Master Thesis
Software Engineering
Thesis no: MSE-2009-22
September 2009

Concurrent Software Testing: A Systematic Review and an Evaluation of Static Analysis Tools

Md. Abdullah Al Mamun
Aklima Khanam

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 2x20 weeks of full time studies.

Contact Information:
Authors:
Md. Abdullah Al Mamun
E-mail: to.mamun@yahoo.com

Aklima Khanam
E-mail: aklima.bth@gmail.com

University advisor:
Prof. Dr. Håkan Grahn
School of Computing, BTH

Examiner:
Dr. Robert Feldt
School of Computing, BTH

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/com
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

Verification and validation is one of the most important concerns in the area of software engineering towards more reliable software development. Hence it is important to overcome the challenges of testing concurrent programs. The extensive use of concurrent systems warrants more attention to the concurrent software testing. For testing concurrent software, automatic tools development is getting increased focus. The first part of this study presents a systematic review that aims to explore the state-of-the-art of concurrent software testing. The systematic review reports several issues like concurrent software characteristics, bugs, testing techniques and tools, test case generation techniques and tools, and benchmarks developed for the tools. The second part presents the evaluation of four commercial and open source static analysis tools detecting Java multithreaded bugs. An empirical evaluation of the tools would help the industry as well as the academia to learn more about the effectiveness of the static analysis tools for concurrency bugs.

Keywords: Systematic review, concurrent software testing, testing techniques and tools, test case generation, benchmark, static analysis tools, concurrency bugs, bug patterns.

CONTENTS

Chapter I - Introduction

Chapter 2 - Systematic Review on the State-of-the-art of Concurrent Software Testing

Chapter 3 – An Evaluation of Static Analysis Tools for Java Multithreaded Bugs

ACKNOWLEDGEMENT

First we would like to thank our supervisor Prof. Dr. Håkan Grahn, for his valuable guidance, and constructive thoughts throughout this thesis. We are grateful to him for his valuable time for the meetings and discussions during the thesis work. We are thankful to him for his enormous support and appreciation throughout the thesis work that helped us to achieve our goal.

We like to give special thanks to Dr. Robert Feldt for his intellectual guidance and acceptable ideas for this work. We wrote several emails to him for his valuable suggestion during the thesis work. All the time, we got prompt reply from him that was really inspiring.

Furthermore, we would like to thank Mr. Dejan Baca at Ericsson and Mr. Mattias Terfelt at Parasoft, for their support with the commercial tools Coverity Prevent and Jtest.

HMT FORMAT

This thesis report follows the 'Hybrid Master Thesis' (HMT) format. It is a hybrid form of master thesis that combines a traditional master thesis format and an IEEE/ACM paper format. The reason to the hybrid format is to make students focus on their writing and express themselves more clearly and to increase the number of theses that are published as papers. The HMT format contains two main parts. The first part contains the report according to IEEE/ACM format focusing relevant areas of the thesis work that can be 15 pages in maximum. The second part contains a number of appendices that demonstrate the detail aspects, particular experimental factors, and specific methodology, etc. The second part can be 15-40 pages.

This thesis report consists of three chapters where chapter 2 and chapter 3 are written as independent articles according to the IEEE paper format with a separate list of reference and appendices. A common introduction of the whole study (i.e. chapter 2 and 3) is given in Chapter 1 that is written in traditional master thesis format.

CHAPTER 1

1. INTRODUCTION

Today we are living in a world where concurrent programs are replacing sequential programs. This change is necessary to utilize the true capabilities of multi-core processors. The multi-core processors are replacing the single-core processors from personal computers to supercomputers, and they execute software code concurrently in different cores. In addition to a lot of opportunities, this transition comes with additional challenges related to architecture, hardware and software that need to be overcome. More attention is needed in the software technology to take full advantages of multi-core processors. In the context of multi-core processing, the hardware technology is more advanced than the software, which needs to be balanced to retain the trend of increasing computing performance with time [3].

2. MOTIVATION

Concurrent programs have more attributes than sequential programs. The most unique property of concurrent software is its non-deterministic execution behavior [11]. It means that same input to a concurrent program might result in different execution sequences. It happens because of the interleaved execution of the concurrent programs. Writing a concurrent program is harder than a sequential program and the non-deterministic execution behavior of a concurrent program makes it difficult to test. Concurrency bugs are one of the most troublesome among all software bugs, which are very difficult to detect and even more difficult to diagnose and repeat.

Verification and validation is one of the most important concerns in the area of software engineering towards more reliable software development. Testing ensures the correctness and reliability of the program. Since concurrency bugs widely exist in concurrent programs, it is important to overcome the difficulties of testing concurrent programs towards the development of reliable concurrent software systems.

The field of concurrent software testing is not new. This is because we have shared memory multiprocessor systems before multi-core systems. Different issues addressed by shared memory concurrent program execution in multiprocessor systems are also applicable for concurrent program execution in multi-core systems. However, unlike multi-processor systems, multi-core systems are being widely used from notebooks to supercomputers. The extensive use of multi-core systems warrants more attention to the concurrent software testing.

Testing concurrent program is more difficult than testing sequential programs [13]. Researchers have already addressed the challenges of concurrent software testing [10,13,15] and there are various tools and techniques available for testing concurrent programs. The available techniques in this field are primarily divided into two sets of technologies namely static and dynamic [5]. The well known techniques under these two major groups are static analysis, formal verification, model checking, dynamic analysis, systematic state space exploration, instrumentation, replay etc. A comprehensive mapping of this field would be helpful for the researchers who are working with concurrent software testing.

Early detection of software bug is a well established fact in software engineering that has a major impact in the reduction of the software development costs. Static analysis (SA) tools are able to detect bugs early [7,8] hence reduce the cost of software development [2]. A number of studies evaluated different static analysis tools for Java programs [4,9,10,12-15]. However, very few studies focus on the evaluation of the static analysis tools for Java multithreaded defects [1, 14]. These attempts are not rigorous in terms of the variety of multithreaded bugs. That means the studies did not cover a wide range of concurrency bug types. More empirical evidence is necessary in assessing the true capabilities of different tools and techniques available in this field to promote the adaptation or improvement of the techniques and tools.

This thesis presents a systematic literature review mapping the area of concurrent software testing and an evaluation of static analysis tools detecting Java concurrency bugs.

3. RESEARCH QUESTIONS

This thesis aims to answer the following research question.

RQ1: What is the state-of-the-art of concurrent software testing in shared memory systems?

RQ1.1: What are the characteristics and bugs of concurrent software?

RQ1.2: What are the testing techniques, testing tools, and benchmarks for the tools of concurrent software?

RQ1.3: What are the techniques and tools for test case generation of concurrent software?

RQ2: How effective are static analysis tools in detecting Java multithreaded bugs and bug patterns?

RQ3: Are commercial SA tools better than open source SA tools in detecting Java multithreaded defects?

RQ1 is addressed in chapter 2, and RQ2 and RQ3 are addressed in chapter 3.

4. INCLUDED STUDIES

4.1. A SYSTEMATIC REVIEW

Chapter 2 describes a systematic review (SR) on the State-of-the-art of concurrent software testing. The study covers related articles in the field of concurrent software testing from the year 1989 to 2009. The systematic review selected in total 109 articles from 4352 articles available in different scholarly databases. The SR answers the RQ1, shown in section 2, discussing about concurrent software characteristics, bugs, testing techniques and tools, tools and techniques for test case generation, and benchmarks for the tools for concurrent software.

4.2. AN EVALUATION OF STATIC ANALYSIS TOOLS

Chapter 3 presents the results of the evaluation of static analysis tools detecting Java Concurrent defects answering RQ2 and RQ3 mentioned in section 2. We selected two

open source and two commercial static analysis (SA) tools and executed a controlled experiment in order to evaluate them. To evaluate the SA tools, we have used a benchmark suite containing Java programs with concurrency bugs [7] and a collection of bug patterns collected from the selected tools and a library of antipatterns [12].

5. MAJOR CONTRIBUTIONS AND RESULTS

The systematic review maps the broad area of concurrent software testing that would be helpful for the researchers working in this field. We have reported the most common discussed difficulties and bugs in this field. We mapped two major testing technology domains, i.e., static and dynamic testing technologies, with their subcategorized testing techniques. The study identifies and reports 51 different testing techniques and 38 testing tools. The study also reports different techniques and tools used for test case generation of concurrent software. We identified a number of benchmarks used by the researchers and the industry. The results show that Java, Ada, and C/C++ are the most frequently discussed languages in the literature.

The results of the evaluation of the static analysis tools show that in overall the commercial tools are better than the open source tools in detecting concurrency defects. The average defect detection ratio of the tools is found as 0.25. This reveals the fact that static analysis tools alone are not sufficient in detecting concurrency bugs. Moreover, the tools reported false positive warnings, which is about the same as the number of defect detected by each tool. The experiment with the bug patterns shows that the selected tools are able to detect a wide range of bug patterns. However, the effectiveness of the tools varies in detecting bugs in different categories and in reporting false positive warnings.

In addition to the evaluation of the tools, we have categorized the rules/checkers/bug patterns used by the tools to detect defects. This categorization would help the users to understand and choose an appropriate set of checkers/bug patterns according to their need. Studying the bug pattern detectors from several sources we have identified a list of 87 unique Java multithreaded bug patterns.

6. CONCLUSION

The pervasive use of concurrent software warrants more research and development activity in the field of concurrent software testing. This study presents the state-of-the-art of concurrent software testing, which would help the researchers to carry out more research in this field. The results of the evaluation of the static analysis tools reveal the effectiveness of the tools in detecting concurrency defects. The results might inspire the industry to adapt static analysis tools with their everyday work and the tool developers to further work to refine several weaknesses of the tools.

7. REFERENCES

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

[2] D. Baca, B. Carlsson, and L. Lundberg, “Evaluating the cost reduction of static code analysis for software security,” *3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security 2008, PLAS'08, June 8, 2008 - June 8, 2008*, Tucson, AZ, United states: Association for Computing Machinery, 2008, pp. 79-88.

[3] M. A. Bauer, "High performance computing: the software challenges," *In Proceeding of the 2007 international Workshop on Parallel Symbolic Computation (London, Ontario, Canada, July 27 - 28, 2007). PASCO '07*. ACM, New York, NY, 11-12.

[4] C. N. Christopher, "Evaluating Static Analysis Frameworks," Carnegie Mellon University Analysis of Software Artifacts, 2006.

[5] Y. Eytani, K. Havelund, S. Stoller, and S. Ur, "Toward a benchmark for multi-threaded testing tools," *Concurrency and Computation: Practice and Experience*, 2005.

[6] H. Hallal, E. Alikacem, W. Tunney, S. Boroday, and A. Petrenko, "Antipattern-based detection of deficiencies in Java multithreaded software," *Proceedings. Fourth International Conference on Quality Software, 8-9 Sept. 2004*, Los Alamitos, CA, USA: IEEE Comput. Soc, 2004, pp. 258-67.

[7] S. Lipner, The Trustworthy Computing Security Development Life Cycle, Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'o4), p.2-13, 2004.

[8] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," *27th International Conference on Software Engineering, ICSE 2005, May 15, 2005 - May 21, 2005*, Saint Louis, MO, United states: Institute of Electrical and Electronics Engineers Computer Society, 2005, pp. 580-586.

[9] F. Painchaud and R. Carbone, *Java software verification tools: Evaluation and recommended methodology,* Technical Memorandum. Defence R&D Canada. Document No. TM 2005-226. March 2006. http://cradpdf.drdc.gc.ca/PDFS/unc57/p527369.pdf

[10] N. Rutar, C.B. Almazan, and J.S. Foster, "A comparison of bug finding tools for Java," *ISSRE 2004 Proceedings; 15th International Symposium on Software Reliability Engineering, November 2, 2004 - November 5, 2004*, Saint-Malo, France: IEEE Computer Society, 2004, pp. 245-256.

