Using Combinatorial Benchmark Construction to Improve the Assessment of Concurrency Bug Detection Tools

Jeremy S. Bradbury[†], Itai Segall[*], Eitan Farchi[*], Kevin Jalbert[†], David Kelk[†]
[†]University of Ontario Institute of Technology, Oshawa, ON, Canada
[*]IBM Haifa Research Laboratory, Haifa, Israel
jeremy.bradbury@uoit.ca, {itais, farchi}@il.ibm.com, {kevin.jalbert, david.kelk}@uoit.ca

ABSTRACT

Many different techniques for testing and analyzing concurrency programs have been proposed in the literature. Currently, it is difficult to assess the fitness of a particular concurrency bug detection method and to compare it to other bug detection methods due to a lack of unbiased data that is representative of the kinds of concurrency programs that are used in practice. To address this problem we propose a new benchmark of concurrent Java programs that is constructed using combinatorial test design. In this paper we present our combinatorial model for creating a benchmark, we propose a new concurrency benchmark and we discuses the relationship between our new benchmarks and existing benchmarks. Specific combinations of the model parameters define different interleaving spaces, thus differentiating between different test tools.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.2.8 [**Software Engineering**]: Metrics

General Terms

Measurement, Verification

Keywords

Benchmark, combinatorial test design, concurrency testing, model checking, static analysis

1. INTRODUCTION

The problem of assessing the fitness of a particular bug detection strategy or tool and comparing it with other strategies and tools is not new and is not isolated to concurrency bug detection. A large part of the solution to this problem is the use of empirical methods. However, without data that is unbiased and data that is representative of the kinds of programs that are used in practice, the ability to generalize empirical results and minimize possible threats to validity is limited. A solution to this problem is the creation of benchmarks.

In this paper, we propose a new benchmark for concurrency bug detection tools. A benchmark can be defined as the composition of three parts [38]:

1. *Creation of a motivating comparison.* The comparison should clearly outline the purpose of the benchmark and motivate its usage. Without a motivating comparison, the purpose of the benchmark may not be clear and the benchmark may be mis-used, thus results obtained from using the benchmark may not accurately reflect the true fitness of the tools under evaluation.

2. *Development of a task sample.* The task sample (i.e., benchmark data) should be representative of the possible tasks that the technique or tool may encounter during actual usage. The task sample in our case will be a set of concurrency programs. These programs should be representative of the different kinds of concurrent software that is being produced in practice and should contain bugs that are representative of the kinds of bugs that have been observed in practice.

3. *Identification or development of performance measures.* The inclusion of performance measures ensure that we can assess the fitness of a technique or tool when used with the task sample. The performance measures are used to define the comparison of different tools when applied to the task samples provided by the benchmark. An agreed upon set of measures ensures that different researchers who use the benchmark will be able to compare results across studies and build a body of literature in the area of concurrency bug detection.

In order to be a true benchmark all three of the above parts must be included – the omission of one of the parts results in a *proto-benchmark*. We strive to satisfy the entire benchmark definition in order to provide a means by which the fitness of existing and new concurrency bug detection tools can be assessed.

The main challenge in our new benchmark is the selection and creation of a representative set of concurrency programs (i.e., task samples). We have identified a set of seven attributes that should be considered when selecting a concurrency program for inclusion in the benchmark. Attributes in our set include program size, number of threads, bug type and more. We quickly realized that it would be infeasible to develop a benchmark consisting of all combinations of these attributes. As an alternative we propose a novel method for construction of benchmarks, that applies combinatorial test design (CTD) – a well known test planning technique [15]. CTD is typically applied for planning tests. In CTD, the test space is modelled as a set of parameters, their respective values, and restrictions on the value combinations. A subset of the space is then automatically constructed such that it covers all valid value combinations (a.k.a interactions) of every t parameters, where t is a user input. The most common instance of CTD is known as pairwise testing, in which the interaction of every pair of parameters must be covered. We propose to apply CTD to the construction of a benchmark for concurrency bug detection tools.

In the next section we will provide an overview of bug detection techniques including concurrency testing, static analysis and software model checking. We also provide background on CTD. In Section 3 we present our proposed CTD benchmark model starting with the parameters that we believe should drive the task sample generation in our benchmark. We demonstrate why we believe CTD to be an appropriate method for planning of the benchmark by exploring how the interactions of parameters differentiate concurrency bug detection tools. Next we present a pairwise selection of model combinations. The pairwise selection of model combinations characterizes each task sample (i.e. concurrent program) in the benchmark by assigning a value to each of the parameters, for example the number of threads. Following the pairwise selection, we describe how existing benchmarks and other sources of concurrent programs may be leveraged in order to simplify the generation of the benchmark. We further discuss some of the obstacles that this approach might face, as well as possible solutions for them. Finally, in Section 4 we summarize our benchmarking approach to concurrency bug detection and we discuss future steps necessary to complete our benchmarking research.

2. BACKGROUND

2.1 Concurrency Bug Detection Techniques

The goal of our research is to develop a benchmark that is appropriate for comparing a variety of concurrency bug detection techniques. In particular, we have focused on a benchmark that is capable of differentiating between different concurrency testing tools, static analysis tools and model checking tools.

