Comparing Four Static Analysis Tools for Java Concurrency Bugs

Md. Abdullah Al Mamun, Aklima Khanam, Håkan Grahn, and Robert Feldt
School of Computing, Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
to.mamun@yahoo.com, aklima.bth@gmail.com, {hakan.grahn,robert.feldt}@bth.se

ABSTRACT
Static analysis (SA) tools are being used for early detection of software defects. Concurrency bugs are different from bugs in sequential programs, and they are often harder to detect. This paper presents the evaluation of four static analysis tools and their capabilities to detect Java concurrency bugs and bug patterns. The tools, i.e., Coverity Prevent, Jtest, FindBugs, and Jlint, are evaluated using concurrent benchmark programs and a collection of multithreaded bug patterns. In addition, we have categorized the bug pattern detectors of the tools and also identified 87 unique bug patterns from the tools’ detectors and literature.

1. INTRODUCTION
Multicore processors have become the main computing platform from small embedded systems to large-scale servers. In order to harvest the performance potential of multicore processors, the software needs to be parallel and scalable. However, writing parallel software is difficult, and detecting concurrency defects is even more difficult.

There has been a significant amount of work on techniques and tools to detect concurrent software defects, e.g., static analyzers, model checkers, and dynamic analyzers [5, 16]. Static analysis tools are able to detect defects in the source code without executing it. Model checkers apply formal verification techniques on the source code using well-defined mathematical formulations [16]. In contrast, dynamic analyzers need to execute the code for detecting defects. The importance of early bug detection in software development is a well-established fact. Static analysis tools are capable of early detection of defects and of reducing the cost of software development [15, 17]. A number of studies have evaluated different static analysis tools for Java programs, but very few studies focus on Java concurrency defects [1, 23]. However, they did not cover a wide range of concurrency defect types, such different types of concurrency bugs and bug patterns.

This study evaluates two commercial, i.e., Coverity Prevent (v4.4.1) [3] and Jtest (v8.4.11) [13], and two open source, i.e., FindBugs (v1.3.9) [8] and Jlint (v3.1) [1], static analysis tools for their ability to detect Java multithreaded bugs and bug patterns. To evaluate the SA tools, we use a benchmark suite containing Java programs with concurrency bugs [5], a collection of bug patterns from a library of antipatterns [10], and the selected tools. We address the following questions:

RQ1: How effective are static analysis tools in detecting Java concurrency bugs and bug patterns?
RQ2: Are commercial SA tools better than open source SA tools in detecting Java concurrency defects?

We conducted an experiment [25] to answer these questions where the SA tools acted as subjects, and the benchmark suite and bug patterns worked as objects. In addition to the evaluation of the tools, we have categorized the rules/checkers/bug patterns used by the tools to detect defects. We studied bug pattern detectors implemented by the tools and an antipattern library [10], and finally, unified them in a list of 87 unique Java concurrency bug patterns.

The results of the study show that the commercial tool Jtest is better than the other tools in detecting Java concurrency bugs, but with the drawback that the false positive ratio reported by this tool is high. It was not possible to draw a clear distinction among the commercial and open source tools as the other commercial tool, Coverity Prevent, detects the lowest number of defects among the tools. Both FindBugs and Coverity Prevent report a low number of false positive warnings.

The rest of the paper is organized as follows. Section 2 introduces concurrency bugs and bug patterns. Section 3 presents the evaluated tools and the set of test programs that we use. Then, we present our bug patterns categorization in Section 4. The experimental methodology is presented in Section 5, while Section 6 presents the experimental results. Related work are discussed in Section 7. Finally, we conclude the study in Section 8.

2. CONCURRENCY BUGS AND PATTERNS
The most general characteristic of concurrent software is non-determinism. The non-deterministic execution of a concurrent program makes it different from a sequential program. A concurrent program holds the non-deterministic characteristic because of the interleaved execution of threads. Due to the interleaved execution, a number of problems
Concurrency problems can be divided into two types of basic properties, safety and liveness [16]. The safety property ensures that nothing bad will happen during the program execution. On the other hand, the liveness property expresses that something good will eventually happen, i.e., the program execution progresses. The most known problems under these properties are race conditions (a.k.a. data races, interleaving problems), deadlocks, livelocks, and starvation [16]. These problems must be absent in the program in order to fulfill the safety and liveness properties. These basic properties are abstract and some concurrency problems overlap between them. Therefore, it is not fruitful to classify concurrency problems based on them.