[11] K. Tai, "Testing of concurrent software," *Proceedings of the 13th Annual International Computer Software and Applications Conference (Cat. No.89CH2743-3), 20-22 Sept. 1989*, Washington, DC, USA: IEEE Comput. Soc. Press, 1989, pp. 62-4.

[12] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb, "An evaluation of two bug pattern tools for Java," *2008 First IEEE International Conference on Software Testing, Verification and Validation (ICST '08), 9-11 April 2008*, Piscataway, NJ, USA: IEEE, 2008, pp. 248-57.

[13] S. Wagner, J. Jurjens, C. Roller, and P. Trischberger, "Comparing Bug finding tools with reviews and tests," *Testing of Communicating Systems. 17th IFIP TC6/WG 6.1 International Conference TestCom 2005. Proceedings, 31 May-2 June 2005*, Berlin, Germany: Springer-Verlag, 2005, pp. 40-55.

[14] M. S. Ware and C.J. Fox, "Securing Java code: heuristics and an evaluation of static analysis tools," *Proceedings of the 2008 workshop on Static analysis*, Tucson, Arizona: ACM, 2008, pp. 12-21.

[15] F. Wedyan, D. Alrmuny, and J. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," *2009 2nd International Conference on Software Testing Verification and Validation (ICST 2009), 1-4 April 2009*, Piscataway, NJ, USA: IEEE, 2009, pp. 141-50.

CHAPTER 2

Paper 1. Systematic Review on the State-of-the-art of Concurrent Software Testing

Systematic Review on the State-of-the-art of Concurrent Software Testing

Aklima Khanam and Md. Abdullah Al Mamun
School of computing
Blekinge Institute of Technology
372 36, Ronneby, Sweden
aklima.bth@gmail.com, to.mamun@yahoo.com

Abstract

The importance of software testing for the correctness of the concurrent program is well established in the discipline of software engineering. The field concurrent software testing is older than three decades and there has been a significant amount of work accomplished addressing different issues in this field. Mapping of important issues within the field can indicate the overall progress accomplished in concurrent software testing. This paper presents a systematic review of concurrent software testing in a set of 109 articles selected from in total 4352 articles from the year 1989 to 2009. Issues covered in this study are concurrent software characteristics, bugs, testing techniques and tools, test case generation, and benchmark for the tools. The study maps and consolidates major work done in this field, which would serve the researchers as a quality source of references. The results also show the recent trend in the concurrent software testing area.

1. Introduction

We are in the middle of an important transition in software and computer engineering. The multi-core processors are replacing the single-core processors from personal computers to supercomputers, and they execute software code concurrently in different cores. In addition to a lot of opportunities, this transition comes with additional challenges related to architecture, hardware and software that need to overcome. However, for taking the full advantages of multi-core processors, more attention is needed in the software technology. In the context of multi-core processing, the hardware technology is significantly advanced than the software, which needs to be balanced to retain the trend of increasing computing performance with time [2].

Among various challenges a common problem is that companies have large code bases that are written for single-core processor architectures, i.e. use few or no constructs to describe concurrent executing pieces of the software. At the same time, customers that buy hardware with multiple cores want their system to have higher performance without any extra development efforts and costs. A key competitive advantage for companies in the coming years will thus be to turn their legacy code into well-tested systems for multi-core architectures.

Fortunately, shared memory or tightly coupled memory systems are nothing new. Different issues addressed by shared memory concurrent program execution in multiprocessor systems are also applicable to concurrent program execution in multi-core systems. Multi-processor systems are being used for high performance computing, on the other hand, multi-core systems are being widely used from notebooks to supercomputers. The extensive use of multi-core systems with shared memory warrants more attention to the concurrent software testing.

Concurrent program testing is more difficult than testing sequential programs [25]. Researchers have already addressed the challenges of concurrent software testing [22,25,27] and there are various tools and techniques available for testing concurrent programs. As far as we know there is no systematic review (SR) performed on concurrent software testing. Presently, it is important to map how much work has been done in this field. The main contributions of this study are:

- Identifying concurrent software characteristics and bugs.
- Identifying the testing techniques, tools, and benchmarks for the tools.
- Identifying the techniques and tools for test case generation of concurrent software.

Concurrent software testing is not a new field. It is more than 35 years since the first problem of concurrent software testing was addressed by the researchers. The field is quite large in terms of concurrency problems, testing approaches, techniques, and tools. The goal of this study is to map this broad area that would be helpful for the researchers working in this field. We have reported the most common discussed difficulties and bugs in this field. We mapped two major testing technology domains, i.e., static and dynamic testing technologies, with their subcategorized testing techniques. We have identified and reported 51 different testing techniques and 38 testing tools. The study also reports different techniques and tools used for test case generation of concurrent software. Benchmarks are important to evaluate the ability of testing techniques and tools. We identified a number of

benchmarks used by the researchers and the industry. The results show that Java, Ada, and C/C++ are the most frequently discussed languages. We associate the testing tools and benchmarks with realated programming languages.

The rest of this paper is organized as follows. The background of the study is discussed in the next section. The phases of SR are discussed in section 3. Section 4 presents the design of the SR. The execution of the review is shown in section 5. Section 6 discusses the results of the study. The overall discussion and validity threats are given in section 7. Finally, we concluded in section 8.

2. Background

Brinch Hansen wrote the first two concurrent software-testing papers in the year 1973 and 1978 [12,13]. In 1985 several issues on testing concurrent programs were addressed and deterministic testing of concurrent software was started [24]. Since then, we observed numerous works covering different issues of this field. The characteristics of concurrent programs are well discussed in the literature. The most common characteristics addressed are non-determinism, data race, and deadlock.

Black box testing which is a very popular testing approach for sequential programs is not appropriate for concurrent program testing. Structural testing or white box testing is more appropriate for concurrent program testing. A testing framework and theoretical foundations for structural testing are presented in [28], which is a first step in all-uses coverage testing. Researchers classified concurrent software testing techniques into several categories like model checking, static analysis, and dynamic analysis. All of these categories addressed different issues of concurrent testing like data race, deadlock, coverage criteria etc.

Eytani et al. [10] conducted a survey on different testing technologies. Concurrent testing techniques are basically divided into two groups of technologies namely static and dynamic. Static techniques analyze the code without executing it [4,23] where the dynamic techniques work in the runtime. Static techniques can be divided into static analysis techniques and formal methods. In formal methods, we mainly have model checkers that analyze the code using state space exploration. In general, static analysis techniques are more scalable than the model checkers.

Model checking covers a number of various techniques those are based on exhaustive or systematic state-space exploration. Model checking techniques with exhaustive state space exploration seriously suffer from scalability issues. In recent years model checkers have been evolved hence they can analyze the programs at run time. Since manual model checkers are expensive and time consuming, researchers are working towards automatic model checkers like Bandera [8], Java PathFinder [15], SLAM [1] and FeaVer [16].

Dynamic or on the fly techniques are able to analyze code in the execution time [3,6,14,19]. Dynamic techniques are getting increasing focus because they are highly scalable and do not consume a lot of time in comparison to other techniques. Dynamic analysis can be two types where one type analyzes and detects bugs in the execution time. The other type collects various data regarding the program in the execution time and later analyzes it for possible bug detection.

The higher complexity of concurrent programs makes the use of manual testing to be infeasible. There are numerous available tools to automate the testing of concurrent programs. However, the advent of multi-core systems demands the availability of such automation tools in the mass market. It is important for the researchers as well as the industry people to check the quality of the available tools. Good benchmark programs are important in this regard.

3. Research methodology

This study demands a detail and comprehensive review of concurrent software testing techniques used in academia and industry. Therefore, a systematic review can be an effective solution for the state of research approaches. SR is a key tool for enabling evidence based software engineering as it combines the findings from different sources. It provides a concise summary from the best available evidence with explicit and rigorous methods and synthesizes the result for a specific topic. In a SR, the total review process is documented in a way so that it can be replicated later when necessary. SR can also find out the existing gaps, critical questions that are not yet addressed or answered in the past empirical researches [9]. For the purpose of SR, we followed the steps designed by Kitchenham et. al. [17]. The main steps and sub steps of the SR are shown in below.

Planning the Review

- Identification of the need of SR
- Defining the research questions
- Develop a review protocol
- Evaluate the review protocol

Conducting the Review

- Identification of research
- Selection of primary studies
- Data extraction
- Study quality assessment
- Data synthesis

Reporting the Review

- Draw conclusion
- Consider threats
- Disseminate results

4. Planning the review

4.1. Motivation for a systematic review

A systematic literature review (aka systematic review) is a form of secondary study that uses a well-defined methodology to identify, analyze and interpret all available evidence related to a specific research question in an unbiased and repeatable (to a degree) way [17]. Literature review does not follow any specified search strategy where SR uses specific search strategy. In a systematic review, the identification of research follows an explicit search strategy. On the other hand, in literature review study sources and selections are potentially biased [20]. The quality and worth of a review depend on the scientific methods used to minimize errors and bias. This is the main difference between SR and literature review. In this sense, systematic review is a research method itself.

We have searched several databases to know whether a SR or literature review on the field "concurrent software testing" is already done or not. We searched for the answers of the following questions.

a) Has any SR been conducted on concurrent software testing, if so in which extent?

b) Has any literature review been conducted on concurrent software testing, if so in which extent?

Databases mentioned in Table 4 are searched, to find out the answers these questions. The following string appended with an AND operator with the search string given in Table 4 is used to search the databases.

(("systematic review" OR "research review" OR "research synthesis" OR "research integration" OR "systematic overview" OR "systematic research synthesis" OR "integrative research review" OR "integrative review")) wn KY

Since no result is found from the selected databases, further search was conducted on Google scholar, which result in 23 articles, those are not relevant to this study. We also searched the databases for the available literature reviews on the field. We got few articles those were not relevant with the research questions. As a result, the topic demands to carry out the SR.

4.2. Research questions

Research Questions	Aim
RQ1: What is the state-of-the-art of concurrent software testing in shared memory systems?	To identify the state-of-the-art of concurrent software testing in shared memory systems.
RQ1.1: What are the concurrent software characteristics and bugs?	To identify the concurrent software characteristics and bugs.
RQ1.2: What are the testing techniques, tools, and benchmarks for the tools of concurrent software?	To identify the testing techniques, tools, and benchmarks for the tools of concurrent software.
RQ1.3: What are the techniques and tools for test case generation of concurrent software?	To identify the techniques and tools for test case generation of concurrent software.

4.3. Review protocol

Review protocol contains detailed steps used to carry out the SR. The review protocol aims to reduce possible bias by the researcher while conducting the SR [17]. Different components of the review protocol are discussed in the rest of the section.

4.3.1. Search strategies. The search strategies are important for successful primary study identification. Search strategies define where to search the primary resources and how to search them.

We have decided to use the following 5 bibliographical databases to search the primary resources.

- Inspec (reference database)
- Compendex (reference database)
- IEEE Xplore (full text database)
- ACM Digital Library (full text database)
- Scopus (reference and citation database)

It should be mentioned that Inspec and Compendex cover all IEEE and ACM publications from the year 1994. Hence, we added IEEE and ACM in the database list. Initially, we decided to include ISI web of science in the searching database list. However, comparing ISI web to Scopus we identified that Scopus is the largest reference database. Scopus cover more peer-reviewed journals than ISI web, where ISI web covers more conference articles. A survey on 16 Portuguese universities from 2000 to 2007 showed that both databases almost equally reference the published studies [26]. Therefore, we decide to include Scopus in our bibliographical databases.

An initial trial is performed to find some relevant articles so that we can see different type of keywords and synonyms used in the literature. We have generated the following group of keywords shown in Table 1 to search the articles related to our research questions. We used these groups of strings using AND, OR and NOT logical operators to make the search more precise.

