Off-The-Shelf Vs. Custom Made Coverage Models, Which Is The One for You?

Shmuel Ur and Avi Ziv
IBM Haifa Research Lab
MATAM
Haifa 31905, Israel
email: {sur, aziv@vnet.ibm.com}

Abstract

We compare the benefits and costs of using off-the-shelf coverage tools vs. application specific coverage tools. We start with an overview of coverage, its benefits and its risks. We elaborate on relatively unfamiliar functional coverage as a way to create custom made coverage models. We explain what functional coverage models are, how to create them and their great benefits with an example from our experience. We provide guidelines on how to decide if coverage should be used at all and whether code based or functional coverage (or both) should be used.

1 Introduction

Testing is one of the biggest problems of the software industry. The cost of testing is usually between 40-80% of the development process (70% for Microsoft) as compared with less than 20% for the coding itself [4]. The practice of letting the users find the bugs and fixing them in the next release is becoming dangerous and costly for three main reasons: reputation and brand-name are harmed, replacing the software can be very costly when there is a large install base, and litigation can be expected if the software error caused harm to the user. Therefore, one has to be certain that testing resources are used efficiently and that the testing is thorough.

The main technique for demonstrating that the testing has been thorough is called test coverage analysis [10]. Simply stated, the idea is to create, in some systematic fashion, a large and comprehensive list of tasks and check that each task is covered in the testing phase. Coverage can help in monitoring the quality of testing, assist in creating tests for areas that have not been tested before, and help with forming small yet comprehensive regression suites [5].

Coverage, in general, can be divided into two types: program-based or functional. Program based coverage concentrates on measuring syntactic properties in the execution, for example, that each statement was executed, or each branch was taken. This makes program based coverage a generic method which is usually easy to measure, and for which many tools are available. Examples include program based coverage tools for C [9], C++ [13], and Java [14]. Functional coverage, on the other hand, focuses on the functionality of the program, and it is used to check that every aspect of the functionality is tested. Therefore, functional coverage is design and implementation specific, and is more costly to measure. Currently, functional coverage is usually implemented manually or by using custom made tools.

In this paper, we describe functional coverage and how it should be used in the testing process, starting from test requirements and specifications, and going all the way until and beyond the release of the product. We discuss the advantages and disadvantages of functional coverage and compare it to program based coverage. Based on this comparison, we provide guidelines on which type of coverage best fits the user's needs and how it should be used.

Since functional coverage models are derived from the specifics of the program specifications and implementation, it is impossible to find off-the-shelf tools that fit the user's coverage models. Moreover, the development of tools for specific models is usually too expensive and time consuming. The result is that functional coverage tools are usually simple and do not have the rich functionality of program based coverage tools. To overcome this problem, we developed, at IBM's Haifa Research Laboratory a new functional coverage tool that separates the coverage models which are defined by the user, from the tool itself. The user can define custom made 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.

The rest of the paper is organized as follows: In Section 2, we provide a short background on what coverage is and what the benefits and risks of using it are. In Sections 3 and 4, we describe program based and functional coverage. In Section 5, we describe Comet, a coverage tool that was developed at IBM's Haifa Research Laboratory. Comet enables the user to define her own coverage models and still enjoy the rich functionality of an off-the-shelf coverage tool. In Section 6, we compare the two coverage methodologies and provide some guidelines on who should use each methodology and how to use it. In Section 7, we provide some general guidelines for the use of coverage. Section 8 concludes the paper.

2 The Benefits and Risks in Using Coverage

Coverage is defined as any metric of completeness with respect to a test selection criteria [3]. Many such metrics have been suggested in the past [3], of which statement coverage is the most common. Full statement coverage means that every statement in the program has been executed by the tests. Coverage is one of the more systematic ways to check that the testing has been thorough. When using any coverage model, of which many are available [11], a metric is created against which the quality and completeness of the testing is measured.

The most commonly used coverage metrics are based on the control flow of the program, such as statement coverage and branch coverage, however, many other metrics exist. Some coverage metrics are based on the data flow of variables, like define-use [3], while others are not based on the program code but on the inputs or the specifications.