A pattern means some common technique to document and reuse specific and important examples [9], and there have been some research regarding concurrent bug characteristics and bug patterns [7, 10]. In a general sense, bug patterns (sometimes called antipatterns) describe common errors that can occur in the program. The terms, bug patterns and antipatterns are similar with the difference that bug patterns are related to coding defects where antipatterns are related to defects in the design pattern or architectural pattern. In the context of concurrent software testing, bug patterns and antipatterns are used interchangeably.

3. TOOLS AND TEST PROGRAMS

3.1 Selection of Java Static Analysis Tools

We have selected four Java static analysis tools as shown in Table 1. Among the tools, FindBugs and Jlint are the most discussed tools in the literature, probably since they are open source. However, very few articles [18, 2] have worked with the tools Coverity Prevent and Jtest. Further, to the best of our knowledge no previous study has evaluated the effectiveness of these two commercial tools for Java concurrency bugs.

Coverity Prevent [2, 3] is a commercial static analysis tool combining statistical and inter-procedural analysis with Boolean Satisfiability (SAT) analysis to detect defects. To infer correct behavior it uses statistical analysis based on the behavioral patterns identified in the source code. Then inter-procedural (whole-program) analysis across method, file, and module boundaries is done to achieve a 100% path coverage. A SAT engine and SAT solvers determine if paths are feasible at runtime or result in quality, security, or performance defects. In addition to Java, Coverity Prevent supports C# and C/C++. Coverity Prevent detects several multithreaded defects like deadlocks, thread blocks, atomicity, and race conditions.

Coverity Prevent works on multiple platforms and compilers like gcc, Microsoft Visual C++, etc. It supports Eclipse and Visual Studio IDEs. It also provides good configurability on product settings like search depth. It is possible to expand the tool by creating custom checkers.

Jtest [13, 18] is a commercial static analysis tool developed by Parasoft. It is an integrated solution for automating a broad range of best practices. Parasoft Jtest supports various code metrics calculations, coding policy enforcement, static analysis, and unit testing for Java. It also provides support for a streamlined manual code review process and peer code review. It performs pattern and flow-based static analysis for security and reliability. Finally, Jtest has a good collection of checkers for detection of multithreaded bugs.

Jtest works on several platforms like Windows, Solaris, Linux, and Mac OS X. It has both GUI and command line (batch processing) support, and works with Eclipse, IBM RAD, and Jbuilder. It allows the creation of custom rules using a graphical design tool by modifying parameters or providing code demonstrating a sample rule violation.

FindBugs [11, 12, 18] is an open source bug pattern based defect detector developed at the University of Maryland. It can find faults such as dereferencing null-pointers or unused variables. It uses syntax and dataflow analysis to check Java bytecode for detecting bugs. FindBugs reports more than 360 different bug patterns. Bug patterns are grouped into categories, e.g., multithreaded correctness, correctness, performance, etc.

FindBugs provides both GUI and command line interfaces. In addition to its own graphical interface, it also works with Eclipse and NetBeans. FindBugs analysis results can be saved in XML. It requires JRE/JDK 1.5 or later versions. FindBugs is platform independent and runs on, e.g., Windows, GNU/Linux and MacOS X platforms. It is possible to expand FindBugs by defining custom detectors.

Jlint [1, 18] is an open source static analysis tool that performs syntactic checks, flow analysis, and builds a lock graph for detecting defects like, e.g., inheritance and synchronization. It can detect data races through the use of global data flow analysis. It can detect deadlocks by inter-procedural and inter-file analysis. Jlint provides a number of checkers for detecting deadlocks in multithreaded Java programs, and it is able to detect 36 different types of bugs. It has a component named AntiC which is a syntax checker for all C-family languages, i.e., C, C++, Objective C, and Java. Jlint has a simple command line interface, and runs on Windows, UNIX, Linux. Finally, Jlint is not easily expandable.

3.2 Selection of Test Programs

We have selected test programs containing both concurrency bugs and concurrency bug patterns. It is necessary to evaluate the tools with both bugs and bug patterns. Bugs can occur due to many reasons, and the purpose of testing the tools is to reveal their effectiveness in detecting bugs. However, collecting real-life buggy programs with a high variety of error reasons is quite challenging and demands a huge amount of time. Therefore, testing the tools with respect to bug patterns is important because bug patterns can re-

<table>
<thead>
<tr>
<th>Tool name</th>
<th>License</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverity Prevent [2]</td>
<td>Commercial</td>
<td>4.4.1</td>
</tr>
<tr>
<td>Jtest [13]</td>
<td>Commercial</td>
<td>8.4.11</td>
</tr>
<tr>
<td>FindBugs [8]</td>
<td>Open Source</td>
<td>1.3.9</td>
</tr>
<tr>
<td>Jlint [1]</td>
<td>Open Source</td>
<td>3.1</td>
</tr>
</tbody>
</table>
flect a high variety of situations where a bug can potentially occur [10]. However, bug patterns do not always lead to actual bugs, hence it is meaningful to test the tools with both bugs and bug patterns.

