Applications of Synchronization Coverage

Arkady Bron
IBM Haifa Research Lab,
Mount Carmel,
Haifa 31905, ISRAEL
bron@il.ibm.com

Eitan Farchi
IBM Haifa Research Lab,
Mount Carmel,
Haifa 31905, ISRAEL
farchi@il.ibm.com

Yonit Magid
IBM Haifa Research Lab,
Mount Carmel,
Haifa 31905, ISRAEL
yonit@il.ibm.com

Yarden Nir
IBM Haifa Research Lab,
Mount Carmel,
Haifa 31905, ISRAEL
yarden@il.ibm.com

Shmuel Ur
IBM Haifa Research Lab,
Mount Carmel,
Haifa 31905, ISRAEL
ur@il.ibm.com

ABSTRACT

Coverage analysis is a useful testing technology. However, some coverage models are more acceptable to the industry than others. In the field of testing multi-threaded applications, there is a need for a coverage model that can be used to evaluate tests for concurrent completeness and to find new testing requirements. We present a new coverage model: synchronization coverage. This model is simple to understand and the action items generated by each uncovered task are clear to testers and developers. We propose that synchronization coverage could, and should, become one of the more commonly used coverage models.

Categories and Subject Descriptors

D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing)*

General Terms

Algorithms, Measurement, Reliability, Experimentation, Theory, Verification

Keywords

Multi-threading, coverage, testing

1. INTRODUCTION

Coverage analysis is a common technique used both to evaluate the quality of testing done and to find areas of the code that need additional testing. No coverage measure dedicated to concurrent aspects of applications has become accepted practice in the industry.

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

In this paper, we first discuss the requirements that a coverage model needs to fulfill to gain acceptance in the industry. We then present a coverage model in detail: synchronization coverage. This model is dedicated to concurrent aspects of programs that fulfill these requirements. It has gained acceptance in IBM and is now being presented outside IBM. We demonstrate several projects in which it was used in different roles and explain the advantage of each role.

This paper is organized as follows: in the second section, we explain basic coverage terms, as well as why coverage is used and for what it is useful. In the third section, we present our insights about what makes some coverage models superior, in practice, to others. In the fourth section, we present synchronization coverage. In the fifth section, we report on the use of synchronization coverage in the field.

2. BACKGROUND ON COVERAGE

Testing is one of the biggest problems of the software industry. The cost of testing is usually between 40 and 80% of the development process, as compared with the coding itself which may be less than 20%[3]. The main technique for demonstrating that testing has been thorough is called *test coverage analysis* [13]. Simply stated, the idea is to create, in a 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, in creating tests for areas that have not been tested before, and with forming small yet comprehensive regression suites [4]. In this section, we discuss coverage and how it is used in the industry.

Coverage is defined as any metric of completeness with respect to a test selection criterion [2]. Many such metrics have been suggested in the past [2], of which statement coverage is the most common. Complete statement coverage means that every statement in the program has been executed by some tests. Coverage is one of the more systematic ways to check that 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 [2], 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 were overlooked in the test plan. Often 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 uses 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 [16].

Another application of coverage that is commonly used is the generation of regression suites [13]. The generation of regression suites has two contradictory requirements: the suite must be small enough to be economical to execute after every design change, yet it must be comprehensive enough to find the bugs that were introduced. Using coverage one can find a relatively small set of tests, which is comprehensive in the sense that it covers the required metric [4].

A number of standards, as well as internal company policies, require that the testing program achieves some level of coverage, under some model. For example, one of the requirements of the DOA standard [1] is 100% statement coverage.

There are many coverage tools that support all major programming languages. Each 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 that can later be used by the tool's user interface to show the coverage status and progress 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.