Coverage is usually used to find new testing requirements that have been overlooked in the test plan. Many times the test requirements are written during the design and do not take into account the details of the implementation. For example, the implementation of a sorting function might use two different algorithms, depending on the size of the array sorted, a detail which is not in the specifications. In this case, statement coverage might show that the inputs never included the case of a short array, that use one of the algorithms, and that a new test is needed. Working with coverage as a guide to improve the quality of testing has been shown to be a cost effective use of resources [12].

Another application of coverage that is commonly used, is generation of regression suites [10]. Generation of regression suites has to deal with two contradictory requirements; the suite must be small so that it is economical to execute it after every design change, yet it must be comprehensive in order to find the bugs that were introduced. Coverage enables us to find a relatively small set of tests which is comprehensive in the sense that it covers the required metric [5].

Besides these uses, coverage provides other benefits to the testing process that are often overlooked. One such benefit is the use of coverage or, more specifically, functional coverage, to assist in defining testing requirements and specifications. Another benefit of functional coverage is that it helps to achieve a better understanding of the tested program during definition of the coverage models.

While the use of coverage as an aid to the testing process has a lot of benefits, centering the testing process around coverage has its own risks. A common misconception about coverage is that the testing methodology should be to decide on an appropriate coverage metric and then generate a set of tests that covers it. This is not advisable for a number of reasons, the main one is demonstrated by the following analogy to wall painting.

Assume that one uses a process which paints a wall with an inch of protective paint. A quality assurance process (coverage) chooses a hundred points at random at which to drill and measure *the paint thickness. If at all of the points there is an inch of paint, then we have a reasonably good assurance that the wall is well painted. However, what do we do if at 50 points the layer of painting is not thick enough? If we tell the painter the location of these 50 points and he fixes them, the fact that the 100 points are* <u>*now*</u> *covered no longer guaranties that the wall is covered because our painting (test generation) process and our coverage process are not independent. One way to overcome this difficulty is to inform the painter that some region of the wall is not well painted. This is equivalent in testing to generating a test requirement (as opposed to a specific test) from an uncovered point.*

Another drawback of coverage is that many coverage models are ill suited to deal with many common problems. For example, control flow models, such as statement and branch coverage, are ill suited to deal with missing code. If, for example, a case statement should have six cases but, in practice, it has only four, statement or branch coverage will not help you find it. One way to overcome this difficulty is to use several coverage models, which are derived from different domains, so that one model will cover the weaknesses of another model.

A different risk in using coverage is setting low coverage goals. It has been shown that using coverage to assess quality with a lower coverage target (50%-90%) is not useful [12]. The reason is that the probability of having bugs in hard-to-cover areas tends to be larger than the probability of bugs in well covered areas. Therefore, it is better to use simpler coverage models with high coverage goals than more complex models with lower coverage goals.

3 Program Based Coverage

Program based coverage, usually just called coverage, is a technique that measures the execution of tests against the source code of the program. For example, one can measure whether all the statements of the program have been executed. The main uses of program based coverage are assessing the quality of the testing, finding missing requirements in the test plan and constructing regression suites.

A number of standards, as well as internal company policies, require the testing program to achieve some level of coverage, under some model. For example, one of the requirements of the DOA standard [15] is 100% statement coverage.

Many coverage tools that support all major programming languages exist. Every tool implements a number of coverage models for a particular combination of operating system, compiler and programming language. Most of them work by instrumenting the source code and adding counters which can later be used by the tool's user interface to show the status and progress of the coverage in some detail. To apply such a tool, one typically has to recompile the software with the tool and execute the tests. After the tests are executed, there is usually some interface that highlights the parts of the program that were not covered.

Almost all coverage tools implement the statement and branch coverage models. Multi-condition coverage, a model that checks that each part of a condition (e.g. A or B and C) had impact, is also implemented by many tools. Fewer tools implement the more complex models such as define-use, mutation, and path coverage variants [6].