Table1. Search Strings

Concurrent, concurrency, thread*, multithread*, multi-thread*
Software, code, program, concurrent system
Test*, "test case", testing, verify*, check, checking, detect*
Determini*, "non-deterministic", interleav*, bug*, fault, error, race, deadlock, synchronization, atom*, "bug pattern", "Design pattern", antipattern, anti-pattern, benchmark, "static analysis", "Dynamic analysis", "Model checker", "formal analysis", "on-the-fly", tool
"Distributed system", "distributed memory", hardware, machine

4.3.2. Study selection criteria. We read the title, abstract of the articles and include them in the SR based on some criteria (inclusion criteria). If the articles discuss about the

research questions and comply with the inclusion criteria, we included it to the selected articles list. The inclusion criteria are given below:

- The article should be a peer-reviewed article.
- The articles' context should be based on correctness issues of concurrent software.
- The articles should be based on any of the research questions.
- The articles should be available as full text.

If the relevancy of the article is not clear by reading the title and abstract, we marked the article as "unsure" so that it can be discussed later with the other member. Then we decided whether the articles should be included into or excluded from the SR.

4.3.3. Study selection procedure. To select the primary studies, first, we performed a pilot selection in order to measure the uniform understanding of both members. This is important to check whether both members have a consistent understanding about the scope and selection criteria of the study.

In the pilot selection, we randomly selected 60 articles among the searched result. Both team members separately performed the article selection according to their personal understanding. After completion of individual selection, we measured the degree of homogeneous understanding of the members using the Cohen's Kappa Method [7]. If the measure, results in a significant difference of the degree of understanding then the members discussed to reduce the gap in understanding the scope.

The final selection of the articles started when the members were agreed upon a common understanding of the scope and article selection. The total number of articles found was divided into two sets and each of the group members checked one set to select the final pool of articles.

4.3.4. Quality selection criteria. We assessed the quality of the articles following the inclusion criteria, while reading the full text of them. The checklist for assessing the quality of the selected articles is given in Table 2.

Table 2. Quality Assessment Criteria

Criteria	Yes/No/Partially
Does the introduction provide good overview?	
Are the aims and objectives clearly reported?	
Is the context of the research is adequately stated?	
Is the data collection method described properly	
Are the validity threats considered in the study?	
Are the results of the study mentioned clearly?	

4.3.5. Data extraction strategy. Throughout the full text reading of the articles, meta-data and data related to the study are extracted. The forms shown in Appendix B are used for data extraction.

4.4. Review protocol evaluation

We updated the review protocol several times all over the planning phase. The reason behind to change or update review protocol was to accurate and concise formulation of each step in the protocol. Mainly, the research questions, search strings, data extraction strategy and data extraction form were the subject for refinement in the review protocol.

The review team fixed schedule for review meetings. Before meeting, the team members prepared a number of questions about review protocol. One member asked the questions to the other member and the other member tried to answer the questions. In this way, the review team performed the verification of the review protocol in several times.

5. Conducting the systematic review

5.1. Identification of research

We conducted the SR to find as many primary studies as possible related to our research questions by following the unbiased search strategy described in the review protocol. We tried our best to remain free from publication bias and collected research from various databases. At first, a pilot selection procedure was performed to verify how much homogeneous interpretation we have on the research selections. We randomly selected 60 papers from Inspec and Compendex using a primary search string. Then we went through the titles, abstracts of these 60 papers and categorized them as accepted, rejected, and unsure based on the study selection criteria. The result is shown in Table 3.

Table 3. Search Strings

	Accepted	Rejected	Unsure	Total
Member 1	5	42	13	60
Member 2	5	44	11	60
Match	4	32	7	

The review members then calculated the Cohen's Kappa value to measure the degree of common understanding between the members. The value obtained from the calculation indicated that the review members have a substantial agreement between them.

Since we have to maintain a large amount of research papers, it is both times consuming and cumbersome to manually handle the references and associate information of the selected articles. Therefore, we used an automated reference management tool called "Zotero" [30]. It facilitates adding notes, tags, and attachments with each article. It also supports to filter the articles by their tags. We used MS. Excel for documenting the search strings along with their results.

5.2. Selection of primary studies

Searching 5 selected bibliographic databases, we found in total 4352 articles. The results of the search are limited from the year 1989 to 2009. We excluded IEEE and ACM publications from the search results of Inspec and Compendex in order to reduce the duplication. The search results from the different databases are given in Table 4.

Table 4. Statistics of Different Databases Searching

Database	Search String	Articles found
Inspec	(((concurrent OR concurrency OR thread* OR multithread* OR multi-thread*) wn KY AND (code OR software OR program OR {concurrent system}) wn KY AND (test OR {test case} OR testing OR verif* OR check OR checking OR Detect*) wn KY AND English wn LA) AND (determini* OR {non-deterministic} OR interleav* OR bug* OR fault OR error OR race OR deadlock OR synchronization OR atom* OR {bug pattern} OR {Design pattern} OR antipattern OR anti-pattern OR benchmark OR {static analysis} OR {Dynamic analysis} OR {Model checker} OR {formal analysis} OR {on-the-fly} OR tool) wn KY) NOT({distributed system} OR {distributed memory} OR hardware OR machine) wn KY	984
Compendex	Equivalent search string	545
IEEE Xplore	Equivalent search string	540
ACM Digital Library	Equivalent search string	300
Scopus	Equivalent search string	1983
Total Article		4352

Reading titles of 4352 articles, we excluded many articles those are not relevant to our study. We imported the relevant articles into the reference management tool Zotero. In Zotero, we sorted the articles by title, which eases the removal of the duplicated items. We got 929 articles after removing the duplicity. Then the articles are divided into two pools. Each member took a pool to check if the articles comply with the inclusion/exclusion criteria. We read abstracts or both abstracts and conclusions of these articles and selected 210 of them. Though this is separately done by each member, the review members were convinced to their homogeneous interpretation from Cohen's Kappa coefficient value found in pilot selection.

We succeeded to download 192 full text articles. These full text articles were assessed for quality by the quality assessment criteria, which are described in section 5.3. We read the articles in more detail during quality assessment and found several similar articles published from the same authors in different publication channels. We decided to reduce such duplications. In this regard, we gave preference to the Journal articles over conference articles. It should be mentioned that we included very few articles into the study those were relevant to the study but not found by the search strings. Finally, we ended up with 109 unique primary articles that undergo the data extraction procedure. A year wise list of the selected primary articles is given in Appendix A. The referenced articles used throughout the result sections as well as in Appendix are mentioned by the number starting with 'S'.

5.3. Study quality assessment

In this stage, the review team assessed the quality of the selected primary studies based on the quality criteria checklist (Table 2). Both the review members individually performed the quality assessment task.

It should be noted that during the assessment process, review members found some studies those are not well described, characteristics of the concurrent software are not clearly mentioned, and testing methods are not stated specifically. Therefore, the review members had to consult to these issues very frequently.

We assess the quality of 192 primary articles and found that most of the articles did not describe anything about the validity threats.

5.4. Data extraction

We have performed the data extraction according to the form given in Appendix B. While extracting data, we went through the full texts of all the 109 primary articles. Then required data is extracted from the full text articles and documented in MS. Excel worksheets for data synthesis.

5.5. Data synthesis

Brereton et al. [5] identified that the nature of the SRs in software engineering is more or less qualitative. Moreover, there are three types of qualitative data synthesis namely reciprocal translation, refutational synthesis and line of argument [21]. We choose qualitative line of argument data synthesis approach in the SR where individual studies are analyzed first and then a group of related studies as a whole. The synthesized data are presented in the results' section.

6. Results

According to the research questions, we present the state-of-the-art of various issues of concurrent software testing from section 6.2 to 6.7.

6.1. Overview of the studies

The study identifies 109 primary studies. An overview of the selected studies according to the publication channel is presented in Appendix C.

6.2. Concurrent software characteristics

Non-determinism is the most general characteristic of concurrent software describes by many articles [S1,S2,S6, S8,S11,S12,S19-S22,S32,S42,S47-S50,S54,57,S67,S78]. The non-deterministic execution behavior makes a concurrent program different than a sequential program. Non-deterministic execution means that a same input to a concurrent program might result in different execution sequence. Concurrent program holds the non-deterministic characteristics because of the interleaved execution of threads. Due to the interleaved execution of threads, there arise a number of issues like data race, atomicity violation, synchronization defect, deadlock, livelock etc.

With the increase of the number of cores in the multi-core processors, there is an increase in the number of executing threads, which results more interleaving in the execution hence more concurrency bugs. Since thread interleaving is an essence of concurrent program execution, above mentioned concurrency issues are a part of the concurrent software. Some of the studies discussed about the rendezvous nature of concurrent software [S2,S5,S6,S7,S9,S11,S13,S28]. We have observed that the characteristics of the concurrent program are more discussed by the earlier studies of the field.

Another characteristic of concurrent software is its enormous state space which makes exhaustive testing techniques less appropriate to test concurrent software. This problem is called state space explosion. A number of articles mentioned the challenges of state space exploration of concurrent software where some of them worked to solve the problem [S21,S47,S48,S54,S59,S63, S86,S88,S92,S96].

6.3. Concurrency bugs

A bug in a concurrent program might lead to serious consequences like data race, deadlock, livelock, and atomicity violation. The most common bugs addressed are data race, and deadlock. It is found that bugs are also described in the form of patterns like bug patterns and antipatterns. In general sense, bug patterns describe errors those can commonly occur in the program. A bug pattern is a recurring correlation between signaled error and underlying bug in a program [29]. Design patterns are solutions to recurring problems. The solution of a design pattern that is decidedly attached with a negative consequence is called an antipattern. Both design patterns and antipatterns can be source of bugs.

Testing concurrent program is difficult due to the nature of concurrent bug patterns. Therefore researches are going on to identify the concurrency bugs, bug patterns, and antipatterns. Recently, a study [S99] has performed on real world concurrency bug characteristics. This study reported 105 real world concurrency bugs analyzing the bug reports of four popular open source software. It also studied the bug patterns, their manifestation, x strategy and other characteristics. Another work [S51] presented a taxonomy of concurrency bug for Java concurrent programs. A study [S66] used 21 multithreaded bug patterns collected from books and articles, researchers and developers, and their own experience. Another study presented and documented 38 antipatterns [S65] with references of tools that are able to detect them.

In the selected studies, we found 29 articles discussing about the data race, 23 articles focused on deadlock, 8 articles on atomicity violation, 6 articles gave attention on livelock, 3 articles on communication error, 3 articles on bug patterns/antipatterns, 2 articles on violation of assertion, and 2 articles on interference properties of the concurrent program. The other bugs referred on different studies are given in Table 5.

Table 5. Concurrent software bugs

Bugs	Articles	Total
Data races/race condition	S4,S13,S15,S20,S37,S38, S40,S43,S44,S52,S55,S56, S58,S59,S65,S67,S69,S75, S76,S89,S90,S94,S97,S100, S102,S103,S104,S107,S108	29
Deadlock	S18,S23,S26,S28,S31,S32, S33,S37,S39,S40,S41,S42, S44,S48,S56,S59,S65,S75, S76,S94,S99,S106,S108	23
Atomicity violations	S61,S69,S71,S81,S87,S89, S99,S103	8
Livelock	S26,S28,S37,S57,S65,S75	6
Communication errors	S7,S9,S27	3
Bug patterns / antipatterns	S51,S65,S66	3
Violation of assertion	S18,S33,S42	2
Local error, synchronization error	S9	1
Intended non-deterministic behavior, harmful non-deterministic behavior, persistent non-deterministic behavior	S21	1
Dead (unreachable) statements	S27	1
Unnecessary synchronization	S29	1
Data protection	S31	1
Memory errors to protocol invariant violations	S47	1
Starvation, dormancy, incoincidence	S75	1
Access violation bugs	S76	1
Non- deadlock (Order violation)	S99	1
Heisenbugs(unexpected interference)	S105	1
Atomic-set serializability violations	S107	1

Data race is a typical multithreaded bug that occurs due to interference between threads. That means, if more than one thread wants to access shared variables, where at least one thread is a write operation and the threads do not use any synchronization mechanism to prevent simultaneous access [S15]. The bug deadlock occurs if a thread holds a lock and acquires another lock without releasing the first lock, while another thread holds the second lock and wants to acquire the first lock without releasing the second [S40]. Livelock happens when a thread is forcedly inactive and no longer activate [S28]. Both deadlock and livelock also can happen due to the improper way of synchronizing threads. Synchronization means to invoke a lock into threads for mutual exclusion. However, unnecessary synchronization decreases performance of the program [S29]. Atomicity is an essential noninterference property, where an atomic method is not affected by other threads during its execution [S61]. In the atomic method, each statement executes sequentially.

