User Defined Coverage - A Tool Supported Methodology for Design Verification

Raanan Grinwald, Eran Harel, Michael Orgad, Shmuel Ur, Avi Ziv
IBM Research Lab in Haifa
MATAM
Haifa 31905, Israel
email: {grinwald, harel, orgad, sur, aziv@vnet.ibm.com}

Abstract

This paper describes a new coverage methodology developed at IBM's Haifa Research Lab. The main idea behind the methodology is a separation of the coverage model definition from the coverage analysis tool. This enables the user to define the coverage models that best fit the points of significance in the design, and still have the benefits of a coverage tool. To support this methodology, we developed a new coverage measurement tool called Comet. The tool is currently used in many domains, such as system verification and micro-architecture verification, and in many types of designs ranging from systems, to microprocessors, and ASICs.

1 Introduction

Functional verification comprises a large portion of the effort in designing a processor [1]. The investment in expert time and computer resources is huge, and so is the cost of delivering faulty products [4]. In current industrial practice, most of the verification is done by generating a massive amount of tests by random test generators [1, 2, 9]. The use of advanced random test generators can increase the quality of generated tests, but it cannot detect cases in which some areas of the design are not tested, while other parts are tested repeatedly.

The main technique for checking and showing that the testing has been thorough, is called test coverage analysis [10]. Simply stated, the idea is to create, in a systematic fashion, a large and comprehensive list of tasks and check that each task was covered in the testing phase. Coverage can help in monitoring the quality of testing, and assist in directing the test generators to create tests that cover areas that have not been tested before.

Many coverage tools exist in the market. Most coverage tools are program-based and check code coverage of execution of programs. Examples include code coverage in C [8] or VHDL [12], and covering states and transitions of state machines [13]. The main disadvantage of code coverage is that it does not "understand" the application domain. It is, therefore, very hard to tune the tools to areas which the user thinks are of significance. To overcome this problem, some domain dependent tools have been developed which measure specific functional events [3]. In all the tools we are familiar with, the coverage models are hard-coded. Therefore, the user is forced to use a tool that does not fit his needs, or put a lot of effort into developing a new tool for his coverage needs.

In this paper, we present a new methodology for coverage that was developed at IBM's Haifa Research Lab, and a coverage measurement tool called Comet (COverage MEasurement Tool) that was developed to support this methodology. The main idea behind the new methodology is separation of the coverage model definition from the tool. This separation enables the user to use a single tool for most of his coverage needs. The user can define coverage models that fit the design in the best way, while enjoying all the benefits of a coverage tool, such as data collection and processing, creation of coverage reports, and generation of regression suites with high coverage. Moreover, the user can change the scope or depth of coverage during the verification process, starting from simple coverage models in the early stages of verification and, based on those models, define deeper and more complex models later on.

Our coverage methodology and Comet are used by several sites in IBM in designs ranging from systems, to microprocessors, and ASICs. The tool is currently used in many different domains, such as architectural verification of microprocessors, micro-architecture verification of units in a processor, system and unit verification of a communication ASIC, and verification of coherency and bus protocols.

The rest of the paper is organized as follows: In Section 2, we describe the functional coverage process and provide some terminology used in this paper. In Section 3, we describe our new coverage methodology in detail. In Section 4, we describe Comet, our coverage tool, and the coverage process as is done in Comet. Finally, Section 5 concludes the paper.

2 Functional Coverage Process

Coverage, in general, can be divided into two types: program-based and functional. Program-based coverage concentrates on measuring syntactic properties in the execution, for example, that each statement was executed, or each transition in a state machine taken. This makes syntactic coverage a generic method which is usually easy to measure. Functional coverage focuses on the functionality of the program,

35th Design Automation Conference ®
Copyright ©1998 ACM
1-58113-049-x-98/0006/$3.50 DAC98 - 06/98 San Francisco, CA USA

and it is used to check that every aspect of the functionality is tested. Therefore, functional coverage is design and implementation specific, and is much harder to measure. Currently, functional coverage is done mostly manually.