3. WHAT MAKES SOME COVERAGE MODELS SUPERIOR?

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., in (A or B) both A and B impacted its value) 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]. An interesting question is what makes a coverage model "good". Comparative work on the strength of coverage models, both theoretical and practical [14][17], has been published. We believe that strength is not the major issue. We have come to the conclusion that for a coverage model to gain acceptance, it must fulfill all of the following requirements:

- The model should be created statically from the code prior to the testing, and each task must be well understood by the user. While this may seem like an obvious requirement it does not hold in [5].
- Almost all coverage tasks must be coverable and for the few that are not, due to practical limitations of testing, a review process should be used. While this is true for statement coverage, it is not the case for multi-condition coverage, define-use coverage, and mutation coverage, and this is probably the reason why those coverage models are not as popular. It turns out that if only most of the coverage tasks are attainable, but the user does not know which ones they are and spends time trying to cover unattainable tasks, the coverage model will be discarded. Work on automatically calculating whether coverage tasks are coverable [15] was partially motivated by making more coverage models conform to this requirement.
- Every uncovered task should yield an action item: either a bug in a program that needs to be fixed, or missing tests that need to be written.
- Some action is taken upon reaching 100% coverage for the model (e.g., the testing phase is complete).

The popularity of statement coverage follows from the fact that all the conditions hold. It is well understood what is meant by a statement being covered or not. A statement is almost always coverable, and if not, it is usually easy to see why. When a statement is not covered, an action item is created; either the statement is dead code and should be removed, or new tests are needed. Many consider 100% statement coverage to be a unit test exit criterion.

While those criteria hold for statement coverage and a few other coverage models in the sequential domain, they do not hold for any of the previously suggested coverage models in the concurrent or multi-threaded domains[18]. Consequently, no coverage model that evaluates the quality of testing with regard to testing multi-threading/concurrent programs is in use. In this paper, we define such a model, *Synchronization Coverage*, and report on its use in the field.

Synchronization coverage is introduced in section 4 in the context of the Java language, although the same ideas were easily transfered to the C/Pthread context. Synchronization coverage is a model whose goal is to check that the synchronization statements in the program, such as `synchronize block`, `wait()`, `notify()`, `notifyAll()` of an object instance and others, have been properly tested. When a block is guarded by a synchronization statement, it is to ensure that only one thread at a time will be able to execute that code. It is standard programming practice to make the synchronization blocks as small as possible in order to enable concurrent execution. Therefore, in most executions, the program will not have a thread switch while executing the area guarded by synchronization, which means that in most cases the synchronization was a no-op. This is not to say that the synchronization statements are not needed but that they rarely impact the execution. To show that a synchronization statement had impact on the execution, a test is needed in which a thread was stopped from progressing (or stopped another thread from progressing) due to a synchronization statement.

An interesting and surprising way to demonstrate that synchronization is not tested is to remove from the program all synchronization protection, in the form of synchronized blocks or synchronized methods, and show that the tests still succeed. When this is done, it clearly demonstrates that synchronization has not been tested at all. We have implemented synchronization coverage in ConTest [7] an in-

ternal IBM tool, which is currently being evaluated by several external users. It is our experience that without measuring synchronization coverage and causing contentions on the resources, most projects have very low synchronization coverage.

In a previous work [8] on the coverage of concurrent programs, an axiomatic approach was taken. Test data adequacy criteria, the axioms, were suggested for concurrent programs. These were used to evaluate two concurrent coverage criteria, covering all the *happen-before* relations and covering, for two events that are not comparable in the *happen-before* relation, their two possible orders of execution. It was found that coverage criteria that combine sequential and concurrent aspects are required in order to satisfy all of the test adequacy properties.

4. SYNCHRONIZATION COVERAGE MODEL

At IBM, we developed a coverage model dedicated to concurrent aspects: the synchronization coverage model. The synchronization coverage model tests that the synchronization primitives used in the program did "interesting" things. Each coverage task is related to a particular program location; specifically, a place in the code where a synchronization primitive is used. We have implemented synchronization coverage for Java and for C/Pthread, with minor differences in details. We present them below using common terminology, while mentioning the specific relevant terms in traditional Java, in the new concurrency libraries in Java 1.5, and in C/Pthread.

The two most important coverage tasks for this model are *blocked* and *blocking*. These tasks are connected to entering "mutual exclusion" code sections. These are sections that can be executed by only one thread at a time; a thread that wants to enter such a section must first obtain some "lock" object. If another thread is holding the lock, the first thread is blocked until the lock is available. In traditional Java, the entrance to a mutual exclusion section is implemented by the start of a synchronized section (block or method); in Java version 1.5, also by calling `java.util.concurrent.locks.Lock.lock()`; in C/Pthread, by calling `pthread_mutex_lock()`. The following description use the traditional Java term. The *blocking* and *blocked* tasks are reported when there is actual contention between synchronized sections – when a thread T1 reaches a synchronized section A, and stops because another thread T2 is inside a section B synchronized on the same lock. In this case, section A is reported as *blocked*, and section B as *blocking*. Note that A and B are often the same section (a synchronized section used by several threads). In this case both tasks related to the same location are reported at once. *Blocking* is also reported for a lock operation if, while a thread T1 is holding the lock following this operation, another thread T2 does try-lock and fails.