We used two sets of programs in our study, where the first set represents concurrency bugs and the second set represents concurrency bug patterns. The first set of programs is taken from a concurrency bug benchmark suite [5]. There is a criticism of using benchmarks for evaluating the effectiveness of verification and validation technologies, because benchmarks may be incomplete in covering several factors that can lead to an incorrect result [14]. However, benchmarks can be used if such limitations are considered [20].

The selected benchmarks are also used in other studies. An experience story [6] of the benchmark reports a list of 14 studies and research centers that have used the benchmark. Experts in concurrent software testing and students of a concurrent software testing course wrote most of the benchmark programs. We selected 19 programs from this benchmark suite that provide precise bug documentation and one additional program. Table 2 shows the selected benchmark programs. Detailed information about these programs are given in the benchmark suite [5].

The second set is a collection of Java concurrency bug patterns and antipatterns. We have collected these patterns from the four evaluated tools, i.e., included those patterns that the tools document and claim to detect, and a collection of antipatterns [10]. We have categorized and identified 87 unique bug patterns from 141 bug patterns that are discussed in Section 4. Then we collected or wrote programs for these bug patterns. These programs are very small; usually 10 to 30 lines of code.

4. BUG PATTERN CATEGORIZATION

The selected tools have more than 100 bug patterns, i.e., bug patterns that the tool vendors claim that their tools are able to detect. In order to carry out the experiment, we need to identify the unique bug patterns. More importantly, we need to categorize these bug patterns under common vocabularies. A person may easily have a general idea about the strength of a tool if the tool describes its checkers/rules/patterns under refined bug categories like deadlock, data race, livelock, etc. Table 3 shows the categories of concurrency checkers/bug patterns described by the tools.

Unfortunately, bug patterns categorized by the tools are not satisfactory. Jlint describes its bug patterns in different categories, and Jtest provides a further categorization of its bug patterns in the bug documentation provided with the tool. Jtest describes 19 bug patterns in the category Deadlocks and race conditions, 6 bug patterns in the category Concurrency, and 18 other bug patterns that are not categorized. It should be mentioned that the five checkers under the category Preview, described by Coverity Prevent, is still under refinement and hence they are not recommended for regular industrial use.

We found two studies [7, 10] that worked with concurrent bug patterns. Among them, Hallal et al. [10] mentioned the advantages of having a good taxonomy of bug patterns and proposed a comprehensive categorization of 38 concurrency antipatterns under 6 categories. They developed the categories keeping the benefit for the developers in mind. We have adopted and extended these categories. In order to develop a unique collection of bug patterns we have used 141 bug patterns, where 103 patterns are collected from the tools and 38 patterns from the antipattern library developed by Farchi et al. [7]. The antipattern library documents 8 bug patterns from FindBugs and 11 bug patterns from Jlint. Since this antipattern library is mostly populated with concurrency bug patterns, this study uses the term 'bug pattern library' to represent it. We have found 87 unique bug patterns from totally 141 bug patterns and categorized them, as shown in Table 4. However, the bug pattern detectors

![Table 2: Selected benchmark programs.](image)

<table>
<thead>
<tr>
<th>Program name</th>
<th>Documented bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProducerConsumer</td>
<td>Orphaned-thread, Wrong lock or no lock*</td>
</tr>
<tr>
<td>SoftWareVerification</td>
<td>Orphaned-thread, Not-atomic, Lazy init*</td>
</tr>
<tr>
<td>BuggedProgram</td>
<td>Not-atomic</td>
</tr>
<tr>
<td>SoftTestProject</td>
<td>Not-atomic, interleaving, Wrong lock or no lock*</td>
</tr>
<tr>
<td>BugTester</td>
<td>Non-atomic</td>
</tr>
<tr>
<td>MergeSortBug</td>
<td>Not-atomic</td>
</tr>
<tr>
<td>Manager</td>
<td>Not-atomic</td>
</tr>
<tr>
<td>Critical</td>
<td>Not-atomic</td>
</tr>
<tr>
<td>Buggy account program</td>
<td>Not-atomic</td>
</tr>
<tr>
<td>Bufwriter</td>
<td>Wrong lock or no lock, Blocking critical section, Wrong lock or no lock*</td>
</tr>
<tr>
<td>Account</td>
<td>Wrong lock or no lock</td>
</tr>
<tr>
<td>Bug1(Deadlock)</td>
<td>Deadlock</td>
</tr>
<tr>
<td>GarageManager</td>
<td>Blocking-critical-section</td>
</tr>
<tr>
<td>TicketsOrderSim</td>
<td>Double checked locking</td>
</tr>
<tr>
<td>Shop</td>
<td>Weak reality (two stage lock), Wrong lock or no lock*</td>
</tr>
<tr>
<td>BoundedBuffer</td>
<td>Notify instead of notifyAll, Data race*</td>
</tr>
<tr>
<td>Test</td>
<td>Weak-reality (two stage access)</td>
</tr>
<tr>
<td>IBM_Airlines</td>
<td>Condition-For-Wait, Wrong lock or no lock*</td>
</tr>
<tr>
<td>Deadlock **</td>
<td>Hold and wait</td>
</tr>
</tbody>
</table>

