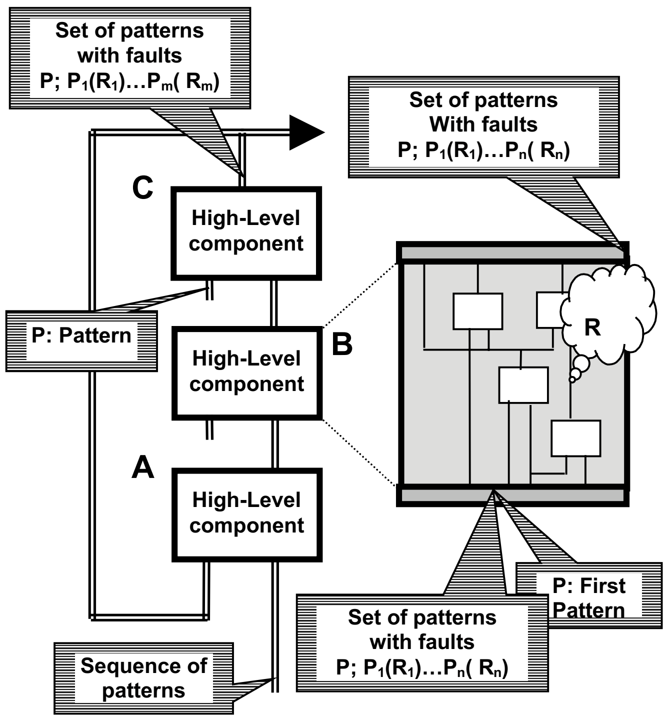

Set of patterns with faults P; P1(R1)...Pm(Rm)
Set of patterns With faults P; P1(R1)...Pn(Rn)
C
High-Level component
P: Pattern
High-Level component
B
R
A
High-Level component
P: First Pattern
Set of patterns with faults P; P1(R1)...Pn(Rn)
Sequence of patterns

Figure 24 Hierarchical simulation of faults

Suppose, a test pattern *P* assigns to a node variable *z*(*m*) a value $e \in V(z(m))$. Suppose also, there is a fault (i.e. defect) *d* which influences on the variable *z*(*m*) and changes the value *e* to *D*. The fault is detected by the pattern *P* if the following conditions are fulfilled:

P activates the paths: $l(m^0, m)$, $l(m^e, m^{Te})$, $l(m^D, m^{TD})$, and

$z(m^{Te}) \neq z(m^{TD})$ is valid.

To *analyse* a test pattern for all faults detected, means:

to find path l activated by the pattern with a terminal node m^T where $z(m^T) = e$, for all nodes $m_k \in l$, find the value $e_k = z(m_k^T)$ where m_k^T is the end node of $l(m_k^{\neg e}, m_k^T)$ activated by the same pattern;

for all nodes $m_k \in l$, the given pattern detects the fault $z(m_k) / \neg e$ if $e_k \neq e$ is valid.

For example, in Figure 23 for the input pattern (001000) we have: $l = \{6, \neg 1, \neg 5\}$. At this pattern, the faults $z(\neg 1) \equiv 0$, $z(\neg 5) \equiv 0$, are detected, but the fault at the node 6 is not detected. The faults in the circuit which are detected by this pattern are spread along the highlighted paths from 1 to y, and from 5 to y. These paths correspond to the nodes ¬1 and ¬5 in SSBDD.

The check of faults propagating at higher level DDs is carried out by simulation of all the patterns $\{P, P_1, \ldots, P_k\}$, and by subsequent comparing the results. For example, in Figure 21 and Figure 22 a clock cycle is simulated where the following calculations are made:

$A := \neg A' + 1$, $q := 4$. The traversed paths are shown in bold.

4.1.4 Experimental results

In Table 11, RT level simulation results in comparison with 2 commercial VHDL simulators for circuits GCD and DIFFEQ (HLSynth benchmarks), MULT8X8 (8-bit multiplier) are given.

Circuit	DD	VHDL - 1		VHDL - 2	
		Time	Ratio	Time	Ratio
GCD	3.89	13.13	3.4	33.98	8.7
DIFFEQ	8.96	81.51	9.1	331.62	37.0
MULT8X8	5.79	25.38	4.4	64.81	11.2
	Inputs	Outputs	States	Gates	F/F
GCD	10	4	8	227	15
DIFFEQ	82	32	6	4195	115
MULT8X8	18	16	8	1058	95

Table 11 High-level simulation efficiency

ISCAS	Gates	Fault cover %	Number of test patterns	Fault sim. gate/macro speed ratio
C432	232	97.33	55	3.86
C880	383	100.00	100	5.15
C1355	546	99.64	52	3.08
C1908	880	99.75	122	6.46
C2670	1193	99.67	119	6.47
C3540	1669	95.58	145	8.81
C5315	2307	99.78	108	8.37
C6288	2416	99.80	33	2.55
C7552	2978	99.46	198	9.04

Table 12 Low-level fault simulation efficiency

FSM	States	Inp	Out	Transitions	Cells	Faults
bbsse	16	7	7	56	80	562
dk16	27	2	3	108	156	1038
ex2	20	2	2	73	77	480
ex3	10	2	2	37	38	274
log	17	9	24	29	76	486
s832	25	18	19	245	162	1090
s1488	48	8	19	251	332	2234
sand	32	11	9	184	259	1622
scf	121	27	56	166	428	2800
styr	30	9	10	166	262	1734

Table 13 Benchmark circuits for hierarchical simulation

			HSIM	GSIM
Fsm	TL	Cover %	Time,s	Time,s
bbsse	300	74.1	0.01	0.38
dk16	150	95.1	0.01	0.55
ex2	600	25.9	0.01	1.21
ex3	1000	46.2	0.01	0.77
log	200	99.6	0.01	0.11
s832	300	59.6	1.00	2.69
s1488	400	63.48	2.00	9.17
sand	400	84.2	1.00	3.18
scf	700	34.5	33.0	39.32
styr	500	74.1	2.00	6.81

Table 14 Hierarchical fault simulation results

In experiments, each circuit was simulated for 200000 random input vectors. Simulation time is given in seconds. “Ratio” is the VHDL simulation time to DD-based simulation time ratio.

In Table 12, the gain in the speed for the low-level fault simulation of SSBDD macros compared to the gate-level fault simulation is demonstrated. For experiments, ISCAS’85 benchmark circuits were used. The differences in speed are between 2.55 and 9.04.

In Table 13 some FSM benchmark circuits are used for evaluating the hierarchical fault simulation approach. The hierarchical fault simulation results (HSIM) compared to the plain gate-level simulation (GSIM) are depicted in Table 14. Here TL is the length of test sequences. Hierarchical fault simulation results compared to the plain gate-level simulation results show speed gain up to 121 times. However, surprisingly in case of largest circuit speed gain is only minimal. Obviously gate-level generator takes the advantage of good observability of the circuit since number of outputs is large.

4.2 Defect oriented mixed-level fault simulation in digital systems

The quality of test generation significantly relies on the efficiency of fault simulation, especially in the case of simulation-based test generators like genetic test pattern generator. Traditionally, fault simulation is performed at the gate-level with using the stuck-at fault (SAF) model. On one hand, the gate-level SAF-based fault simulation is time-consuming; on the other hand, the SAF model doesn’t represent adequately the physical defects in transistor circuits. To overcome these disadvantages, mixed-level simulation is needed which allows to use the advantage of high level information to speed up the simulation process while analyzing the quality of tests still in relation to realistic physical defects.

In this section, a new method for parametric defect modeling is presented for calculating the conditions for activating physical defects in the modules (e.g. library components) of digital circuits. A new method for multi-level defect-oriented fault simulation based on Decision Diagrams (DD) is proposed. We suppose that a register transfer level (RTL) information along with gate-level descriptions for blocks of the RTL structure are available. For defect simulation a new functional fault model is used which can be handled on the gate- and RT levels. Decision diagrams (DDs) are exploited as a uniform model for describing systems on both, RT and gate levels.

In this section, mapping of physical defects onto the logical level is described, the general method of the hierarchical defect-oriented fault simulation is presented, then same simulation on the model of DDs is considered. Finally, experimental results are presented.

4.2.1 Mapping Defects onto the Logical Level

Subsequently we present a general fault model for describing and modeling arbitrary physical defects in the components of digital circuits and for mapping them onto the logical level.

Consider a Boolean function $y = f(x_1, x_2, ..., x_n)$ implemented by an embedded component in a digital circuit. Introduce a Boolean variable d for representing a given defect in the component or in the neighbourhood layout of the component, which may affect the value y by converting the Boolean function f into another function $y = f^d(x_1, x_2, ..., x_n, x_{n+1}, ... x_p)$. Here, the new variables $x_{n+1}, ... x_p$ may be introduced to describe the influence of the neighbourhood layout of the component in the presence of the physical defect d.

For example, assume there is a short between x_1 and x_5 in the circuit in Figure 25.
The faulty function $y = f(x_1,x_2) = \neg(x_1 \wedge x_2)$ in the case of the defect d can be represented as $y = f^d(x_1,x_2,x_3,x_4) = \neg(x_1 x_5) \vee x_2 = \neg(x_1 \neg(x_3 \wedge x_4)) \vee x_2)$.

Introduce now a generalized parametric function
$y^* = f^*(x_1, x_2, ..., x_n, x_{n+1}, ... x_p, d) = \neg d \,\&\, f \vee d \,\&\, f^d$
as a function of a defect variable d, which describes the behavior of the component simultaneously for both possible cases. For the erroneous case the value of the *defect variable d* as a parameter is equal to 1, and for the nonerroneous case $d = 0$. In other words, $y^* = f^d$ if $d = 1$, and $y^* = f$ if $d = 0$.

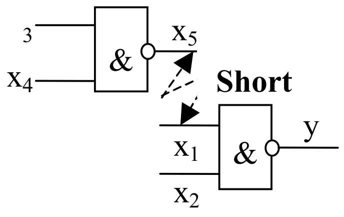


Figure 25 A short between two signal leads

The solution of the Boolean differential equation

$$W^d = \partial y^* / \partial d = 1 \qquad (1)$$

describes the conditions which activate the fault d on a line y. For example for the short in Figure 25 we have

$$y^* = \neg d f \vee d f^d = \neg d \neg(x_1 \wedge x_2) \vee d(\neg x_1 \neg(x_3 \wedge x_4) \vee \neg x_2).$$

To find the conditions for activating the short to the line y we have to solve the logical equation $W^d = \partial y^* / \partial d = x_1 x_2 x_3 x_4 .= 1$.

The method of parametric defect modeling by logical conditions W^d can be generalized for the purpose of hierarchical fault simulation. A component of a circuit can be preprocessed by lower level defect simulation with the goal to generate a set of conditions W for all possible lower level defects d of the component. Each condition as a solution of $W^d = 1$ can be regarded as a higher level functional fault model for a given defect d, since in the presence of this defect the functional behavior of the component at the input where $W^d = 1$ will be erroneous. The functional fault model concept is illustrated in Figure 26.

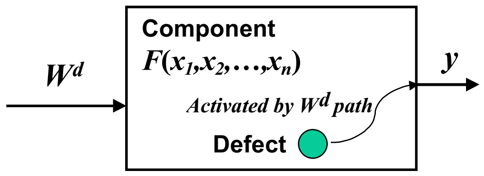


Figure 26 Functional fault model for a physical defect

The relationships between the functional faults (patterns) W^d and the defects d for all the logic level simple or complex gates g in the library L are given by defect tables $DT_g = || g_{id} ||$, $g \in L$, where an entry $g_{id} = 1$ means that the input pattern i (solution of $W^d = 1$) of the gate detects the defect d, otherwise $g_{id} = 0$.

4.2.2 Defect Oriented Hierarchical Fault Simulation

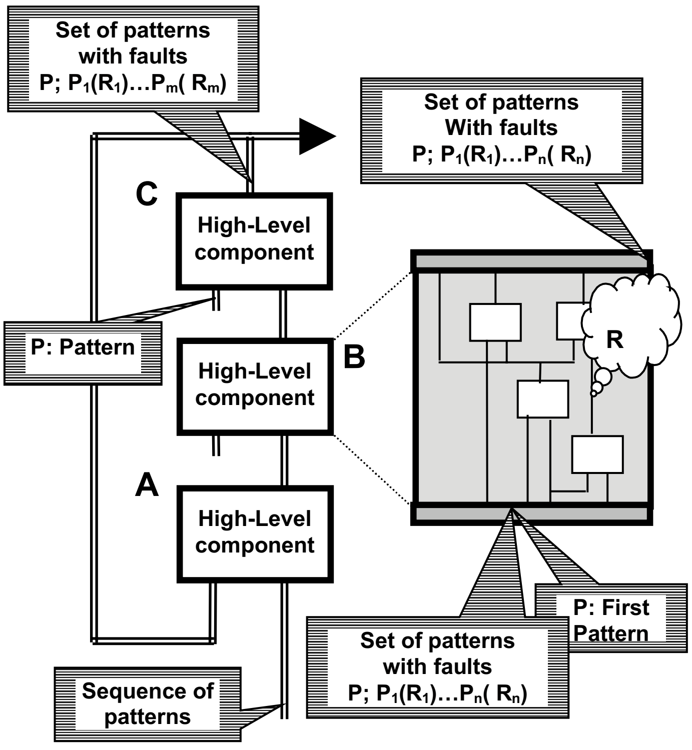
Consider a task of defect oriented fault simulation in a system represented on three levels: register transfer, gate and defect levels.

Formally, if Y is the system RTL variable representing an observable point of the system, y_M is an output variable of a gate-level module and y_C is the output of a component (complex gate) in the module with a physical defect d, then the condition of detecting the defect d on the observable test point Y can be represented as

$$W = \partial Y/\partial y_M \wedge \partial y_M/\partial y_C \wedge W^d = 1, \qquad (2)$$

where $\partial Y/\partial y_M$ is the Boolean derivative calculated by the high-level simulation, $\partial y_M/\partial y_C$ is the Boolean derivative calculated by the gate-level simulation, and W^d is the functional fault condition found by the gate defect-level preanalysis.

In the hierarhical (i.e. multi-level) fault simulation approach proposed earlier in 4.1.3, the defect analysis is made module by module in the higher RT level network. An example of a RTL system to describe the main principles of the approach is illustrated in Figure 27. The network of the system consists of 3 blocks (modules): *A, B*, and *C*. The block *B* is taken currently as the target for defect oriented fault analysis, and therefore is represented at the lower gate-network level. The test sequence before reaching the target block is simulated on the RT level, pattern by pattern, starting from the inputs of the system. Let *D* be the set of all defects to be simulated in the target block *B*. The low-level fault simulation for a given input pattern *P** in the target block B is carried out by the following procedure.