The main advantage of program based coverage tools is their simplicity of use. The tools come ready for the testing environment. No special preparations are needed in the programs and understanding the feedback from the tool is straightforward. The main disadvantage of code coverage tools is that the tools do not "understand" the application domain. Therefore, it is very hard to tune the tools to areas which the user thinks are of significant.

4 Functional Coverage

Unlike program based coverage, where the execution of tests is measured against the program source code, functional coverage focuses on the functionality of the program, and it is used to check that every aspect of the functionality is tested. Therefore, functional coverage is design and implementation specific, and is harder to measure. Currently, functional coverage is mostly done manually.

Functional coverage is considered by some to be black-box testing [6], since it involves models based on the specifications of the application. We believe that functional coverage is much more varied. Functional coverage models can be based on the specifications of the application, but they can also be derived from the implementation. Functional coverage models have many flavors. Models can cover the inputs and outputs of the program or they can look at the internal state of the program (e.g., values of variables). Functional coverage models can be snapshot models, that look at the state of the program at a certain time, or they can be temporal models that deal with scenarios. Usually, functional coverage models involve looking at several properties in parallel. Our experience shows that many bugs can be found only when a number of events happen concurrently [1]. Therefore, covering each event on its own is not sufficient. A simple example for a snapshot model is covering all the possible values of the input parameters of a function. An example for a temporal model is looking at the changes in the values of global variables between consecutive activations of a function. Thread interleaving and synchronization in a multi-threaded system is a source of many bugs. Therefore, a coverage model that looks at all the reasons for thread switching is a good example for a coverage model that is based on a bug model (A bug model is a set of requirements for finding bugs of a type that have been uncovered before.)

The first and most important step in the functional 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 testing 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 risky or error prone. Next, the size of the model should be chosen in accordance with the testing resources. The model size (number of tasks) should not be too large, making it impossible to cover the model, given the testing time and resources available. From our experience, we found that the best way to create effective models is to start from small models and later refine them or combine them to create bigger and more complex models.

Often, some of the tasks in the coverage model are *illegal tasks*, that is, coverage tasks that should not occur. The reasons could be limitation on the inputs - the sum of angles of a triangle is 180 - or implementation details - two threads that write to the same resource should never be in the write stage of the semaphore at the same time. Specifying the illegal tasks is an important part of coverage model definition.

After definition of the coverage models, the next step is data collection. There are two broad categories of coverage data collection techniques. The first, which is used in most of the program based coverage tools, is to create a counter for each task and modify it on the fly whenever the task is found. The second is to print the necessary data to a trace file that is later processed.

One of the historical problems of functional coverage, which stems from the fact that the models are implementation specific, is lack of automation. However, although each model is unique, the coverage processes of different models have much in common. Tasks have to be updated in tables, regression suites have to be created, coverage reports on sub-models and on progress have to be made. We have created a coverage tool, named Comet [7], that handles all the common requirements, and we have created over a hundred coverage models for a number of customers for widely varying applications (mainly in hardware). A more detailed description of Comet and the methodology behind it is given in Section 5.

The ability to focus on points of concern, which is one of the main advantages of functional coverage, carries a risk. The risk is that only functional coverage will be used, and therefore, *only* the parts which are of concern will be tested thoroughly. Since creation of coverage models requires some effort, the parts which are not of special concern could be neglected. For these parts, it is better to use generic, off-the-shelf, program based coverage models than nothing at all.

4.1 Example - Parcel Sorting System

We demonstrate applying a functional coverage to a parcel sorting system that was developed at IBM's

