Abstract

This paper describes an environment for internet-based collaboration in the field of design and test of digital systems. Automatic Test Pattern Generation (ATPG) and fault simulation tools at behavioral, logical and hierarchical levels available at geographically different places running under the virtual environment using the MOSCITO system are presented. The interfaces between the integrated tools and also commercial design tools were developed. The tools can be used separately, or in multiple applications in different design and test flows. The functionality of the integrated design and test system was verified in several collaborative experiments over internet by partners locating in different geographical sites.

1. Introduction

In the field of digital design, system-on-chip (SoC) technology is becoming state-of-the-art. The design of SoCs raises a lot of EDA problems: HW/SW codesign, high-level synthesis, testability evaluation, test pattern generation, fault simulation, physical defect analysis with respect to the whole system to be integrated. Usually not all the needed EDA tools are available for a designer in his working site.

The Internet opens a new dimension, and offers new chances using tools from different sources. The basic idea of this paper aims at exploiting an Internet-based tool integration. For that purpose several design and ATPG tools implemented at geographically different places were successfully integrated into the new virtual environment MOSCITO [1,2]. The essential features due to this integration environment were experimentally proved. The results obtained are presented also.

The paper is organized as follows. The MOSCITO system is described in Section 2. The functionality of the environment via tools description is given in section 3. Experimental results obtained by cooperative use of the environment are shown in section 4.

2. MOSCITO

2.1. General Concept of MOSCITO

Starting from the idea to connect tools via the Internet to form an appropriate workflow for solving dedicated design problems the MOSCITO system at IIS/EAS was developed and implemented.

The MOSCITO system works as an Internet-based multi-agent system which can be controlled and observed by the user’s front end program – the MOSCITO desktop. Over the last years several approaches for coupling tools and providing services via Internet were developed [19]-[24]. Powerful middleware such as CORBA, Java RMI, COM/DCOM or HLA is available today for implementing such distributed infrastructures. MOSCITO is mainly based on the general ideas of these approaches but it uses a very small, pure Java-based implementation. Some reasons for this design decision are:

---

1 This work was partially supported by EC Copernicus project VILAB, the Swedish National Board for Industrial and Technical Development (NUTEK) and Swedish Academy of Engineering Sciences, and the Estonian Science Foundation (Grants No 3658 and 4003), and by the Slovak VEGA project grant 26091/20.
CORBA is very complex. It supports components which can be implemented in different programming languages. In MOSCITO this feature is not necessary because each tool is encapsulated in a Java wrapper class – the MOSCITO Agent.

COM/DCOM is available on Microsoft platforms only. MOSCITO is needed at least on Solaris, Linux and Windows systems.

Main focus of HLA is coupling of simulators. MOSCITO isn’t restricted to this application area.

In some systems only a white-box integration of tools is possible. MOSCITO Agent interface is targeted to support black-box integration. So also commercial tools can be encapsulated with low effort.

Systems like [22] are based on very fine-grained, powerful workflow concepts. MOSCITO uses very simple, dataflow-based workflows which are controlled by the MOSCITO Agents themselves.

The main emphasis in the MOSCITO tool integration was put on the following aspects:

- Encapsulation of design tools and adaptation of the tool-specific control and data input/output to the MOSCITO framework
- 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.

Moreover, an important goal is to provide the functionality of a tool to a potential user as a service in a local area network (LAN). This approach is similar to the Application Service Provider (ASP) idea or the recent approach of Web Services.

In the present system the following tools have been integrated in MOSCITO:

- Interface to system-level HW/SW co-design environment (C2VHDL code migration tool) [3]
- Behavioral level Automated Test Pattern Generation (ATPG) [4]
- High-level synthesis system CAMAD [5]
- Interface from RTL VHDL to hierarchical ATPG
- Interfaces from low-level EDIF and ISCAS formats to ATPGs and fault simulators
- Hierarchical ATPG DECIDER [6,7]
- Logic level fault simulation and test generation tools Turbo-Tester [8,9]
- Tst2Alb - a data converter between ATPG tools
- DefGen - ATPG for I\textsubscript{DDQ} and voltage testing of digital circuits [10,11].
- ALB - an automatic fault library builder [12].