Reporting these tasks in runtime is achieved by instrumenting the program and wrapping the calls to the synchronization primitives. We hold data structures that track the state of the various locks. Just before a thread attempts to lock, we check in our data structures whether the lock is held by another thread.

Before describing the rest of the coverage task types, we show how the requirements specified in the introduction for a good coverage model are fulfilled by these two types of coverage tasks.

1. **Each task must be well understood by the user**: admittedly, understanding what the *blocking* and *blocked* tasks mean is not as trivial as understanding statement or method coverage tasks. However, a developer of a multi-threaded application does need to understand how the mutual exclusion works in general, and in her program in particular; that a thread may stop at the entrance to a synchronized section, etc. Given such understanding, the meaning of each task is not hard to understand.

2. **Each coverage task must be coverable**: if the synchronization protocol is implemented correctly, then the vast majority of synchronized sections can sometimes block and sometimes be blocked. As in statement coverage, there are exceptions: for example, if a synchronized method is necessarily the first to be performed in the program, it can only be *blocking*, and never *blocked*. But this is rare; and where it is the case, the developer should be aware of it.

3. **Every uncovered task should yield an action item**: if a synchronization block was never seen to be *blocking*, but (as is usually the case) it is possible for it to be *blocked*, it means that some interleaving scenario has not been tested. Most multi-threaded bugs occur only in some interleavings, sometimes only in rare ones; so such an untested interleaving is a cause for concern, and the test should be strengthened. Similarly for an uncovered *blocked*. Alternatively, perhaps there can possibly not be a contention; this means that the synchronization is redundant, and (since it costs in runtime) should usually be removed.

4. **Some action is taken upon reaching 100% coverage for the model**: this can be one of the criteria for exiting the unit test when testing concurrent programs.

We now describe the other types of coverage tasks included in the model. The first two are implemented in ConTest; the other three are planned for future implementations.

- **Try-lock** is an entrance to a mutual exclusion section, which may not block but may succeed or fail (depending on whether another thread is holding the lock). It does not exist in traditional Java; in Java 1.5, this is `java.util.concurrent.locks.Lock.tryLock()`; in C/pthread, `pthread_mutex_trylock()`. For each try-lock operation, we define the tasks *failed* and *succeeded*. Normally a try-lock may sometimes succeed and sometimes fail; if one of them never happened, then either the test is insufficient or the operation is redundant.

- For each *wait* on a condition variable (`java.lang.Object.wait()`, `java.util.concurrent.locks.Condition.wait()`, `pthread_cond_wait()`) we define a task `repeated`. Concurrent protocols use *wait* to make a thread wait for a certain condition to hold, this condition depending on actions executed by other threads. When another

thread makes this condition true, it calls `notify()` (`signal(), broadcast(), notifyAll(), signalAll()`) to make the waiting thread wake up. A programmer might assume, in the code following the *wait*, that the condition holds (relying on the *notify*). But this is a known bug pattern: between the notification and the time the waiting thread resumes running, other threads may have operated and made the condition false again. Therefore, *wait* should usually be done in a loop, testing the condition before continuing. Furthermore, in some systems, *wait* may wake-up "spuriously", without notification (or timeout or interrupt); this behavior is part of the spec in Java version 1.5, for example. It is then nearly always mandatory to test the condition in a loop. The *wait repeated* coverage task is reported if this *wait* was called twice within a single run of the synchronized block it is in (a *wait* must be called inside a block synchronized on the same lock). That is, the thread that was in a synchronized block called *wait* (thereby relinquishing the lock), finished the *wait* (thereby reacquiring the lock), and called the *wait* again, before leaving the synchronized block.

- `Semaphore-wait` (`java.util.concurrent.Semaphore.acquire()` in Java 1.5, `sem_wait()` in C) *blocked* and *not-blocked*. A call to *semaphore-wait* will report the first task if it had to stop because the semaphore was not immediately available, and the second otherwise.