The manual process through which one gains confidence that testing has been thorough, is composed of the following steps: first, one prepares a list of testing requirements in a test plan which contains the events that have to happen during testing. Then, one executes the tests and checks that for every requirement there is a test in which the particular event happened and that this test has been executed successfully. In general, this is a very labor intensive effort, as the tests are crafted in order to fulfill specific requirements. If a test fulfilled a requirement "by chance", it will usually not be noticed.

Automation can be added to this process in the following way: a *coverage task,* a binary function on a test or a trace which specifies whether an event occurs or not, is created for each test requirement. Examples of coverage tasks are: "statement 534 was executed", and "the instructions in the first two stages of a pipe were add and divide writing to the same target". A cohesive group of coverage tasks is called a *coverage model.*

The second example of a coverage task can be generalized into coverage models over attributes. Each task is an instantiation of these attributes and the coverage model is the cross product of the possible values. For example, (first instruction, second instruction) is a coverage model over two attributes, and (ADD, DIV) is a task in that model. Comet uses coverage models of this type.

In order to find out if a coverage task happens in a test, we create a trace called *event trace.* Rows in the event trace usually contain the values of the attributes at the time the row was produced. Coverage is measured by activating the coverage models on the event trace.

We divide coverage models into two types: *snapshot* and *temporal.* Snapshot models are coverage models whose attributes are taken from a single row in the event trace, for example, the combination of two instructions that are at different stages of a pipe at the same time. Temporal coverage models are coverage models whose attributes are taken from different rows in the event trace. A coverage task in these models is a specific scenario that occur during the test, for example, the combination of two instructions that were fetched by the processor with a distance of less than 5 cycles between them.

Often, some of the tasks in the coverage model, defined by the cross product of the attributes, are illegal tasks, that is, coverage tasks that should not occur. For example, two instructions that write to the same resource should never be in the write-back stages of pipes in the same cycle. Comet reports to the user on any illegal tasks found in the event trace. It also reports coverage statistics on which tasks out of the task coverage list (TCL), a list of all the (legal) tasks in the coverage model, have been covered.

3 User Defined Coverage

Coverage measurement and the use of coverage as an indicator of the quality of testing and reliability of the design are growing rapidly. More tools for coverage measurement are becoming available, both for software and hardware testing. These tools provide the user with many features that are necessary for an efficient coverage measurement, such as data gathering, updating of the coverage task list, and reports on the coverage status.

A common property of most of these tools is that the coverage models which the tool is designed to handle, are hard-coded into the tool. Therefore, in order to make the tools applicable to many users, the models that are implemented in these tools are generic. This leaves a user that wants to cover models that are specific to his design with two options: to measure coverage manually or to build his own coverage tool for these models. Both options require a large effort and are error prone. Therefore, specific models are rarely used.

We propose a different methodology for coverage. This methodology calls for separation of the coverage model from the coverage measurement tool. The idea behind the methodology is that most of the functionality provided by existing coverage tools, such as data gathering and coverage reports, is similar in all tools. The main difference between tools is the models they implement. Therefore, a single general purpose tool, that will be oblivious to the coverage model and provide all the functionality of existing tools, can be used to provide all the coverage needs of a user, both for generic and specific models.

To provide all the needed functionality of a coverage tool to a specific model, the tool has to be aware of the exact specification of the model. Therefore, one of the inputs to the tool is the definition of the model. The definition of a model has to be done in a language which is simple enough for a user to use, yet rich enough to describe all the models that the user wants to cover. Our experience shows that a language that contains the predicates used in first order temporal logic (*and, not, exist, for all, before, after,* etc.) combined with the use of simple arithmetic (natural numbers, arithmetic and relational operators), is sufficient.