The listed tools can act as MOSCITO agents and each of them provides a demanded service. The user are empowered 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. Due to that fact the user’s effort for installation, configuration and maintenance of software will be drastically reduced. 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). So the entire workflow will speed
up. To facilitate 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 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).

2.2. Software architecture

MOSCITO consists of three software layers: kernel layer, interface layer, and user extensions.

The kernel provides functionality for basic object and data management, file handling, XML processing, and communication. Due to the fact that 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 helps the user to integrate a new tool or workflow in less than one or two days.

2.3. Tool encapsulation

For the integration of tools with MOSCITO a sophisticated agent interface was introduced. A tool is embedded into a MOSCITO agent for:

- adapting the input data to the embedded tool,
- converting the tool-specific data (simulation results, logfiles, test vectors),
- mapping the control information to the embedded tool and the transfer and conversion of status information (warning and error messages) to be submitted to the user.

For embedding programs into a MOSCITO agent there are three ways:

- Integration of the entire program: the software has to be run capable as a batch job (e.g. ATPG). In this way the integration of commercial tools is possible.
- Embedding of a library via the Java Native Interface (JNI): e.g. C, C++ or FORTRAN routines can be embedded.
- Direct integration of Java-classes and applications, respectively, in particular for software in JAVA.

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

2.4. 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. To minimize the implementation effort for parsers, translators and converters, the format for all data transmitted in MOSCITO was set to a special XML format, the MOSCITO Markup Language (MoscitoML).

2.5. Graphical User Interface (GUI)

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.

2.6. 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 initiate 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.
2.7. Tool environment

To validate the MOSCITO system and to collect experiences while using it for real-life applications an experimental tool environment for design and test pattern generation (Fig. 1) was developed and mapped to a MOSCITO workflow. In the following the functionality of the tools will be explained in detail.

Design information can be generated in different ways, by VHDL files to be processed by commercial or experimental high-level or logic synthesis systems, or provided manually by schematic editors. The gate-level design is presented in the EDIF format. In university research practice, ISCAS benchmark families with a dedicated ISCAS format are widely used. For linking test generation and fault simulation tools with all the needed formats, different translators and interfaces were developed (Blocks 5, 6, 10, 11 in Fig. 1). These interfaces make possible to design a circuit in one geographical site, generate test patterns in another site, and to analyze the quality of patterns in a third site. In such a way, joint experiments were carried out in the field of defect-oriented test where high-level synthesis was performed in Germany and Sweden, defect level analysis for complex gates was performed in Poland, and defect oriented fault simulation and test generation were carried out in Slovakia and Estonia.

3. Tool descriptions

3.1. Interface to system-level HW/SW co-design

The initial entry to the testability driven design flow is represented by a system-level HW/SW co-design backend tool (Block 1 in Fig. 1).

3.2. Behavioral ATPG

There has been a lot of research devoted to solve the test generation problem for gate-level circuits. By working at this level of abstraction, an ATPG can generate high quality tests but is computationally expensive in the case of large circuits. We have developed a technique to generate tests at the behavioral level, and therefore the ATPG algorithm works directly on the behavioral specification of the digital circuit to be designed (Block 2 in Fig. 1).

Beside the reduction of complexity, such a behavioral ATPG technique gives the designer an opportunity to perform design for testability already in the early design stages, since testability evaluation can
be based on the ATPG results obtained directly from the behavioral specification.

In the proposed approach, decision diagrams (DDs) [5,6] are used for design modeling for the behavioral specification, which is originally given in VHDL. One important advantage of modeling the behavior with DDs is that DDs can be used to capture a digital design at different abstraction levels, and therefore a hierarchical test generation approach can also be developed. For every internal variable and primary output of the design a data-flow DD will be generated. Terminal nodes of the DD represent arithmetic expressions. Further, an additional DD, which describes the control-flow, will be generated. The control-flow DD describes the succession of statements and branch activation conditions.