2.1.1 Concurrency Testing

Concurrency Testing is a class of testing techniques specifically targeted at exposing faults in multi-threaded source code. Due to the non-determinism of the execution of concurrent source code and the high number of possible interleavings, concurrency testing can not rely on source code coverage metrics alone to guarantee that code is correct. In addition to ensuring that all code is covered we must also provide some probabilistic confidence that bugs that manifest themselves in only a few of the interleavings are found. This highlights the primary difference between sequential testing and concurrency testing – concurrency testing provides coverage of the interleaving space in addition to coverage of the source code. Various testing techniques exist to detect concurrency bugs:

- *Coverage-Based Testing with Manual Interleaving Exploration:* the simplest form of concurrency testing, this approach uses coverage techniques from sequential testing with a manual approach to covering the interleaving space. On the one hand the manual approach can be passive such as using different operating systems, different versions of language interpreters and different hardware configurations to potentially cause different interleavings to occur [14]. On the other hand, the manual approach can be active and involve hand-instrumentation of delays in the source code to affect thread scheduling. [14].
- *Testing with Noise makers:* an alternative to hand-instrumentation is to use a noise making tool like IBM's Concurrent Testing Tool [19] that provides randomized scheduling by automatically and systematically inserting random delays into Java bytecode. Tools like IBM's Concurrent Testing Tool also provide heuristics to increase the confidence that interleavings with a high-risk of bugs are explored.

The use of heuristics and different scheduling strategies in concurrency testing tools provide a variety of tools that need benchmarks in order to understand trade-offs.

2.1.2 Static Analysis

Static analysis bug detection tools identify potential bugs without executing the program. Static analysis is an attractive alternative to concurrency testing, which requires many executions of each test case, however it is susceptible to spurious result. Examples of static analysis tools include FindBugs [25], JLint [4, 10], Chord [3, 8, 34], JSure [5], JTest [7] and RSAR [31]. All of these tools aim to detect specific kinds of concurrency bugs or patterns of code that are associated with concurrency bugs. For example, Chord is a newer static analysis tool designed to detect concurrency bugs including data races [33] and deadlocks [34]. Chord combines 4 different types of static analysis to detect bugs: call-graph analysis, alias analysis, thread-escape analysis and lock analysis.

2.1.3 Software Model Checking

Software model checking is a formal methods approach that typically involves developing a finite state model of a software system and specifying properties (e.g., assertions) that the software system should satisfy. Traditional use of model checkers involves proofs of correctness via an exhaustive state space search. An exhaustive search explores *all* possible thread interleavings of the concurrent software model. The trade-off in using an exhaustive search is that it usually requires a long time to complete.

Software model checking advances in the last decade have lead to another important use of model checking tools as debuggers. Software model checkers including Java PathFinder (JPF) [23, 40], developed at NASA, now work directly on the source code, allow for simulation (i.e., path executions) and can be integrated into popular IDEs. Other software model checkers, including Bogor [35], also provide enhanced customization capabilities which allow them to be customized to software in a particular domain.

2.2 Combinatorial Test Design (CTD)

Combinatorial Test Design (CTD), also known as *Combinatorial Testing*, is a well-known test planning technique [15]. In CTD, the test space is modelled by a set of parameters, their respective values, and restrictions on the value combinations. A test in this setting is an assignment of exactly one value to each parameter, such that the combination is valid according to the restrictions. A subset of the test space is then automatically constructed such that it covers all valid value combinations (a.k.a. interactions) of every t parameters, where t is a user input. In other words – for every set of t parameters, all valid combinations of t values for them will appear in at least one test in the test plan. The most common application of CTD is known as pairwise testing, in which the interaction of every pair of parameters must be covered (i.e., $t = 2$).

Consider a test scenario in which a file is opened then read. The file could be of two types, a directory or a regular file. The open operation can open the file for reading, writing or both. When reading, the file can be accessed sequentially or randomly. The result of the operation is that the file was either read successfully or not. This results in a CTD model. There are four parameters, namely, the open type, the file type, the read type and the scenario result. The possible values for each parameter are as follows the *openType* $= \{read, write, readAndWrite\}$, the $fileType = \{directory, regular\}$, the way the file is read *readType* $= \{sequential, random\}$ and the *readResult* $= \{success, fail\}$. Thus, a priori, there are $3*2*2*2 = 24$ potential test scenarios but some of them are impossible. For example, the read operation will fail only if the file was opened for a write, thus, the following combination is not possible (*write*, *regular*, *sequential*, *success*). Appropriate logical restrictions are defined to exclude the impossible combinations. Next, we may require that a subset of the possible tests is chosen so that pair-coverage is obtained (i.e., each combination of two values of different parameters will have at least one test in which it appears). Such a combination is $(read, directory)$ another is $(read, sequential)$ and another is $(read, success)$. They can be satisfied by the test (*read*, *directory*, *sequential*, *success*). A CTD algorithm will choose a subset of the possible tests that will meet the pair coverage requirement. Our example is small, but typical test spaces are huge and manual selection of the tests is not practical.

The reasoning behind CTD as a functional test planning technique is the observation that in most cases the appearance of a bug depends on the combination of a small number of parameter values of the system under test. Experiments show that a test set that covers all possible pairs of parameter values can typically detect 50% to 75% of the bugs in a program [39, 16]. Other experimental work has shown that typically 100% of bugs can be revealed by covering the interaction of between 4 and 6 parameters [27].

In this paper we argue that CTD, though typically applied for test planning, may also be a suitable technique for systematic benchmark construction.

3. TOWARDS A CONCURRENCY BENCHMARK

3.1 The Combinatorial Model

In this section we review the parameters and values used to characterize programs in the benchmark. The parameters and values listed below are intended as a starting point for discussion within the concurrency testing community and we expect that these may change as more input from the community is provided.

3.1.1 Program Size – Number of Statements

Some concurrency testing approaches are highly sensitive to the size of the program. This parameter measures program size in its simplest form – the number of statements. Size is subdivided into three categories:

- `small`: $<$ 10k statements
- `medium`: 10-100k statements
- `large`: $>$ 100k statements

There is no upper bound on the `large` category. Programs millions of statements long are possible. Initially, the majority of tools using this benchmark are not expected to work at this scale. We hope the inclusion of large programs encourages practicioners to investigate scaling to this domain.