Set of patterns with faults P; P1(R1)...Pm(Rm)
Set of patterns With faults P; P1(R1)...Pn(Rn)
C
High-Level component
P: Pattern
B
High-Level component
R
A
High-Level component
P: First Pattern
Set of patterns with faults P; P1(R1)...Pn(Rn)
Sequence of patterns

Figure 27 Hierarchical fault modeling in a digital system

Procedure 1

When the target block B is reached by a pattern P^* first, the fault-free output pattern P is calculated. Then, the low level defect analysis is carried out, and the set of all defects $d \in D_B \subset D$ activated in B by the pattern P^* is calculated by checking if at least for one output y_B of the block B the condition

$$W = \partial y_B / \partial y_G \wedge W^d = 1 \qquad (3)$$

is fulfilled. For each activated defect $d \in D_\mathrm{B}$ in B, the corresponding faulty output pattern of the block B is calculated. Then the all activated defects $d \in D_\mathrm{B}$ for which the same (faulty) output pattern of B is produced are grouped into the same subset $D_{Bi} \subseteq D_B$. As the result, a complex test pattern $T = \{P, (P_1,D_1), \ldots, (P_k,D_k)\}$ at the output of B will be generated where $D_1 \cup D_2 \cup \ldots \cup D_k = D_\mathrm{B} \subset D$.

Suppose now, a block C (which is not the target block) is to be simulated at the higher level. All the input patterns of the block C can be regarded in a general case as a set of complex test patterns $TS = \{T_1, T_2, \ldots, T_\mathrm{m}\}$ where $T_i = \{P_{i,0}, (P_{i,1,}D_{i,2}), \ldots, (P_{i,k},D_{i,k})\}$. This set can be easily reformed as a single joint complex pattern $T^* = \{P^*, (P^*_1,D^*_1), \ldots, (P^*_n,D^*_n)\}$. The high-level (RT-level) fault simulation for a given complex input pattern T^* in the non-target block C is carried out by the following procedure.

Procedure 2

For each joint input pattern from the set $\{P^*, P^*_1, \ldots, P^*_n\}$ at the high-level, the corresponding output complex pattern $T' = \{P, (P_1,D^*_1), \ldots, (P_n,D^*_n)\}$ is calculated. If two

input patterns P^*_i and P^*_j produce the same output pattern P_h then the two pairs (P_i,D^*_i) and (P_j,D^*_j) in T' should be merged, and the pattern P_h should be linked to a joint set of detects $D_h = D_i \cup D_j$. If the fault-free input pattern P^* and a faulty input pattern P^*_j produce the same output pattern P, then all the defects in D^*_j are self-masked, and the component (P_j,D^*_j) should be removed from the complex pattern T'. As the result, a new simplified complex test pattern $T = \{P, (P_1,D_1), \ldots, (P_k,D_k)\}$ at the output of B where $k \leq n$ may be generated where $D_1 \cup D_2 \cup \ldots \cup D_k \subseteq D_B \subset D$.
When during the fault simulation the target block B is again reached via the feedback loops by a complex pattern $T^* = \{P^*, (P^*_1,D^*_1), \ldots, (P^*_n,D^*_n)\}$, all the patterns $\{P^*, P^*_1, \ldots, P^*_n\}$ should be fault simulated on the low-level at the presence of corresponding defects. For P, new defects activated by P are calculated by using Procedure 1. After that, a new complex pattern T^* will be created at the output of B using the operations described in Procedures 1 and 2.

4.2.3 Defect Oriented Fault Simulation on Decision Diagrams

Fault simulation on DDs is carried out by tracing the activated paths on DDs in according to the given values of variables as specified in Definition 1 in subsection 4.1.1.

For example, at the given input (state) pattern $P = \{q'=1, x_A=0\}$ of the block A we reach the terminal node m^T of the graph G_A with label $A' + 1$ (see the highlighted path in Figure 28 below). The new value of A will be $A = A' + 1$.

In high-level fault propagation in the digital system $S=(Z,F)$ through a block with function $z = f(z_1, z_2, \ldots, z_n) = f(Z'), Z' \subseteq Z$, which is represented by a decision diagram G_z, we proceed from the fact that the defects may have been propagated to all of the variables $z_i \in Z'$ used in labels of nodes in the graph. To each node m of the DD with the label $z(m)$, a complex pattern $T_{z(m)} = \{P_{z(m),0}, (P_{z(m),1},D_{z(m),1}), \ldots, (P_{z(m),kz},D_{z(m),kz})\}$
corresponds. From this pattern, it results that a set of defects $D_{z(m)} = D_{z(m),1} \cup \ldots \cup D_{z(m),k}$ has been propagated to the node m. Let D be the set of all faults currently activated and listed in $T_{z(m)}$.
Consider the fault simulation on the decision diagram G_z as the following set of procedures.

<u>Procedure 3</u>
First, the fault-free path is simulated in accordance to the fault-free input pattern $P_{z(m),0}$,, and the fault-free value of $z=z(m^{T,0})$ is calculated, where $m^{T,0}$ is the terminal node of the fault-free activated path.

Denote the set of all nodes traced in the fault-free path up to the node m (m itself not included) by $M_{FF}(m)$. Let $D_{FF}(m)$ be the set of all faults propagated to the nodes $m \in M_{FF}(m)$. The condition of reaching the node m in the fault-free path during fault simulation is the absence of all the faults in $D_{FF}(m)$. Denote by $D_{CF}(m)$ the set of faults consistent to the current faulty path from the initial node m^0 up to the node m. For the nodes m on the fault-free path we have $D_{CF}(m) = D - D_{FF}(m)$.

Denote by L the list of all nodes of the DD to be fault simulated. All the nodes on the fault-free path are included into L. For carrying out fault simulation of the nodes in L, either Procedure 4 or Procedure 5 will be used. As the result of the procedure the list L will be updated. Fault simulation is terminated when the list L gets empty.

Procedure 4

Fault simulation of a terminal node $m^{T,0} \in L$ with the function $z = z(m^{T,0}) = f(z_1, \ldots, z_p)$ for the set of complex input patterns $T = (T_1, \ldots, T_p)$, $T_i = \{P_{i,0}, (P_{i,1}, D'_{i,1}), \ldots, (P_{i,ki}, D'_{i,ki})\}$, $i = 1,2, \ldots, p$, where $\forall i,j$: $D'_{i,j} = (D_{i,j} - D_{FF}(m^{T,0})) \cap D_{CF}(m)$
is equivalent to Procedure 2 of high-level fault simulationdiscussed in Section 4.

Procedure 5

Fault simulation of a nonterminal node $m \in L$ with the variable $z(m)$ for the complex pattern $T_{z(m)} = \{P_{z(m),0}, (P_{z(m),1}, D'_{z(m),1}), \ldots, (P_{z(m),km}, D'_{z(m),km})\}$ where $\forall j$: $R'_{z(m),j} = (R_{z(m),j} - R_{FF}(m)) \cap R_{CF}(m)$, consists in the following:

if m belongs to the fault-free path, and if $D'_{z(m)} = D'_{z(m),1} \cup \ldots \cup D'_{z(m),km} = \varnothing$ no nodes will be included into L;

if m does not belong to the fault-free path, and if $D'_{z(m)} = \varnothing$, the node m^e where $e = P_{z(m),0}$, will be included into L; for the new node m^e in L we calculate: $D_{FF}(m^e) = D_{FF}(m) \cup D_{z(m^e)}$, and $D_{CF}(m^e) = D_{CF}(m)$,

if $D'_{z(m)} \neq \varnothing$, all the nodes m^e, where $e = P_{z(m),i}$, i: $D'_{z(m),i} \neq \varnothing$, will be included into L; for all these nodes we calculate $D_{CF}(m^e) = D_{CF}(m) \cap D'_{z(m),i}$, $D_{FF}(m^e) = D_{FF}(m)$.

As the result of the fault simulation by Procedures 4 and 5 we create a complex pattern for the graph variable z: $D_z = \{P_{z,0}, (P_{z,1}, D_{z,1}), \ldots, (P_{z,kz}, D_{z,kz})\}$. All the pairs $(P_{z,i}, D_{z,i})$ where $P_{z,i} = P_{z,0}$ are eliminated since the defects $D_{z,i}$ are self-masked at this point. All the groups of pairs $\{(P_{z,i}, D_{z,i}), (P_{z,j}, D_{z,j})\}$ where $P_{z,i} = P_{z,j}$ are merged into a single pair $(P_{z,i}, D_{z,i})$ where $D_{z,i} = D_{z,i} \cup D_{z,j}$.

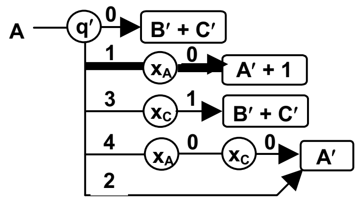


Figure 28 DD for the subcircuit of A in the system in Figure 21

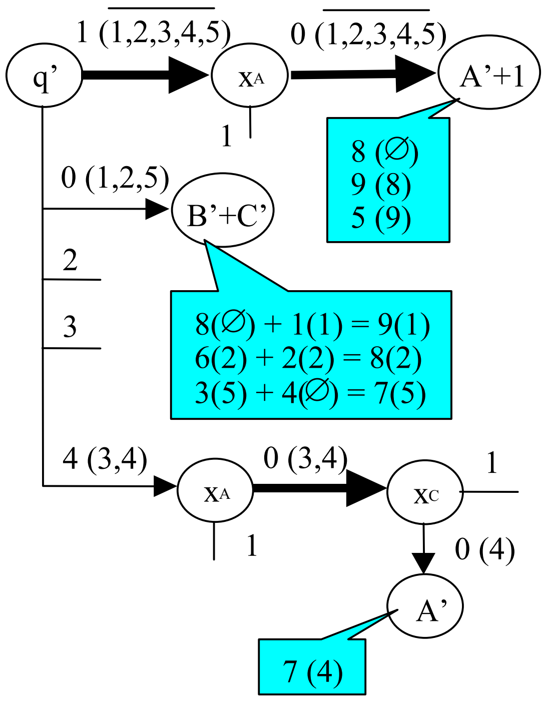


Figure 29 Fault simulation on the graph G_A on Figure 28

Example

Consider the DD G_A in Figure 28 with a set of complex patterns: T_q = {1, 0 (1,2,5), 4 (3,4)}, T_{xA} = {0, 1 (3,5)}, T_{xC} = {1, 0 (4,6)}, T_A = {7, 3 (4,5,7), 4 (1,3,9), 8 (2,8)}, T_B = {8, 3 (4,5), 4 (3,7), 6 (2,8)}, T_C = {4, 1 (1,3,4), 2 (2,6), 5 (6,7)}. All the paths traced during the fault simulation are highlighted and marked by details of simulation in Figure 29. The fault free paths are shown by bold lines both, in Figure 28 and Figure 29. The edges on paths in Figure 29 are labelled by pairs *e*,(*D*), where *e* is the value of the node variable when leaving the node at this direction, and *D* is a subset of defects: $D_{FF}(m)$ for the next node *m* on the fault-free path, and $D_{CF}(m)$ for the next node *m* on the faulty paths. Since D_{FF}(x$_A$) = {1,2,3,4,5} includes both of the defects 3 and 5 propagated to x$_A$, no faulty paths are simulated from the node x_A: for the value x_A = 1: D'_{xA} = (D_{xA} - D_{FF} (x_A)) = ∅. From all the defects propagated to *A'*, only the defects 8 and 9 are simulated at the node *A'*+1. At the terminal node *B*'+*C*' only the defects 1,2,5 are simulated, since only they are consistent to the condition of leaving the node *q'* at this direction.

After fault simulation of all 3 terminal nodes reached at the given complex pattern we compose the final result as follows: the defect 2 propagated to the node *B'*+*C'* is self-masked because the value *B'*+*C'* = 8 calculated for the defect 2 is equal to the fault-free value calculated at the node *A*'+1. The defects 4 and 5 propagated to different terminal nodes are merged into the same group because they produces the same new value 7 for *A*. Also the defects 1 and 8 are merged into the same group. The final value of the new complex pattern for A is: T_A = {8, 5(9), 7 (4,5), 9 (1,8)}.

4.2.4 Experimental results

For investigation the correlation between fault coverages for stuck-at faults (SAF) and the defects, two benchmark circuits C1 and C2 were created – both, tree-like combinational networks, the first with 2-levels (5 complex gates, 16 inputs, 100 defects) and the second with 3-levels (21 complex gates, 64 inputs, 420 defects). Both circuits were simulated for two tests T_{min} (optimized test: 8 patterns for C1, and 16 patterns for C2) and T_{max} (not optimized test: 19 patterns for C1, and 70 patterns for C2) which both had 100% coverage for stuck-at faults. The results of the defect oriented simulation for the given tests are depicted in Table 15.

Circuit	Number of defects	Stuck-at Fault Coverage	Defect coverage, %	
			T_{min}	T_{max}
C1	100	100,00	81,00	83,00
C2	420	100,00	84,29	84,76

Table 15 Comparison of Defect and SAF simulation

From these experiments we see that the stuck-at-fault based fault coverage is overestimated compared to the realistic defect coverage, and that the difference between stuck-at fault and physical defect coverages reduces when the complexity of the circuit increases. In the worst case we have noticed that the 100% SAF test may cover only 50% of realistic physical defects.

In Table 16 the results of multi-level simulation for FSM benchmark circuits are shown for evaluating the proposed mixed-level fault simulation approach. A hierarchical multi-level fault simulator (HSIM) is compared to the plain gate-level simulator (GSIM). Here we see that the mixed-level fault simulation can be carried out with significally higher speed than in the case of plain gate-level simulation, the difference is between 2,7 and 121 times, or in average 35 times.

FSM circuit	Number of faults	Test length	Fault cover %	Time,sec	
				HSIM	GSIM
bbsse	562	300	74.1	0.01	0.38
dk16	1038	150	95.1	0.01	0.55
ex2	480	600	25.9	0.01	1.21
ex3	274	1000	46.2	0.01	0.77
Log	486	200	99.6	0.01	0.11
s832	1090	300	59.6	1.00	2.69
s1488	2234	400	63.48	2.00	9.17
Sand	1622	400	84.2	1.00	3.18
Styr	1734	500	74.1	2.00	6.81

Table 16 Mixed-level fault simulation results

4.3 Discussion

Experiments with hierarchical simulation of digital circuits show considerable speed gain. For instance, RT level (high level) simulation results on decision diagrams in comparison with two commercial VHDL simulators reveal up to 37 time boost up in simulation speed. Speed difference increases with the size of the circuit.