There are two types of tests which we consider in the current approach. One set targets nonterminal nodes of the control-flow DD (conditions for branch activation) and the second set aims at testing operators, depicted in terminal nodes of the data-flow DD.

The test generation task is formulated as the following problem: tests will be generated sequentially for each nonterminal node of the control-flow DD. Symbolic path activation is performed and functional constraints are extracted. Solving the constraints gives us the path activation conditions to reach a particular segment of the specification. In this way, the test generation problem is formulated as a constraint-satisfaction problem. This problem is solved by a constraint solver based on SICStus Prolog [4].

In order to test the operations, presented in the terminal nodes of the data-flow DD, different approaches can be used. In our work, we use mutation testing [13] for test generation for the operations at the terminal nodes. For the nonterminal nodes of the control-flow DD, conformity tests will be applied. The conformity tests target errors in branch activation.

3.3. High-level synthesis system CAMAD

The CAMAD high-level synthesis system (Block 3 in Fig.1) is built around an internal design representation, called ETPN (Extended Timed Petri Net), which has been developed to capture the intermediate results during the high-level synthesis process. The use of Petri nets provides a natural description of concurrency and synchronization of the operations and processes of a hardware system. It gives thus a natural platform to represent and manipulate concurrent processes of VHDL specifications.

ETPN is used as a unified design representation which allows the synthesis algorithm to employ an iterative improvement approach to carry out the synthesis task. The basic idea is that once the VHDL specification is translated into the initial design representation, it can be viewed as a primitive implementation. Correctness-preserving transformations can then be used to successively transform the initial design into an efficient implementation. CAMAD integrates the operation scheduling, data path allocation, control allocation and, to some degree, module binding sub-tasks of high-level synthesis. This is achieved by developing a set of basic transformations of the design representation which deals simultaneously with partial scheduling and local data path/control allocation. An optimization algorithm is then used to analyze the (global) design and select transformations during each step of the iterative improvement process.

Fig. 4 illustrates the basic structure of the CAMAD system. The first step of CAMAD is to map the VHDL specification into ETPN and to perform automatic parallelism extraction. After the transformation steps a RTL hardware implementation is generated which consists of a data path netlist and a controller specified in the form of a finite state machine. The final RTL implementation is converted into structural VHDL which, as well as the input system specification, can be simulated for verification.

Fig. 4. Overview of CAMAD

3.4. Logic-level ATPG tools

The Turbo Tester ATPG software (Block 8 in Fig.1) consists of a set of tools (Fig.5) for solving different test related tasks by different methods and algorithms:

- test pattern generation by deterministic, random and genetic algorithms
- test optimization (test compaction)
- fault simulation and fault grading for combinational and sequential circuits
- defect-oriented fault simulation and test generation
- multi-valued simulation for detecting hazards and analyzing dynamic behaviour of circuits
- testability analysis and fault diagnosis.

Fig.5. A set of low-level ATPG tools Turbo-Tester

All the Turbo Tester tools operate on the model of Structurally Synthesized Binary Decision Diagrams (SSBDD) [8,9]. The tools run on the structural logic level. Two possibilities are available - gate-level and macro-level. In the second case, the gate network is transformed into macro network where each macro represents a tree-like sub-network. Using the macro-level helps to reduce the complexity of the model and to improve the performance of tools. The fault model used is the traditional stuck-at model. However, the fault simulator and test generator can be run also in the defect-oriented mode, where defects in the library components can be taken into account. In this case, additional input information about defects in the form of defect tables for the library components is needed.

3.5. Hierarchical ATPG DECIDER

In addition to the gate-level tools, a hierarchical test generation system DECIDER (Block 7 in Fig.1) has been developed and linked to MOSCITO. DECIDER includes a Register-Transfer Level (RTL) VHDL interface for importing high-level design information, and also an EDIF interface for importing gate-level descriptions of logic.