- `Semaphore-try-wait` (`java.util.concurrent.Semaphore.tryAcquire()`, `sem_trywait()`) *succeeded* and *failed* - similarly to *try-lock*.

- `Notify` (`java.lang.Object.notify()/notifyAll()`, `java.util.concurrent.locks.Condition.signal()/signalAll()`, `pthread_cond_signal()/broadcast()`) *had-target* and *had-no-target*. A notify call will report *had-target* if there was another thread in a corresponding wait, and *had-no-target* if not. In the latter case, the *notify* will have no effect. Sometimes this is a bug, because the programmer assumed a corresponding *wait*; but the *wait* was actually called after the *notify*, and there is no subsequent notification: the program hangs (this is the "lost notify" bug pattern). To increase the chance of finding such bugs, each *notify* should be tested when it doesn't have a target *wait* (for most *notifies*, this is possible). Naturally, most *notifies* should have a target, or else they are redundant; that's what the *had-target* tests. These tasks are optional and the user may decide if the tool will create them, as for some testers they require too deep an understanding of the multi-threading issues involved.

One can perhaps think about other types of synchronization tasks, depending on the primitives available in the language. However, care must be taken with the requirements for a good coverage model. For example, one can implement an *interrupted* task for *wait*, reported when a *wait* call terminates by *interrupt*. But this is not a good idea, since in many applications, interruption is never used, and so this task cannot be covered (the second requirement is violated). Even if *interrupt* is used, it may have as possible targets only a subset of the wait calls in the application.

5. PILOT REPORT

In this section, we give a few examples of how specific projects benefited from using synchronization coverage. Subsection 5.1 describes how a unit test suite was developed to quickly obtain 100% synchronization coverage. In 5.2 we show how synchronization coverage was used to guide review and in 5.3, we show how synchronization coverage was used to evaluate risks in a function test.

5.1 Unit testing with synchronization coverage

Various business applications require the execution of some business logic portion of code on a periodical base. An example of such a business application is the Video On Demand (VOD) application. It contains content management, media streaming and client portions. On the client side, a multimedia processing framework performs decryption, parsing, decoding and rendering of the media. Each of the mentioned operations may be designed and implemented as independent threads. Major synchronization protocols controlling the thread execution may be gathered under the Thread Manager abstraction.

For instance a video renderer is responsible for pulling decoded frames from its input queue, performing color conversion and rendering them into the clients rendering device. If the frame rate of the incoming video stream is 20 frames per second, the video renderer has to execute the *runBusinessCode()* method at least every 50 milliseconds. It is the responsibility of the Thread Manager controlling the video renderer to start the rendering process when the decoded media starts to arrive, to pause it and to stop it according to VOD application user requests.

We performed unit testing with synchronization coverage of the Thread Manager classes in a middle sized multimedia-related project with 160 classes. We also applied the Interleaving Review Technique [10] and concurrent bug pattern based review [9] to the thread manger protocol identifying several bugs. One type of bugs was related to the memory model and to the wrong/no-lock bug pattern [9], others were related to not handling interrupts appropriately. The Thread Manager served as a base class for various modules of the project. The Thread Manager was written from scratch, and the thread that performed the *runBusinessCode()* logic was implemented as an inner class.

At the unit test level, we wrapped an abstraction of the test manager protocol and tested it. The Thread Manager was easily abstracted from business code. Then, we implemented a tester class that created several concurrent threads executing different scenarios of starting, pausing, and stopping threads. Finally, we applied ConTest to obtain 100% synchronization coverage. This was easily obtained after a short period of time – less than an hour. The same experience is confirmed in other examples; when a few classes are used to implement the synchronization protocol, as should be the case in correct concurrent programming, 100% synchronization coverage is easily obtainable at unit test using the abstraction method just described. We did not find additional bugs but this is expected due to the previous review stages. The abstracted unit test ensures high quality and is then used in regression testing.

For example, in the following Thread Manager inner class code segment the real call to *runBuissnessCode*() was substituted with a method that only printed a message "run business code".

```
while (!needToStop) {
    runBuisnessCode();
    //code abstracted to a print statement
    synchronized (synchronizationObject) {
        if (pause  == true) {
           pause = false;
```

There are other ways in which abstraction can be obtained. For example, sometimes business code if statements can be substituted by random decisions. At other times, a return code from some function call can be simply hard coded, etc. Of course, this abstraction should be done with care by someone who understands the protocol but that is why this technique is applied in the unit test in the first place!