At the same time, the low level fault simulation of SSBDD macros in comparison to the gate-level fault simulation on ISCAS'85 benchmark circuits demonstrated the differences in speed between 2.55 and 9.04 times. In addition, here, speed difference increases with size of the circuit in favor of SSBDD macros.

Hierarchical fault simulation results compared to the plain gate-level simulation results show speed gain up to 121 times. However, surprisingly in case of largest circuit speed gain is only minimal. Obviously gate-level generator takes the advantage of good observability of the circuit since number of outputs is large.

Hierarchical defect oriented fault simulation method for digital systems introduced here helps to reduce dramatically the computation cost of test quality analysis in digital systems. Decision diagrams are used as a mathematical model for systematic multi-level solution for fault simulation at three levels of abstraction - RT, gate- and defect levels respectively.

Experiments show that the stuck-at-fault based fault coverage is overestimated compared to the realistic defect coverage, and that the difference between stuck-at fault and physical defect coverages reduces when the complexity of the circuit increases. In the worst case, it was observed that the 100% stuck-at-fault test may cover only 50% of realistic physical defects.

5 Design Flow with Test Tools

5.1 Generating data for high-level synthesis and test generation

Nowadays system design flow starts at higher abstraction level because of the design complexity problem. Namely, recent developments in microelectronic technology allow implementing a whole digital system as a single integrated circuit. Various hardware description languages (HDL) have gained therefore much popularity among designers because of the advantages they offer over traditional schematic techniques. The main advantage is the possibility to use the same description both to model the behavior and as a starting point for schematic synthesis. Another important feature of most of the HDL-s is the possibility to describe an algorithm at higher abstraction levels thus hiding target technology dependent hardware implementation details.

VHDL (VHSIC Hardware Description Language) is one of the most popular languages, if not the most popular. VHDL supports top-down, bottom-up and mixed development methodologies. Designs in VHDL can be made technology-independent. That is, no redesign is needed, or a very limited redesign is required, when switching to a new technology. Development times in VHDL are shorter and maintenance is simpler compared to the traditional schematic design [52].

The main problem when using VHDL is that only a limited set of constructions can be mapped onto hardware without different possible interpretations. This limited set is called synthesizable subset and it is defined differently for different abstraction levels. Subsequently in this section, we describe the principles of mapping a VHDL subset onto flow-chart like internal representation used by an academic HLS tool xTractor [53]. Important is that the ongoing research in high-level test generation makes it necessary to have a VHDL front-end available also for test generation tools.

The used internal representation, a synthesizable subset of IRSYD [55], is in essence a directed graph in which the arcs explicitly represent the flow of control. Four types of nodes are used currently: *entry* and *exit* nodes to mark beginning and end of the control flow, *operation* nodes to encapsulate computational activity, and *condition* nodes to describe branching on a variable. Edges of the graph can have special associations that represent either wait statements of the source code or states of the Mealy FSM corresponding to the IRSYD. An example code with the corresponding piece of flow-chart is shown in Figure 30. Flow chart is actually represented textually during computations.

Rest of the chapter is organized as follows. Supported VHDL constructs are described in section 5.1.1. The principles of mapping VHDL constructs onto IRSYD ones is explained in section 5.1.2. In section 5.1.3, the control flow extraction process is described. A brief overview of the xTractor tool is given in section 5.1.4 together with some characteristic synthesis results. Finally, decision diagram synthesis from RTL level VHDL is discussed.

5.1.1 Supported VHDL constructs

Since VHDL is a very complex language and it was originally designed for simulation and not for synthesis, a synthesizable subset of the language had to be defined. The abstraction level used in HLS simplified the task - only constructs used for behavioral descriptions are needed (see, e.g., [56]). We have excluded constructs that cannot be mapped directly onto bit-vectors or operations with them because additional decisions are needed. For instance, encoding is needed for enumerated data types or bit-layout is needed for floating point numbers. The front-end (compiler) is oriented to VHDL'93 standard because VHDL'86 lacks few constructs often used by hardware designers, e.g., built-in shift operations. The supported constructs are listed as follows.

Data types

The supported types are *bit*, *bit_vector*, *boolean*, *integer*, *signed*, and *unsigned* - all of them can be mapped directly onto signed and/or unsigned bit-vectors (sets of wires in hardware). Conversion functions, needed by the strict rules of VHDL, are ignored because all supported data types map onto the same underlying type. Both *variable*s and *signal*s of supported types can be declared. One-dimensional arrays of allowed types are also supported.

Operations

All operations with supported data types are allowed - they map onto basic operations with bit-vectors. The only difference is between few signed and unsigned operations, e.g., comparisons must treat the most significant bits differently.

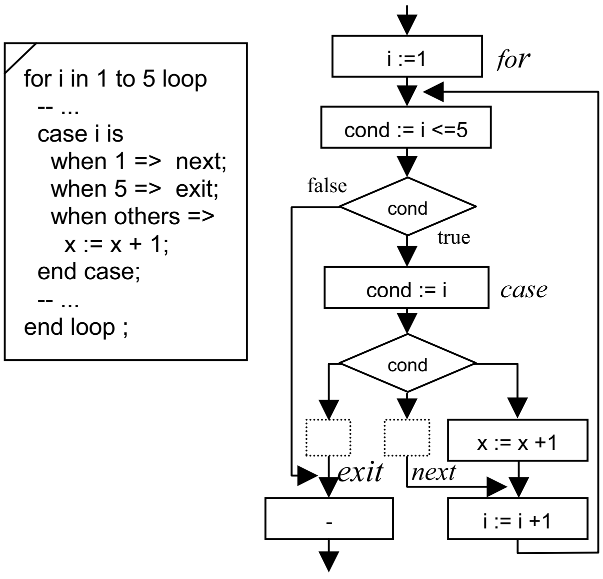

```
for i in 1 to 5 loop
  -- ...
  case i is
    when 1 =>  next;
    when 5 =>  exit;
    when others =>
      x := x + 1;
  end case;
  -- ...
end loop ;
```

Figure 30 Control flow mapping example

Structure

From the structural constructs, *entity* and *architecture* are supported. Currently only port declarations are allowed in entities.

Concurrent statements

Only a single process is allowed per architecture because of the underlying single thread execution model of HLS.

Behavioral statements

All statements that can be mapped onto control flow represented by a flow-chart are supported (see section 3 for details):

- *assignments* are divided into atomic operations and mapped onto sequences of operation blocks;
- *if-then-else* statements are mapped sequences of decision and operation blocks;
- *case* and *loop* statements are first mapped onto *if-then-else* constructs and then onto flow-chart blocks; and
- *wait on* statements are bound to special associations on the corresponding edges of the flow-chart.

Some of the unsupported constructs, especially those that support behavioral hierarchy, will be added later. They are not necessary to describe modules at behavioral level but allowing them makes descriptions more flexible and generic. The constructs to be added later are *library*, *package* (user defined), *generic*, *function*, and *procedure*. We do not intend to allow constructs that are either simulation oriented, e.g., configuration and file operations, or are at non-behavioral abstraction levels, e.g., components and data-flow constructs. The higher level constructs should be replaced with supported constructs during earlier design phases, and the lower level constructs can be easily handled by corresponding back-end synthesis tools.

5.1.2 Mapping VHDL onto IRSYD

This section describes the correspondence between VHDL and flow chart constructs.

Entity

All ports in a VHDL entity are kept as ports also in IRSYD. All data types are converted into signed or unsigned bit-vectors (IRSYD data types).

Architecture

Signals and variables remain intact in IRSYD, only data types are converted into signed or unsigned bit-vectors when necessary. VHDL constants are preserved also as constants in IRSYD but necessary data type conversions will be performed.

Process

A process with the sensitivity list is first transformed into a process without the sensitivity list. The sensitivity list of signals is replaced with equivalent *wait on* statement in the very beginning of the process. After that, such transformed VHDL process or any other process with wait on statements is handled as simple process.

Behavioral statements

Behavioral constructs in VHDL process are translated into operation and control blocks of IRSYD. Figure 30 illustrates how *loop* and *case* constructs are mapped onto a sequence of IRSYD blocks. The other statements are also mapped onto sequences of control flow nodes. An *if-then-else* statement is modeled like the *case* statement except the condition node has only two successor nodes.

5.1.3 Control flow extraction

The IRSYD flow-chart is extracted from the VHDL source file with the help of a special translator that consists of a symbol table, lexers and parsers. The translator has four passes and there is a dedicated parser for each pass. Several passes are needed because some features of the VHDL are difficult to handle in a single pass.

Symbol table

The symbol table contains information about identifiers and types in the source file. The table is an important component of interface between different translator passes. It is organized as a binary tree. During parsing, the data is searched and updated in the symbol table. The table is created and predefined data types are initialized at the beginning of the translation.

Lexers

The task of the lexers is to recognize the tokens in the source code and pass them to parsers when asked. The translator is switching between lexers depending on the detected tokens. A separate lexer is used to handle VHDL comments, that is, to ignore them.

Parsers

The parsers used in the translator are of recursive descent type. The parser for the first pass, which reads the VHDL source, has to look at least two tokens ahead. In some cases, when this is not enough special syntactic predicates are used to determine correct parser rule production i.e. a concurrent grammar rule. A syntactic predicate specifies the syntactic context under which production will match successfully.

Pass one**:** The first parsing pass builds up an intermediate representation (IR). We use abstract syntax tree (AST) as IR (see Figure 31). It should be noted that it contains not only the content of the input stream but the structure of the underlying language as well. For example, a linked list of the input tokens has complete content but it has no structure to indicate how the input was parsed. The purpose of IR is to simplify the parsing during the subsequent translation phases. AST structure is carefully crafted to simplify parsing, i.e., any parsed token should match only one production at time.

During the first pass, some delimiters, e.g., brackets, colons, and obsolete keywords, are skipped. Several new tokens (tree nodes) are inserted into AST to simplify later parsing passes. In such a way, we can reduce parsing to look only one token ahead. All expressions will be translated into prefix notation - operation comes before arguments. The first pass starts also to fill up the symbol table. While parsing VHDL declarations - architecture, process, etc. - appropriate entries into the symbol table are made. That is, the name of the constant, signal or variable, its type, and value, if applicable, are inserted. Also, necessary type conversions are performed. At the same time, the syntax is checked and errors are reported.

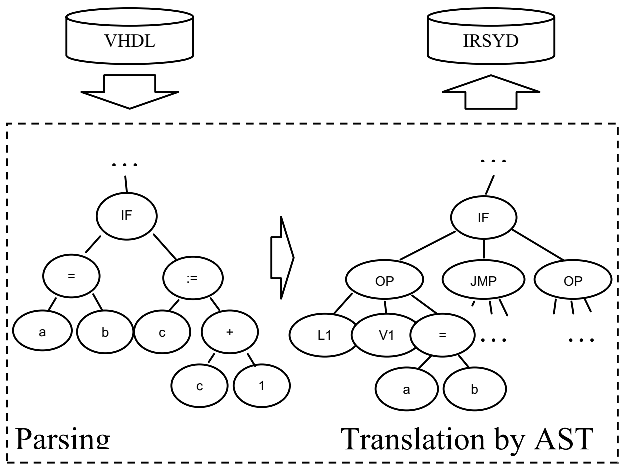


Figure 31 Compilation flow

Pass two: The IR, created during pass one, is kept in the main memory throughout all translation passes, temporary files are not used. During pass two, the IR is parsed again and further modifications are carried out using a special tree parser. The principles of the tree parsing are similar to that of ordinary parsing, only instead of tokens in a text file we have nodes in AST. During the second phase of translation, all sequential VHDL constructs - *if*, *case*, *for*, etc. - are replaced with Control-and-Data-Flow-Graph (CDFG) primitives. That is, jump, conditional jump, and label tokens are added to the intermediate tree in order to model control flow of sequential VHDL. Complex expressions are decomposed into simple two operand expressions. Intermediate variables are introduced when necessary, inserted into the symbol table, and propagated throughout decomposed expressions. For instance, an expression "y:=(a+b)*c;" is replaced with two expressions: "y:=v_1*c;" and "v_1:=a+b;".

Pass three: The order of these atomic expressions is corrected in pass three.

Pass four: The IRSYD output generation. First, all constant, port, signal and variable declarations are dumped from the symbol table into the target file. Then, once more the current AST is traversed. Since there is now one to one mapping between AST sub-trees and IRSYD constructs, translation to IRSYD is a straightforward process. Following synthesis steps of xTractor use the generated IRSYD flow-chart.

A compiler construction tool-set PCCTS [57] from Purdue University was used for building lexers and parsers. PCCTS is similar to a highly integrated version of LEX and YACC [58], but it offers some additional features that make it easier to use. Now there is also Java based version of PCCTS called ANTLR available.

5.1.4 HLS tool xTractor

High-level synthesis tool xTractor [55] uses IRSYD flow-chart produced as described above. xTraxtor is an academic high-level synthesis tool that was developed to test synthesis methodology of control and memory intensive systems (CMIST). The overall synthesis flow (see Figure 32) is similar to the synthesis flow of any HLS approach (see, e.g., [56]). The three main steps can be outlined as follows:

- in the partitioning phase memories are extracted from the initial behavioral;
- operations are assigned to states (control steps) during the scheduling phase; and
- unified allocation and binding assigns operations to specific functional units.

The separate steps of the flow are executed by component programs that input, output, and manipulate synthesizable subset of IRSYD. Two of the component tools are used as input and output of the tool-set. One of the programs - IRSYD Generator - compiles either a subset of C or VHDL into IRSYD. The second tool - RTL HDL Generator - generates register-transfer level VHDL or Verilog code for back-end logic level synthesis tools. Some simpler Data-Path Transformations can be applied before (or after) every main step. Although there are many different transformations are available, only the most obvious ones have been implemented - constant propagation, variable propagation, and simplification of operations with constants. Memory Extractor lists arrays and/or maps them onto memories. It should be noted that although most of the synthesis steps can be skipped, the Scheduling phase is an exception because it is the only step that assigns states to the behavioral control flow. So-called as-fast-as-possible (AFAP) scheduling strategy is used [55]. Allocator/Binder allocates and binds operations and variables into functional units and registers. Interconnections (multiplexers) are allocated and bound together with related functional units and/or registers.