Bug analysis facilitates the programmers to be aware of the bugs while writing code and it inspires the testing tool developers to create or update tools finding more bugs. It is necessary to explore specialized bugs available in different concurrent programming languages.

6.4. Concurrent testing techniques

The concurrent testing techniques can be divided into two major domains, namely static and dynamic. Static techniques check programs statically without executing it. The dynamic techniques need to execute the program and the analysis can be in run time or after the execution.

Static techniques are mainly two types [S92], static analysis techniques, and formal methods. Formal methods are model checking and theorem prove. Model checkers basically work in three steps, modeling, specification, and verification. Model checkers use exhaustive or systematic state space exploration to verify the correctness of the program. It can be both static and dynamic. The dynamic or on the fly model checkers uses systematic state space exploration, and they are more scalable than the static model checkers. However, in general, static analysis techniques are faster and more scalable in comparison to the model checkers.

There are two basic approaches for testing concurrent software discussed in the literature are non-deterministic testing [S19,S30,S78] and deterministic testing [S1,S6,S8,S72,S85]. Since, both approaches describe program testing in the run time; they can be connected with the dynamic testing techniques. In non-deterministic testing, a parallel program P is tested with input X. P is executed several times with input X to see if there is any problem in the program P with the input X. Non-deterministic execution may be used both for testing a concurrent program P and selecting SYN-sequence [S1,S8] (synchronization sequences). SYN-sequences are necessary for various concurrent testing techniques to check if the sequences generate bug or not.

In deterministic execution or forced execution approach a program P is executed in a SYN-sequence S with an input X. Here the SYN-sequence is forcedly executed to check if S is a safe or feasible execution path of P. Though deterministic testing of a concurrent program suffers from several problems [S1], it has advantages over non-deterministic execution. In deterministic testing, if a SYN-sequence S can detect a bug in program P with input X, it can be reproduced later when necessary.

We want to mention that reproducibility of concurrency bug is one of the challenging issues of concurrent testing. It is possible to combine both deterministic and non-deterministic approaches together to take advantage of both [S22]. In the recent years, we have observed several attempt in reachability testing [S8,S60,S64,S67,S91] which is a testing technique combining both deterministic and non-deterministic testing.

Dynamic techniques are faster and more scalable than static techniques. Generally, dynamic techniques demand instrumentation of the programs to make the analysis better. Instrumentation is the insertion of additional instruction into the program code that can guide the execution behavior of the program. Dynamic techniques are built on several techniques like instrumentor, noise maker, replay etc.

Table 6 shows the distribution of static and dynamic testing techniques used in the studies between the years 1990 to 2009. It is clear that until 2002 dynamic techniques were seldom used and static techniques were frequently used. From 2003 to 2005, more attention is given on both techniques separately. However, from 2006 to 2009 the trend is towards the dynamic techniques. It is interesting that some recent studies [S79,S94] proposed to combine both techniques together to take the advantage of both. We found 4 such articles where researchers show how to use both techniques in combined. From Table 6, it is clear that in recent years dynamic testing techniques are getting increased focus.

We found about 51 different testing techniques in the selected studies used for concurrent software testing. Table 7 shows that some studies worked on more than one testing techniques. Another interesting thing is that few articles used similar testing techniques. Most of the articles discussed about different testing techniques, based on different algorithms.

Eraser is one of the most used techniques referred in the articles [S15,S31,S40,S61,S63,S72,S89,S100]. It is a dynamic race detection algorithm based on lock set algorithm [S15]. Eraser uses binary rewriting technique to monitor that all shared memory access follows a

consistent locking discipline. Different tools like ConTest [S89], Helgrind [S100], JPAX [S40,S63], Jnuke [S72], Atomizer [S61] uses Eraser's lockset algorithm to detect data race. The lockset algorithm that is used in Eraser, is powerful algorithm to detect data race [S52,S56,S87]. Lockset with Djit is used to detect data race on the fly [S52]. Multilockset is an improved version of the lockset algorithm that misses fewer races and gives fewer false alarms [S87]. Reachability is another popular testing technique referred in several articles [S8,S60,S64,S67,S91]. Reachability testing is generally used to collect feasible SYN-sequence sets and to execute those sets deterministically [S60]. By this way it combines deterministic and non-deterministic testing [S8], to generate test sequences [S64]. Moreover, it does not demand to save synchronization sequence history [S67].

For testing concurrent program as easy as sequential program, hypersequential programming technique [S17,S21] is used. This technique first serializes the original concurrent program, then tests and debugs as a sequential program, and finally restores the program as its original version. State-space exploration is a powerful verification technique that is used to explore state-space [S18,S48,S88]. State-space exploration systematically or exhaustively searches possible state/behavior of a program for correctness. It checks the behavior during program execution or after execution. A new state-space exploration algorithm combines state less search and two partial order algorithms (persistent set and sleep sets search techniques) for efficiently exploring the state-space [S18]. A tool JPF uses state-less state space exploration search technique with the technique *reduction* to systematically explore different schedules [S48]. The tool Vyrd also incorporates state space exploration search technique [S88]. A structural convenient method Petri-net [S24,S75] is used to model check the safety and liveness properties of Ada program, and modeling the locking objects. Different Petri nets such as Colored Petri Nets graph (CPG) and Colored Petri Nets Tree (CPT) [S10], Time Petri Net and Temporal Logic are used for test suite generation [S10,S19]. Some articles also showed the use of the type system. Type system follows a locking discipline [S44,S58] in the form of a type declaration for preventing data race and deadlock. Another type system, Extended Parameterized Atomic Java (EPAJ) for specifying and verifying atomicity in Java programs for analyzing data race. A number of articles [S48,S83,S87] presented the testing technique based on reduction based algorithm. Reduction technique is used to reduce the size of SYN-sequences or state space to be searched. It uses locks to protect shared variables and to utilize common patterns of synchronization [S48]. Another reduction technique based on call stack criteria [S83] reduces the test suite size.

Due to space limitations, we could not describe all of the techniques. Therefore, we have briefly described them in Table 7. It is difficult to find out a trend of techniques used in different years from Table 7. A large number of techniques are identified. A reason to this is that an existing technique is used with a modification or extension or in combination of other techniques and given a different name. We found several examples of such cases. However, from the analysis it is found that different race detection techniques, atomicity violation, lockset analysis, reachability testing, coverage based techniques, reduction based technique, healing mechanism, type based techniques are the most widely used testing techniques.

Table 6. Static & Dynamic Testing Techniques by Year

Year	Static		Dynamic				Both
	Static analysis	Formal method/ Model checking	Dynamic analysis	State-Space Exploration/ Model checking	Noise maker	Replay	
1990	S2		S3				
1991			S4				
1994			S7,S8				
1996			S12				
1997	S16		S15	S18			
1998		S21	S20,S22			S23	
1999	S28,S29	S24					
2000	S30			S33,S34		S32	
2001	S38			S39,S40		S36	S37
2002	S44,S45		S46	S47,S48	S42		
2003	S56,S58	S53	S49,S51,S52,S57	S54, S59	S55		
2004	S65,S66		S61,S62	S63			
2005	S69,S71,S77		S67,S80	S74,S76		S72	S75,S79
2006			S81,S84,S85,S87,S88				
2007	S90		S89				S94
2008			S100,S103,S104	S96,S101			
2009	S106,S107					S105	

Table 7. Testing techniques used in the studied articles.

Techniques	Year	Description	Articles
Path analysis	1990	A path analysis testing method is described where the execution behavior of a concurrent program are modeled by flowgraph and rendezvous graph for static and dynamic structure of the program respectively.	S2
	2006	A novel method BPEL(Buisness Proceess Execution Language) test case generation used path analysis approach	S82
	2008	UML activity diagram test case generation framework used path analysis approach	S102
Behavioral testing	1990	Two behavioral testing (passive and forced execution) paradigms are described in an efficient way that can provide reproducible information for parallel software validation.	S3
Fork-join parallelism	1991	A new access history protocol for detecting data races dynamically in executions of programs with nested fork-join parallelism graph	S4
DEMT	1993	A deterministic execution mutation testing (DEMT) is described that is a combination of deterministic execution and mutation based testing.	S6
OSC (ordered sequence criteria)	1994 1996	A new technique for test coverage measurement of concurrent program. It concerns the execution order of the concurrent statements	S7*,S12
	1999	A concept of OSC is applied on EIAG for test case generation	S27
Reachability testing	1994	A technique that combines deterministic and non-deterministic testing.	S8*
	2004	Reachability testing collects feasible synchronization sequence sets and implements deterministic testing on them	S60
	2004	The reachability testing algorithm is applied on a tool RichTest to generate test sequences dynamically.	S64
	2005	A new on the fly reachability testing algorithm that does not save history of synchronization sequences that have already exercised.	S67
	2007	A new reachability testing based on extended synchronization sequence (ESYN-sequence) to measure the test coverage rate	S91
Rendezvous style mechanism	1996	Four testing criteria (Static analysis, Deterministic testing, testing based on execution trace, petri nets testing based on controlled execution) based on the rendezvous for concurrent programs are proposed. Rendezvous means process synchronization and inter task communication.	S13
Coverage criteria	1996	A new coverage criteria that combine sequential (statement and path coverage criteria) and concurrent (ordered and complementary ordered coverage criteria) aspect of the program	S14
	2007	Seven coverage criteria hierarchy are discussed	S93
Eraser	1997	Eraser is a race detection algorithm uses binary rewriting technique to monitor that all shared memory access follow a consistent locking discipline (lockset algorithm)	S15*
	2007	Eraser is used to implement the race detector of the dynamic tool ConTest	S89
	2008	Helgrind uses the Eraser's algorithm	S100
	2004	A tool Atomizer uses Eraser's lockset algorithm and Lipton's reduction theory	S61
	2000	The eraser algorithm is used for developing the data protection rules of the tool Visual Threads.	S31
	2001	Eraser data race algorithm has been implemented in JPAX	S40,S63
	2005	In Jnuke framework on top of JVM, Eraser is used to detect data race.	S72
SnapShot	1997	The SnapShot mechanism instruments the program with SnapShot statements during program execution that are used to generate a trace and then analyze the trace.	S16*
Hypersequential programming	1997 1998	Hypersequential programming makes concurrent programming testing and debugging as easy as sequential programming	S17,S21
State space exploration algorithm	1997	A new state space exploration algorithm combines state less search and persistent set and sleep sets search techniques, built upon existing partial order methods for efficiently exploring the state space	S18
	2002	A state space exploration tool JPF uses state less search with the reduction to systematically explore different schedules	S48
	2006	A refinement checking tool Vyrd incorporates the state space exploration technique	S88
Specification based methodology	1998	Specification based method uses sequencing constraint and combination of deterministic and non-deterministic testing	S22
	2006	A specification based testing of concurrent program with generated test sequences from Statechrats. A state space exploration technique with PMSs (possible macro steps) in Statecharts is developed to generate interleaving.	S86
Petri nets	1995	CPT and CPG methods are used to test case generation by using the equivalent marking technique on CPN	S10
	1997	Petri nets by introducing scenario equivalence is used to formalize the scenario based hypersequential programming	S17
	1998	A dual language approach (temporal logic and time petri nets) is used for real time software design and analysis	S19
	1999	High level petri nets uses structural methods and model checking to verify liveness and safety properties of Ada programs	S24
	2005	Petri-nets-a convenient mechanism for modeling the locking object	S75
CSP	1999	CSP (Communicating sequential process) is designed to handle systems that communicate only via explicit messages	S26
FLAVERS	1999	It uses data the flow analysis algorithm TFG (trace flow graph) based on CFGS (control flow graph algorithm) to check error prone thread communication patterns in Java.	S28
Escape analysis	1999	A new interprocedural algorithm uses a data flow analysis framework and maps escape analysis to a simple reachability problem over a connection graph abstraction	S29
	2002	To avoid inserting calls in the scheduling function escape analysis is used.	S42
	2006	A dynamic escape analysis determines when an object becomes accessible to other threads. It used to supports	S87