The advantages of such a general purpose coverage tool are enormous. First, it allows its users to define their own models, according to their specific needs, and still enjoy all the functionality of a dedicated coverage tool without the need to develop such a tool. It, therefore, provides users with the means to measure coverage on models which were not available to them before. The tool also allows organizations to use a single tool for most of their coverage needs, and not be forced to buy and maintain a set of tools, one for each coverage model. Another advantage for explicit and external model definition is that it enables sharing of coverage models between projects. Finally, the tool enables its users to adapt their coverage models to their testing resources, and refine these models during testing. For example, users that can afford only quick and dirty testing can define coarse grain models with a small amount of tasks, while users that want to do comprehensive testing can define finer grain models [11].

4 Comet (COverage MEasurement Tool)

Based on the methodology described above, we developed a coverage measurement tool called Comet. Comet enables users to define their own coverage models, gather and process traces to measure the coverage of these models, and generate coverage reports on the results of the measurement. Comet depends on a relational database in order to supply a comprehensive and stable environment needed for the coverage measurement process itself and the analysis of the coverage results.

The relational database provides to Comet all the data management it needs. It stores the traces on which coverage is measured and the TCLs, and provides access and processing of the data using standard SQL [5]. It also enables the

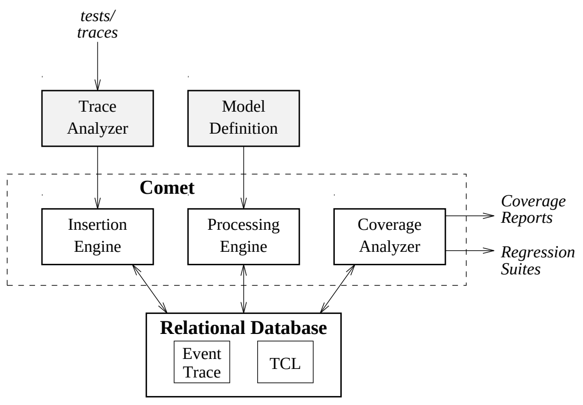


Figure 1: Comet Structure

user to produce non-standard reports, that are not provided by Comet, by directly querying the database itself. Another advantage of the relational database is that it enables us to implement our requirements for model specification using standard SQL [5], without the need to invent a new specification language and write an interpreter for it.

Comet consists of three major parts, as shown in Figure 1: the *Insertion Engine,* whose task is to insert events from input traces into the database, the *Processing Engine,* whose task is to process the traces in the database in order to detect coverage tasks according to the model definition, and the *Coverage Analyzer,* which analyzes the measurement results and prepares coverage analysis reports according to the user definition.

Since Comet is oblivious to the coverage model and it can receive many types of traces, it requires two user-provided additions that do not exist in other tools. The first addition is definition of the coverage model which is done using Comet's GUI. The second addition is a trace analyzer that converts the format of the traces to a standard format that Comet can handle.

Below, we explain in more detail the coverage measurement process, as it is done with Comet. We use two examples of models to illustrate the main features of the tool, one that was used to build a set of architectural tests for PowerPC processors, called *IIwI* (Interleaved Instructions with Interrupts), and the other which was used during the testing of a branch prediction unit in a processor, called *branch unit pipe.* The two examples also help to illustrate the versatility of Comet as a coverage measurement tool.

4.1 Model Definition

The first and most important step in the coverage process is deciding what to cover or, more precisely, on what coverage models to measure coverage. In order to make coverage successful and use it to influence the verification process, it is important to choose the correct types of coverage models. First, it is important to choose coverage models for areas which the user thinks are of significance. Next, the size of the model should be chosen according to the testing resources. The model size (number of tasks) should not be too large, such that it is impossible to cover the model, given test time and resource constraints. From our experience, we found that the best way to create effective models, is to start with small models and later on to refine or combine them to create bigger and more complex models.