* - Bugs not documented by the benchmark suite.  
** - Program not collected from the benchmark suite.

![Table 3: Bug patterns and categories.](image)

<table>
<thead>
<tr>
<th>Tools</th>
<th>Number of checkers/bug patterns</th>
<th>Concurrency checkers/rules/bug patterns by bug categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverity</td>
<td>10 checkers</td>
<td>Concurrency: 4 regular checkers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preview: 6 non-regular checkers</td>
</tr>
<tr>
<td>Jtest</td>
<td>43 bug patterns</td>
<td>Thread safe programming: All 43 bug patterns</td>
</tr>
<tr>
<td>FindBugs</td>
<td>23 checkers with 40 bug patterns</td>
<td>Multithreaded correctness: All 23 checkers</td>
</tr>
<tr>
<td>Jlint</td>
<td>12 bug patterns</td>
<td>Deadlock: 7 bug patterns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Race condition: 4 bug patterns waitNoSync: 1 bug pattern</td>
</tr>
</tbody>
</table>
implemented by different tools may vary to some extent, although they are listed as a common bug pattern.

5. EXPERIMENTAL METHODOLOGY
The study is conducted as an experiment. The subjects of the experiment are open source and commercial static analysis tools for testing multithreaded Java programs. The objects of the experiment are a collection of Java multithreaded programs that will be analyzed by the testing tools.

The primary measure in the study is the defect detection ratio of the tools, as defined below. Further, we also study and categorized all warnings generated by the tools in order to evaluate the number of false positives generated by the tools.

\[
\text{Defect detection ratio} = \frac{\text{No. of defects detected by SA tool}}{\text{Total number of defects}}
\]

During the evaluation, we activated all concurrency related checkers/rules and set the tools in full analysis mode. Coverity Prevent is used with both concurrency and preview (a collection of checkers still under development) checkers. Jtest uses a set of rules named thread safe programming in order to evaluate our test programs. In FindBugs, the multithreaded correctness bug category will be used with minimum priority to report level as low and analysis effort as maximal. Jlint will be used with the +all command line option.

The experiments are executed in a Windows environment, on a system with an Intel Core 2 Quad CPU and 3GB of main memory. JRE 1.6 is used as the Java virtual machine, and Microsoft Excel is used to collect the experimental data. We use Eclipse (version 3.4.2) for the tools Coverity Prevent, FindBugs, and Jtest since all of them provide plugins for Eclipse. Jlint is used in the command line mode because it does not provide a graphical user interface.

Though the benchmark programs cover a variety of concurrency bugs, they are not evenly distributed in different categories. Table 5 shows that the number of bugs in the deadlock and livelock categories is 4 and 2, respectively. Larger bug samples within these categories would make the result more general.

6. RESULTS
6.1 Testing Concurrency Bugs
We tested the four static analysis tools on 20 Java programs containing 32 concurrency bugs. Table 5 shows how many bugs that are detected by each of the SA tools. The selected benchmark programs contain 11 different types of bugs. Detailed descriptions of these bug types are available in a study by Farchi et al. [7], which are the researchers who developed the benchmark suite.

<table>
<thead>
<tr>
<th>Category</th>
<th>Coverity Prevent</th>
<th>Jtest</th>
<th>FindBugs</th>
<th>Jlint</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>5</td>
<td>146</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>True</td>
<td>4</td>
<td>21</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>False positive</td>
<td>4</td>
<td>16</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>Undetermined</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>16</td>
<td>181</td>
<td>16</td>
<td>34</td>
</tr>
</tbody>
</table>

The category data race and atomicity violation, FindBugs also detected 5 bugs, where 4 bugs are in the data race and atomicity violation category and 1 bug in the deadlock category. Jtest detects 8 bugs, which is more than both Coverity Prevent and FindBugs.