		the reduction based and block based algorithm.	
SFSM (Synchronizing finite state machine)	2000	The behavior of each thread is described by Synchronizing finite state machine. The transition information for thread synchronization is expressed by the implementation-language-dependent synchronization mechanism.	S30
Bandera	2000	A component based model extractor using finite state verification techniques such as model checking	S34
	2003	JPF uses slicing tool of the BANDERA toolset. BANDERA also supports partial symbolic evaluation.	S59
EE method	2001	EE (Entity Event) method uses event graph for systematically deriving partial oracles for the concurrent program	S35
Automata based testing	2001	A kind of deterministic execution method	S36
TLVA	2001	TLVA (three valued logic) used for verifying safety properties of Java concurrent program	S39
Scheduling function	2002	Scheduling function works as context switch	S42
Fault insertion techniques	2002	It is related to mutation analysis with the goal of measuring testability of a concurrent program. The technique is used to detect race condition.	S43
Seeding techniques	2002	Seeding the program with sleep(), yield(),priority() primitives increase the probability of finding faults in Java concurrent program	S46
Reduction	2002	Reduction exploits a common pattern of synchronization, namely the use of locks to protect shared variables.	S48
	2006	A new reduction technique based on call stack criterion that reduces the test suite size	S83
	2006	for runtime analysis of atomicity reduction based algorithm is used	S87
SYN-sequence selection strategy for dynamic testing	2003	The strategy is based on OBJSA net/CLOWN specifications that support components based analysis and incremental development of specification with good reusability and modularity.	S49
All uses coverage	2003	A structural testing based on all uses criteria (all-branches and all statements criteria)	S50
Djit and Lockset algorithms	2003	These two are very powerful algorithms for on the fly race detection	S52
	1997	Eraser uses the lockset based algorithm to detect data race.	S15
	2003	lockset algorithm is used in RacerX to detect deadlocks and races	S56
	2006	An improved version of lockset algorithm called multilockset is used that misses fewer races and reports fewer false alarms	S87
	2008	Helgrind is an implementation of lockset based algorithm that observes each shared memory location should be protected by lock and happens before relation that defines a partial ordering of events in the distributed system	S100
Symbolic execution	2003	A well-known program analysis technique that represents program variables with symbolic values	S54
Heuristics	2003	Two level scheme heuristics (biased and scheduling heuristics) are used to detect race. Scheduling heuristics uses seeding testing technique	S55
Structural heuristics	2004	Based on branching structure, thread interdependency structure, abstraction structure to show control flow properties of a program for model checking	S63
Type system	2002	A new static type system follows a locking discipline in the form of a type declaration for preventing data race and deadlock	S44
	2003	A lock based synchronization discipline in which programmers assign a lock to each data object	S58
	2005	A static analysis technique capable of verifying atomicity by applying reduction to a program abstraction	S71
	2005	A type system, Extended Parameterized Atomic Java (EPAJ) for specifying and verifying atomicity in Java programs that combines Flanagan and Qadeer' s [11] atomicity type system for analyzing data race	S69
Multiplexer	2005	A method based on a multiplexing algorithm to correctly order events of multiple threads for model checking	S74
Coverage testing	2005	Based on Synchronization coverage testing	S77
	2008	A coverage based testing using block based coverage modeling and coverage criteria	S97
Generic analysis	2005	A graph-free static analysis which is also applicable to dynamic analysis	S79
Aspect oriented programming (AOP)	2005	A relatively new technology for instrumentation	S80
	2006	The aspects used to control the value returned for the read of shared variables to deterministically execute test cases	S85
	2008	AOP is used as a technique for monitoring and controlling threads in a transparent and modularized way.	S98
Runtime atomicity detection technique	2006	Based on commit-node algorithms, which can check conflict atomicity and view atomicity	S81*
	2006	This technique is based on reduction and block-based algorithms for runtime analysis of atomicity. To improve the algorithms two other algorithm: multilockset algorithm and happens before analysis also incorporated	S87
CHESS	2006	Thread scheduling based on two techniques ;Iterative context bounding and partial order reduction	S84
	2009	CHESS is used to thread scheduling and searching possible thread interleaving with its two components, the scheduler, and the search module.	S105
CSSAME and AOP	2006	CSSAME is an intermediate compiler form that analyzes explicitly parallel shared-memory programs. AOP is used to control CSSAME execution.	S85
Healing mechanisms	2007	A set of techniques: 1.problem detection (based on eraser algorithm). Problem localization (developing an oracle specialized on a particular class of concurrent bug), 3. Problem healing (by an appropriate synchronization) and 4, healing assurance (by formal verification)	S89
	2008	A novel algorithm AtomRace incorporates a self healing architecture for detecting and healing data race and atomicity violations on the fly in Java	S103
MUVI	2007	MUVI (Multi-Variable Inconsistency) combined static analysis and data mining techniques. It is also extended by two classic race detection methods (lock-set and happens-before) to detect multi-variable related data races.	S90
TestCon	2007	The TestCon method combines code inspection, the automated static analysis tools FindBugs and Jlint, and dynamic analysis using ConAn	S94

An integration method	2008	The method combines testing techniques with model checking techniques of Java PathFinder (JPF). The method is based on localization of the search performed by JPF. A test suite generation technique, UniTESK with state space exploration technique also used with JPF.	S96
ThreadControl-ForTests	2008	A Generic Approach that ensures test assertion is performed in a safe moment using AOP to avoid in correct results in the automated multithreaded test execution.	S98
UML activity diagram	2008	A Transformation-based approach Generates Scenario-oriented Test Cases for Concurrent Applications that are modeled by UML Activity Diagrams	S95
	2008	Unified modeling language based testing that can be used to testing Java program for data race and inconsistencies. UML can also use to test generation and test oracle.	S102
AtomRace	2008	A novel algorithm for dynamic race detection using noise injection technique that forces different legal interleaving for particular executions of a test in order to increase the concurrent coverage	S103*
Extended dynamic slice	2008	Extended dynamic slicing algorithm is developed to consider RAW, WAW and WAR dependences to capture data race in the dynamic slices	S104
Deadlock detection algorithm	2009	It uses four static analyses namely, a call-graph analysis, a may-alias analysis, a thread-escape analysis, and a may-happen-in-parallel analysis	S106*
Random isolation	2009	New abstraction principle allows strong updates to be performed on the abstract counterparts of each randomly-isolated object	S107
Fractional permission system	2009	To assure the correct usage of mutual-exclusion locks. Permission is an abstract token associate with some portion of a program permitting certain operations	S108
Total 51 techniques			

* indicates new (original attempt) techniques.

6.5. Testing tools

The study identified 38 different testing tools from the selected studies that are shown in the Table 8. The tools are presented with their supported programming language. It is found that many tools are developed for the language Java followed by C/C++. Some of the promising tools are discussed in the rest of this section.

The tool JPF (Java PathFinder) integrates model checking, analysis and testing. It is built on top of java virtual machine. It is developed at the NASA Ames research center. It has two components: state generator and analysis algorithm [S63]. The race detection algorithm Eraser has been implemented in JPF to identify data race and deadlock [S59]. It uses partial order and symmetry reduction, slicing, abstraction, and runtime analysis techniques to reduce the state space [S59]. It supports various heuristics such as testing related coverage, which can guide model checker's search ability [S54]. JPF performs verification of the concurrent system by exploring a directed graph, called state space, and presents the behavior of the system [S96]. JPF is described as a translator, which translates a Java program into PROMELA model and the model checked by SPIN model checker [S33]. We got 7 articles that discuss about JPF. ConTest is one of the most popular tools used in several studies [S46,S51,S62,S77,S89]. ConTest consists of a number of components like instrumentor, a noise generator, a replay and coverage component, listener architecture. It is developed by IBM, which implements the popular dynamic race detection algorithm Eraser [S15]. It works for Java concurrency bugs.

Another popular static analysis tool is FindBugs. It is a bug pattern detector [S66]. ConAn (Concurrency Analyzer) is specifically designed to test component those are accessed by multiple threads. ConAn can successfully detect liveness and safety related errors [S57]. The static tool Jlint was specially designed to detect deadlock in synchronization block [S37]. Jlint detected several deadlock and synchronization errors from 38 antipatterns discussed in [S65]. TestCon is a method for validating concurrent Java components. TestCon proposes to use static analysis tools FindBugs, Jlint and dynamic tool ConAn.

Eraser is a dynamic race detector tool that checks all shared memory accesses; follow a consistent locking discipline [S15]. The idea from Eraser's lock set algorithm and Lipton's theory [18] of reduction is used in a dynamic atomicity checker Atomizer [S61]. A tool [S17,S21] is developed to support hyper sequential programming based on a sequential execution history called 'scenario' for C language. Wolf is a model checker that uses a symbolic BDD based algorithm, which enable faster image computation for software models [S76]. RacerX [S56] is a static analysis tool that uses flow sensitive, interprocedural analysis to detect both race conditions and deadlocks. TDCAda is an Ada source transformation tool [S6,S22]. FLAVERS follows the data flow analysis algorithm TFG (trace flow graph based on CFG (control flow graph)) to model the communication among threads in Java programs [S28].

Jnuke is the tool that first integrates static and dynamic analysis in a novel way [S79]. The goal of Jnuke is to develop a framework for both static and dynamic analysis of Java's byte code [S72]. It uses Eraser algorithm to detect data race.

Bandera [S34] is a model checking tool that extracts model from program source code. Bandera slices a program code where all irrelevant elements can be removed [S59]. A tool Visual Threads [S37] is based on powerful race/deadlock detection algorithm, can work in C, C++, FORTRAN and Java.

Most of the identified tools detect concurrency defects of Java programming language. A number of tools found for C/C++ where a few tools are identified for Ada, FORTRAN and Microsoft .NET languages.