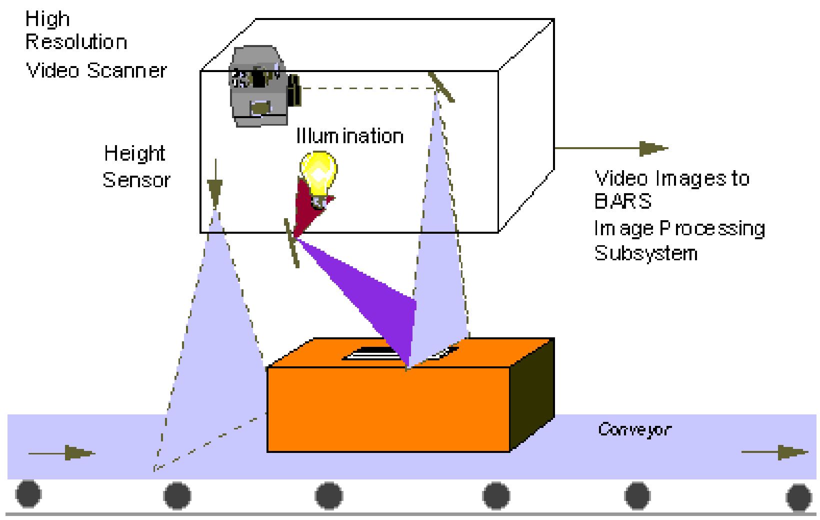

Figure 1. Structure of the parcel sorting system

Haifa Research Laboratory. In this system, a 6 PowerPC SP system (a multi processor system) is used to sort parcels. The parcels are sorted in a number of phases. In the first phase, the top surface of the parcel is scanned, its dimensions are measured, and the lenses are focused. Then, a processor is assigned to the parcel. In the second stage, a high resolution image of the top surface is created, the image is decoded so that the address and bar-code on it are understood. In the third stage, the weight of the parcel is acquired. The information including the size, weight and address is then delivered to the host. The parcel sorting system is depicted in Figure 1. This system contains many technologies, such as voting between two powerful OCR engines, fuzzy dictionary search for address validation, parallel processing and leading edge image data transfer (50MB/sec).

The coverage model described below is designed to check that the system has been stressed by different inputs. The model checks that every type of parcel size and every type of address, is encountered. The model verifies that a parcel has been internally processed in every way and by every processor, and that the processing can be correctly interrupted at any stage. The correlation between success (correct decoding of addresses), parcel type, and system load is measured.

A coverage model is composed of a list of tasks. A convenient way to specify tasks is as tuples of variables or attributes. The list of tasks in the coverage model is the cross product of the lists of all the possible values for each attribute. For example, the coverage model that we illustrate here, has attributes of a single parcel and the load of the system during the processing of that parcel. Figure 2 gives the list and description of the attributes for the parcel sorting system model. Some of these attributes, such as `Num_addresses`, are values of variables in the program. Some, like `Width`, are calculated from a single variables, and others, such as `Num_processors` and `Flush`, require processing of the trace. Not all the tuple values are possible. Restrictions on the possible tuples are imposed by limitations on the inputs caused by the environment, or by the implementation. Examples of restrictions for the parcel sorting model are:

- There are no more than six busy processors at one time
- When a parcel is sent to the master, all the slaves are busy.

Note that the process of finding restrictions is an iterative process. An initial set of restrictions is

```
Attribute list:
 (Height, Width, Length, Master, Num_processors, Num_addresses, Flush,
  Finish, Success)

Attribute description and values:
 •Height - (Height of parcel in milimiters)/100+1
 •Width - (Width of parcel in milimiters)/100+1
 •Length - (Length of parcel in milimiters)/100+1
 •Master - 1 if parcel processed by master else 0
 •Num_processors - Number of busy processors when the parcel started (1-6)
 •Num_addresses - Number of addresses are on the parcel (0-4)
 •Flush - Name of the stage prior to Flush
 •Finish - Name of the stage prior to Finish
 •Success - confidence above 0.8 in recognition
```

Figure 2. Attribute list for parcel sorting system model

```
Model 1:Parcel Size - (Height, Width, Length)
Model 2:Number of addresses - (Num_addresses)
Model 3:Load - (Master, Num_processors)
Model 4:Messages - (Flush, Finish)
Model 5:Success - (Success)
Model 6:All Parcel types - Model 1 + Model 2
Model 7:Load and Messages - Model 3 + Model 4
Model 8:Messages and Success - Model 4 + Model 5
Model 9:Complete model - Model 6 + Model 7 + Model 5
```

Figure 3. Models hierarchy for parcel sorting system

generated. After looking at the traces, exceptions to some of the restrictions are found. Each exception signals a bug in the implementation, a problem in the trace generation, or an incorrect restriction, all of which happen in practice. Also, holes in the coverage measurement that are found may indicate restrictions that were previously overlooked.