The definition of a coverage model in Comet is done in three steps: defining the attributes of the model, specifying how to extract the attributes from the event trace, and separating between the legal and illegal tasks for the model. These steps are explained in detail below.

4.1.1 IIwI model

The IIwI coverage model was designed as a part of an Architecture Verification Suite (AVS) for PowerPC processors. Since incomplete or wrong cleanup after interrupts is a source of many bugs in processors, we decided that it is important to test all possible pairs of instructions with all types of interrupts between them. The IIwI model is an example of a temporal model. The model is defined by a scenario of three events: the first instruction, the second instruction, and an interrupt between the two instructions, with additional constraints on the timing of the events.

The first step in the definition of the IIwI model is selection of the attributes of the model. The natural choice for attributes for the model are the instruction name for the first and second instructions, and the interrupt type for the interrupt. The problem with that choice is the size of the model. There are more than 400 PowerPC instructions and 30 types of interrupts, and therefore more than 5 million tasks (400x400x30). We, therefore, decided to group the instructions into 34 groups, each group containing instructions of similar type. This is done using the grouping option of Comet. The attributes of the IIwI model can be seen in the top part of Figure 2. The attributes are `First` and `Second`, which are obtained from the `InstName` attribute in the event trace using the `IIwIGroup` grouping function, and `Interrupt`, which is the `InterruptBasic` attribute, taken directly from the event trace.

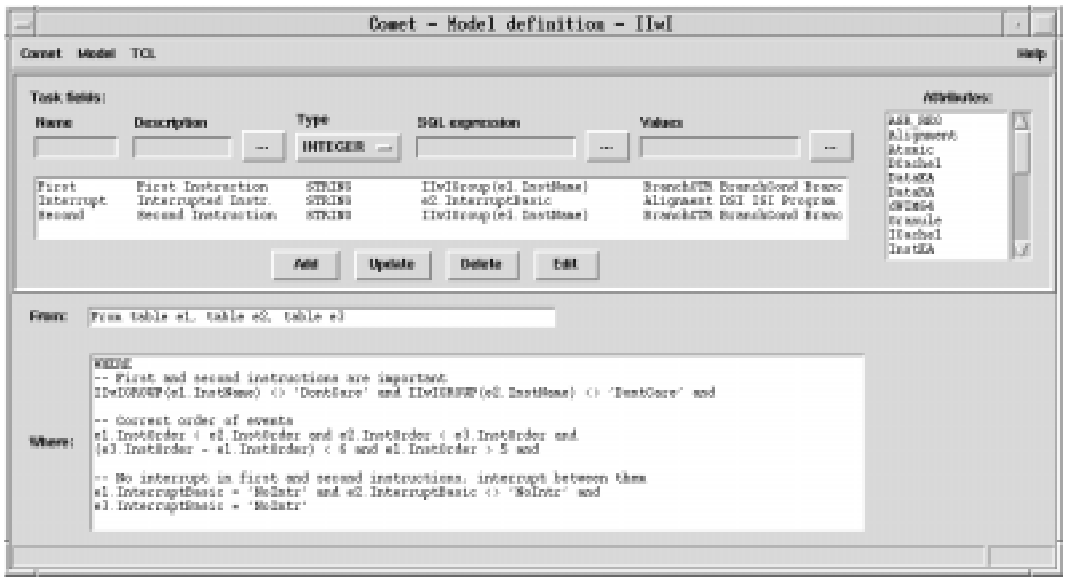
The second part of model definition is to provide Comet with a method to extract tasks from the event trace. This is done by specifying a `WHERE` clause of an SQL query, that defines the conditions on the events that create the tasks. Since the IIwI is a temporal model, with tasks consisting of events occurring at three different times, the SQL query involves three views of the event trace, denoted by `e1` for the first instruction, `e3` for the second one and `e2` for the interrupt. The conditions for the IIwI model appear at the bottom of Figure 2. The conditions are divided into 3 sub-conditions. The first states the instructions that are of interest to us (they do not belong to the `DontCare` group). The second sub-condition deals with the timing of the instructions (the first instruction occurs before the second one with a maximal distance of 6 instructions, and the interrupt is between them). The last sub-condition states that the first and second instructions are not interrupted, while the middle event is an interrupt.