3.1.2 Program Size – Number of Critical Regions

Another aspect of program size, important for concurrency testing, is the number of critical regions in a program. Some approaches may be agnostic to the number of statements in the program, yet suffer from very poor performance when the number of critical regions grows. Proposed categories are:

- `small`: $<$ 5 critical regions
- `medium`: 5-20 critical regions
- `large`: $>$ 20 critical regions

We are interested in assessing testing tools targeting business applications, middleware and operating systems and our experience with these programs was the basis for the above categories. Other kinds of programs may lead to different categories and attribute values. For example, scientific applications can often have thousands of critical regions.

3.1.3 Program Size – Percentage of Statements in Critical Regions

Related to the previous, this category considers the number of statements in critical regions. Proposed categories are:

- `small`: $<$ 5%
- `medium`: 5-15%
- `large`: $>$ 15%

The range of values selected for these categories emphasizes programs with low synchronization. This is common in professional business applications where performance optimizations place only the minimum number of statements within critical regions.

3.1.4 Number of Threads

Following the suggestion by Dwyer et al. in [18], concurrent programs are classified by the number of threads used:

- `small`: < 5
- `medium`: 5-10
- `large`: 10-100
- `very large`: > 100

3.1.5 Path Error Density

The path error density is defined as "the probability of a thread schedule exhibiting an error" [18]. For some testing tools, the more thread schedules exhibiting an error the easier it is for them to detect it. Path error density may be estimated by executing a significantly large set of random thread schedules (e.g., 5000) then determining the percentage exhibiting an error. Proposed categories are:

- `very low`: $< 2\%$
- `low`: 2-25%
- `medium`: 25%-75%
- `high`: $> 75\%$

A `very low` category is included to accomodate testing tools designed to detect heisenbugs[1].

3.1.6 Bug Depth

Bug depth is defined as the minimum depth along a path that a bug can be exhibited. Bug depth is measured in number of context switches between the start of execution and the bug occurring. Proposed categories are:

- `low`: < 25 context switches
- `medium`: 25-50 context switches
- `high`: > 50 context switches

3.1.7 Bug Patterns

Bug patterns refer to the kind of bug exhibited within a program:

- **Nonatomic operations assumed to be atomic:** *"...an operation that 'looks' like one operation in one programming model (e.g., the source code level), but actually consists of several unprotected operations at the lower abstraction levels" [22].* Nonatomic floating point operations are also included in this pattern.
- **Two-state access:** *"Sometimes a sequence of operations needs to be protected but the programmer wrongly assumes that separately protecting each operation is enough" [22].*
- **Wrong lock or no lock:** *"A code segment is protected by a lock but other threads do not obtain the same lock instance when executing. Either these other threads do not obtain a lock at all or they obtain some lock other than the one used by the code segment" [22].*
- **Double-checked lock:** *"When an object is initialized, the thread local copy of the objects field is initialized, but not all object fields are necessarily written to the heap. This might cause the object to be partially initialized while its reference is not null " [22].*
- **sleep:** *"The programmer assumes a child thread should be faster than the parent thread in order that its results be available to the parent thread when it decides to advance. Therefore, the programmer sometimes adds an 'appropriate' `sleep()` to the parent thread. However, the parent thread may still be quicker in some environment. The correct solution would be for the parent thread to use the `join()` method to explicitly wait for the child thread" [22].*
- **Losing a notify:** *"If a `notify()` is executed before its corresponding `wait()`, the `notify()` has no effect and is 'lost' ... the programmer implicitly assumes that the `wait()` operation will occur before any of the corresponding `notify()` operations" [22].*
- **A "blocking" critical section:** *"A thread is assumed to eventually return control but it never does" [22].*
- **The orphaned thread:** *"If the master thread terminates abnormally, the remaining threads may continue to run, awaiting more input to the queue and causing the system to hang" [22].*
- **Notify instead of notify all:** *If a `notify()` is executed instead of `notifyAll()` then threads with some of its corresponding `wait()` calls will not be notified [28].*
- **Interference:** A pattern in which *"...two or more concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the access from being simultaneous."* [29]. The interference bug pattern can also be generalized from classic data race interference to include high level data races which deal *"...with accesses to sets of fields which are related and should be accessed atomically"* [11].
- **Deadlock (deadly embrace):** *"...a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something in a deadlock cycle ... For example, this occurs when a thread holds a lock that another thread desires and vice-versa"* [29].

[1] Concurrency bugs that appear only in one or a few interleavings of a program [32].

3.2 Differentiating Bug Detection Tools Based on Parameter Values

As stated in Section 2.2, the reasoning behind CTD as a test planning technique is that most functional bugs in software systems are a result of an interaction between a small number of parameter values. We argue here that CTD may also be an effective method for systematic construction of benchmarks, specifically for concurrent bug detection tools.

A benchmark is effective if it is capable of differentiating between tools. That is – for a pair of tools, we would like to have at least one task sample in which one tool performs better than the other. In most cases one tool will not be strictly superior to the other, but rather have several benefits and drawbacks. In such cases, we would also like to have a task sample in which the second tool is shown to be superior to the first.

In this section, we demonstrate how interactions between a small number of parameters in our model differentiate between concurrent bug detection approaches. This will support the claim that CTD is an effective method for constructing the set of concurrency programs that will constitute the task samples in our benchmark.

Model checking tools may exhaustively search the possible space of interleavings of a concurrent program. Different interleavings result from different scheduling decisions. On the other hand, other tools may introduce noise that changes scheduling decisions of the program under test. In such cases, some heuristics are applied to direct the manner in which the scheduler decisions are impacted. These heuristics may take into account known concurrent bugs [26] or bug patterns such as the ones mentioned in the previous section. Alternatively, they may focus on general heuristics such as the increase of lock contention. Static analysis tools will not execute the program but typically may apply fixed point analysis that mimic the concurrent execution to some extent. They may also look for simple program patterns typically associated with bad programming practices or known bugs (e.g., a Java $wait()$ that is not within a loop).