5.2 Guiding review with synchronization coverage

The project in which this was done is a middle size project with 575 classes. The goal of this application is to identify complex events as they occur, in order to give timely warnings. Synchronization became a big issue when the application was enhanced to support distributed events, for example identifying when many resources are getting close to their limit, when each is on a different CPU. After the new capabilities were added, class coverage was measured on the regression suite. Tests were created to cover the classes not initially covered in the regression. After this first stage, synchronization coverage was measured and evaluated.

First, we looked to see how many classes had synchronization primitives and how many there were in each. Of the 575 classes in the project 16 had synchronization primitives. This is reasonable and shows that a common design problem, that of having too many classes with synchronization, is not present. However, nine classes had three or fewer primitives. First, we checked if we could reduce the number of files with synchronization primitives. The advantage of having fewer files with synchronization primitives is having fewer files to review. It also improves maintainability at a later stage.

In an initial review, we found that two of the four files with a single synchronization statement can be ignored as they belong to a different code base. In one, the synchronization statement was not needed and the last is still under review. We found a few additional files from which we can remove the synchronization statements which will make the design cleaner.

The synchronization coverage showed that 25% of the synchronization statements that were reached were also exercised. We started examining individual synchronization statements to see why they had the coverage results indicated in the coverage. After the first hour of review, we had examined a few statements with the following results:

- Two synchronized statements were in dead code so they could not be reached
- One was suspected as unnecessary and the decision delayed until a thorough review could be completed.
- One was found to be unnecessary
- One was in a place that was very hard to test as it was hard to create the specific scenario needed. This one will also be inspected further as it is not going to be tested
- Four statements had missing tests. This requires the design of new tests, one of the major reasons for measuring coverage to begin with.
- One of the synchronization statements was missing the application code required to enable it to be tested.
- After finding a problem with every uncovered synchronization statement we looked at, we decided to look at one of the covered ones. It had no problems.

There were a number of actionable items as a result of this review. We found areas that needed more tests, as there were some synchronizations that could not be activated. We found synchronization statements that needed to be removed, code that needed to be written, and design that needed to change. We also found a few bugs, and locations where further reviews were needed. In this review, we learned that after some testing was done, every synchronization point that was not covered was indicative of some problem. As the measuring phase is very fast and inexpensive, the benefits of reviewing the synchronization coverage was very clear, and it will be continued in the future. The cost effectiveness is very impressive.

While it is probably the case that it is cost effective to review every synchronization coverage, regardless of coverage information, it is also the case that uncovered synchronization statements supply a specifically attractive target.

5.3 Evaluating the risk of multi-threaded testing using synchronization measures

We were asked to evaluate the cost of testing customer code (prior to engagement) with respect to synchronization coverage. The code was not particularly big, being about 6000 classes, but it had 482 classes with synchronization, out of which 293 had a single synchronization element in them. The code had 1080 synchronization statements and 140 *wait* statements. From this we learned that reviewing will be very expensive as 482 classes need to be reviewed. The design does not contain encapsulation of the synchronization protocols to a few classes, which is indicative of a bad design that could have many synchronization issues. Our recommendation is to do a redesign, reducing the number of classes containing synchronization elements, and if this is not feasible, to put a lot of resources into testing multi-threaded issues. These recommendations, with the rational behind them, were used by our service branch while bidding for this specific work.

It would have been better if, as part of the evaluation, we could run the regression and measure synchronization coverage. Then, we would have been able to give more specific reports, naming the components with the highest risk, but this could not be done in this assessment. One of the capabilities of FoCus [12], our coverage measurement tool, is to help in simple data mining and finding the packages or key words (for example if the class name contains "exception" in it) that contain many uncovered tasks, or a higher percentage of uncovered tasks. Those are risk areas as they have not been properly tested.