After defining the attributes of the model and its conditions, we need to create the TCL containing all the legal tasks of the model. This step is explained below, using the branch unit pipe model.

4.1.2 Branch unit pipe model

The branch unit pipe model is part of a set of coverage models we designed to measure coverage on the branch unit of a processor. This model was designed to check the amount and locations of branch instructions in a nine stage pipe inside the branch unit. The specification of the unit is for a maximum of seven simultaneous branches in the pipe.

The branch unit pipe model is a snapshot model, and its attributes are taken directly from the event trace. The first



```
First       First Instruction    STRING    IIwIGroup(e1.InstName)    BranchGTN BranchCond Branc
Interrupt   Interrupted Instr.   STRING    e2.InterruptBasic         Alignment DSI ISI Program
Second      Second Instruction   STRING    IIwIGroup(e1.InstName)    BranchGTN BranchCond Branc
```

```
ASR_REG
Alignment
Atomic
DCachel
DataEA
DataRA
dWIMG4
Granule
ICachel
InstEA
```

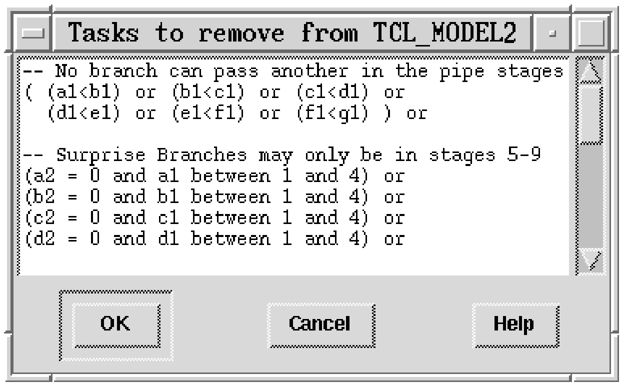
```
From table e1, table e2, table e3
```

```
WHERE
-- First and second instructions are important
IIwIGROUP(e1.InstName) <> 'DontCare' and IIwIGROUP(e2.InstName) <> 'DontCare' and

-- Correct order of events
e1.InstOrder < e2.InstOrder and e2.InstOrder < e3.InstOrder and
(e3.InstOrder - e1.InstOrder) < 6 and e1.InstOrder > 5 and

-- No interrupt in first and second instructions, interrupt between them
e1.InterruptBasic = 'NoIntr' and e2.InterruptBasic <> 'NoIntr' and
e3.InterruptBasic = 'NoIntr'
```

Figure 2: IIwI model definition

Attribute	Description	Values
a1	Stage of 1st branch	0–9
a2	Prediction of 1st branch	0–1
b1	Stage of 2nd branch	0–8
b2	Prediction of 2nd branch	0–1
c1	Stage of 3rd branch	0–7
c2	Prediction of 3rd branch	0–1
d1	Stage of 4th branch	0–5
d2	Prediction of 4th branch	0–1
e1	Stage of 5th branch	0–4
f1	Stage of 6th branch	0–4
g1	Stage of 7th branch	0–2
field13	Reset type	0–2

Figure 3: Branch unit pipe model attribute definition



```
-- No branch can pass another in the pipe stages
( (a1<b1) or (b1<c1) or (c1<d1) or
  (d1<e1) or (e1<f1) or (f1<g1) ) or

-- Surprise Branches may only be in stages 5-9
(a2 = 0 and a1 between 1 and 4) or
(b2 = 0 and b1 between 1 and 4) or
(c2 = 0 and c1 between 1 and 4) or
(d2 = 0 and d1 between 1 and 4) or
```

Figure 4: Restrictions for branch unit pipe model

two steps in the model definition are similar to the steps for the IIwI model, and thus are omitted. Figure 3 gives the list of attributes for the model. The attributes describe, for each possible branch in the pipe, the stage it is in and whether it is a predicted or a surprise branch. Another attribute for the model is a reset signal, that can have three values.