For a given concurrent program, consider the space of possible program interleavings. This space is exponential in the program length and its size is monotonically influenced by at least three of the model parameters, namely, number of program statements, number of critical sections, and number of threads. Testing tools that exhaustively search the interleaving space, or some abstraction of the interleaving space, may encounter an exponential explosion due to some combinations of the above three parameters and their performance will be impacted. In contrast, noise making tools and static analysis tools may still function under these conditions. Therefore it is important to have programs in our benchmark that vary the number of program statements, number of critical sections and the number of threads to cover their possible combinations so that we can explore the tradeoffs between the different kinds of concurrent bug detection tools.

It is well known that typical concurrent bugs materialize due to the occurrence of a small set of critical events, access to shared variables or usage of synchronization primitives, in some predefined order. For example, in the case of a lost notify bug pattern, the $notify()$ critical event should occur before the corresponding $wait()$ event – thus only two events occurring in a certain order are required for the bug to materialize. Even in this simple bug pattern it may be difficult to exhibit this bug through testing because the occurrence of the bug is impacted by a number of other factors. One such factor is the length of the interleaving that leads the software under test to the state in which the required events are about to occur (referred to as the bug depth). In the case of a notify bug, a long interleaving might be required before two threads are about to execute the $notify()$ and its corresponding $wait()$. Another factor is a masking factor – two reads to a shared variable might need to execute one after the other in two threads in order to exhibit the bug while requiring that no write to the shared variable will occur in between the two reads. Including parameters in our CTD model such as bug depth, path error density, and covering their possible combinations will help us detect bugs such as the example notify bug.

Two other CTD model parameters, the number of threads and number of critical sections, can also create more masking effects with respect to concurrent bugs. As the percentage of critical section code and path error density decreases the bug depth will typical increase. Noise making tools that apply different search heuristics may attempt to create a certain order of events based on a bug pattern but will be less or more effective based on how they handle the bug depth and masking effects. Since there are a variety of heuristic techniques that can be used in noise making tools we feel the inclusion of number of threads and number of critical regions as parameters are important for differentiating these heuristics.

Although many static analysis tools use a pattern-based approach to bug detection others may still apply a fixed point analysis that mimics the interleaving space execution and as such may be sensitive to whether or not the interleaving space is huge. In such cases parameters that influence the size of the interleaving space will impact their performance.

3.3 Pairwise Benchmark Construction

We used IBM's CTD tool [2, 37] to construct a pairwise plan for the model given in Section 3.1. The resulting plan is given in Table 1. The table consists of 44 combinations, each representing a program in our proposed benchmark. Since CTD was applied, with $t = 2$, every combination of two values for the parameters listed in Section 3.1 appears in at least one row in this table.

In Section 3.2 we discussed how the parameter values in our CTD model can be used to differentiate between bug detection tools. We also need to ensure that the pairwise plan for identifying the 44 concurrency programs in our benchmark contains programs with combinations of parameters that can differentiate different concurrency bug detection tools.

A general comment regarding bug detection tools that target specific kinds of bugs is that the 44 programs in Table 1 include examples of *all* of the bug patterns listed in Section 3.1.7. Therefore, our proposed benchmark is capable of differentiating tools that excel at detecting certain kinds of

Table 1: Pairwise plan for our combinatorial model

	Program Size – # Statements	**Program Size – # Critical Regions**	**Program Size – % Statements in Critical Regions**	**# Threads**	**Path Error Density**	**Bug Depth**	**Bug Pattern**
1	Small	Small	Large	Small	Medium	High	TwoStageAccess
2	Medium	Small	Medium	Large	VeryLow	Medium	NonAtomicAssumedAtomic
3	Large	Medium	Small	Small	Low	Medium	BlockingCriticalSection
4	Medium	Large	Large	Medium	Low	Low	Interference
5	Small	Medium	Medium	VeryLarge	High	Low	OrphanedThread
6	Large	Large	Small	Large	High	High	NoLock
7	Medium	Large	Small	VeryLarge	Medium	Medium	LostNotify
8	Small	Medium	Small	Medium	VeryLow	High	NotifyInsteadOfNotifyAll
9	Large	Small	Medium	Medium	Medium	Low	SleepInsteadOfJoin
10	Large	Large	Large	VeryLarge	VeryLow	High	Deadlock
11	Medium	Small	Large	Medium	High	Medium	DoubleCheckedLocking
12	Small	Large	Medium	Small	VeryLow	Low	DoubleCheckedLocking
13	Small	Medium	Large	Large	Low	Medium	SleepInsteadOfJoin
14	Medium	Medium	Small	Large	Medium	Low	Deadlock
15	Large	Small	Medium	VeryLarge	Low	Medium	NotifyInsteadOfNotifyAll
16	Large	Small	Medium	Small	High	High	LostNotify
17	Medium	Small	Small	Small	Low	High	OrphanedThread
18	Small	Large	Medium	VeryLarge	VeryLow	Low	BlockingCriticalSection
19	Small	Medium	Large	VeryLarge	Low	Low	NonAtomicAssumedAtomic
20	Large	Medium	Medium	Small	VeryLow	Medium	Interference
21	Large	Large	Small	Small	Medium	High	NonAtomicAssumedAtomic
22	Small	Medium	Large	Large	VeryLow	Low	LostNotify
23	Medium	Small	Large	Medium	Medium	Medium	NoLock
24	Small	Medium	Medium	VeryLarge	Low	Low	NoLock
25	Small	Small	Medium	Small	Low	Medium	Deadlock
26	Small	Small	Small	Large	High	High	Interference
27	Large	Large	Small	Medium	High	Medium	TwoStageAccess
28	Medium	Medium	Medium	Large	VeryLow	Low	TwoStageAccess
29	Large	Large	Large	Large	Medium	Medium	OrphanedThread
30	Large	Medium	Small	Large	Medium	High	DoubleCheckedLocking
31	Medium	Large	Large	Small	High	Low	NotifyInsteadOfNotifyAll
32	Medium	Small	Large	Large	Medium	High	BlockingCriticalSection
33	Medium	Large	Small	VeryLarge	VeryLow	High	SleepInsteadOfJoin
34	Small	Large	Small	VeryLarge	Low	High	DoubleCheckedLocking
35	Small	Large	Small	Large	Medium	Low	NotifyInsteadOfNotifyAll
36	Small	Medium	Large	Small	High	Medium	SleepInsteadOfJoin
37	Medium	Medium	Medium	Small	VeryLow	High	NoLock
38	Small	Large	Small	VeryLarge	Low	Low	TwoStageAccess
39	Small	Large	Large	VeryLarge	Medium	High	Interference
40	Large	Medium	Large	Medium	VeryLow	High	OrphanedThread
41	Medium	Medium	Large	Medium	High	High	Deadlock
42	Small	Medium	Small	Medium	High	High	NonAtomicAssumedAtomic
43	Small	Small	Large	Medium	Low	Medium	LostNotify
44	Small	Medium	Medium	Medium	High	High	BlockingCriticalSection