Table 8. Concurrent software testing tools and their supported language

Language	Tool	Focus	Articles
Java	JPF	A dynamic model checker to detect deadlock, data race and assertion violation	S33,S48,S54,S59, S63,S72,S96
	ConTest	A dynamic tool for detecting deadlock data race, atomicity violation etc	S46,S51,S62,S77,S89
	FindBugs	A bug pattern based static analysis tool	S65,S66,S75,S94
	ConAn	It is good at detecting liveness and safety errors and deadlock	S57*,S75,S78,S94
	Jlint	A static analysis tool for deadlock and race detection	S37,S65,S75,S94
	Jnuke	The first tool that integrates static and dynamic analysis in novel ways	S72,S79
	RaceFinder	It is used to detect race condition related bugs	S55,S62
	Empire	To detect atomic serializability violation and data race	S107
	JADE	A deadlock detection tool	S106
	A tool based on atomrace algorithm	Detect data race and atomicity violation	S103
	Toc4j	Check data race and inconsistency of a concurrent program	S102
	RichTest	A dynamic tool that can check deadlock and assertion violation	S67
	Atomizer	An atomicity checker	S61*
	Rstest	Detect deadlock and assertion violation	S42
	PathExplorer(JPAX)	Data race and deadlock detector	S40
	Rccjava	An effective tool for identifying race in large scale software	S38
	Rivet, ESC/Java	ESC/Java can detect potential race condition	S37
	Bandera tool set	Bandera automatically extracts finite state model from source code to check correctness properties.	S34,S59
	DJVM	A replay tool for distributed and multi-threaded applications that provides a deterministic replay of a non-deterministic execution	S32*
	CSP Case tools	Verify concurrent software design against deadlock and livelock related dangers	S26
	DejaVu	It provides deterministic replay of a program's execution	S23
	JaDA	Able to collect SYN-sequence during program execution and race analysis	S20*
C	Scenario based hypersequential programming tool	Detect and remove harmful non-deterministic behavior of concurrent programs	S17,S21
	Eraser	A dynamic race detector	S15*
	Wolf	A model checker that can detect access violation bugs, data races, and deadlocks	S76
	Promela/SPIN model checker	Verification complexity of model checkers in terms of uncontrolled concurrency	S101
C (Linux, FreeBSD)	RacerX	Static detection of race condition and deadlock using flow sensitive, inter procedural analysis	S56
Ada	TDCAda	A monitoring tool for applying OSC (ordered sequence criteria)	S6,S22
Java, Ada	FLAVERS	Flavers follows trace flow graph (TFG) based on the control flow graph (CFG) to model the communication among threads	S28
C, C++	CMC	It is used to find bugs in network protocol implementations	S47
	Verisoft	It systematically explores the state spaces	S18
C, C++	MUVI	It detects inconsistent update and multi variable access bug like data race	S90
C++	MultiRace	It detects data race, which combines improved versions of two powerful dynamic algorithms named Djit and Lockset.	S52*
C, C++, FORTRAN Java	Visual Threads	A dynamic checker that catches deadlock, data race, and potentially hazardous locking scheme. Data protection rules of this tool are based on the algorithm and implementation of the eraser tool	S31,S37
POSIX thread API	Helgrind	Open source dynamic race detection tool	S101
UNIX C	A monitoring tool	A monitoring tool for applying the test criteria OSC	S12
All .NET language	CHESS	For finding and reproducing heisenbugs (unexpected interference among threads)	S105
C++, Java	VyrdMC	A runtime refinement model checker	S88
Total 38 tools			

* indicates new (original attempt) tools

6.6. Test case generation

Test case generation is another challenging issue of concurrent software testing. The rendezvous nature of the concurrent programs needs huge amount of test cases in comparison to sequential programs. Automated software test case generation is essential because concurrent programs are quite complex to analyze manually. Automated test case generation needs to select different execution traces and then the quality of the trace is checked to narrow down the number of traces. It is not possible to test every single trace and doing such will be a primary obstacle toward the scalability of testing. In order to narrow down the number of selected traces, different techniques are applied. We have found about 11 (similar techniques are not counted separately) different techniques and 8 tools for automated test case generation. The test case generation techniques and supporting tools for test case generation are given in Table 9 and Table 10.

Event InterAction Graph (EIAG) is used to generate test case for concurrent programs [S9,S11,S27]. A new test case generator tool, TCgen (test case generator) is developed based on EIAG for Ada in [S9]. Another technique called ISTC (Interaction Sequences Testing criteria) is applied with EIAG [S27]. Labeled Transition System (LTS) is used to select the test path, but it has state explosion problem. An Annotated Labeled Transition System (ALTS), which is based on ALTS reduced algorithm, is used for test case generation [S41].

By using Reachability Graph (RG), ALTS solves the state explosion problem of LTS. RG is a common approach for selecting test sequences from a set of LTSs. RG can be derived through a global LTS and then force execution is performed according to paths selected from the graph. The tool TSG (an incremental testing tool) and TDCAda for deterministic execution are used to an empirical study [S41]. Another approach to effectively generate a small set of test sequences combined RG with four another methods based on the hot spot prioritization and topological sort [S70]. Both RG and stubborn set technique for generating test sequences were applied in a tool, which can generate automata [S86].

We have found three Petri net techniques: Colored Petri Nets graph (CPG) and Colored Petri Nets Tree (CPT) [S10], Time Petri Net and Temporal Logic [S19]. A tool TSG (Test Suites Generator) is based on (CPG) and (CPT) [S10]. HSP is a Hypersequential Programming development tool by Toshiba software laboratories [S25]. An UML approach is applied on the tool SAL and a transformation-based approach is applied for generating scenario-oriented test cases from UML Activity Diagrams for Concurrent Applications [S68,S95]. A reachability testing [S64], which can generate on the fly test sequence is applied in a tool called RichTest.

Some other techniques and associated tools are found such as the Predicate Sequencing Constraint Logic (PSCL) [S73], Extended Control Flow Graph (XCFG) using a tool BoNuS to solve the constraint of the test paths and generate test cases [S82], and a new reduction technique based on the call stack coverage criteria [S83]. For call stack coverage analysis, a Java Virtual Machine Tool Interface (JVMTI) is built [S83].

Table 9. Test case generation techniques

Techniques used to generate test case	Articles
EIAG (Event InterAction Graph)	S9,S11,S27
ISTC (Interaction Sequences Testing criteria)	S27
CPG (Colored Petri net Graph) and CPT (colored Petri net tree)	S10
Time petri net and logic specifications(temporal logic)	S19
Test case generation from hypersequential programming	S25
A new RG named ALTS (Annotated labeled transitions systems)	S41
RG (reachability graph)	S70
RG and stubborn set technique	S86
Reachability testing	S64
UML approach	S68
A transformation-based approach to generating scenario-oriented test cases from UML activity diagrams	S95
A predicate sequencing constraint logic (PSCL)	S73
Extended control flow graph(XCFG)	S82
A new reduction technique based on the call stack coverage criterion	S83

Table 10. Test case generation tools

Tools for test case generation	Articles
TCgen (test case generator)	S9,S27
TSG (Test Suites Generator)	S10,S41
HSP (Hyper Sequential Programming) Development Tool	S25
A tool that generates automata	S86
RichTest	S64
SAL (Symbolic Analysis Laboratory) model checker	S68
A test case generation tool	S73
BoNuS	S82

6.7. Benchmark

Through the SR, we identified a number of benchmarks used for evaluating different tools to measure their testing capabilities. We identified 5 well known benchmarks, used among 14 articles. A tool DejaVu, based on an idea of logical thread schedule used a benchmark named SPLASH-2 (Stanford Parallel Applications for Shared Memory) [S23]. This benchmark was also used in [S104]. A race detector rccjava [S38] has evaluated on SPEC benchmark. A synthetic, multithreaded, client-server benchmark [S32] was tested in a new tool DJVM. A Java benchmark suite was created and used by ConTest in [S62,S92]. The tool Empire [S107] also used some programs from this benchmark. The Java Grande Benchmark Suite [S72,S81,S106] was used by JPF and JADE. JADE also tested by a Successive Over-Relaxation

(SOR) benchmark [S106]. In [S79], they create their own benchmark and tested on Jnuke tool. Another benchmark, SPEC also used in rccjava. There are some other benchmark applications that are used in [S52,S61,S69]. It should be noted that, most of the benchmarks are based on Java Language. We found only one benchmark for C++ [S52].

7. Discussion

This systematic review, maps a number of important issues in the area of concurrent software testing. The primary articles are selected in the context of the correctness of concurrent software for shared memory systems. Since the field is quite large, further systematic reviews are possible toward more specific directions such as static analysis techniques, model checking, and dynamic techniques etc.

As a whole, we found that non-determinism is the key characteristic of concurrent software. The non-deterministic execution behavior of the program results in a bug that may occur in one program execution instance and may disappear on the next program execution instance. Among the concurrency bugs data race, deadlock, livelock, and atomicity violation are the most discussed bugs in the articles. The study found in total 66 articles discussing about these four bugs. Data race is the most discussed bug that is mentioned in 29 articles. In recent years, some studies focused towards bug patterns and antipatterns discussing more refined level reasons of bugs in the concurrent programs.

The study results show that among different testing techniques, dynamic techniques are getting increased focus. The study identified 51 different techniques. Some of the commonly used techniques in the studies are Eraser, Reachability testing, state space exploration, escape analysis, etc. The systematic review identified 38 different testing tools. Most of the tools are developed for Java language followed by C/C++ and Ada. Some studies used well known benchmarks such as SPLASH-2, Java Grande suite, SPEC etc. to evaluate the effectiveness of the tools. However, a number of articles are found where researchers used their own developed benchmarks without providing details about the benchmark programs.

The study found 15 articles discussing about the test case generation techniques such as EIAG model, Petri net, RG model and so on. Besides the techniques, a number of tools for automated test case generation like TCgen, TSG, BoNuS, etc. are identified.

It is found that the tools and techniques are developed in combination of various techniques. It might be wondering why the tools and techniques combine various techniques. This is necessary, because concurrent software testing has many issues like SYN sequence collection/generation, instrumentation, noise making, replay that need to be considered when developing tools and techniques.

According to the research questions, we have mapped a number of concurrent software testing issues in the selected studies through a systematic review. We believe the results of the SR will help the researchers for further investigations on this field.

7.1. Threats to the validity

Validity factors more or less affect every research study. The results of a research become more accurate and acceptable if it considers the validity threats. We observed couple of threats during the SR process.

Though we carefully developed the search strings, some of the related articles are not found through the search strings. We included few such studies that are not found from the searches. While collecting the full text articles, we could not collect about 18 articles. It is possible that some related articles are not included to the study due to a certain degree of different understanding between the members. However, the pilot selection of the articles shows that the members had a good consistency in their understanding about the study selection criteria.

We strongly believe that the study result is generalized in the context of the correctness of concurrent software for shared memory system.

8. Conclusion

This systematic review investigated the state-of-the-art of concurrent software testing from the year 1989 to 2009. We used a structured approach for this review as detailed in section 3. We have identified the characteristics of concurrent programs, bugs, bug patterns, antipatterns, different testing techniques, testing tools, different techniques and tools for test case generation and benchmarks used to evaluate the capability of the tools/techniques. We have classified different testing techniques according to their analysis behavior. The testing tools and benchmarks are described according to their supported programming languages. The results also show the current trend of research in the area of concurrent software testing. We strongly believe that the results present in this study will be helpful for the researches and practitioners working in this field.

We have observed that many techniques are developed in combination of different techniques. In recent years researchers combined dynamic and static techniques together in order to take advantages of both. Future works can be undertaken exploring the relative advantages of different testing techniques that can inspire more hybrid techniques.

9. References

[1] T Ball, R Majumdar, T Millstein, SK Rajamani, "Automatic predicate abstraction of C programs," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM Press: New York, 2001; 203–213.

[2] M. A. Bauer, "High performance computing: the software challenges," *In Proceeding of the 2007 international Workshop on Parallel Symbolic Computation (London, Ontario, Canada, July 27 - 28, 2007). PASCO '07*. ACM, New York, NY, 11-12.

[3] Y Ben-Asher, Y Eytani, E Farchi, "Heuristics for finding concurrent bugs," *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003) PADTAD Workshop*, April 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003.

[4] C Boyapati, R Lee, M Rinard, "Ownership types for safe programming: Preventing data races and deadlocks," *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM Press: New York, 2002; 211–230.

[5] P. Brereton, B. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of Systems and Software*, vol. 80, Apr. 2007, pp. 571-83.