Specification of the unit and its environment puts many restrictions on combinations of possible locations and types of branches in the pipe. The restrictions affect which tasks in the model are legal and which are not. Separation between the legal and illegal tasks is done while building the TCL of the model, which is a list of all legal tasks in the model. Figure 4 shows the window for restriction specification with some restrictions for the branch unit pipe model.

4.2 Data Collection

After defining the coverage models, the next step is to collect event data from tests and measure the coverage. The events can come directly from the test programs or from trace files that are created by tracing the behavior of the design during the test. For example, the input for the IIwI model are the instructions and interrupts in the test, which are provided directly by the test generator, while the input for the branch unit pipe model are the stages of instructions in the branch pipe, which are provided by monitoring the pipe during execution of a test.

The decision on which attributes to monitor is taken in parallel with the model definition or even before it, and is affected by the requirements of the model and the ability to monitor the attributes. The exact format of the trace also depends on the environment and mechanisms that are used to generate the test or the trace.

To measure coverage, the traces are inserted into a table called `EventTrace` in the database. Since Comet needs to handle many types of traces, the insertion of data is done in two steps. The first step is to convert the data to a single format using the Trace Analyzer, and the second step is to insert the data into the database using the Insertion Engine (See Figure 1). The Trace Analyzer is specific to each type of trace and should be provided by the user, usually in the form of a simple script. The Insertion Engine, that writes

InstName	InstOrder	InstEA	InstRA	InterruptBasic	DataEA	DataRA	...
mtmsr	0	8085B660	03C1664	-	0	0	...
isync	1	8085B664	03C1660	-	0	0	...
ld	2	8085B668	03C166C	-	21466048	0263D48	...
fnmsub.	3	8085B66C	03C1668	FPUI	0	0	...

Figure 5: Event Trace example

the trace into the EventTrace table in the database, is a part of Comet.

Figure 5 provides an example of a list of attributes that are stored in the EventTrace table for the IIwI model. Some attributes are used directly by the model, such as InterruptBasic, while other attributes are used indirectly, such as InstName that is translated into the instruction group. Other attributes are not used at all by the IIwI model, and are stored for the sake of other models.

After the trace data is stored in the database, it can be processed by Comet's Processing Engine. This engine executes the query of the model on the trace to detect the list of tasks that occurred in the test. For each of these tasks, if it appears in the TCL, Comet increases its coverage count. If the task does not appear in the TCL, Comet reports it as an illegal task, as explained in the next section. Besides updating the coverage counters for tasks, Comet also maintains other data for each trace and each task, such as the first and last time each task occurred, for reporting and analysis purposes.

4.3 Coverage Analysis

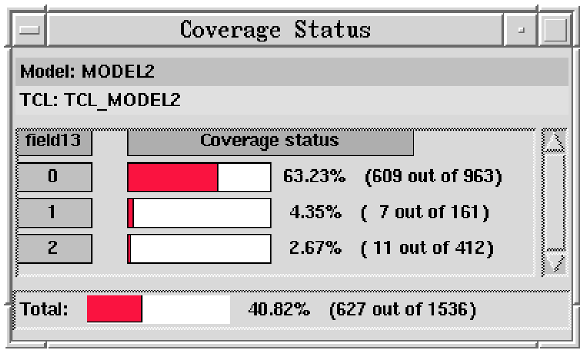
The output of coverage measurement can be used in several ways to improve the verification process. First, the coverage tool can detect illegal events that occur and help find bugs in the design. Coverage measurement can also provide the user with information on the status of the verification process. It can help to find holes in the testing, i.e. areas that are not covered. It can also help detect when the testing process runs out of steam, and no more new tasks are being covered. Coverage measurement can also assist the verification process more directly, by directing the test generation program to create tests that check uncovered areas, and by creating good regression suites.