Functional coverage models are usually built in a hierarchical way, one on top of each other. We start with small models that examine specific areas, such as the dimensions of a package in the parcel system. Later on, these small models are merged to create larger models. Figure 3 shows the hierarchy of the models for the parcel sorting system. The advantage of using hierarchical models is that problems can be found at an earlier stage of the testing. New rules on sub-models imply new rules on the containing models. Another advantage is that the coverage process is tested first on simple models which makes it easier to debug and cover. Note that not all the models in the hierarchy need to be implemented explicitly. Some models, especially very simple ones, like Model 2 in Figure 3, can be viewed as projections of larger models. Usually, smaller models are implemented if we want to use them to find new restrictions and estimate the size of the big model, so that we will be able to assess if the large model is feasible.

Whenever the parcel sorting system is operating, it creates, as a by-product, a trace of the messages between the processes in the system. This trace includes information such as dimensions for every incoming parcel. It also includes messages which contain the arrival and departure time of a parcel, which processor processed it, and intermediate status of the parcel processing. This trace is processed and, from the information relating to each parcel, we create a tuple that is used in the coverage.

After collecting coverage information for some time we look for coverage holes which we do not succeed to find when generating tests. We first look at each variable separately as there are a fewer tasks to look at this way. We may find out, for instance, that we never saw a parcel on the master processor (`Master = 1`). After we make sure that we have full coverage for each variable, we look at combinations of some of the variables.

There are several possible reasons for holes in the coverage. We have already discussed missing restrictions, in which case we simply add new restrictions. Bugs in the application or the coverage process are also a possibility. However, the most common cause, and the reason for coverage in the first place, is that some tests are missing. We use coverage holes as pointers to areas which need more testing. We iterate the process until the coverage is complete or, more commonly, resources or time are exhausted. Eventually the restrictions will be almost accurate.

In many cases, the number of tasks in the coverage model seems daunting. For example, for the coverage model in Figure 2 with no restrictions, the number of tasks is 6*6*6*2*6*5*7*7*2=1,375,920. In practice, the number of tasks is usually not a problem. The first reason is that the restrictions in many cases reduce the number of tasks by a number of orders of magnitude. In [7] we discuss a model in which the restrictions reduce the number from 30,000,000 to 1,500. The second reason is that one does not start with the full model but with partial models which have a smaller number of tasks.

5 General Purpose Tool for User Defined Coverage Models

One of the properties of functional coverage is that the coverage models are application specific. On the other hand, a common property of most coverage 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. The result is that there are almost no commercially available coverage tools for functional coverage in general, and specifically for the coverage models that the user needs.

This leaves a user who wants to use functional coverage with two options: to do coverage manually, or to build her own coverage tool for her models. The first option makes the coverage work tedious and time consuming and creates a severe limit on the size of the models that can be implemented. The second solution may require a large development effort for a tool that might be used once for a single program. Usually, tools that are built for a single use are less elaborate in the functionality they provide to their users. Such tools also tend to be error-prone. The result is that such tools cost much more than commercial tools and provide much less functionality.