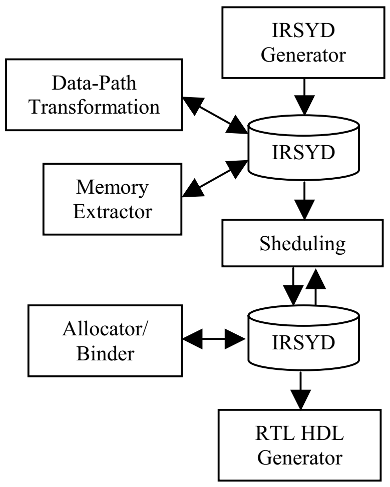


Figure 32 xTractor synthesis flow

A separate component, not shown in Figure 32, is an interactive shell that organizes the overall synthesis flow, that is, in which order and with which parameters the component programs are called. The shell organizes also the graphical user interface (GUI) and menu systems. To illustrate the power of CMIST approach, synthesis results of few sub-modules from Operation and Maintenance (OAM) module of ATM switch are presented in Table 17. The results characterize the area of modules in the number of equivalent gates for AMS 0.8-micron target technology. The columns "Tool 1" and "Tool 2" show results from two commercial HLS tools.

Design	Tool 1	Tool 2	xTractor
Full OAM	22434	22184	14663
buffer	365	311	129
FMCG	1222	1303	722
input handler #1	6310	5783	4366
input handler #2	3728	3423	2781
input handler #3	3027	3203	1688
output handler #1	3740	2891	2351
output handler #2	351	519	248

Table 17 Synthesis results of OAM module

5.1.5 Decision diagram synthesis from RTL level VHDL

The main goal of representing VHDL descriptions by decision diagrams (DDs) for test generation is to extract the functionality of the design for setting up targets (primitive functions or operations) for testing and for carrying out the functional test generation or to guide efficiently the hierarchical test generator by using high-level diagnostic information.

Here we assume that the architecture of the circuit is described at the Register-Transfer Level (RTL) as data path and a control path as shown in Figure 33 [59]. Control path is a Finite State Machine (FSM) with a state register x_S, next state logic and output logic. Input signals to the FSM are the primary inputs of the design (variables x_I), conditional signals originating from the data path (variables) and current value of the state variable x_S. Outputs of the FSM are the primary outputs of the design (variables x_O), control signals (variables x_C) and the next value of x_S.

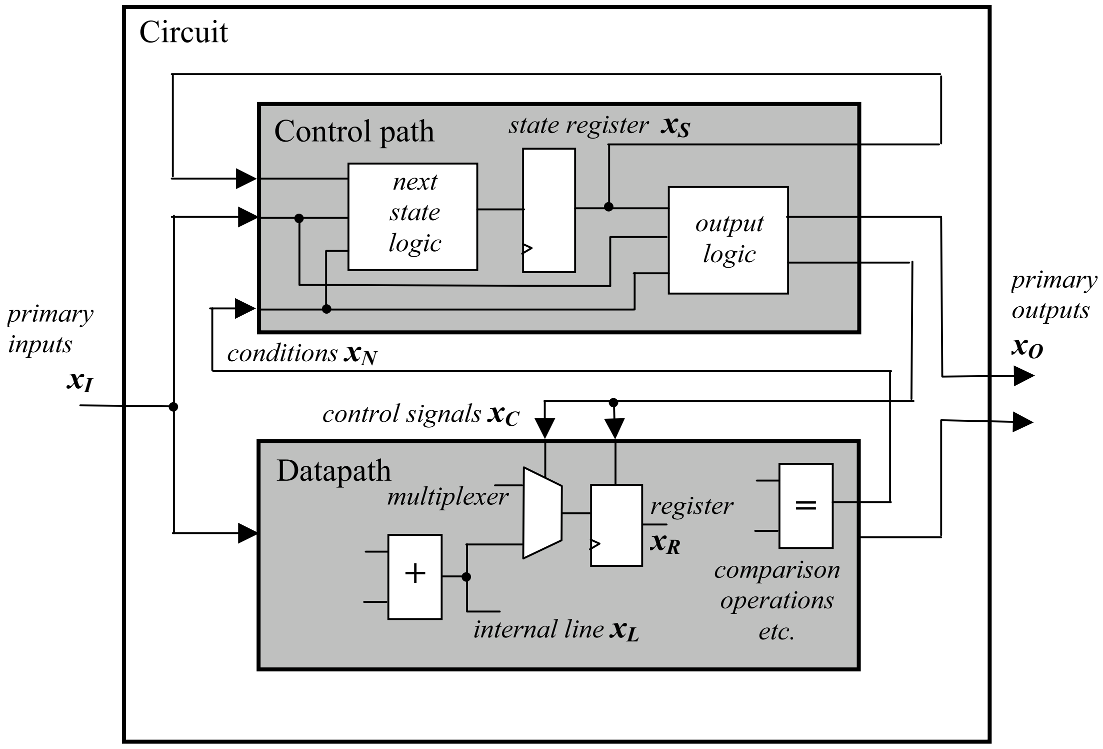


Figure 33 Register-transfer level view of a digital circuit

Data path can be viewed as a network consisting of modules or blocks. These include registers, multiplexers and functional units (for implementing operations). All the registers and some internal lines of the data path can be represented by variables in the RTL DD model (variables x_R and x_L, respectively). Inputs for the data path are the primary inputs x_I and control signals x_C (e.g. multiplexer addresses and register enable signals). Outputs are the primary outputs x_O as well as conditional signals x_N (e.g. from comparison operators) leading to the control part FSM.

In DD models representing the data path, the non-terminal nodes correspond to control signals (labeled by variables x_C). The terminal nodes represent operations (functional units). Register transfers and constant assignments are treated as special cases of operations. Figure 36 shows a simple example of a DD representation for a data path fragment.

At the RT-level, data path is generally represented by a system of DDs. For each primary output, fan-out signal and register a DD corresponds. In addition, multiplexers that are connected to an input of an FU are represented by a separate DD.

The DD synthesis from VHDL consists of several steps. Data path and control path will be converted into DDs separately. Additionally the entity declaration, signal declarations and constants should be converted. After initial synthesis the data path DD can have some redundancy which should be removed. As a final step, the DD model will be written to the file.

Control path DD model generation

The control path of the design should be represented as a finite state machine (FSM) transition table, behaviorally described in VHDL like in Figure 34. The FSM state transition table is presented as one process, describing next state value and output signal values in relation with the current state value and input values.

```
entity fsm is
    port( rst, clk       : in bit;
          in1            : in bit;
          y1, y2, y3, y4 : out bit
    );
end fsm;

architecture fsm_arc of fsm is
    type states is (s1,s2,s3,s4,s5,s6,s7,s8,s9);
    signal nState, cState: states;

begin
    process( rst, clk )
    begin
      if( rst = '1' ) then
          cState <= S1;
      elsif( clk'event and clk = '1' ) then
          cState <= nState;
      end if;
    end process;

    process(in1, cState)
    begin
      case cState is

        when s1 =>
                     nState <= s2;
                     y1 <= '1';
                     y2 <= '0';
                     y3 <= '0';
                     y4 <= '0';

        when s2 =>
                   if ( in1 = '1' ) then
                     nState <= s8;
                     y1 <= '0';
```

```
                        y2 <= '0';
                        y3 <= '1';
                        y4 <= '0';

                      elsif ( in1 = '0' ) then
                        nState <= s9;
                        y1 <= '0';
                        y2 <= '0';
                        y3 <= '1';
                        y4 <= '0';

                    end if;
            .
            .
            .
        end case;
    end process;
end fsm_arc;
```

Figure 34 Example of FSM description in VHDL

The DD model is synthesised in two steps. At the first step, the RTL level VHDL description is converted into an intermediate memory structure. At the second step, the intermediate memory structure is converted into the DD. In Figure 35 is shown how the DD model is created from FSM state table description.

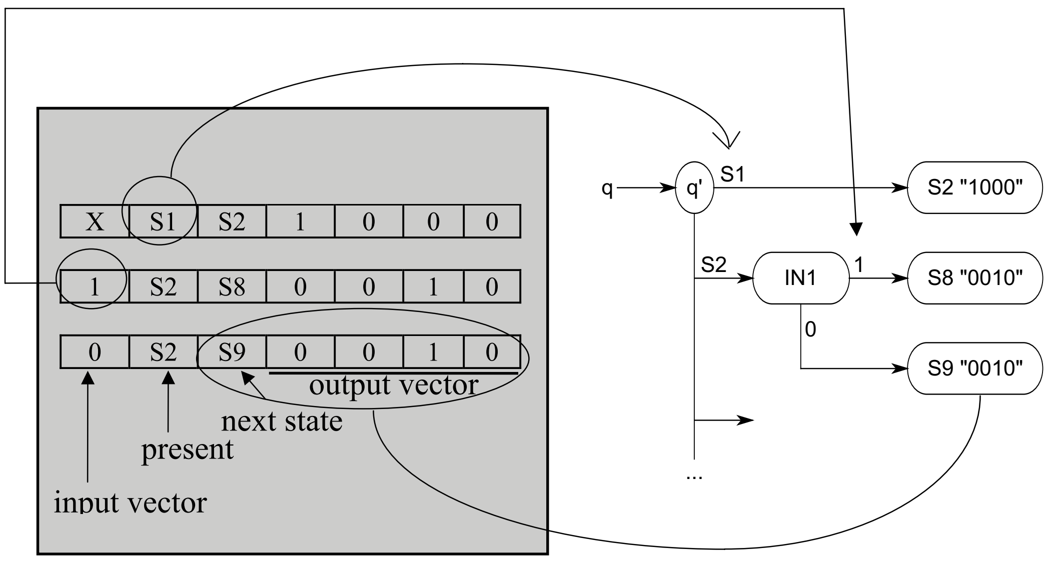


Figure 35 DD generation from the FSM State Transition Table

Data path DD-model generation

The data path DD-model is created from the device data-path specification. The data-path VHDL description is based on the library of predefined modules (registers, multiplexers, arithmetic and logic blocks). To provide flexibility, there is similar library of DD modules, which is used during the DD synthesis process. Such approach allows introducing additional modules, whenever needed.

At the first step of the data path DD synthesis, for every functional unit (multiplexer, register, arithmetic or logic block) the corresponding DD from the library will be selected. At the second step minimization of the initially generated data path DD model will be done.

For example, in Figure 36a the sample data path fragment is depicted, which contains two registers, one multiplexer and one adder. This fragment may initially be represented by three DDs - one for register reg2, one for multiplexer and one for adder. After selecting DD models for every VHDL component instantiation statement, obtained model should be minimized, using superposition process. Final, minimized DD is depicted in Figure 36b.

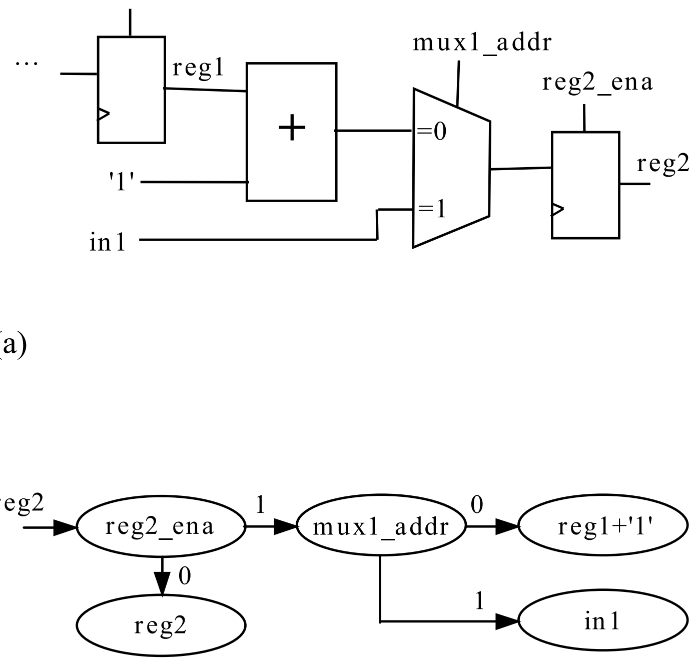


Figure 36 A data path fragment (a) and its DD representation (b)

5.2 Digital design flow with automated test generation

5.2.1 Test generation flow with ATPG

Test flow with gate level test generator and hierarchical test pattern generator is shown in Figure 37. In case of gate level test generation, design specified as register transfer VHDL can be synthesized with Synopsys Design Compiler. As a result, we get flattened gate level circuit model which in turn is converted into decision diagram (SSBDD) model. Then gate level ATPG like genetic test generator can start. Second flow can be with hierarchical (two level) test generator [59]. Here, both RTL level decision diagram model and gate level decision diagram models (SSBDD models) for every functional unit in design under test are needed. Finally, test patterns generated by hierarchical ATPG, must be evaluated at gate level. Therefore, test patterns must be converted accordingly and fault simulated.