The ATPG uses a top-down approach, with a novel method of combining random and deterministic techniques. Tests are generated for each functional unit (FU) of the system separately. First, a high-level symbolic test frame (test plan) is created for testing the given FU by deterministic search. As the result, a symbolic path for propagating faults through the network of components is activated and corresponding constraints are extracted. The frame will adopt the role of a filter between the random TPG and the FU under test. If the filter does not allow to find a random test with 100% fault coverage for the component under test, another test frame will be chosen or generated in addition to the previously created ones. In such a way, the following main parts in the ATPG are used alternatively: deterministic high-level test frame generator, random low-level test generator, high-level simulator for transporting random patterns to the component under test and low-level fault simulator for estimating the quality of random patterns.

These test patterns are the input stimuli for the RTL design. Since the test generation implements also high-level fault models, we do not know the precise gate-level stuck-at fault coverage of these tests. Therefore, the test patterns have to be converted in order to correspond to the stimuli for the gate-level netlist of the entire design. This is required for gate-level fault simulation in order to measure the quality of tests.

3.6. Defect-oriented ATPG

The DefGen ATPG system (Block 12 in Fig.1) is a hierarchical ATPG for combinational circuits for IDDQ and/or voltage testing. The random, deterministic TPG algorithms and a fault simulator are involved in the ATPG. The TPG process uses the functional fault model and runs over the functional test set specified for each functional cell of a CUT structure. The deterministic TPG techniques are based on justification and propagation of the predefined test patterns for each cell in a circuit. The functional test set for each cell is named a list of faults conditions and it is a part of the fault conditions library for DefGen. These lists can be created e.g. from a defect analysis of circuits cell at the low level or can be specified by the designer with regards to the used fault model for the investigated cells. The input format for circuit description is the language from ISCAS’85 benchmark circuits. The EDIF-ISCAS translator from Turbo tester (Block 8 in Fig.1) can be used as the interface to DefGen.

An Automatic Fault Library Builder (ALB) has been developed and implemented for finding an optimal functional patterns for cells in the CUT structure [14]. The test patterns are generated from different defect/fault tables for selected cell, and they form the fault conditions library of the DefGen ATPG system. Some defect tables have been created for several combination standard gates (e.g. from the 0.8 µm CMOS library) and lists of optimal patterns have been generated by ALB for the ATPG experiments.

Test Pattern Generation (TPG) technique at higher levels of abstraction rests upon a functional fault model and physical defect - functional fault relationships in the form of a defect coverage table at the lower level. Each table (one for a given cell) includes the following information:
- list of all possible faults (shorts, bridges between nonequipotential conducting paths, resulting in a short circuit caused by physical defects);
- erroneous functions performed by the faulty gate;
- list of input patterns detecting physical defects.
- (optional) probabilities of occurrence of the physical defects.

The lists of erroneous functions and test patterns were generated by electrical simulations at the transistor level by WUT [15]. Probabilities of defects occurrence were calculated by layout probabilistic analysis at the physical level taking into account defect density and size distribution.

4. Experiments

The described environment has been tested in the frame of European project VILAB by the partners IIS/EAS and TUD (Germany), IIN (Slovakia), LIU (Sweden), TTU (Estonia) with input from WUT (Poland) for several designs according to the following general algorithm (the reference to exploited tools in Fig. 1 is given in parentheses):

1. The high-level designs were produced in the HW/SW codesign environment (1) or by high-level synthesis system (3).
2. The high-level testability was evaluated at the early design stage by behavioral level ATPG (2).
3. The user evaluates the quality of his own functional test for the new design (4) by using Turbo Tester fault simulator (6,8). Alternatively, he can also make use of other university fault simulators (13).
4. If the results are acceptable (test has obtained the demanded quality) go to END, else go to Step 5.
5. The user can work with the implemented ATPGs 7, 8 or 12. If the stuck-at-fault model is accepted, the user can work with the ATPGs 7, 8 or 12. If the circuit is a simple sequential or combinational one (e.g. only FSM without data-path) go to Step 7. If the circuit consists of the control- and data-paths the user can work with the hierarchical ATPG (5,6,7). In this case, both the RTL description from high-level synthesis system (3) and the gate-level description from logic synthesis system (4) are needed. If the results are acceptable (test has the needed quality), go to END, else go to Step 7.
6. If the stuck-at-fault model is accepted, the user can work with the ATPGs 7, 8 or 12. If the circuit is a simple sequential or combinational one (e.g. only FSM without data-path) go to Step 7. If the circuit consists of the control- and data-paths the user can work with the hierarchical ATPG (5,6,7). In this case, both the RTL description from high-level synthesis system (3) and the gate-level description from logic synthesis system (4) are needed. If the results are acceptable (test has the needed quality), go to END, else go to Step 7.
7. The user can work with the gate-level ATPG (6,8). If the results are acceptable (test has the needed quality), then go to END, else go to Step 8.
8. The testability should be now improved by redesign. Some flip-flops can be included, for example, into the scan-path. For testing the new full or partial scan-path design the user can work again with the gate-level ATPG (4,6,8).
9. Depending on the results, the step 8 can be repeated till the demanded test quality has been obtained.