[6] Jong-Deok Choi and H. Srinivasan, "Deterministic replay of Java multithreaded applications," *Proceedings of SPDT 98. Parallel and Distributed Tools, Symposium, 3-4 Aug. 1998*, New York, NY, USA: ACM, 1998, pp. 48-59.

[7] J. Cohen and others, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, 1960, pp. 37–46.

[8] JC Corbett, M Dwyer, J Hatcliff, C Pasareanu, S Robby, S Laubach, H Zheng, "Bandera: Extracting finite-state models from Java source code," *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. ACM Press: New York, 2000.

[9] T. Dybaç[a] , T. Dingsøyr, G.K. Hanssen, Applying systematic reviews to diverse study types: an experience report, *in Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM'07)*, IEEE Computer Society, Madrid, Spain, 2007, pp. 225–234.

[10] Y. Eytani, K. Havelund, S. Stoller, and S. Ur, "Towards a framework and a benchmark for testing tools for multi-threaded programs," *Concurrency and Computation Practice & Experience*, vol. 19, Mar. 2007, pp. 267-79.

[11] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03),* USA: ACM, 2003, pp. 338-49.

[12] P. Hansen, "Testing a multiprogramming system," *Software - Practice and Experience*, vol. 3, Apr. 1973, pp. 145-50.

[13] P. Hansen, "Reproducible testing of monitors," *Software - Practice and Experience*, vol. 8, Nov. 1978, pp. 721-9.

[14] J. J. Harrow, "Runtime checking of multithreaded applications with visual threads," *SPIN Model Checking and Software Verification Lecture Notes in Computer Science*, vol. 1885, Springer: Berlin, 2000; 331–342.

[15] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, 2000, pp. 366–381.

[16] G. J. Holzmann, MH Smith, "Software model checking: Extracting verification models from source code," *Proceedings of the International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV)*. Kluwer: Dordrecht, 1999; 481–497.

[17] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," *Software Engineering Group, School of Computer Science and Mathematics, Keele University, and Department of Computer Science, University of Durham*, 2007.

[18] R.J. Lipton, "Reduction: A method of proving properties of parallel programs," *Communications of the ACM*, vol. 18, 1975, pp. 717-721.

[19] Y. Malaiya, N. Li, J. Bieman, R. Karcich, B. Skibbe, and S. Tek, "Software test coverage and reliability," *Colorado State University, Fort Collins, Colorado,* 1996.

[20] C.D. Mulrow and D. Cook, *Systematic reviews: synthesis of best evidence for health care decisions*, Amer College of Physicians, 1998.

[21] G.W. Noblit and R.D. Hare, *Meta-ethnography: synthesizing qualitative studies*, Sage Publications Inc, 1988.

[22] R. Peri, "Software development tools for multi-core/parallel programming," *in Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (Seattle, Washington*), *PADTAD '08*. ACM, New York, NY, 1-1.

[23] C Von Praun, T Gross, "Static conflict analysis for multi-threaded object-oriented programs," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press: New York, 2003; 115–128.

[24] K. Tai, "On testing concurrent programs," *Proceedings of COMPSAC 85. The IEEE Computer Society's Ninth International Computer Software and Applications Conference, 9-11 Oct. 1985*, Washington, DC, USA: IEEE Comput. Soc. Press, 1985, pp. 310-17.

[25] J. Takahashi, H. Kojima and Z. Furukawa, "Coverage Based Testing for Concurrent Software," *in Proceedings of the 28th international Conference on Distributed Computing Systems Workshops - Volume 00 (June 17 - 20, 2008). ICDCSW*. IEEE Computer Society, Washington, DC, 533-538.

[26] E.S. Vieira and J.A. Gomes, "A comparison of Scopus and Web of Science for a typical university," *Scientometrics*, pp. 1–14.

[27] Lei Wang, Jia-yong Fang, and Cheng-jin Gao, "Parallel test tasks scheduling on multi-core platform," *IEEE AUTOTESTCON 2008, 8-11 Sept*. 2008, Piscataway, NJ, USA: IEEE, 2008, pp. 504-7.

[28] C.D. Yang and L.L. Pollock, "All-uses testing of shared memory parallel programs," *Software Testing Verification and Reliability*, vol. 13, 2003, pp. 3-24.

[29] L. Yu, J. Zhou, Y. Yi, P. Li, and Q. Wang, "Ontology Model-Based Static Analysis on Java Programs," Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference-Volume 00, IEEE Computer Society Washington, DC, USA, 2008, pp. 92-99.

[30] Zotero. [Online].Viewed 2009 August 12. Available: http://www.zotero.org/

Appendix A. The studies referred in systematic review.

[S1] K. Tai, "Testing of concurrent software," *Proceedings of the 13th Annual International Computer Software and Applications Conference (Cat. No.89CH2743-3), 20-22 Sept. 1989*, Washington, DC, USA: IEEE Comput. Soc. Press, 1989, pp. 62-4.

[S2] R. Yang and C. Chung, "A path analysis approach to concurrent program testing," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1990, pp. 425-32.

[S3] J. Walker, "A proposed parallel software testing paradigm," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1990, pp. 167-72.

[S4] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1991, pp. 24-33.

[S5] R. Taylor, D. Levine, and C. Kelly, "Structural testing of concurrent programs," *IEEE Transactions on Software Engineering*, vol. 18, Mar. 1992, pp. 206-15.

[S6] R. Carver, "Mutation-based testing of concurrent programs," New York, NY, USA: IEEE, 1993, pp. 845-53.

[S7] E. Itoh, Y. Kawaguchi, Z. Furukawa, and I. Ushijima, "Ordered sequence testing criteria for concurrent programs and the support tool," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1994, pp. 236-45.

[S8] Gwan-Hwan Hwang, Kuo-Chung Tai, and Ting-Lu Huang, "Reachability testing: an approach to testing concurrent software," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1994, pp. 246-55.

[S9] T. Katayama, Z. Furukawa, and K. Ushijima, "Event interactions graph for test-case generations of concurrent programs," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1995, pp. 29-37.

[S10] H. Watanabe and T. Kudoh, "Test suite generation methods for concurrent systems based on coloured Petri nets," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1995, pp. 242-51.

[S11] T. Katayama, Z. Furukawa, and K. Ushijima, "A method for structural testing of Ada concurrent programs using the Event Interactions Graph," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996, pp. 355-64.

[S12] E. Itoh, Z. Furukawa, and K. Ushijima, "A prototype of a concurrent behavior monitoring tool for testing of concurrent programs," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996, pp. 345-54.

[S13] T. Shih, Chi-Ming Chung, Ying-Hong Wang, Ying-Feng Kuo, and Wei-Chuan Lin, "Software testing and metrics for concurrent computation," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996, pp. 336-44.

[S14] M. Factor, E. Farchi, Y. Lichtenstein, and Y. Malka, "Testing concurrent programs: a formal evaluation of coverage criteria," Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996, pp. 119-26.

[S15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, vol. 15, Nov. 1997, pp. 391-411.

[S16] M. Factor, E. Farchi, and S. Ur, "Rigorous testing using SnapShot," Los Alamitos, CA, USA: IEEE Comput. Soc, 1997, pp. 12-21.

[S17] N. Uchihira and H. Kawata, "Scenario-based hypersequential programming: concept and example," Los Alamitos, CA, USA: IEEE Comput. Soc, 1997, pp. 277-83.

[S18] P. Godefroid, "Model checking for programming languages using VeriSoft," *Proceedings of POPL '97: 24th ACM SIGPLAN-SIGACT, 15-17 Jan. 1997*, New York, NY, USA: ACM, 1997, pp. 174-86.

[S19] I. Ho and Jin-Cherng Lin, "A method of test cases generation for real-time systems," Los Alamitos, CA, USA: IEEE Comput. Soc, 1998, pp. 249-53.

[S20] A. Bechini and K. Tai, "Design of a toolset for dynamic analysis of concurrent Java programs," Los Alamitos, CA, USA: IEEE Comput. Soc, 1998, pp. 190-7.

[S21] N. Uchihira, "How to make concurrent programs highly reliable-more than state space analysis," Los Alamitos, CA, USA: IEEE Comput. Soc, 1998, pp. 16-23.

[S22] R. Carver and Kuo-Chung Tai, "Use of sequencing constraints for specification-based testing of concurrent programs," *IEEE Transactions on Software Engineering*, vol. 24, Jun. 1998, pp. 471-90.

[S23] Jong-Deok Choi and H. Srinivasan, "Deterministic replay of Java multithreaded applications," *Proceedings of SPDT 98. Parallel and Distributed Tools, Symposium, 3-4 Aug. 1998*, New York, NY, USA: ACM, 1998, pp. 48-59.

[S24] E. Bruneton and J. Pradat-Peyre, "Automatic verification of concurrent Ada programs," Berlin, Germany: Springer-Verlag, 1999, pp. 146-57.

[S25] C. Rudram, P. Croll, and N. Uchihira, "Building test cases for use in Hypersequential Programming," 1999, pp. 205-209.

[S26] P. Welch, G. Hilderink, A. Bakkers, and G. Stiles, "Safe and verifiable design of concurrent Java programs," Anaheim, CA, USA: IASTED-Acta Press, 1999, pp. 20-6.

[S27] T. Katayama, E. Itoh, Z. Furukawa, and K. Ushijima, "Test-case generation for concurrent programs with the testing criteria using interaction sequences," Los Alamitos, CA, USA: IEEE Comput. Soc, 1999, pp. 590-7.

[S28] G. Naumovich, G. Avrunin, and L. Clarke, "Data flow analysis for checking properties of concurrent Java programs," *Proceedings of the 21st International Conference on Software Engineering (ICSE '99), 16-22 May 1999*, New York, NY, USA: ACM, 1999, pp. 399-410.

[S29] J.D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff, "Escape analysis for Java," *ACM SIGPLAN Notices*, vol. 34, 1999, pp. 1–19.

[S30] J. Chen, "A study on static analysis in network of synchronizing FSMs," Los Alamitos, CA, USA: IEEE Comput. Soc, 2000, pp. 489-93.

[S31] J. Harrow, "Runtime checking of multithreaded applications with Visual Threads," *SPIN Model Checking and Software Verification. 7th International SPIN Workshop, 30 Aug.-1 Sept. 2000*, Berlin, Germany: Springer-Verlag, 2000, pp. 331-42.

[S32] R. Konuru, H. Srinivasan, and Jong-Deok Choi, "Deterministic replay of distributed Java applications," *Proceedings 14th International Parallel and Distributed*

Processing Symposium. IPDPS 2000, 1-5 May 2000, Los Alamitos, CA, USA: IEEE Comput. Soc, 2000, pp. 219-27.

[S33] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, 2000, pp. 366–381.

[S34] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and Hongjun Zheng, "Bandera: extracting finite-state models from Java source code," *Proceedings of International Conference on Software Engineering, 4-11 June 2000*, New York, NY, USA: ACM, 2000, pp. 439-48.

[S35] C. Hunter and P. Strooper, "Systematically deriving partial oracles for testing concurrent programs," Los Alamitos, CA, USA: IEEE Comput. Soc, 2001, pp. 83-91.

[S36] Heui-Seok Seo, In Sang Chung, Byeong Man Kim, and Yong Rae Kwou, "The design and implementation of automata-based testing environment for Java multi-thread programs," Los Alamitos, CA, USA: IEEE Comput. Soc, 2001, pp. 221-8.

[S37] C. Artho, "Finding faults in multi-threaded programs," *Master's thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin*, 2001.

[S38] C. Flanagan and S. Freund, "Detecting race conditions in large programs," *2001 ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 18-19 June 2001*, USA: ACM, 2001, pp. 90-6.

[S39] E. Yahav, "Verifying safety properties of concurrent Java programs using 3-valued logic," *ACM SIGPLAN Notices*, vol. 36, 2001, pp. 27–40.