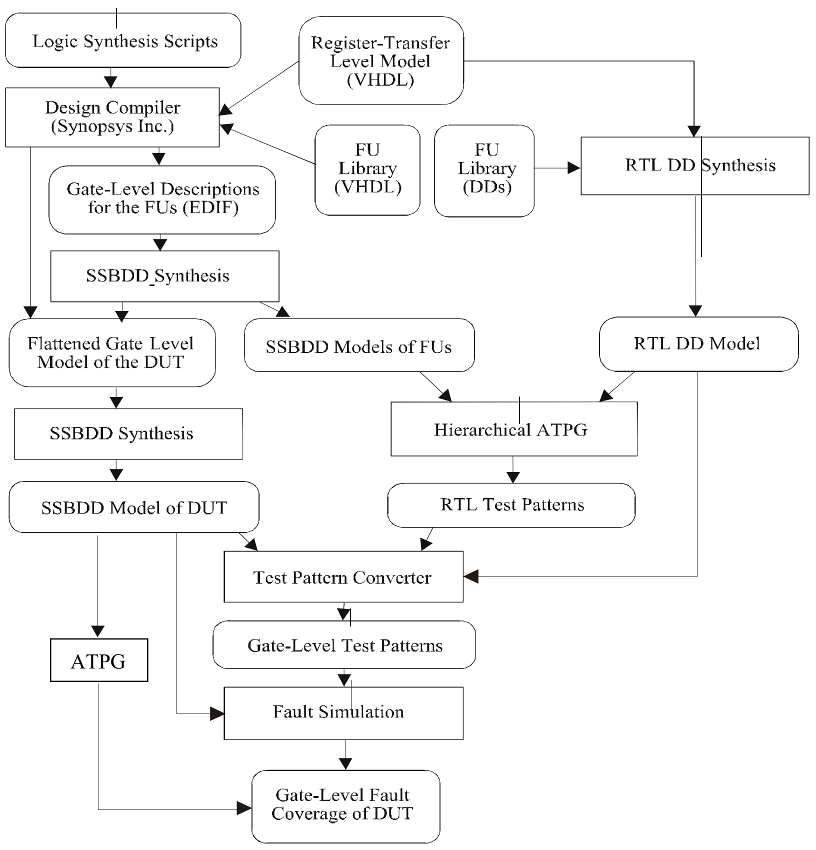


Figure 37 Test generation flow

5.2.2 Description of the complete Design flow

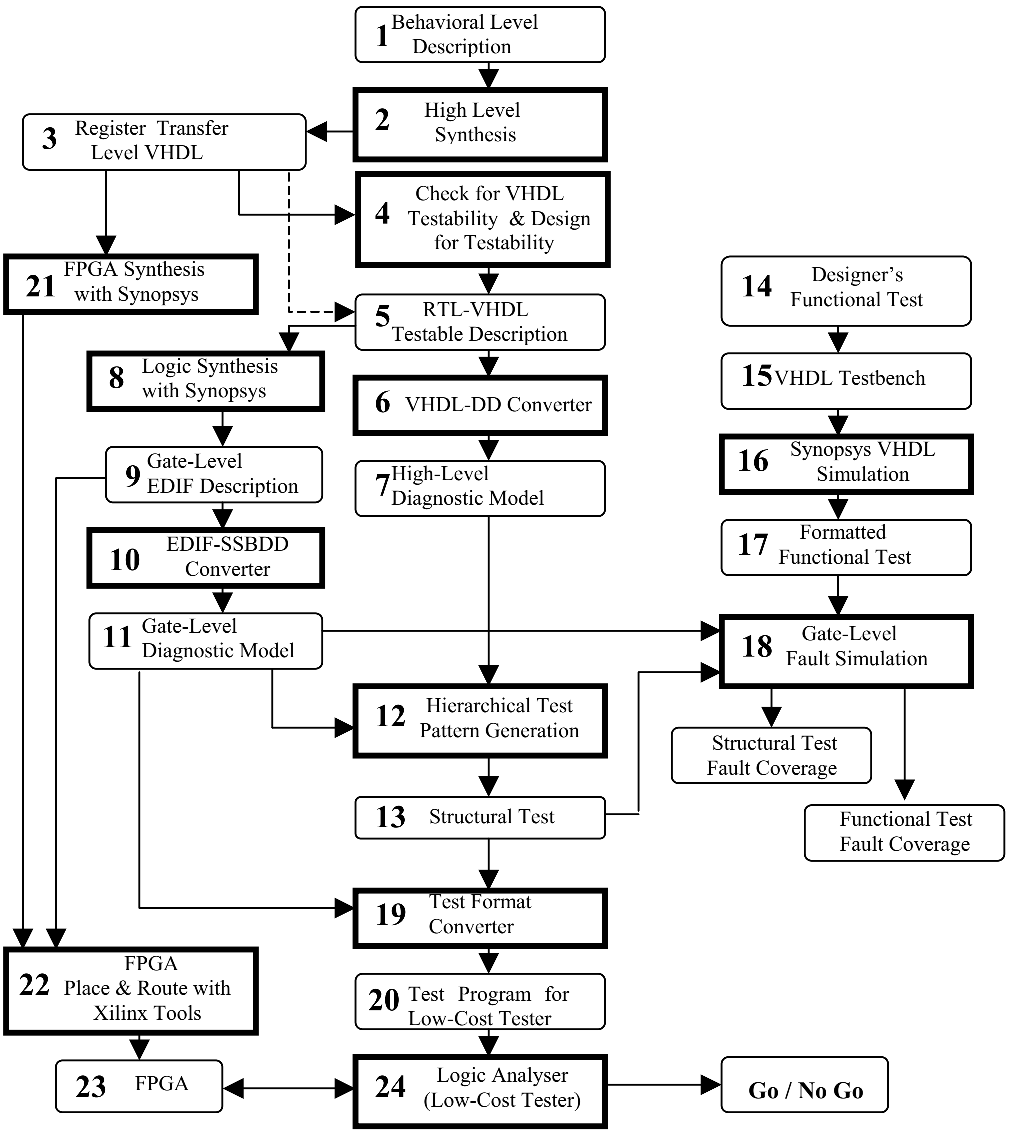
Generally, the FPGA development cycle can be interpreted as a sequence of the following design steps (Figure 38):

working out the system specification as a behavioral description i.e. a VHDL system model (in most cases on algorithm level or register transfer (RT) level) and simulation patterns (1,2,3,14);

working out a synthesizable description (VHDL system model) with using methods and tools for modeling, simulation and analysis (4,5);

synthesis by using methods and tools - high level synthesis, logic synthesis (8,9,21);

partitioning, place and route with using tools for layout generation, back annotation and simulation (22); programming the FPGA (23); testing by a low-cost tester or logic analyzer (24).



1 Behavioral Level Description
2 High Level Synthesis
3 Register Transfer Level VHDL
4 Check for VHDL Testability & Design for Testability
5 RTL-VHDL Testable Description
6 VHDL-DD Converter
7 High-Level Diagnostic Model
8 Logic Synthesis with Synopsys
9 Gate-Level EDIF Description
10 EDIF-SSBDD Converter
11 Gate-Level Diagnostic Model
12 Hierarchical Test Pattern Generation
13 Structural Test
14 Designer's Functional Test
15 VHDL Testbench
16 Synopsys VHDL Simulation
17 Formatted Functional Test
18 Gate-Level Fault Simulation
Structural Test Fault Coverage
Functional Test Fault Coverage
19 Test Format Converter
20 Test Program for Low-Cost Tester
21 FPGA Synthesis with Synopsys
22 FPGA Place & Route with Xilinx Tools
23 FPGA
24 Logic Analyser (Low-Cost Tester)
Go / No Go

Figure 38 The combined FPGA Design and Test Flow

Each design step starts up from a system model given as representation (fitted to) appropriate to different abstraction levels to be passed through within the design process, and it ends up with a model of an implementation. Based on that, the next design step can be executed, and will lead to a more refined model of implementation, and so on.

5.2.3 Experiments on proving the motivation of using the ATPG

In the frame of EC COOPERNICUS JEP 977133 VILAB project, following experiments [60,61] were carried out in order to check the efficiency of using the ATPG. The Huffman encoder circuit developed in EAS IIS Dresden, a complex multiplier circuit *mult8x8*, and two well-known international high-level synthesis benchmark circuits *diffeq* and *gcd* were chosen as testing samples. The Huffman encoder is a part of the sender side of a video signal transmission system. The register-transfer level VHDL description of the encoder was synthesized by a high-level synthesis tool from Univ. Tuebingen, C-LAB Paderborn and Univ. GH Paderborn. The description was further applied to logic synthesis with SYNOPSYS Design Compiler and FPGA placement and routing with XILINX software. The resulting FPGA had a complexity of about 1300 gates. To show the real need of developing a dedicated compact structural test for a FPGA design, the quality and fault coverage of the designer's functional test for the Huffman Decoder was investigated for comparison against the automatically synthesized structural tests.

For this experiment, at first, the procedure of logic synthesis by Synopsys CAD tools (8) for the design (3) created at the EAS Dresden, was carried out. Then, a gate-level diagnostic model (11) of the logic design in the form of SSBDDs was generated by the converter (10). Finally, the fault simulation experiment (blocks 14-18) was carried out. A Test Format Converter (19) was needed for transforming the test sequence produced by the ATPG into the test program for the logic analyzer to carry out the test experiments on a low-cost tester (logic analyzer). Experimental results on structural test generation for three highly sequential circuits (with global feedback loops embracing the control and data paths) shown in Table 18. The experiments showed that the DD based test generation algorithm runs (on the example of *gcd*) an order of magnitude faster than previously published approaches [13]. The functional test of the Huffman encoder with a length of 3,5 millions of patterns led to fault coverage of 61,5% only.

Circuit	**Gates**	**Faults**	**PIs**	**POs**	**FFs**	**Control states**	**Fault cover %**	**Test length**	**Times**
Functional test									
Huff.enc.	~1300	5336	31	30	118	27	61,5	9658*	-
Structural tests									
Huff.enc.	~1300	5336	31	30	118	27			
diffeq	4195	15836	81	48	115	6	95.4	3277	20.4
mult8x8	1058	3975	17	16	95	8	79.5	2846	14.8
gcd	227	844	9	4	15	8	91.0	924	5.6

* Fault simulated part of the total 3,5 millions patterns (the last 5000 patterns gave only 1%)

Table 18 Test Generation Results of the ATPG

5.3 Discussion

Nowadays design flow often starts with description of the system on the behavioral level using hardware description languages like VHDL. We have presented here the key issues of translating descriptions written in behavioral VHDL into IRSYD, an internal representation format suitable for control and memory intensive digital systems synthesis used by academic high level synthesis tool xTractor. The main concern was the mapping of VHDL constructs onto IRSYD equivalents while preserving the semantics of the original VHDL code, and extracting the control flow from source code. A simple synthesizable subset was selected while keeping in mind possible future extensions. The front-end compiler is built with a popular compiler construction tool-set PCCTS. The compiler translates the behavioral VHDL subset into control oriented flow-chart like description IRSYD that is used by the synthesis tool for data exchange. Current design-pattern of compiler can be used for other HDL input languages as well. The compiler prototype has been tested in the design flow of an academic high-level synthesis tool xTractor which will output RTL level VHDL code what can be further used for Decision Diagrams synthesis, which in turn are used by test generators.

Complete flow with design and test tools presented in this chapter, can be regarded as valuable result by itself. Usually design and test fields develop separately and results are therefore hard to unite. Here is created a link between results of design process and test generators. Consequently, ATPG-s elaborated in current thesis, are practically usable in automatic design and test flow, there is no need for manual intervention while preparing input for ATPG.

During work with VILAB project several design and test tools were integrated into complete automatic flow functioning remotely over the Internet, it was shown that it is practically feasible. Different tools were situated geographically different places. Many experiments were carried out, among them with industrial example.

Experiments with encoder circuit for telecommunication have shown that functional tests developed based on designers' experience by hand are tainted with two essential drawbacks: they are much longer than the automatically synthesized structural tests, and generally, functional tests do not offer a sufficient structural fault coverage. The new automated approach tends to drastically reduce the test cost and brings out a remarkable progress to control (master) the test problem.

6 Web-based Environment for Digital Design and Test

6.1 Internet-based collaborative design and test with MOSCITO

The Internet offers a new dimension and new solutions in engineering works and gives new possibilities to use tools from different sources. Subsequently there is described how the new communication technologies were applied in the area of digital systems testing with MOSCITO system [62] developed at Fraunhofer IIS/EAS, Dresden, Germany. Several TPG tools running at geographically different places have been selected for integration into the new virtual environment.

6.1.1 Overview of MOSCITO

MOSCITO system was implemented with the purpose to connect different tools together via network to form a uniform workflow for solving problems in electronic circuit design. The software was developed regarding the following aspects:

- Encapsulation of design tools and adaptation of the tool-specific control and data input/output to the MOSCITO framework (MOSCITO agent, see below).
- Communication between the tools for data exchange to support distributed Internet-based work.
- Uniform graphical user front-end program for the configuration of the tools, the control of the whole workflow and the visualization of result data. One important goal is to provide the functionality of a tool (e.g. fault simulator, a test pattern generator, a netlist converter, ...) to a potential user as a service in a local area network (LAN) or in the entire Internet.

This approach is similar to the ASP idea (Application Service Provider). In the present system the following tools have been integrated in MOSCITO:

- several converters for EDIF, ISCAS, and VHDL design description formats
- Turbo-Tester tools for logic level fault simulation and test generation [54]
- DECIDER - a hierarchical test pattern generation for digital systems [63]
- DefGen - ATPG for IDDQ and voltage testing of digital circuits [64,65]
- Tst2Alb - a data converter between ATPG tools
- ALB - an automatic fault library builder [66]

All the tools can act as MOSCITO agents and each of them provides a certain service. The user should be able to combine all the services to a problem specific workflow. That means, the needed tools have not to be installed on the users local computer. It is sufficient, that the services are available via the network. Due to that fact for the user the effort for installation, configuration and maintenance of software will decrease. Furthermore, specialized tools can be executed on their native platform with a high performance (e.g. supercomputer with fast CPUs and large memory, Workstation-Cluster). This will speed up the entire workflow.

Remote computing in this way is important for application with huge amount of computing time: e.g. fault simulation as well as test pattern generation.

The MOSCITO framework was implemented in JAVA and can be used on different computing platforms. At the moment MOSCITO is used on SUN workstation (Solaris) and on PCs (Microsoft Windows and LINUX). In the following, some of the most important concepts of the implementation are discussed.

6.1.2 Software architecture

MOSCITO consists of three software layers:

- kernel layer,
- interface layer,
- extensions.

The kernel provides functionality for basic object and data management, file handling, XML processing, and communication. Because MOSCITO is an open system, a special interface layer provides programming interfaces for integration of new tools, new workflows and appropriate viewers such as for diagrams, plain text and images. Each interface is represented by a Java class, which contains the basic functionality. The user only needs to extend this class and can implement its own extension. A large number of templates and example implementations help the user to integrate a new tool or workflows.

6.1.3 Tool encapsulation

In order to integrate design tools (e.g. fault simulators, test pattern generators, netlist converters) into other systems usually it is necessary to implement an additional software layer. In MOSCITO, this layer is realized as a special agent interface (MOSCITO agent). An agent must carry out the following tasks:

- adaptation of input data to the embedded tool, e.g. generation of configuration scripts or input files
- adaptation of output data, the tool-specific data formats (simulation results, log files, test vectors) have to be converted
- the mapping of control information to the embedded tool and the transfer and conversion of status information (warning and error messages), which have to be submitted to the user.

To provide the opportunity of the integration of a broad spectrum of tools and use them as a service in MOSCITO there are three ways for embedding programs into a MOSCITO agent:

• Integration of the entire program: the software has to be able to run as a batch job (e.g. DefGen, ATPG). In this way the integration of a large number of commercial tools is possible.

• Embedding of a library via the Java Native Interface (JNI): this way functions (optimisation algorithms etc.) written in e.g. C, C++ or FORTRAN can be called.

• Direct integration of Java-classes and applications, respectively - an easy way when the software which is to be integrated is written in JAVA.