We documented and inspected each warning generated by the tools. An overview of the warnings is shown in Table 6. The general warning category contains the warnings that are not exactly related to the correctness of the program. Jtest reports a large number of warnings, totally 181 warnings. More than 75% were general warnings, and among the 136 general warnings, 50 warnings are generated from a single Jtest rule named TRS.NAME that checks whether a thread initializes its name. Jlint does not have quality and styles related concurrency rules, and hence it does not produce any general warnings. FindBugs reports only 2 general warnings, even though it has 23 checkers with 40 bug patterns, where several address quality and style problems.

Looking at the number of false positive warnings, we observe that Jtest and Jlint have significantly more false positives than the other tools. FindBugs and Coverity Prevent have almost the same number of false positive warnings, but FindBugs can be considered as better since it checks for a larger number of bug patterns as compared to Coverity Prevent. Similarly, Jtest can be seen as more powerful than Jlint as it checks for more bug patterns than Jlint.

6.2 Testing Concurrency Bug Patterns
We tested the tools with 87 unique bug patterns, see Table 3. It is expected that every tool should be able to detect a bug pattern that it claims to detect. The tools were almost able to detect “their own” bug patterns, as promised. Five cases are documented where Coverity Prevent (3 cases) and Jlint (2 cases) fail to detect bug patterns, though they have detectors for these bug patterns. Further, a few cases are observed where the strength of the bug pattern detectors differs though they are described in a similar way. Table 7 shows the number of bug patterns detected by the tools in different categories. We observe that JTest detects most concurrency bug patterns, even though it only detects less than half of the bug patterns. The other commercial tool, Coverity Prevent, only detects 8% of the bug patterns.

7. RELATED WORK
Artho [1] evaluated three dynamic analysis tools (MaC, Rivet, Visualthreads) and two static analysis tools (Jlint and ESC/Java) for finding concurrency bugs in Java program. The results of the study show that none of the tools is a clear winner. A
major part of this study is about extending the Jlint tool.

A study by Rutar et al. [19] used five bug finding tools, namely Bandera, ESC/Java 2, FindBugs, Jlint, and PMD, to cross check their bug reports and warnings. This study identified the overlapped warnings reported by the tools. They divided the warnings into different bug categories, where concurrency was identified as one of the categories. Finally, a meta-tool was proposed, which combines the warnings of all the five tools used.

In addition, we found several studies evaluating static analysis tools from different perspectives other than detecting concurrency bugs. A study by Painchaud et al. [18] evaluated four commercial and seven open source static analysis tools. Their study also recommended a six steps methodology to assess the software quality.

Two industrial case studies are described in [21], where two static analysis bug pattern tools are evaluated. However, the paper do not discuss any concurrency issues. Another industrial case study [22] analyzes the interrelationships of static analysis tools, testing, and reviews. The results show that static analysis tools detect a subset of the defects detected by reviews with a considerable number of false warnings. However, the static analysis tools detect different types of bugs than testing. Hence a combined approach is suggested by the study. A third industrial case study [4] surveys three static analysis tools along with an experience evaluation at a large software development company.

F. Wedyan et al. [24] evaluated the usefulness of automated static analysis tools for Java program. They evaluate the effectiveness of static analysis tools for detecting defects and identifying code refactoring modifications.

8. CONCLUSIONS

We have evaluated four static analysis tools for detecting Java concurrency bugs and bug patterns. A total number of 141 bug patterns is collected from the tools and from a library of antipatterns. We identified and classified 87 unique bug patterns and tested the tools against them. Finally, we inspected each warning reported by the tools, and classified them as true or false positive warnings.

The defect detection ratio of the best tool, Jtest, is 0.48 and the average defect detection ratio of the tools is 0.25. This reveals the fact that static analysis tools alone are not sufficient in detecting concurrency bugs. Moreover, the tools report a number false positive warnings, which is about the same as the number of defect detected.
The experiment with the bug patterns shows that the selected tools are able to detect a wide range of bug patterns. In general, we can not say that the commercial tools are better than the open source tools, since one of the commercial tool is best and the other one worst in detecting concurrency bugs. However, the effectiveness of the tools varies in terms of detecting bugs in different categories and in reporting false positive warnings. It would be more beneficial if the users take the respective advantage of several tools for detecting bugs in different categories.

9. REFERENCES


[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.