bugs when compared to tools that excel at detecting other kinds of concurrency bugs.

When comparing model checking tools using exhaustive search versus noise maker tools that use heuristics or static analysis tools we can differentiate these tools by exposing the limitations of exhaustive techniques using programs that have a large number of statements, number of critical regions or number of threads. In our benchmark 5 programs (6, 10, 21, 27, 29) all have a large program size (number of statements and number of critical regions). Our benchmark also contains 11 programs with a large number of threads (2, 6, 13, 14, 22, 26, 28, 29, 30, 32, 35) and 11 programs have a very large number of threads (5, 7, 10, 15, 18, 19, 24, 33, 34, 38, 39). Furthermore, our proposed concurrency benchmark contains 3 programs that are both large in size (number of statements and number of critical regions) and have a large or very large number of threads.

If we reflect back on our notify bug example in Section 3.2 we can observe that this bug can be caused by the LostNotify bug pattern and that the bug depth and path error density can affect the ability to detect the bug. Our proposed benchmark contains several programs with LostNotify bugs that can help to assess bug detection tools capable of finding this bug in different situations:

- Program 7: medium path error density and bug depth
- Program 16: high path error density and bug depth
- Program 22: very low path error density and low bug depth
- Program 43: low path error density and medium bug depth

In summary, we can see that the pairwise benchmark construction based on the parameters in our CTD model preserves the ability of the underlying parameters to differentiate between different concurrency bug detection tools.

3.4 Acquiring Benchmark Example Programs

Currently, we have proposed how to construct a concurrency benchmark using combinatorial test design and have produced a table describing a pairwise plan consisting of 44 programs (see Table 1). To complete the construction of the benchmark we need to select an existing program, or generate a new program, that matches the specification of each program in the table. For example, we need an actual program that satisfies the criteria of *Program 1* in the table. That is, a program that has a TwoStageAccess bug, a small number of statements, a small number of critical regions, a large percentage of statements in critical regions, a small number of threads, a medium path error density and a high bug depth.

The combinatorial model described above spans a space of over 14,000 different programs, out of which the CTD process chose 44. The chances for finding programs that exactly match the required combinations are therefore very slim – for each of the 44 combinations, one has to find a program that exactly matches all values for the seven attributes. A considerable amount of program analysis is required only to determine if a given program meets the proposed criteria. Therefore, our approach to generating the programs involves two steps:

1. leveraging programs from existing benchmarks and open source repositories.
2. modifying existing programs using program mutation.

We will now discuss each of these two steps in more detail.

3.4.1 Leveraging Existing Programs

The majority of benchmarks used by researchers to assess concurrency testing tools are actually proto-benchmarks. In general these benchmarks provide a task sample, a set of concurrency programs, but do not include a motivating comparison or performance measures.

The most widely used benchmark in the concurrency testing community is the IBM Benchmark [24, 21, 20], which is publicly available from researchers at IBM's Haifa Research Laboratory. Although this benchmark was primarily developed by researchers at the Haifa Research Laboratory, it has benefited from community consultation and feedback as well as the inclusion of task programs from other sources such as NASA. The public IBM Benchmark contain 28 programs and is representative in terms of the bug patterns found in the programs (see Table 2). Although this benchmark is a good first step towards a comprehensive concurrency benchmark it is missing several key elements:

1. an explanation of the purpose and usage of the benchmark
2. additional programs that would make it representative of concurrency programs that vary in path error density, size, etc.
3. a set of metrics to support the comparison of tools that are applied to the benchmark

We recognize the contributions of the IBM Benchmark and plan to use it as a resource for a number of the smaller programs included in our new benchmark.

Rungta and Mercer propose another benchmark of multithreaded programs for assessing model checking search algorithms [36]. This line of research is closely related to benchmarks for concurrency bug detection tools – in fact, the Rungta and Mercer benchmark includes a number of programs from the IBM Benchmark. We plan to also utilize the Rungta and Mercer benchmark as a source for programs in our new benchmark.

Finally, BugBench [30] is a general defect detection benchmark of 17 programs with real bugs. Four of the programs in the BugBench benchmark have a concurrency bug. An HTTP server (HTTPD) contains a data race and three different versions of the database MYSQL contain a data race

Table 2: IBM Concurrency Benchmark programs [24, 21, 20]