Encapsulation of the tools as a MOSCITO agent guarantees a uniform interface to the framework. All tool specific details are in a special agent description file. Such file is used to create tool specific dialogs for the configuration of the tool via the front-end program.

Originally, following tools were integrated in MOSCITO:

• SABER (Analogy Inc.) and ELDO (Mentor Inc.) - mixed-signal circuit simulation supporting analog behavioral models

• SPICE (University of California, Berkeley) for die analog circuit simulation

• KOSIM (Fraunhofer IIS) - simulation of mixed-signal circuits and heterogeneous systems

• ANSYS (ANSYS Inc.) - FEM-Simulation

• MATLAB (MathWorks Inc.) - mathematical problems, block oriented simulation

• OPAL (Fraunhofer IIS) - optimization module

6.1.4 Communication

The implementation of the tool communication is based on TCP/IP-sockets. The tools can be executed on different computers or on different computing platforms (e.g. UNIX, Windows).

All we need for communication is a LAN or Internet access. Many problems caused by the limited availability of the tools (e.g. incompatible computing platform, insufficient resources) can be prevented in this way. A complicated aspect of communication is the format of data, which has to be submitted. Usually it is necessary to adapt/convert input as well as output data for each tool. To decrease the implementation effort for parsers and converters, the format for all data transmitted in MOSCITO was set to a special XML-Format, the Moscito Markup Language (MoscitoML). The main advantages of XML are:

• XML is an ISO standard

• XML allows the definition of application specific data formats (like used here for the Moscito Mark up Language)

• for XML-based data formats there is free parsing software available. Thus the implementation effort decreases considerable

• Data formats can be expanded in the future without changes in the parsers

• The integration of any kind of data is possible. It is no problem to include model descriptions or configuration scripts into XML.

6.1.5 General concept

MOSCITO uses a Client-Server concept. There is one Master Server, several Slave servers and arbitrary number of clients. The requested service is provided by Slave servers. That is because so-called Agents were attached to each Slave server. The Agents encapsulate service providing work tools (program executables). An Agent can be seen as an intelligent wrapper around a stand-alone program, which is capable of communicating with the Servers. All Slave servers are registered at the Master Server, so all Agents (i.e. services) are also registered at the Master server. Users access first the Master server and will get a list of available services. After selecting a service (Agent), the user is automatically re-directed to the Slave server, and after that, the work with the service providing tool can start.

The MOSCITO framework was implemented in JAVA and can run on different computing platforms. The only prerequisite is an installed Java Virtual Machine. At the moment MOSCITO is used on SUN workstation (Solaris) and on PCs (Microsoft Windows and LINUX).

6.1.6 Graphical User Interface

To offer a uniform and consistent concept for the user interaction the MOSCITO system has been provided with a graphical front-end with the following functionality:

- The problem description including all data can be read in from a MOSCITO project file.
- Workflows can be chosen from a set of predefined flows for the specific problem.
- A browser supports the choice of agents (tools) needed for the solution of the problem from the set of available services.
- With buttons for start, pause, resume and stop the workflow can be controlled by the user.
- A console window collects all messages from the running tools and allows the observation of the proper operation or trouble shooting, respectively.
- The visualization module MOSCITO Scope supports the display of all result data (test vectors, statistic information).

The graphical front-end aims at using design tools via the Internet in a simple and efficient manner. Actually, the front-end is available as a JAVA application and has to be installed together with the MOSCITO software.

6.1.7 Internet-based usage

At first, it is necessary to start one MOSCITO server on each host belonging to a domain of services. After that, an administrator has to register one or more MOSCITO agents so that they are available as remote services via LAN or Internet. Now a user can start the MOSCITO front-end program (GUI) and can browse through registered agents, can select, configure, and initialise the appropriated workflow and the needed agents. MOSCITO automatically calls remote tools and establishes direct connections between the tools for data transfer. Furthermore, the GUI allows the user to control and observe the data processing provided by a certain workflow. Result data are transmitted to the front-end and displayed by appropriate viewers. Finally, MOSCITO closes the connections between all remote tools and organizes correct termination of them.

6.1.8 Enhancements: working with firewall protection

Nowadays it is common to protect computers and especially intranets against viruses and hackers by so called firewalls.

Firewall can be regarded as filter, which allows certain type of communication (e.g. TCP/IP protocol based) “go through” certain configurable chock points, called ports [67]. Firewall is implemented for example as specialized software running on a well-secured computer or in hardware. Firewall has its filter rules. Internet is accessible only via that computer and vice versa- any computer in intranet is accessible only via that computer (through firewall). While speaking about opening a port in a firewall, then usually is meant that firewall filtering rules are configured appropriately.

At first, MOSCITO was intended for local area network use, not for Internet based use across firewall-protected systems. Therefore, it randomly used arbitrary number of the non-restricted communication ports above 1024. The problem nowadays is that that many other network applications also use these so-called free ports. There is no harm in internal network generally, but there will be security problems when such programs are directly exposed to Internet. The reason is that some of them are known to be vulnerable, i.e. they can be misused to attack the host computer they are running on. Tolerating one of such vulnerable programs will compromise the host computer and finally entire network. Consequently, in a restrictive firewall protected system there are only few ports left open for incoming Internet connections (like port 80 for http web server). In the case of restrictive firewall such MOSCITO solution would not work, because firewall blocks all the communication. In order to comply with firewall requirements, the major MOSCITO communication scheme was modified.

Simplified communication schema for MOSCITO in a firewall-protected environment is shown in Figure 39. Direct connections between subcomponents are not allowed. All the communication has to be organized through predetermined communication ports. Random port numbers are also not allowed. All the traffic goes only through firewall ports.

One possible solution for solving the firewall traversal problem is to implement MOSCITO proxy as Java application (Figure 41). Here we relay on usual MOSCITO socket based communication (TCP/IP sockets). By default, only some vital ports for computer system are configured to be open while all the rest are blocked. This means it is necessary to open up at least one dedicated port in a firewall for MOSCITO communication needs.

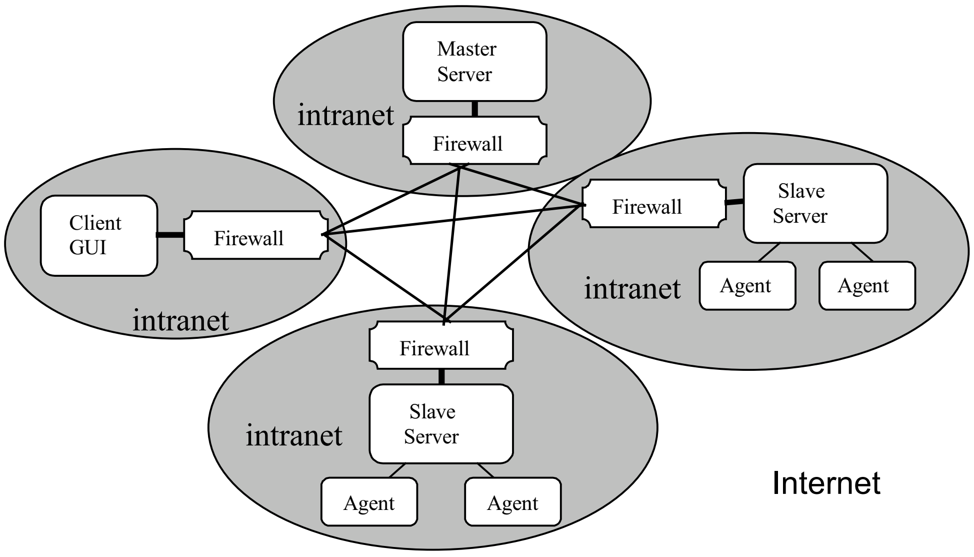


Figure 39 Communication between firewall protected MOSCITO subsystems: connections are allowed only between dedicated communication ports

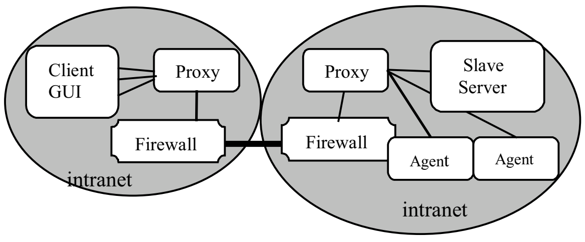


Figure 40 Communication between client and agents via proxy

Proxy mechanism (Authenticated Firewall Traversal) enables hosts in one side of proxy server to gain full access to hosts in the other side of the proxy server without requiring direct IP reach ability. It works by redirecting connection requests from hosts in one side to hosts in the other side to a proxy server, who authenticates and authorizes the requests, establishes a proxy connection and passes data back and forth.

6.1.9 Integrating test tools into the MOSCITO environment

Here is described the results of the COPERNICUS europroject JEP-97-7133 VILAB (Virtual LABoratory) on creating an environment for internet-based collaboration in the field of design and test of digital systems. Different CAD tools at geographically different places running under the virtual environment using the MOSCITO system can be used for microelectronics design, fault simulation and test generation purposes. The interfaces between the integrated tools were developed during the project work. The tools can be used separately, or in multiple applications in different complex flows. The functionality of the integrated design and test tools was verified in several collaborative experiments over Internet by partners located in different geographical sites.

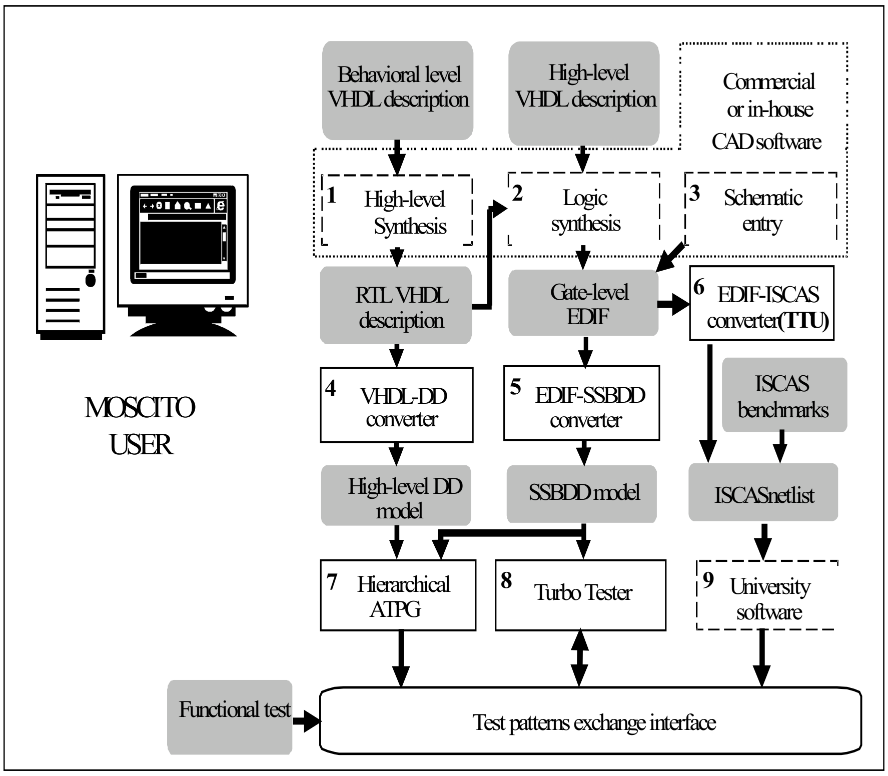


Figure 41 Work flows integrated to the MOSCITO environment

On Figure 41 there is shown, which tools were integrated with MOSCITO and what workflows could be formed. For example, in case of using genetic test generation flows could be: 1, 2, 5, 8 or 2, 5, 8. When using hierarchical test generator, then flow is more complicated since both high level and low level models are necessary: 1, 4, 7 and 2, 5. As we see hierarchical, ATPG (7) needs two inputs.

6.2 Enhanced WEB-based environment

6.2.1 Motivation

During the experiments with MOSCITO [62] virtual environment, the biggest problem was caused by corporate firewalls. It was hard to set up installations in a proper way. A proxy extension as Java program was proposed for MOSCITO, but it added undesirable effect of communication overhead. In addition, considerable amount of man-months would be needed to build next production quality MOSCITO release based on proxy solution. Therefore, more flexible solution using HTTP protocol and reusing some of general ideas of MOSCITO is presented subsequently.

6.2.2 General concept

A virtual environment to support the research and teaching of digital system testing is described below. User will be able to use test tools remotely over the Internet. System is based on an open architecture that allows easily add new tools later.

System core for remote tool usage has client-server concept similar to MOSCITO. There is one master server, several application servers and arbitrary number of clients (see Figure 42). Master server holds the information about application servers, which provide service. On application server so called agents can be invoked. Agents encapsulate actual test tools (executables). User first accesses the master server and gets a list of services available. After selecting appropriate service, user is automatically re-directed to application server, and then the work with the actual tool can start. The big difference from MOSCITO is HTTP protocol based communication and use of Java applets and servlets. In addition, user tracking is unique. There is no need to install tools on the user's local computer. Therefore, user's effort for installation, configuration and maintenance of software will be drastically reduced. The system is implemented in Java and can therefore run on different computing platforms. Actual work tools must run on their native platform of course.

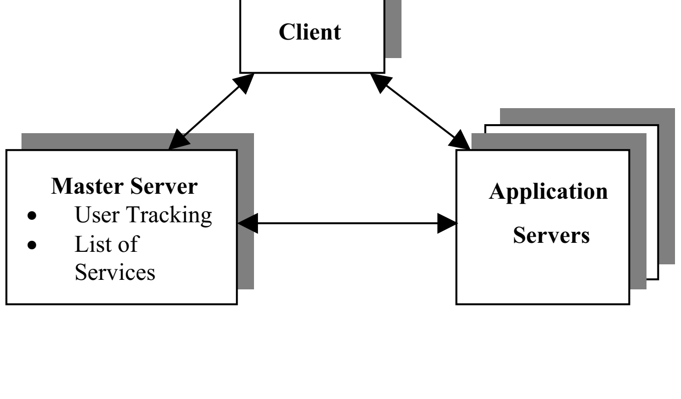


Figure 42 General concept of WEB based system

Each tool will be wrapped into Java agent. Encapsulation of entire tools guarantees a uniform interface to the framework. There will be no need to reprogram existing tools. Only requirement for tool encapsulation is that tool is able to run from command line (in ‘text mode’). All tool-specific details are stored in a description file. This allows automatically display appropriate tool configuration dialogs for end user.