4.3.1 Detection of Illegal Events

The most direct way in which coverage can assist the verification process, is by detecting illegal events. Detection of illegal events can be difficult, since they may consist of a series of sub-events, each occurring in different parts of the design or at different times.

Comet detects illegal events during processing of the event trace. If it discovers a coverage task that is not part of the TCL for the model it is currently processing, it marks this task as an illegal task. For example, if in a given cycle the branch unit pipe contains $a(1) = 4$ and $a(2) = 0$, Comet will detect it as an illegal task, since all the tasks with these values were removed from the TCL due to the 2nd restriction in Figure 4.

When such an illegal task is detected, Comet stores it in a special table, and notifies the user of detection of the task. Along with the illegal task, Comet also stores other information that can help the user to identify the task easily. This information includes the test in which the task was detected, the cycle in which it occurred, and the restrictions that the task violates.


Coverage Status

Model: MODEL2

TCL: TCL_MODEL2

field13	Coverage status	
0	63.23%	(609 out of 963)
1	4.35%	(7 out of 161)
2	2.67%	(11 out of 412)

Total: 40.82% (627 out of 1536)

Figure 6: Simple coverage report for branch unit pipe

4.3.2 Coverage Reports

Analysis of coverage data can provide the user with vital information on the status of the verification process. For example, by looking at a map of covered and uncovered tasks, the user can detect areas that have not been tested. By examining the progress of coverage over time, one can find out when the testing is not producing any new uncovered tasks. We can also see the effects of changes in the testing strategy on the progress of coverage.

Comet provides several analysis and reporting mechanisms to the user. The most basic one is examination of the coverage map in a specific cross section, for example, the coverage of a single attribute. Figure 6 shows the coverage of the field13 attribute of the branch unit pipe model. The figure shows that the coverage when field13 is 0 is high (more than 60% of the legal tasks with this value are covered), while the coverage for the other two values is very low.

This type of coverage analysis can be used to improve the quality of testing by directing the test generator to generate tests that will reach tasks that are not currently covered. It can also assist the designer in finding new restrictions, which were overlooked previously, and thus, get better understanding of the design and the environment.

Another type of report that Comet can generate is the progress in coverage as a function of time (measured either by testing days, testing cycles, or the number of tests). This type of report can help detect the affects of changes in the testing strategy on the progress of coverage and the time when testing stops being effective since it is not reaching new tasks. For example, Figure 7 shows the progress in coverage of the IIwI model as a function of the number of generated tests. The figure shows that, after about 30,000 tests, the rate of new tasks became low. This caused us to

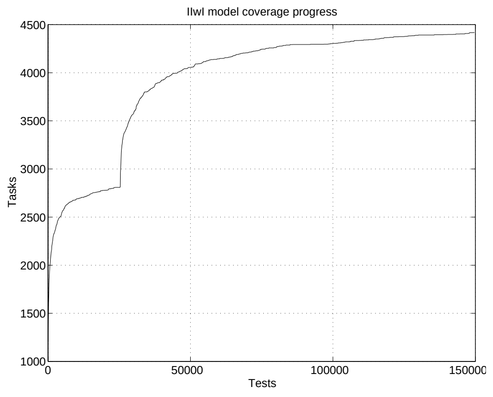


Figure 7: Coverage progress for IIwI model

change the test specification for the test generator which, in return, increased the rate of new tasks dramatically.

Comet can generate other types of reports, such as a list of uncovered tasks, tasks that were covered more than a minimal number of times, and the number and list of new tasks that were covered between certain dates.