10. END.

This environment has been utilized for research purposes. For example, the performance of the hierarchical ATPG (7) was compared against the existing university tools GATEST [16] and HITEC [17]. For that the translator 10 was necessary. The results of comparison of different ATPGs are given in Table 1.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>DECIDER</th>
<th>GATEST</th>
<th>HITEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCD</td>
<td>91.0</td>
<td>3.4</td>
<td>92.2</td>
</tr>
<tr>
<td>Mult 8x8</td>
<td>79.4</td>
<td>13.6</td>
<td>77.3</td>
</tr>
<tr>
<td>Dileg</td>
<td>95.8</td>
<td>15.8</td>
<td>96.0</td>
</tr>
</tbody>
</table>

Table 1. Experimental results for hierarchical ATPG

Some experiments have been performed with the ATPG tools running at the Tallinn Technical University and the Institute of Informatics of the Slovak Academy of Sciences using defect tables created at the Warsaw Technical University separately.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Defect coverage for OR-type shorts, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counted defects</td>
<td>Probabilistic defects</td>
</tr>
<tr>
<td>C17</td>
<td>92.59</td>
</tr>
<tr>
<td>C432</td>
<td>99.38</td>
</tr>
<tr>
<td>C499</td>
<td>92.80</td>
</tr>
<tr>
<td>C880</td>
<td>95.95</td>
</tr>
<tr>
<td>C1355</td>
<td>93.42</td>
</tr>
<tr>
<td>C1908</td>
<td>92.91</td>
</tr>
<tr>
<td>C3540</td>
<td>94.21</td>
</tr>
<tr>
<td>C5315</td>
<td>94.71</td>
</tr>
<tr>
<td>C6288</td>
<td>92.59</td>
</tr>
</tbody>
</table>

Table 2. Data of defect-oriented fault simulation

The purpose of experiments was to compare the quality of 100% stuck-at test patterns in relation to physical CMOS defects. For that we used the data produced at WUT by probabilistic analysis of physical defects for a restricted library of complex gates. Then we resynthesized the ISCAS'85 circuits using only components from the analysed library. The problem to investigate was to determine how good are the 100% stuck-at tests in detecting physical defects in complex gates. The results in Table 2 show the low quality of 100% stuck-at tests in detecting real physical defects.

The new result of these experiments was to show that the quality of tests in terms of defect coverage is higher when the defect probabilities are not taken into account. From that we can conclude that the traditional methods of test coverage measuring based on simply
counting of not detected defects, where all the faults are assumed to have the same probability, are tending to give overestimated quality measures.

5. Summary

In the paper an Internet-based environment based on MOSCITO system [18] is presented. The environment is focused on providing SW/HW codesign, high-level and logic level design flows with test pattern generation and fault simulation at behavioral, register-transfer, gate and physical defect level operational activities. The main effort was put on linking together test generators and fault simulators with varying functionalities and diverse fault models available at geographically different sites. The system provides interfaces and links to commercial design environments and also to other university tools. The functionality of the integrated design and test system was verified by several benchmark circuits and by different design and test flows. Furthermore, authors believe that the MOSCITO architecture is powerful enough to solve similar problems in other application areas of automated system design. Future work will continue in this direction.

References