Several tools can be started simultaneously. One servlet will serve many client applets in parallel. There is task queue management. Results reside initially on the server computer where servlet is running. Each user has its own server-side workspace. In the database, there is user’s workspace folder name where results for certain task id can be found. It is possible to query on results, make statistics. Subsequently, general concept is elaborated in detail.

6.2.3 Implementation

The environment for remote use of Test tools is built according to the client-server three-tier concept using HTML pages, Java applets/servlets and MySQL as database backend for user tracking and management tasks. General solution in details is given in Figure 43. Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and Java Server Pages technologies. Tomcat and servlets running on it play important role while gaining access to intranet resources on application servers and MySql database (platform independent open source DB).

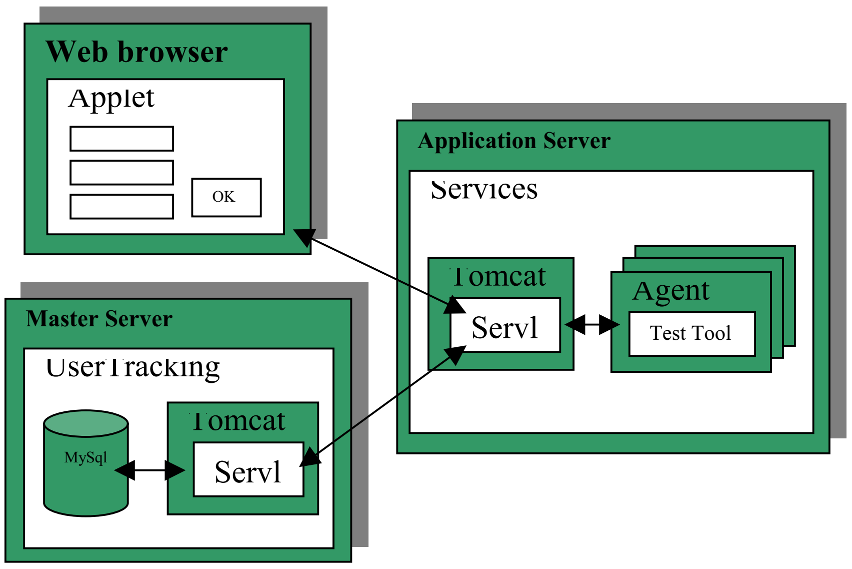


Figure 43 Implementation details

6.2.4 Usage scenario

Below is described simplified scenario for end user (see Figure 43), let's assume here that user has account already:

1. User logs in with login name and password. Information is sent to servlet which accesses database and verifies user. If user exists, then servlet sends confirmation message back to applet. Login screen is dismissed;

2. Applet displays tool's parameters dialog;

3. Tool's parameters are sent to servlet, which launches appropriate program and makes a new entry to tracker database "Tasks" table;

4. Client applet will be notified about successful start or failure;

5. Client can/must time-to-time check status of his task(s). E-mail could be sent to client when task is ready;

6. When status of the task is "completed" then user can see the results on his applet, can save them onto his computer.

6.2.5 Tool encapsulation

In order to integrate different tools, it is necessary to implement additional software layer. Each tool has to be wrapped into Java agent, which allows to adapt the input data to the embedded tool, convert the tool-specific data, simulation results (log files, test vectors, etc), map the control information to the embedded tool, transfer and convert status information (warning and error messages) to be submitted to the user.

Technically simplest way is to encapsulate tool as an entire program. Tool has to be able to run as a batch job. Integration of commercial tools is then also possible. Also embedding of a library (e.g. C, C++ routines) via the Java Native Interface (JNI) could be thinkable and also direct integration of Java-classes and applications (especially for Java software).

6.2.6 Communication

General communication is based on HTTP protocol. The tools on different computers and on different computing platforms (UNIX, Linux, Windows) can easily change data as serialized Java objects (datagrams). To minimize the implementation effort for parsers, translators and converters XML mark up language is used for configuration files and transmitted data. HTTP protocol allows us also easy firewall traversal as we can use default web server port and Java servlet extensions on web servers as sort of proxies in order to reach intranet resources. There is no need for opening extra ports in the firewall as it is the case in TCP/IP based communication.

6.2.7 User tracking

User management module is described in this section. Without proper user management, anybody in the Internet could possibly use valuable computer resources. Better practice would be to allow registered users access the resources. User tracking system allows us to monitor and control the usage of services. It may allow also billing the business customers. Main goal here was to provide sufficient set of basic functions to allow support user registration, tracking and management. User tracking is database based. Tool execution and data base access over Internet is carried out via Java servlet technology. Below the implementation specifics are given.

As we know, web-based http communication is stateless. This means that we have to keep track about all necessary information. As work tools tend to run long, then normal user's http session is not valid for such time period and result data is lost. We want to provide a possibility for user to come back online later to check his results. Therefore, we need to identify (track) users and save all their relevant data. Using so called "cookies" could be one solution, but database approach offers many advantages like powerful SQL query mechanism, speed, reliability, and consistency of data and ease of use.

User tracing module has open architecture, general API (application programming interface). With slight modifications, it is also usable for any similar web-based system, where user tracking is needed. It has three layers: presentation layer (user tier), business logic tier (data base queries, etc.), physical database (MySQL- platform independent open source DB).

First two layers are implemented in Java programming language. User is accessing database via presentation layer, not directly. This makes architecture open. User tier consists several functions to run business layer queries. For example, we could have different user interfaces for different applications. Then if database structure or business logic changes, we don't have to change our user interfaces. More over- it is easy to introduce user-tracking facility to new applications. It is much easier to invoke appropriate function (command), than construct a new query every time a new application needs one.

6.2.8 User interface

Graphical User interface (GUI) is based on collection of Java applets, which can be integrated into HTML page when needed (e-learning solutions). GUI applet reads the layout properties (field names, default values, etc) from initialization file. It would be easy for non-qualified Java programmer to introduce new tools into web-based environment by modifying initialization fail only. Features of GUI are following:

– Reading in problem description including data from project file;
– Selecting a service (tool) from the set of available services;
– Buttons to start and stop the tools;
– A console window collects all messages from the running tools;
– The visualization of all results (test vectors, statistic information);
– Downloading results: user clicks appropriate button which displays html page containing appropriate link;

6.2.9 Concept of grid computing for Test tools

Often testing tasks involve lots of computing power. Test tools tend to run too long on a single computer to obtain satisfactory test coverage or simulation results in everyday testing tasks. Therefore, a grid computing solution was worked out by extending a Web-based solution described above. This will allow running one task on several computing stations in parallel. Below is given overview of the concept (see Figure 44).

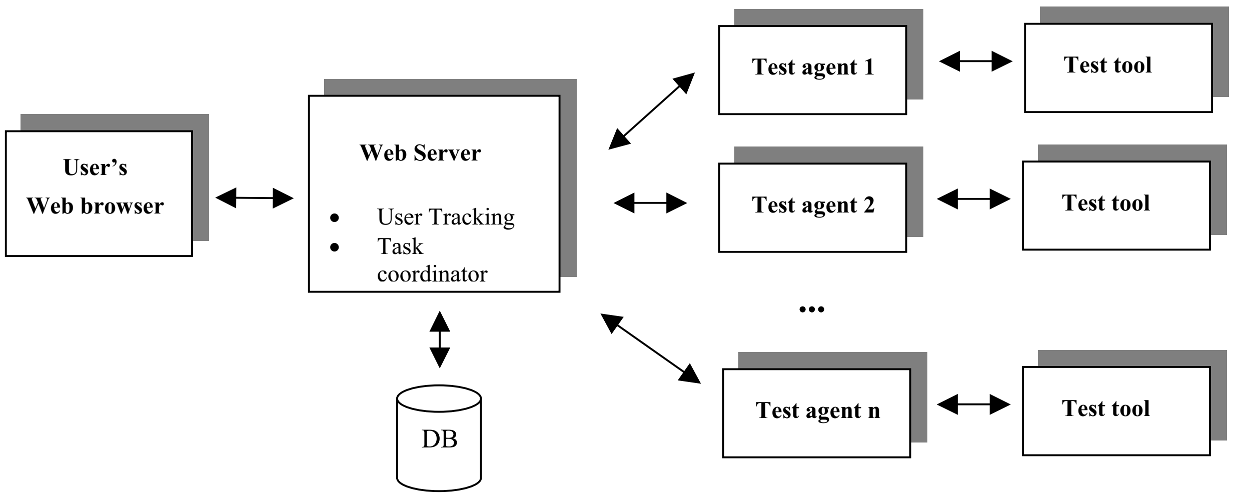


Figure 44 General Web-based grid computing concept

User specifies parameters and design file location for certain test tool. Thereafter user GUI contacts with coordinating web server and described parameters along the model are passed automatically. Task coordinator service process on Web server records all requests from user(s) and devides the task into subtasks. So called test agents poll constantly web server and if any subtask is sheduled by coordinator process, then test agent(s) receives the appropriate parameters and design file and will start actual native test tool. Test agent waits then until task will be completed and reports results back to coordinating web server which in turn assembles subresults into final result and forwards it to user when requested.

Test agent will not accept several tasks at a time unless the computer on it runs has several processors. Test agent and native test tool must reside on the same computer. Web server resides separately from test agents, they may not reside in the same local area network. Each agent can be on different local area network.

This solution is flexible and effectively works across the internet and through the firewalls as long as dedicated communication port for task coordinator service process on coordinating Web server is opened by administrator. On the client side there is no firewall configuration problem as communication uses standard port 80 for http communication.

6.3 Discussion

In the field of digital design and test, many different software tools are needed, but usually not all of them are available for a designer in his working site. Internet opens a new dimension offering new chances using tools from different sources without installation.

The ultimate goal of the work with MOSCITO platform was to have Internet based work environment (virtual laboratory) for design and test tools. User had to install only GUI environment. Main effort here was analysis of requirements and redesign of MOSCITO. New workflows were introduced in order to carry out collaborative design and test at geographically different places. Second challenge was integrate all the single tools into one automatic workflow. Several data exchange problems had to be solved. The essential features of this integration environment were experimentally proved in the frame of VILAB project. The results obtained are presented also in several papers.

During the integration, biggest problem was caused by corporate firewalls. It was hard to set up MOSCITO installations in a proper way. It very much depended on firewall configurations. Not all the systems allowed free outgoing Internet connections (i.e. connections with arbitrary port numbers). Therefore, proxy extension was proposed and implemented in Java for MOSCITO system. Connecting problem was solved in new prototype version of MOSCITO, but drawback was that such extra layer of Java software also slows down data exchange, although not critically. In section 6.2 there was proposed another possible solution to Internet based tool usage.

Compared to initial MOSCITO system, enhanced system prototype has following advantages: firewall traversal is not major problem anymore as communication is HTTP based i.e. web browsers can be used now. Use of Java applets as graphical user interface ensures feature rich, responsive working environment as it was in case of MOSCITO desktop installation. Installation overhead is much smaller compared to MOSCITO since it is sufficient if end user has installed Java runtime environment, which normally is easy to do. Every time when user starts new working session, fresh, up-to-date Java applet is loaded to users machine. Since applets are light, i.e. small in size by multi tier system design, start up is fast. System also has built in, database based user tracking ensuring that licensed users have priority access to system resources. System has also merits to provide SSL encrypted communication. Described enhanced solution was approved as a paper at IFIP 18th World Computer Congress, at Conference on Virtual Enterprises and Collaborative Networks.

7 Summary

Current thesis deals with digital testing in a web environment. Proposed were three genetic test generation software tools, simulation methods with experimental data for multi-level simulation and defect oriented simulation using decision diagrams, behavioral level VHDL front-end compiler for high level synthesis tool xTractor. Several design and test tools were integrated into complete automatic workflow based on the state-of the-art network collaboration platform MOSCITO in the frame of VILAB project. MOSCITO system was extended by proxy solution for firewall traversal. Entire design and test flow was verified for feasibility with sample designs. Based on extensive experience and feedback new enhanced http protocol based virtual environment for remote test system was finally proposed. Subsequently, results are given in detail.

Comparison of random and genetic test generators reveals that test sets of genetic generator are always more compact. During genetic test generation, dynamic test vector 'packing' occurs because vectors are carefully chosen all the time. Genetic generator performs better than random in last stadium in test generation when only hard-to-test faults are left. Shortly, genetic test generator is justified for large circuits.

For control path of digital system testing an approach based on genetic algorithms to generate test vectors was presented in order to detect hard-to-test faults. Circuits must be presented as state transition tables (functional level) and same information must presented also as gate level as hierarchical fault simulation was used to evaluate test sets. Program works standalone or together with the test generator introduced in [39][37]. Evolutionary program tries to detect faults, which had remained undetected. Comparison with deterministic test generator HITEC was made. Prototype program developed here received smaller run time and smaller number of vectors for benchmark circuits.

The main contribution of the work regarding genetic test generation is that differently from the known genetic algorithms, a fault oriented genetic approach for sequential gate level circuits is developed. Unique feature is the use of knowledge about the circuit under test. For example, input of reset signal is not altered during genetic manipulation because otherwise essential building blocks of test vector set are destroyed and noise is introduced to the algorithm which decreases convergence. Reset signal is made active once and then kept non active in the test sequences. Experiments show that targeting single faults can improve the convergence of a genetic algorithm. In comparison with other GA based generator GATEST, considerably better results were obtained, especially for Huffman encoder circuit. The experiments have shown that targeting single faults however suffers loss in run times in comparison to the other compared approaches. Good news is that better fault coverages are obtained by this technique compared to other solutions. Using more internal knowledge by doing some circuit preprocessing prior to test generation will probably have some potential in order to further limit the search space and improve the convergence of genetic algorithm. This method can be used when only gate level netlist for circuit is available. In comparison, genetic FSM based testing is faster since it is working also on higher level to make decisions, but gate level sequential method is more robust in terms of usability. Since circuit models are different, there is no comparison information of fault coverage available yet.

Experiments with hierarchical simulation methods for digital system presented in current thesis, show considerable speed gain. For instance, RT level (high level) simulation results on decision diagrams in comparison with two commercial VHDL simulators reveal up to 37 time boost up in simulation speed. Speed difference increases with size of the circuit.

At the same time, the low level fault simulation of SSBDD macros in comparison to the gate-level fault simulation on ISCAS'85 benchmark circuits demonstrated the differences in speed between 2.55 and 9.04 times. In addition, here, speed difference increases with size of the circuit in favor of SSBDD macros.

Hierarchical fault simulation results compared to the plain gate-level simulation results show speed gain up to 121 times. However, surprisingly in case of largest circuit speed gain is only minimal. Obviously gate-level generator takes the advantage of good observability of the circuit since number of outputs is large.