	Program Size – (sloc, statements)	Program Size – # Critical Regions	Program Size – % Statements in Critical Regions	Bug Pattern
account	Small(139,77)	Medium(7)	Small(11.6883)	NoLock
airlinestickets	Small(61,34)	Small(0)	Small(0)	Interference
allocationvector	Small(163,83)	Small(3)	Large(22.8916)	TwoStageAccess
boundedbuffer	Small(328,192)	Medium(5)	Large(20.3125)	NotifyInsteadOfNotifyAll
bubblesort	Small(236,84)	Small(3)	Medium(11.9048)	NonAtomicAssumedAtomic, OrphanedThread
bubblesort2	Small(98,43)	Small(1)	Medium(6.97674)	Initialization-Sleep
bufwriter	Small(170,72)	Small(3)	Medium(11.1111)	NoLock
Cmgtt-examples	Small(317,138)	Small(1)	Small(2.17391)	Stack Overflow
critical	Small(54,25)	Small(0)	Small(0)	NonAtomicAssumedAtomic
dcl	Small(138,70)	Small(2)	Small(4.28571)	DoubleCheckedLocking
deadlock	Small(95,40)	Small(2)	Large(27.5000)	Deadlock
deadlockexception	Small(149,64)	Medium(5)	Large(17.1875)	BlockingCriticalSection, Deadlock
filewriter	Small(235,98)	Small(3)	Medium(11.2245)	NonAtomicAssumedAtomic
ftp_server	Medium(11981,-)	-	-	-
garagemanager	Small(398,182)	Medium(7)	Large(20.3297)	BlockingCriticalSection, Deadlock
linkedlist	Small(241,97)	Small(1)	Small(2.06185)	NonAtomicAssumedAtomic
liveness	Small(135,60)	Small(4)	Large(18.3333)	Dormancy
lottery	Small(150,66)	Small(3)	Small(4.54545)	NotAtomicAssumedAtomic, NoLock, BlockingCriticalSection
manager	Small(158,60)	Small(4)	Medium(13.3333)	NonAtomicAssumedAtomic
mergesort	Small(279,129)	Small(3)	Large(16.2790)	NonAtomicAssumedAtomic
mergesortbug	Small(146,62)	Small(0)	Small(0)	NonAtomicAssumedAtomic
pingpong	Small(126,50)	Small(0)	Small(0)	Null-Pointer Exception
piper	Small(116,63)	Small(4)	Large(23.8095)	Condition-For-Wait
producerconsumer	Small(223,112)	Small(0)	Small(0)	OrphanedThread
shop	Small(185,63)	Medium(5)	Large(20.6349)	Sleep, Weak-Reality
suns_account	Small(66,28)	Small(0)	Small(0)	NonAtomicAssumedAtomic
xtangoanimation	Small(1547,641)	Large(64)	Large(37.4415)	Deadlock

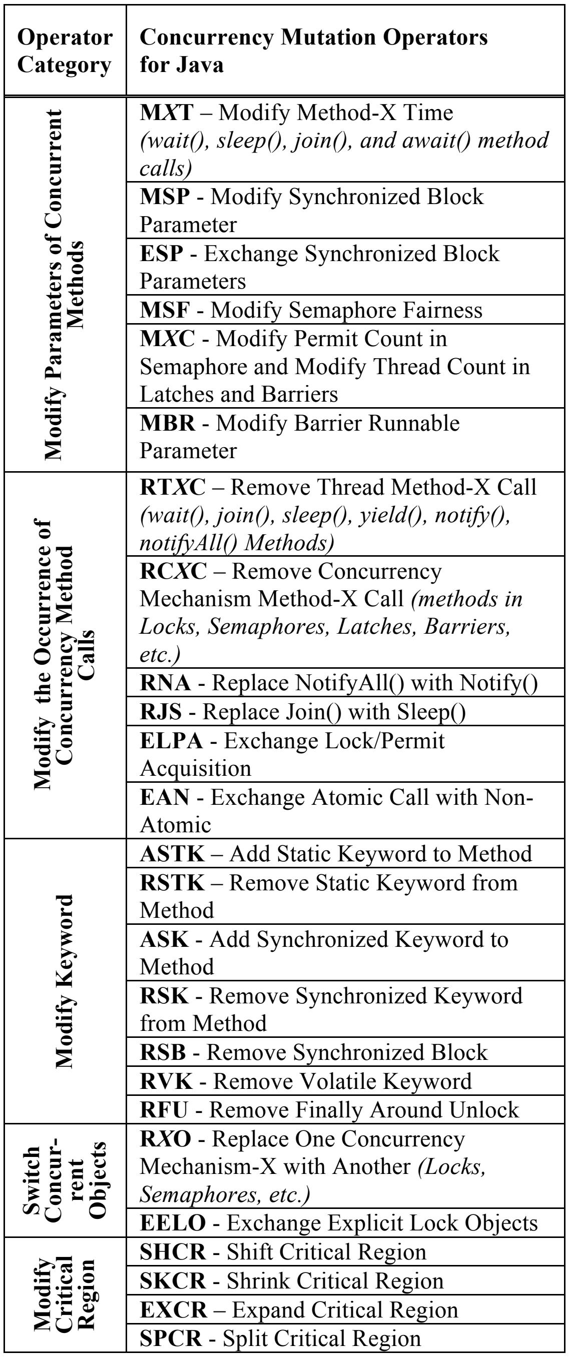
and two atomicity bugs. The concurrency examples found in BugBench are interesting because of their size and scale however we can not consider them for inclusion in our own benchmark because they are not written in Java.

In summary, we plan to leverage the programs already collected in existing concurrency benchmarks, like the IBM Benchmark, to satisfy as many of the program specifications as possible in our pairwise plan. The IBM Benchmark provides many programs that are small in size and may satisfy some of our needs. Unfortunately, there are no large programs in the benchmark that we can use in our work. We have identified a number of existing large open source projects (e.g., MOBAT [6]) available in online repositories that we plan to analyze for conformance to our program specifications. However, we do not expect to find programs that exactly and completely match the required combinations. Therefore, a mutation step is also required.