6. PREVIOUS FAILED ATTEMPTS

We have been trying to create a good synchronization coverage for some time. It may be instructive to explain what our previous models did, and why we think they have not done well. Both of the models which we tried are implemented in ConTest and are used from time to time. The first model is *concurrent pair of events*. Each task of this coverage model is composed of a pair of program locations that were encountered consecutively in a run, and a third field which is *true* or *false*. It is *false* if the two locations were run by the same thread, and *true* otherwise - that is, *true* means there was a context switch there. There are two problems with this model, the first is that it is too *hard*. There are too many tasks, covering all of them is very difficult and not very important. The second is that the analysis of which tasks are coverable and which are not is not automatic and therefore it is not possible to know when a 100 percent is reached. The concurrent pair of events model has been used to test progress in testing, mainly to verify that we get interleavings we didn't get before - i.e. that we have context switches in new places. However due to its deficiencies it is very rarely used.

The second, semi-successful attempt, was the *shared variables* coverage model. A variable is covered if an instance of this variable has been accessed by two different threads. Knowing which variables are covered and which are not is very useful for the system designer in assessing the quality of testing and has been used for this purpose. However, like the previous model we do not know which tasks (variables) can be covered (accessed by two threads) and which can not which limited its popularity as a coverage model.

7. CONCLUSION

Coverage is a useful testing technology; however, some coverage models seem to be more useful than others. We started by analyzing the requirements of a coverage model that make it acceptable to the industry. The set of requirements that we suggest is based on our significant experience with coverage and is very different from previous comparisons of coverage models based on their strength or bug finding power. We believe that for a coverage model to gain acceptance, it has to fulfill all four specified requirements. We have been working in the field of testing multi-threaded applications for quite some time and have felt the lack of such a coverage model, which can be used to evaluate tests for concurrent completeness and to find new testing requirements.

When we came up with the idea of synchronization coverage, we thought that this will succeed where previous coverage models for concurrency have failed, which is in getting user acceptance. Synchronization coverage is simple to understand, every task in it is coverable, and if one is not covered, this generates an action item for the tester. We have managed to convince developers to reach 100% synchronization coverage as part of the unit testing methodologoy that we teach. So indeed, once we found a good model, people started using it.

We teach developers and testers how to develop, review, and test concurrent applications. Synchronization coverage plays a key role in a number of these activities. Synchronization coverage is new and is currently known only within the communities in which we are active. We believe that once synchronization coverage becomes better known, it could become one of the more commonly used coverage models.

Synchronization coverage definition depends on the specific synchronization coverage primitives used, and to some extent, on the bug patterns common for the user. Our tools are currently used in the Java and C/Pthread environments. As we branch to new environments, and as language synchronization primitives evolve (such as in Java 5.0) the exact implementation of synchronization coverage will evolve. The basic idea though, that of testing that the synchronization primitives did "interesting" things, will remain the same.

8. REFERENCES

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

[2] Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.

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

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

[5] H. Chockler, O. Kupferman, and M.Y. Vardi. Coverage metrics for formal verification. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 2003.

[6] S. Cornett. Software test coverage analysis,. http://www.bullseye.com/webCoverage.html.

[7] O. Edelstein, E. Farchi, Y. Nir G Ratzaby, and S Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(3):111–125, 2002.

[8] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka. Testing concurrent programs: a formal evaluation of coverage criteria. In *Proceedings of the Seventh Israeli Conference on Computer Systems and Software Engineering*, pages 119 – 126, June 1996.

[9] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *PADTAD II in the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, April 2003.

[10] Amiram Hayardeny, Shachar Fienblit, and Eitan Farchi. Concurrent and distributed desk checking. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 16*, April 2004.

[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] Oded Lachish, Eitan Marcus, Shmuel Ur, and Avi Ziv. Hole analysis for functional coverage data. In *Proceedings of the 39th conference on Design automation*, pages 807–812. ACM Press, 2002.

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

[14] A. Mathur and W. Wong. Comparing the fault detection effectiveness of mutation and data flow testing: An empirical study, 1993.

[15] G. Ratzaby, S. Ur, and Y. Wolfsthal. Coverability analysis using symbolic model checking. Submitted to Charme 2001.

[16] R. Stewart. Unit test coverage as leading indicator of rework. In *proceedings of EuroSTAR 97*, NOvember 1997.

[17] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Commun. ACM*, 31(6):668–675, 1988.

[18] Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. All-du-path coverage for parallel programs. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 153–162. ACM Press, 1998.