In [7], we propose a different methodology for functional coverage. This methodology calls for separation of the coverage model from the coverage measurement tool. The idea behind this methodology is that most of the functionality provided by existing coverage tools, such as data gathering and coverage reports, is independent of the coverage models, and is thus similar in all tools. The main difference between coverage 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 specifications of the model. Therefore, one of the inputs to the tool is the definition of the models. 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

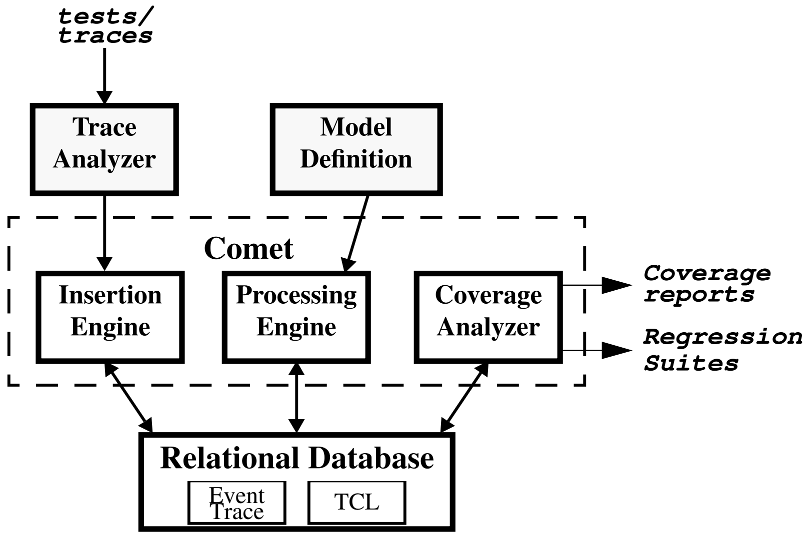

Figure 4. Comet Structure

sufficient.

The advantages of using 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 a 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 of 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.

Based on the methodology described above, we have developed a coverage measurement tool named 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 relies on a relational database in order to supply a comprehensive and stable environment needed for the coverage measurement process itself and an analysis of the coverage results.

Comet consists of three major parts, as shown in Figure 4: the *Insertion Engine* whose task it 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 the 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. A more detailed description of Comet can be found in [7].

6 Comparison Between Program Based and Functional Coverage

In this section, we compare the advantages and disadvantages of program based coverage and functional coverage. The comparison is based on the following criteria: availability of tools, cost, ease of use, learning curve, portability, ability to focus on areas of concern, experience gathered, and ability to cover the entire design. Based on this comparison, we suggest several guidelines on where and when these coverage methods should be used. The summary of the comparison is shown in Table 1.

Property	Program based coverage	Functional coverage
Availability	Many available commercial tools	Almost no commercial tools available
Cost	Cost of tools is usually low, no additional costs	Higher cost to develop tools and models
Ease of use	Easy to use. Plug and play	Need expertise in definition and implementation of models
Learning curve	Users can benefit from tool almost immediately	Need some time to learn how to define and implement models
Experience gathered	Vast experience on usage, coverage targets, etc.	Little experience, harder to reflect from one project to another
Focus	Uniformly spread on all the entire program	Focuses on areas of concern
Adapting to testing resources	Hard to adapt because of hard coded models	Easy to adapt. Can use right number of models and of the appropriate complexity
Missing code	Not fitted to deal with missing code	Can deal with missing code
Impact on project	Impact only on testing and testing requirements	Many areas of impact

TABLE 1. Comparison between program based and functional coverage

Program based coverage is a well established method. Many tools are available for almost any programming language. Program based coverage has been extensively used in the industry. It is estimated that 20-30% of the testers have used it at least once. Most of the program based coverage tools are inexpensive, easy to use, and do not require any (manual) changes in the code. The results they provide are easy to understand. We have had experience in the past in which we have found bugs using branch coverage tools less than an hour from the time the tool was introduced. These tools give a general sweep of the entire code for areas which where not used or not sufficiently used. The cost of the tools is relatively small, and their running time overhead is usually between 20% to 100%.

Another big advantage for program based coverage is the vast experience that has accumulated from using it. There are many experts and publications, such as [3] and [10], that provide guidelines on how to deploy coverage, which coverage models should be used, what the accepted levels of coverage for each model are, and how to interpret coverage results.

Program based coverage tools have some built-in faults. The first is that these tools do not find missing code. Another problem is that a coverage tool is not available for some environments (e.g., language, operating system) and therefore, since program based coverage tools are expensive to create, in this environments coverage is not used.