4.3.3 Creation of Regression Suites

Coverage can also be used to find good regression suites. A regression suite [10] is a set of tests that is run periodically or after major changes to the design, to check that no new bugs were introduced to the design. A regression-suite must, on one hand, be economic so that it can be executed many times. On the other hand, a regression suite must be comprehensive, so that it will be likely to discover new bugs. A good example for a compact and comprehensive regression suite is a minimal set of tests that covers all the tasks in a coverage model (or a set of models). We have developed a new algorithm for creating small regression suites incrementally on the fly [6]. The algorithm creates small regression suites that cover all the tasks that were covered in the past. For example, we created a regression suite of 742 tests for 5 models of microprocessor pipelines containing 9810 tasks out of 500,000 tests used. We incorporated the algorithm into Comet.

5 Conclusions

In this paper, we presented a new methodology for coverage, that separates the coverage model from the coverage tool. The advantage of this methodology is that it enables us to define coverage models according to the specific needs of each unit, while keeping the benefits of a coverage tool, such as data collection and processing, and coverage analysis.

Based on this methodology, we developed a general purpose coverage measurement tool called Comet. Comet enables us to do coverage work on many types of designs, ranging from ASICs to systems, and in many verification domains, such as system verification and micro-architecture verification. Our experience from using Comet as our coverage tool, shows that it can be used for a large majority of our coverage needs, and that its users can benefit from the advantages of a coverage tool without losing the flexibility of defining their own models.

We are currently working to enhance the capabilities of Comet in many areas, such as automatic feedback to the test generator, enhanced reports capabilities, and capabilities to process very large amounts of data (insertion and processing of traces from hundreds of simultaneous sources). Another area which we are working on, is using data mining techniques [7] for analysis of coverage status.

References

[1] A. Aharon, D. Goodman, M. Levinger, Y Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of PowerPC processors in IBM. In *Proceedings of the 32nd Design Automation Conference*, pages 279–285, June 1995.

[2] A.M. Ahi, G.D. Burroughs, A.B. Gore, S.W. LaMar, C.R. Linand, and A.L. Wieman. Design verification of the HP9000 series 700 PA-RISC workstations. *Hewlett-Packard Journal*, 14(8), August 1992.

[3] J. Baumgartner and R. Raghavan. Method to compute test coverage in complex computer system simulation. *IBM Technical Disclosure Bulletin*, 40(3):1–4, March 1997.

[4] B. Beizer. The Pentium bug, an industry watershed. *Testing Techniques Newsletter, On-Line Edition*, September 1995.

[5] J.S. Bowman, S.L. Emerson, and M. Drnovsky. *The practical SQL handbook: using structured query language*. Addison-Wesley, 1996.

[6] E. Buchnik and S. Ur. Compacting regression-suites on-the-fly. In *Proceedings of the 4th Asia Pacific Software Engineering Conference*, December 1997.

[7] P. Cabena, P. Hadjinian, R. Stadler, J. Verhees, and A. Zanasi. *Discovering Data Mining, from Concept to Implementation*. Prentice Hall, 1997.

[8] J.R. Horgan, S. London, and M.R. Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9):60–69, September 1994.

[9] A. Hosseini, D. Mavroidis, and P. Konas. Code generation and analysis for the functional verification of microporcessors. In *Proceedings of the 33nd Design Automation Conference*, pages 305–310, June 1996.

[10] B. Marick. *The Craft of Software Testing, Subsystem testing Including Object-Based and Object-Oriented Testing*. Prentice-Hall, 1985.

[11] S. Ur and A. Ziv. Off-the-shelf vs. custom made coverage models, which is the one for you? In *proceedings of STAR98: the 7th international conference on software testing analysis and review*, May 1998.

[12] Vhdlcover: How thoroughly have you simulated your vhdl code? http://www.veda.co.uk/vhdlcov.html.

[13] Visual StateScore. http://www.summit-design.com/ products/vcd/index.html.