3.4.2 *Using Program Mutation to Modify Existing Programs*

As mentioned above, it is certain that we will not find existing programs that will exactly and completely match the program specifications in our concurrency benchmark. Several of the attributes considered in our benchmark (program size and number of threads) appear to occur with wide diversity in existing concurrency programs. Other attributes (path error density, bug depth and bug pattern) are not necessarily as easy to identify or find. We plan to use program mutation of existing programs to generate new programs with different path error densities, bug depths and bug pattern types. Program mutation (i.e., mutation analysis) has been previously used in the comparison of different testing techniques [9, 17] and in the comparison of different concurrency bug detection tools [13]. Mutation analysis uses a set of mutation operators in which each operator corresponds to a syntactic bug pattern. A mutation operator is applied to a program and generates a set of mutant programs – each mutant contains a single syntactic change from the original program.

Table 3: Concurrency mutation (ConMAn) operators for Java [12]



Operator Category	Concurrency Mutation Operators for Java
Modify Parameters of Concurrent Methods	**MXT** – Modify Method-X Time *(wait(), sleep(), join(), and await() method calls)*
	MSP - Modify Synchronized Block Parameter
	ESP - Exchange Synchronized Block Parameters
	MSF - Modify Semaphore Fairness
	MXC - Modify Permit Count in Semaphore and Modify Thread Count in Latches and Barriers
	MBR - Modify Barrier Runnable Parameter
Modify the Occurrence of Concurrency Method Calls	**RTXC** – Remove Thread Method-X Call *(wait(), join(), sleep(), yield(), notify(), notifyAll() Methods)*
	RCXC – Remove Concurrency Mechanism Method-X Call *(methods in Locks, Semaphores, Latches, Barriers, etc.)*
	RNA - Replace NotifyAll() with Notify()
	RJS - Replace Join() with Sleep()
	ELPA - Exchange Lock/Permit Acquisition
	EAN - Exchange Atomic Call with Non-Atomic
Modify Keyword	**ASTK** – Add Static Keyword to Method
	RSTK – Remove Static Keyword from Method
	ASK - Add Synchronized Keyword to Method
	RSK - Remove Synchronized Keyword from Method
	RSB - Remove Synchronized Block
	RVK - Remove Volatile Keyword
	RFU - Remove Finally Around Unlock
Switch Concurrent Objects	**RXO** - Replace One Concurrency Mechanism-X with Another *(Locks, Semaphores, etc.)*
	EELO - Exchange Explicit Lock Objects
Modify Critical Region	**SHCR** - Shift Critical Region
	SKCR - Shrink Critical Region
	EXCR – Expand Critical Region
	SPCR - Split Critical Region

To generate our new programs we plan to use the concurrency mutation tool ConMAn [1, 12] and apply it to the programs discussed in Section 3.4.1. The ConMAn mutation operators were designed to generate mutant programs that cover the set of bug patterns listed in Section 3.1.7 and the set of concurrency constructs defined in Java (version J2SE 5.0). ConMAn contains 25 mutation operators that can be classified into 5 general categories (see Table 3):

1. Modify parameters of concurrent methods
2. Modify the occurrence of concurrency method calls (removing, replacing, exchanging)
3. Modify concurrency keywords (addition and removal)
4. Switch concurrent objects
5. Modify critical regions (shift, expand, shrink, split)

One example ConMAn mutation operator is MSP or Modify Synchronized Block Parameter. The MSP operator is applied to all synchronized blocks and will modify the current lock object to another object. For example:

Original Code:

```
private Object lock1 = new Object();
private Object lock2 = new Object();
....
public void methodA() {
    synchronized(lock1) { ... }
}
...
```

MSP Mutant:

```
private Object lock1 = new Object();
private Object lock2 = new Object();
...
public void methodA() {
    synchronized(lock2) { ... }
}
...
```

Another MSP Mutant:

```
private Object lock1 = new Object();
private Object lock2 = new Object();
...
public void methodA() {
    synchronized(this) { ... }
}
...
```

The MSP operator can result in a wrong lock bug. The bug depth and path error density of the bug in the mutant program will vary depending on where in the program the lock object is modified.

Another example of a ConMAn operator is SPCR or Split Critical Region which divides one critical region into two distinct critical regions. The consequences of splitting a critical region into 2 regions may cause a set of statements that were meant to be atomic to become nonatomic. For example:

Original Code:

```
<statement n1>
synchronized (this) {
    //critical region
    <statement c1>
    <statement c2>
}
<statement n2>
...
```

SPCR Mutant:

```
<statement n1>
synchronized (this) {
    //critical region
    <statement c1>
}
synchronized (this) {
    <statement c2>
}
<statement n2>
...
```

3.5 Performance Measures

In order to assess the fitness of a given concurrency bug detection tool using our proposed benchmark we need to define a set of performance measures. In particular, we need to assess the fitness of a given tool with respect to its ability to find bugs (effectiveness) and its efficiency with which the bug detection is carried out. Community consultation may lead to the inclusion of other kinds of performance measures related to accuracy or other attributes of the tools under evaluation.

3.5.1 Effectiveness Measures

We propose the following effectiveness measures [13]:

bug detection rate of t =
the percentage of bugs detected by a tool t.

ease to kill a kind of bug by t =
the percentage of bugs of a given kind that are detected by a tool t.

3.5.2 Efficiency Measures

We propose the following efficiency measures [13]:

cost (in time) to detect a bug by t =
the total time to detect the bug by a tool t

path cost to detect a bug by t =
the number of interleaving schedules analyzed/executed in order to find the bug by a tool t

4. CONCLUSIONS & FUTURE WORK

We have proposed a new benchmark to assess the fitness of a concurrency bug detection tool and to compare it with other tools. Specifically our benchmark is intended to compare all kinds of concurrency bug detection tools including model checking tools, testing tools and static analysis tools. We have motivated the need for a new benchmark, have proposed how to systematically assemble a set of concurrency programs (task samples) that will effectively cover the space of possible programs and we have included effectiveness and efficiency measures for comparing tools using our benchmark programs.