Traditionally functional coverage was outside of the commercial tools domain. When functional coverage was needed, a domain specific tool was written in order to check it. Tools written for single applications are usually not very elaborate as far as their report generation capability or graphical user interface are concerned. The main advantage that these tools have is that they do exactly what the user wants. The disadvantages are cost, as they do not have the economy of scale, and that the tools are relatively simple and bug prone. With the introduction of functional coverage tools, such as Comet, functional coverage becomes a viable option in many more circumstances. The advantages of such a general purpose coverage tool over domain specific tools are enormous, as explained in Section 5.

More effort, more time, and more expertise are required in the deployment of functional coverage. The technical work of implementing a functional coverage model takes several hours, and requires a better knowledge of the tool than the use of a program based coverage tool. Moreover, the knowledge on how to create good functional coverage models requires additional expertise and special training. This means that the learning time for using functional coverage is much longer than for program based coverage.

As far as ease of use, learning curve, and cost are concerned, program based coverage has the advantage. However, it is our experience that functional coverage models have many advantages as well. Creating a functional coverage model, as opposed to using an off-the-shelf tool, gives the application engineer a fresh point of view on his applications. It is our experience that creating the coverage models by itself, without even collecting the information, is considered worthwhile by our customers. We have found, through many examples, that the people who design and code the program cannot predict which configurations or scenarios are possible in the programs. Analyzing possible scenarios will, in some cases, find bugs. For this reason, functional coverage is now being incorporated into the high level design stage of a very large project that we work with.

Functional coverage methodology enjoys a number of other advantages over program based coverage. The first and most important is the ability to create coverage models for areas of concern. For example, in maintenance where most of the total cost is spent, small changes are made to existing, tested, software. One would like to be able to focus the tests and coverage only on the areas of these changes. By using functional coverage, one can focus the resources where they are needed. Other examples include applications which contain a complex logic part, such as a scheduler in an operating system. One would like to ensure that the level of testing for such sections of the application is very high.

The ability to focus on certain areas also helps with adapting the coverage to the available testing resources. With program based coverage, if you do not have the amount of resources needed to do statement coverage you will probably not do coverage at all. If your resources are more than sufficient to cover everything your coverage tool marked, you will not be able to use your additional resources. Functional coverage models can be as detailed or as high-level as desired. This advantage has an accompanying disadvantage: if you want to test the application, and there is nothing specific you want to focus on, then program based coverage is better than functional coverage.

Functional coverage is less vulnerable to the problem of missing code then program based coverage, since it is usually derived from the specification, not the code itself. Missing specification is still an area of concern but it can be circumnavigated in most cases if the coverage model is well defined.

Program based coverage should be used for a uniform check on the test plan. Whenever the testing resources are enough to complete the test plan, program based coverage is a cost effective way to find missing requirements which should be added. Functional coverage, on the other hand, should be risk driven. A few areas of the application which either contain risk or are very complex and error prone should be identified and functional coverage should be applied to them. If there is no specific area which is of concern, then only program based coverage should be used. If the resources are not enough for using coverage for the entire code base, but there are areas of specific risk, only functional coverage should be used. This can be the case when maintenance is done where only small changes in tested software need to be tested thoroughly. For these kinds of changes, specific models can be made and the testing will focus on the changes.

7 General Guidelines for Usage of Coverage

In this section, we try to share some of our experience as to how coverage should be applied. Coverage should not be used if the resources used for it can be better spent elsewhere. This is the case when the budget is very tight and there is not enough time to even finish the test plan. In such a case, designing new tests is not useful as not all the old tests will be run. Coverage should be used only if there is a full commitment to make use of the data collected. Measuring coverage in order to report coverage percentile is practically worthless. Coverage points out parts of the application that have not been tested and guides test generation to these parts. Moreover, it is very important to try to reach full coverage or at least set high coverage goals, since many bugs hide in hard-to-reach places. It is usually worthwhile to create an automatic test generator instead of generating all the tests manually. The combination of an automatic test generator and coverage is very potent as the tests can be biased in the direction in which coverage is missing.