Hierarchical defect oriented fault simulation method for digital systems introduced here helps to reduce dramatically the computation cost of test quality analysis in digital systems. Decision diagrams are used as a mathematical model for systematic multi-level solution for fault simulation at three levels of abstraction - RT, gate- and defect levels respectively.

Experiments show that the stuck-at-fault based fault coverage is overestimated compared to the realistic defect coverage, and that the difference between stuck-at fault and physical defect coverages reduces when the complexity of the circuit increases. In the worst case, it was observed that the 100% stuck-at-fault test may cover only 50% of realistic physical defects.

Nowadays design flow often starts with description of the system on the behavioral level using hardware description languages like VHDL. In order to link the VHDL descriptions to automatic design and test flow, compiler front-end for academic high-level synthesis tool xTractor was created. Key issues of translating descriptions written in behavioral VHDL into IRSYD, an internal representation format suitable for control and memory intensive digital systems synthesis used by xTractor were presented here. The main concern was the mapping of VHDL constructs onto IRSYD equivalents while preserving the semantics of the original VHDL code, and extracting the control flow from source code. A simple synthesizable subset was selected while keeping in mind possible future extensions. The front-end compiler was built with a popular compiler construction tool-set PCCTS. The compiler translates the behavioral VHDL subset into control oriented flow-chart like description IRSYD that is used by the synthesis tool for data exchange. Current design-pattern of compiler can be used for other HDL input languages as well. The compiler prototype has been tested in the design flow with xTractor synthesis tool, which will output RTL level VHDL code what can be further used for decision diagram synthesis, which in turn can be used by test generators.

Complete, collaborative, Internet based workflow with several design and test tools as a result of integrative research in frame of VILAB project presented in this thesis can be regarded as valuable result. Usually design and test fields develop rather separately and are therefore hard to unite. Here is created a link between results of design process and test generators. Consequently, ATPG-s elaborated for example in this thesis, are practically usable in automatic design and test flow, there is no need for manual intervention while preparing input for ATPG. During VILAB project several design and test tools were integrated into automatic flow functioning remotely over the Internet using MOSCITO software platform, it was shown that such integration it is practically feasible. Several challenges had to be solved. First, tools had to be linked together within proper work flows, secondly MOSCITO platform had to be tuned accordingly to support the flows over Internet, last but not least – firewall traversal problems had to be solved. Different tools were situated geographically different places. Many experiments were carried out, among them with industrial example.

Experiments with encoder circuit for telecommunication have shown that functional tests developed based on designers’ experience by hand are tainted with two essential drawbacks: they are much longer than the automatically synthesized structural tests and generally, functional tests do not offer a sufficient structural fault coverage. The new automated workflow approach tends to drastically reduce the test cost and brings out a remarkable progress to control (master) the test problem.

Finally, based on VILAB project experience, new enhanced web-based system was proposed in this thesis. All the critical components and communication are tested for feasibility in a prototype solution. Compared to initial MOSCITO system, enhanced system has following advantages: firewall traversal is not major problem anymore as communication is HTTP based i.e. web browsers can be used now. Use of Java applets as graphical user interface ensures feature rich, responsive working environment as it was in case of MOSCITO desktop installation, actually as it is case in any installed application. Installation overhead from user perspective is much less compared to MOSCITO since it is sufficient if user has installed Java runtime environment, which normally is easy to do. Every time when user starts new working session, fresh, up-to-date Java applet is loaded to users machine. Since applets are light, i.e. small in size by multi tier system design, start up is fast. System also has built in, database based user tracking ensuring that licensed users have priority access to system resources. System has also merits to provide SSL encrypted communication to protect sensitive designs.

References

1. M. Niermann, J.H. Patel, "HITEC: A test generation package for sequential circuits", *EDAC*, 1991.

2. M.S. Hiao, et al., "Sequential circuit test generation using dynamic state traversal", *EDTC*, 1997.

3. D. Brahme, J. A. Abraham, "Functional Testing of Microprocessors", *IEEE Trans. Comput.*, C-33, 1984.

4. A. Gupta, J. R. Armstrong, "Functional fault modeling", *30th ACM/IEEE DAC*, pp. 720-726, 1985.

5. K.-T. Cheng, J.Y. Jou. Functional Test Generation for Finate State machine. ITC '90: International Test Conference 1990 pp 162-168

6. J. Lee, J.H. Patel, "Architectural level test generation for microprocessors", *IEEE Trans. CAD*, no.10, 1994.

7. J.Raik, R.Ubar. "Sequential Circuit Test Generation Using Decision Diagram Models", *DATE*, 1999.

8. S. Seshu and D. N. Freeman, "The diagnosis of asynchronous sequential switching systems". *IRE Trans. Electronic Computing,* Vol. EC-11, pp. 459-465, August 1962.

9. V.D. Agrawal, K-T. Cheng, P.Agrawal, "CONTEST: a concurrent test generator for sequential circuits". *DAC'25: 25th IEEE/ACM Design Automation Conference,* Anaheim, CA(USA, pp. 84-89), June 1988.

10. E. M. Rudnick, J. H. Patel. E. M., G. S. Greenstein, and T. Niermann, "Sequential circuit test generation in a genetic algorithm framework", *Proceedings of Design Automation Conference*, pp. 698-704, 1994.

11. D. G. Saab, Y. G. Saab, J. A. Abraham 1992. CRIS: a test cultivation program forsequential VLSI circuits. In Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '92, Santa Clara, CA, Nov. 8–12), L. Trevillyan, Ed. IEEE Computer Society Press, Los Alamitos, CA, 216–219.

12. F.Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, "A Genetic Algorithm for Automatic Test Pattern Generation for Large Synchronous Sequential Circuits". *IEEE Trans. on CAD/ICAS*, Vol. CAD-15, No. 8, August 1996

13. M. S. Hsiao, E. M. Rudnick, J. H. Patel. Automatic Test Generation Using Genetically – Engineered Distinguishing Sequences. In Proceedings of the Symposium on VLSI Test, 216–223, 1996

14. L.Ghanmi, A.Ghrab, M.Hamdoun, B.Missaoui , G.Saucier, K.Skiba. E-Design Based on the Reuse Paradigm. DATE'02. Paris, March 4-8, 2002, pp 214-220.

15. K. Kloekner. Collaboration Suport in Design – What is the Message from CSCW and Knowledge Co-Production? International Workshop on IP-Based SoC Design, Grenoble, October 30-31, 2002, pp 51-55.

16. Leandro Soares Indrusiak, Florian Lubitz, Ricardo Reis, Manfred Glesner. Ubiquitous Access to Reconfigurable Hardware: Application Scenarios and Implementation Issues. DATE’03. Munich, March 3-7, pp 940.

17. Wolfgang Mueller, Tim Schattkowsky, Heinz-Josef Eikerling, Jan Wegner. Dynamic Tool Integration in Heterogeneous Computer Networks. DATE’03. Munich, March 3-7, pp 946.

18. Tom Kazmierski, Neil Clayton. A two-tier distributed electronic design framework. DATE’02. Paris, March 4-8, 2002, pp 227-231.

19. Achim Rettberg, Wolfgang Thronicke. Embedded System Design based on Webservices. DATE’02. Paris, March 4-8, 2002, pp 232-236.

20. Robert Orfali, Dan Harkey, Jeri Edwards, Robert Crfali. Instant CORBA. John Wiley & Sons. 1997. ISBN 0471183334.

21. Technical Report: SOAP Version 1.2 Working Draft. http://www.w3.org/TR/soap12

22. Technical Report: Web Services Description Language (WSDL) 1.1.

23. Universal Description, Discovery and Integration. http://www.uddi.org

24. http://java.sun.com/javase/index.jsp

25. http://java.sun.com/products/javabeans/index.jsp

26. http://www.w3.org/

27. http://www.w3.org/TR/xbc-use-cases/

28. J.H. Holland, “Adaptation in Natural and Artificial Systems”, University of Michigan Press, 1975

29. Brad L. Miller, David E. Goldberg “Genetic Algorithms, Tournament Selection, and the Effects of Noise”, IlliGAL Report No.95006, July 1995

30. Goldberg, D. E., Deb., Thierens, D. (1993). Toward better understanding of mixing in genetic algorithms. *Journal of the Society of Instrument and Control Engineers*, 32(1) 10-16

31. Goldberg “Genetic algorithms”, Addison-Wesley USA,1991

32. S. R. Ladd, “Genetic Algorithms in C++”, M&T Books, 1996

33. R.Ubar, J.Raik, P.Paomets, E.Ivask, G.Jervan, A.Markus. “Low-Cost CAD System for Teaching Digital Test”, .Microelectronics Education. World Scientific Publishing Co. Pte. Ltd. pp. 185-188, Grenoble, France, Feb. 1996

34. F. Berglez, H. Fujiwara, “A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran”, Proc. of the Int. Test Conf., pp. 785-794, 1985.

35. I. Pomeranz, S. M. Reddy, “On improving Genetic Optimization based Test Generation”, Proc. of European Design and Test Conference 1997, pp. 506-511

36. Schulz, M. et al., “SOCRATES: A highly efficient automatic test pattern generation system.” IEEE Trans. on CAD., (Jan.1988), pp.126-137.

37. Brik M, Ubar R “An Improved Test Generation Approach for Sequential Circuits using Decision Diagrams”

38. Cheng, K.-T., Jou J.-Y., “Functional test generation for finite state machines”.*IEEE International Test Conference*, (1990), pp.162-168.

39. Brik M., Ubar R., “Hierarchical test generation for finite state machines”. *Proc. of the 4th Baltic Electronics Conference*.Tallinn

40. R.Ubar. Combining Functional and Structural Approaches in Test Generation for Digital Systems. Microelectronics and Reliability. Elesevier Science Ltd. No.1, pp.1-13, 1998.

41. R.Ubar, Test Synthesis with Alternative Graphs. IEEE Design&Test of Computers, Spring 1996, pp. 48-57.

42. J.Raik, R.Ubar. Feasibility of Structurally Synthesized BDD Models for Test Generation. Compendium of Papers of the European Test Workshop, pp. 145-146, Barcelona, May 27-29, 1998.

43. J.Raik, R.Ubar. Test Generation with Structurally Synthesized BDD Models. Proc. of the 5th Int. Conf. on Electronic Devices and Systems, pp. 66-69, Brno, Czech Republic, June 11-12, 1998.

44. R Guo, I. Pomeranz and al., “A Fault Simulation Based Test pattern Generator for Sequential Circuits”. 17th IEEE VLSI Symposium, April 25-29, 1999, California.

45. D.Krishnaswamy, M. S. Hsiao and al. “Parallel Genetic Algorithms for Simulation-Based Sequential Circuit test Generation”. IEEE VLSI Design Conference, 1997. pp. 475-481

46. S.Minato. Binary Decision Diagrams and Applications for VLSI CAD. Kluwer Acad. Publishers, 1996, 141 p.

47. R.Drechsler, B.Becker. BDDs. Theory and Implementation. Kluwer Academic Publishers, 1998, 200 p.

48. R.Ubar. Vektorielle Alternative Graphen für digitale Systeme. Nachrichtentechnik/ Elektronik, (31) 1981, H.1, pp.25-29.

49. R.Ubar. Test Synthesis with Alternative Graphs. IEEE Design and Test of Computers. Spring, 1996, pp.48-59.

50. R.Ubar,A.Morawiec, J.Raik. Cycle-based Simulation with Decision Diagrams. DATE, Munich, 1999, pp.454-458.

51. R.Ubar. Multi-Valued Simulation of Digital Circuits with Structurally Synthesized Binary Decision Diagrams. OPA (Overseas Publishers Assotiation) N.V. Gordon and Breach Publishers, Multiple Valued Logic, Vol.4 pp. 141-157, 1998.

52. B. Cohen, VHDL Coding Styles and Methodologies. Kluwer Academic Publishers, 1999.

53. P. Ellervee, "xTractor: An Academic High-Level Synthesis Tool for Control and Memory Intensive Applications." The 20th NORCHIP Conference, Copenhagen, Denmark, pp. 253-258, Nov. 2002.

54. M.Aarna, E.Ivask, A.Jutman, E.Orasson, J.Raik, R.Ubar, V.Vislogubov, H.D.Wuttke. Turbo Tester – Diagnostic Package for Research and Training. J. of Radioelectronics and Informatics, No3 (24), July – September, 2003, pp. 69-73.

55. P. Ellervee, High-Level Synthesis of Control and Memory Intensive Applications. Ph.D. Thesis ISRN KTH/ESD/AVH--2000/1--SE, Stockholm, 2000.

56. P. Eles, K. Kuchcinski, Z. Peng, System Synthesis with VHDL, Kluwer Academic Publishers, 1998.

57. Compiler construction toolset PCCTS -- http://www.polhode.com/pccts.html

58. A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques and Tools. Addison-Wesley Publishing Company.

59. J. Raik Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. PhD thesis, 2001

60. G.Elst, K-H.Diener, E.Ivask, J.Raik, R.Ubar. FPGA Design Flow with Automated Test Generation. Proc. of German 11th Workshop on Test Technology and Reliability of Circuits and Systems. Potsdam, 1999, pp. 120-123

61. K.-H.Diener, G.Elst, E.Ivask, G.Jervan, Z.Peng, J.Raik, R.Ubar. Digital Design Flow with Test Activities. VILAB User Forum, Smolenice, April 8, 2000, 11 p.

62. MOSCITO. http://www.eas.iis.fhg.de/solutions/moscito

63. J.Raik, R.Ubar: Fast Test Pattern Generation for Sequential Circuits Using DD Representations. J. of Electronic Testing: Theory and Applications. Kluwer Acad. Publishers. Vol. 16, No. 3, pp. 213-226, 2000.

64. E. Gramatova, T. Cibakova, P. Miklos: Defect Oriented TPG for combined IDDQ - Voltage Testing of Combinational Circuits. Proc. of ETW'2000.

65. E. Gramatova, J. Gaspar, T. Cibakova: Fault Simulation for Combined IDDQ - Voltage Tesing of Combinatorial Circuits, Proc. of DDECS'00, pp. 52-58.

66. T. Cibáková, E. Gramatová, W. Kuzmicz, W. Pleskacz, J. Raik, R. Ubar: Defect-Oriented Library Builder and Hierarchical Test Generation. Proceedings of DDECS'2001, Gyor, Hungary, pp.163-167.

67. S. Garfinkel, G. Spafford. Practical Unix & Internet Security. Second edition. O’Reilly, 1996