In addition to simply proposing a new benchmark we have also developed a new approach to benchmark construction based on combinatorial test design (CTD). Our approach defines a set of parameters that characterize the set of concurrency programs that our benchmark should represent. We use a pairwise plan to combine the parameters into a set of 44 concurrency programs. Currently we have not identified 44 actual programs to complete the pairwise plan constructed from our CTD model. Prior to completing the plan and constructing the actual benchmark we want to obtain feedback from the concurrency bug detection community (i.e., PADTAD participants). Community consultation and feedback is important for the construction and adoption of any benchmark [38].

The next step in our research is to identify a set of concurrency programs that will complete the pairwise plan generated using our CTD model. We plan to leverage existing benchmarks and open source repositories, as well as apply program mutation, in order to generate a set of programs that satisfy the specifications for the 44 programs. Our benchmark assumes that existing programs contain a single documented concurrency bug, however in practice a program may contain many bugs. One of the future challenges is to address the use of open source programs that may contain more than one bug and the effect of these programs on the benchmark.

5. ACKNOWLEDGEMENTS

This research was partially funded by the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257574 (FITTEST) and partially by the Natural Sciences and Engineering Research Council of Canada (NSERC).

6. REFERENCES

[1] ConMAn: Concurrency mutation analysis operators. Web page: `https://github.com/sqrg-uoit/ConMAn` (last accessed Apr. 29, 2012).

[2] IBM Test Planning and Coverage Analysis Tool. Web page: `http://researcher.ibm.com/view_project.php?id=1871` (last accessed Apr. 27, 2012).

[3] jChord - a static and dynamic program analysis framework for Java. Web page: `http://code.google.com/p/jchord/` (last accessed Jun. 21, 2012).

[4] Jlint. Web page: `http://artho.com/jlint/manual.html` (last accessed Jun. 21, 2012).

[5] JSure for concurrency. Web page: `http://www.surelogic.com/concurrency-tools.html` (last accessed Jun. 21, 2012).

[6] MOBAT. Web page: `http://sourceforge.net/projects/mobat/` (last accessed Apr. 29, 2012).

[7] Parsoft JTest - Java testing, static analysis, code review. Web page: `http://www.parasoft.com/jsp/products/jtest.jsp/` (last accessed Jun. 21, 2012).

[8] Conditional must not aliasing for static race detection. *ACM SIGPLAN Notices*, 42(1), 2007.

[9] J. H. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Trans. Softw. Eng.*, 29(7):634–648, 2003.

[10] C. Artho. Finding faults in multi-threaded programs. Master's thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, 2001.

[11] C. Artho, K. Havelund, and A. Biere. High-level data races. In *Proc. of the 1st Int. Workshop on Verification and Validation of Enterprise Information Systems (VVEIS'03)*, 2003.

[12] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation operators for concurrent Java (J2SE 5.0). In *Proc. of the 2nd Workshop on Mutation Analysis (Mutation 2006)*, pages 83–92, 2006.

[13] J. S. Bradbury, J. R. Cordy, and J. Dingel. Comparative assessment of testing and model checking using program mutation. In *Proc. of the 3rd Workshop on Mutation Analysis (Mutation 2007)*, pages 210–219, Sept. 2007.

[14] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on Martian Rover software. *Formal Methods in Systems Design Journal*, 25(2-3):167–198, 2004.

[15] M. B. Cohen and S. Ur. Combinatorial test design in practice. In *Proc. of the 32nd ACM/IEEE Int. Conf. on Software Engineering - Volume 2*, ICSE '10, pages 495–496, 2010.

[16] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. of the 21st Int. Conf. on Software engineering*, ICSE '99, pages 285–294, 1999.

[17] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Proc. of ICSM 2005*, pages 411–420, 2005.

[18] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proc. of the 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*, pages 92–104, 2006.

[19] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[20] Y. Eytani, R. Tzoref, and S. Ur. Experience with a concurrency bugs benchmark. In *Proc. of the 1st Software Testing Benchmark Workshop (TESTBENCH'08)*, 2008.

[21] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proc. of the 2nd Int. Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, 2004.

[22] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. of the 1st Int. Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003)*, 2003.

[23] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2(4), 2000.

[24] K. Havelund, S. D. Stoller, and S. Ur. Benchmark and framework for encouraging research on multi-threaded testing tools. In *Proc. of the 17th Int. Symp. on Parallel and Distributed Processing (IPDPS '03)*, page 286.1, 2003.

[25] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[26] K. Jalbert and J. S. Bradbury. Using clone detection to identify bugs in concurrent software. In *Proc. of 26th IEEE Int. Conf. on Software Maintenance (ICSM 2010)*, pages 1–5, Sept. 2010.

[27] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30:418–421, 2004.

[28] B. Long, R. Duke, D. Goldson, P. A. Strooper, and L. Wildman. Mutation-based exploration of a method for verifying concurrent Java components. In *Proc. of the 2nd Int. Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, 2004.

[29] B. Long, P. Strooper, and L. Wildman. A method for verifying concurrent Java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*, 19(3):281–294, 2007.

[30] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Proc. of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[31] Z. Luo, L. Hillis, R. Das, and Y. Qi. Effective static analysis to find concurrency bugs in Java. In *Proc. of 10th IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM 2010)*, 2010.

[32] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI 08)*, 2008.

[33] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. *ACM SIGPLAN Notices*, 41(6), 2006.

[34] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *2009 IEEE 31st Int. Conf. on Software Engineering*, pages 386–396, 2009.

[35] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. of the 9th European Software Engineering Conf. held jointly with the 11th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (ESEC/FSE-11)*, pages 267–276, 2003.

[36] N. Rungta and E. G. Mercer. Understanding hardness in models used for benchmarking model checking techniques. Dept. of Computer Science, Brigham Young University, 2007.

[37] I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA'11)*, pages 254–264, 2011.

[38] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *Proc. of the 25th Int. Conf. on Software Engineering (ICSE 2003)*, pages 74–83, May 2003.

[39] K. C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, Jan. 2002.

[40] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.