Coverage is a very useful criteria for test selection for regression suites. Whenever a small set of tests is needed, the test suite should be selected so that it will cover as many requirements or coverage tasks as possible.

When coverage and reviews are used for the same project reviews can put less emphasis on things that coverage is likely to find. For example, a review for dead code is unnecessary if statement coverage is used, a review for boundary error in loops is redundant if the appropriate mutation coverage model [8] is used, and manually checking that some values of variable can be attained is not needed if the appropriate functional coverage model is used.

When there is a coverage expert who helps a number of projects, such as in our organization, an effort should be made to make the coverage reports as succinct as possible. It is more useful to have feedback of the form function `SORT` was never called than line 2 in function `SORT` was never executed (as well as lines 5,8,11...). For functional coverage this comment is even more important. A functional coverage model is usually an N dimensional space over some variables. The information that a single point has not been covered is not as useful as the information that a sub-space (as large as possible) was not covered.

The following guidelines are applicable only to functional coverage:

- First, the most complex, error prone part of the application has to be identified. Coverage models should be written for these parts at the high level design stage. The benefits are a fresh look at the design, a good start for the automation of the test plan and creating a model that will fit the implementation as it evolves, assuming that it still implements the same design in contradiction to program based coverage, where coverage can start having an impact only after tests have been executed.
- Coverage models should be created hierarchically. One should start with simple small models and combine them to create larger models. Figure 3 illustrates a hierarchy of functional coverage models. Using hierarchical models enables us to debug the coverage process on simple cases and find problems in the coverage models early.
- Restrictions should be created at model creation time. It is our experience that programmers have a very hard time specifying the correct restrictions (They never do.) but that they find that they learn a lot about the design. Before coverage is measured, sample traces should be looked at to check that the restrictions are correct.

8 Conclusions

In this paper, we compared functional coverage to program based coverage. We have shown that each has its own merits and drawbacks. It has been our experience, as well as the experience of anyone that we know has used coverage, that coverage is worth doing. Almost anyone, and under any timing and budget consideration, can benefit from some form of coverage. However, one has to commit to coverage and use it properly.

Functional coverage is a more powerful testing technique then program based coverage, since it can focus on areas of concern, and contribute to the design and verification processes in many more ways. On the other hand, functional coverage is more complicated and requires more resources than program based coverage. We therefore recommend that it should be reserved to those parts of a program that are of special concern. Program based coverage, on the other hand, can and should be uniformly applied to the entire application.

References

[1] Y. Abarbanel-Vinov, and S. Ur. Processor Bug Classification and Modeling, IBM's Haifa Research Lab internal document, 1996.

[2] 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.

[3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[4] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1995.

[5] E. Buchnik and S. Ur. Compacting regression-suites on-the-fly. *In Proceedings of the 4th Asia Pacific Software Engineering Conference*, pages 385-94, December 1997.

[6] S. Cornett. Software Test Coverage Analysis, http://www.bullseye.com/webCoverage.html

[7] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage - a tool supported methodology for design verification, to appear in *Proceedings of the 35th Design Automation Conference*, June 1998.

[8] W.E. Howden. Weak mutation testing and completeness of test sets, *IEEE Transactions on Software Engineering*, 8(4):371-379, July 1982.

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

[10] B. Marick. *The Craft of Software Testing, Subsystem Testing Including Object-Based and Object-Oriented Testing*. Prentice-Hall, 1995.

[11] C. Kaner. Software negligence and testing coverage, In *proceedings of STAR 96: the Fifth International Conference, Software Testing, Analysis and Review*, pages 299-327, June 1996.

[12] R. Stewart. Unit test coverage as leading indicator of rework, *EuroSTAR 97*, November 1997.

[13] C-Cover - Test Coverage Analyzer for C/C++, http://www.bullseye.com/webCcover.html

[14] DeepCover for Java, http://www.rstcorp.com/DCJava.html

[15] Software test and evaluation guidelines, Department of the Army, Pamphlet 73-7