[S40] K. Havelund and G. Rosu, "Monitoring Java programs with Java PathExplorer," *RV'2001, Runtime Verification (in Connection with CAV '01), July 23, 2001 - July 23, 2001*, vol. 55, 2001, pp. 200-217.

[S41] P. Koppol, R. Carver, and Kuo-Chung Tai, "Incremental integration testing of concurrent programs," *IEEE Transactions on Software Engineering*, vol. 28, Jun. 2002, pp. 607-23.

[S42] S.D. Stoller, "Testing concurrent Java programs using randomized scheduling," Amsterdam, 1000 AE, Netherlands: Elsevier, 2002, pp. 149-164.

[S43] S. Ghosh, "Towards measurement of testability of concurrent object-oriented programs using fault insertion: a preliminary investigation," Los Alamitos, CA, USA: IEEE Comput. Soc, 2002, pp. 17-25.

[S44] C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe programming: Preventing data races and deadlocks," *17th ACM Conference on Object-Orientated Programming Systems, Languages, and Applications, November 4, 2002 - November 8, 2002*, Seattle, WA, United states: Association for Computing Machinery, 2002, pp. 211-230.

[S45] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv, "Deriving specialized program analyses for certifying component-client conformance," *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02), June 17, 2002 - June 19, 2002*, Berlin, Germany: Association for Computing Machinery, 2002, pp. 83-94.

[S46] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation," *IBM Systems Journal*, vol. 41, 2002, pp. 111-25.

[S47] M. Musuvathi, D.Y. Park, A. Chou, D.R. Engler, and D.L. Dill, "CMC: A pragmatic approach to model checking real code," *ACM SIGOPS Operating Systems Review*, vol. 36, 2002, pp. 75–88.

[S48] S. Stoller, "Model-checking multi-threaded distributed Java programs," *International Journal on Software Tools for Technology Transfer*, vol. 4, 2002, pp. 71-91.

[S49] Huo Yan Chen, Yu Xia Sun, and TH Tse, "A strategy for selecting synchronization sequences to test concurrent object-oriented software," Los Alamitos, CA, USA: IEEE Comput. Soc, 2003, pp. 198-203.

[S50] C.D. Yang and L.L. Pollock, "All-uses testing of shared memory parallel programs," *Software Testing Verification and Reliability*, vol. 13, 2003, pp. 3-24.

[S51] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," Los Alamitos, CA, USA: IEEE Comput. Soc, 2003, p. 7 pp.

[S52] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs," *SIGPLAN Notices*, vol. 38, Oct. 2003, pp. 178-89.

[S53] B. Long and B. Long, "Formal specification of Java concurrency to assist software verification," Los Alamitos, CA, USA: IEEE Comput. Soc, 2003, p. 8 pp.

[S54] S. Khurshid, C. Pasareanu, and V. Willem, "Generalized symbolic execution for model checking and testing," Berlin, Germany: Springer-Verlag, 2003, pp. 553-68.

[S55] Y. Eytani, E. Farchi, and Y. Ben-Asher, "Heuristics for finding concurrent bugs," Los Alamitos, CA, USA: IEEE Comput. Soc, 2003, p. 8 pp.

[S56] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," *SIGOPS Oper. Syst. Rev.*, vol. 37, 2003, pp. 237-252.

[S57] B. Long, D. Hoffman, and P. Strooper, "Tool support for testing concurrent Java components," *IEEE Transactions on Software Engineering*, vol. 29, Jun. 2003, pp. 555-66.

[S58] D. Grossman, "Type-safe multithreading in cyclone," *ACM SIGPLAN International Workshop on Types in Language Design and Implementation, 18 Jan. 2003*, USA: ACM, 2003, pp. 13-25.

[S59] W. Visser, K. Havelund, G. Brat, Seungjoon Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, Apr. 2003, pp. 203-32.

[S60] Shuang Quan Li, Huo Yan Chen, and Yu Xia Sun, "A framework of reachability testing for Java multithread programs," Piscataway, NJ, USA: IEEE, 2004, pp. 2730-4.

[S61] C. Flanagan and S. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," USA: ACM, 2004, pp. 256-67.

[S62] Y. Eytani and S. Ur, "Compiling a benchmark of documented multi-threaded bugs," Los Alamitos;Massey University, Palmerston, CA 90720-1314, United States;New Zealand: IEEE Computer Society, 2004, pp. 3641-3648.

[S63] A. Groce and W. Visser, "Heuristics for model checking Java programs," *International Journal on Software Tools for Technology Transfer*, vol. 6, Dec. 2004, pp. 260-76.

[S64] Yu Lei and R. Carver, "Reachability testing of semaphore-

based programs," Los Alamitos, CA, USA: IEEE Comput. Soc, 2004, pp. 312-17.

[S65] H. Hallal, E. Alikacem, W. Tunney, S. Boroday, and A. Petrenko, "Antipattern-based detection of deficiencies in Java multithreaded software," *Proceedings. Fourth International Conference on Quality Software, 8-9 Sept. 2004*, Los Alamitos, CA, USA: IEEE Comput. Soc, 2004, pp. 258-67.

[S66] D. Hovemeyer and W. Pugh, "Finding concurrency bugs in Java," *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs, St. John's, Newfoundland, Canada*, 2004.

[S67] Yu Lei and R. Carver, "A new algorithm for reachability testing of concurrent programs," Los Alamitos, CA, USA: IEEE Computer Society, 2005, p. 10 pp.

[S68] S. Kim, L. Wildman, and R. Duke, "A UML approach to the generation of test sequences for Java-based concurrent systems," Los Alamitos, CA, USA: IEEE Comput. Soc, 2005, pp. 100-9.

[S69] A. Sasturkar, R. Agarwal, L. Wang, and S.D. Stoller, "Automated type-based analysis of data races and atomicity," New York, NY 10036-5701, United States: Association for Computing Machinery, 2005, pp. 83-94.

[S70] W. Wong, Yu Lei, and Xiao Ma, "Effective generation of test sequences for structural testing of concurrent programs," 2005, pp. 539-548.

[S71] C. Flanagan, S. Freund, and S. Qadeer, "Exploiting purity for atomicity," *IEEE Transactions on Software Engineering*, vol. 31, Apr. 2005, pp. 275-91.

[S72] V. Schuppan, M. Baur, and A. Biere, "JVM Independent Replay in Java," *Electronic Notes in Theoretical Computer Science*, vol. 113, 2005, pp. 85-104.

[S73] Peng Wu and Huimin Lin, "Model-based testing of concurrent programs with predicate sequencing constraints," Los Alamitos, CA, USA: IEEE Comput. Soc, 2005, pp. 3-10.

[S74] C. Campbell, M. Veanes, J. Huo, and A. Petrenko, "Multiplexing of partially ordered events," Heidelberg, D-69121, Germany: Springer Verlag, 2005, pp. 97-110.

[S75] 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 & Experience*, vol. 19, 2005, pp. 281-94.

[S76] S. Barner, Z. Glazberg, and I. Rabinovitz, "Wolf - bug hunter for concurrent software using formal methods," *Computer Aided Verification. 17th International Conference, CAV 2005. Proceedings, 6-10 July 2005*, Berlin, Germany: Springer-Verlag, 2005, pp. 153-7.

[S77] A. Bron, E. Farchi, Y. Magid, Y. Mir, and S. Ur, "Applications of synchronization coverage," *2005 ACM SIGPLAN Symposium on Principles and Practise of Parallel Programming, PROPP 05, June 15, 2005 - June 17, 2005*, Chicago, IL, United states: Association for Computing Machinery, 2005, pp. 206-212.

[S78] L. Wildman, B. Long, and P. Strooper, "Dealing with non-determinism in testing concurrent Java components," *12th Asia-Pacific Software Engineering Conference, APSEC'05, December 15, 2005 - December 17, 2005*, Taipei, Taiwan: IEEE Computer Society, 2005, pp. 393-400.

[S79] C. Artho and A. Biere, "Combined static and dynamic analysis," *Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL 2005), January 21, 2005 - January 21, 2005*, Elsevier, 2005, pp. 3-14.

[S80] S. Copty and S. Ur, "Multi-threaded testing with AOP is easy, and it finds bugs!," *11th International Euro-Par Conference, Euro-Par 2005, August 30, 2005 - September 2, 2005*, Lisbon, Portugal: Springer Verlag, 2005, pp. 740-749.

[S81] L. Wang and S.D. Stoller, "Accurate and efficient runtime detection of atomicity errors in concurrent programs," New York, NY 10036-5701, United States: Association for Computing Machinery, 2006, pp. 137-146.

[S82] J. Yan, Zhongjie Li, Yuan Yuan, Wei Sun, and Jian Zhang, "BPEL4WS unit testing: test case generation using a concurrent path analysis approach," Los Alamitos, CA, USA: IEEE Computer Society, 2006, p. 10 pp.

[S83] S. McMaster and A. Memon, "Call stack coverage for GUI test-suite reduction," Los Alamitos, CA 90720-1314, United States: IEEE Computer Society, 2006, pp. 33-42.

[S84] M. Musuvathi and S. Qadeer, "CHESS: systematic stress testing of concurrent software," Berlin, Germany: Springer-Verlag, 2006, pp. 15-16.

[S85] S. MacDonald, Jun Chen, and D. Novillo, "Deterministically executing concurrent programs for testing and debugging," USA: CSREA Press, 2006, pp. 844-50.

[S86] H. Seo, I.S. Chung, and Y.R. Kwon, "Generating test sequences from statecharts for concurrent program testing," *IEICE Transactions on Information and Systems*, vol. E89-D, 2006, pp. 1459-1469.

[S87] L. Wang and S. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Transactions on Software Engineering*, vol. 32, Feb. 2006, pp. 93-110.

[S88] T. Elmas and S. Tasiran, "VyrdMC: Driving Runtime Refinement Checking with Model Checkers," *Electronic Notes in Theoretical Computer Science*, vol. 144, 2006, pp. 41-56.

[S89] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar, "Healing data races on-the-fly," New York, NY 10036-5701, United States: Association for Computing Machinery, 2007, pp. 54-64.

[S90] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, R. Popa, and Yuanyuan Zhou, "MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," *Operating Systems Review*, vol. 41, Dec. 2007, pp. 103-16.

[S91] Xufang Gong, Yanchen Wang, Ying Zhou, and Bixin Li, "On testing multi-threaded Java programs," Piscataway, NJ, USA: IEEE, 2007, pp. 702-6.

[S92] Y. Eytani, K. Havelund, S. Stoller, and S. Ur, "Towards a framework and a benchmark for testing tools for multi-threaded programs," *Concurrency and Computation Practice & Experience*, vol. 19, Mar. 2007, pp. 267-79.

[S93] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," New York, NY 10036-5701, United States: Association for Computing Machinery, 2007, pp. 533-536.

[S94] M.A. Wojcicki and P. Strooper, "Maximising the information gained from a study of static analysis technologies for concurrent software," *Empirical Software Engineering*, vol. 12, 2007, pp. 617-645.

[S95] Chang-ai Sun, "A transformation-based approach to generating scenario-oriented test cases from UML activity diagrams for concurrent applications," Piscataway, NJ, USA: IEEE, 2008, pp. 160-7.

[S96] V. Mutilin, "Concurrent testing of Java components using Java PathFinder," Piscataway, NJ, USA: IEEE, 2008, pp. 53-9.

[S97] J. Takahashi, H. Kojima, and Z. Furukawa, "Coverage based testing for concurrent software," Piscataway, NJ 08855-1331, United States: Institute of Electrical and Electronics Engineers Inc., 2008, pp. 533-538.

[S98] A. Dantas, F. Brasileiro, and W. Cirne, "Improving automated testing of multi-threaded software," Piscataway, NJ, USA: IEEE, 2008, pp. 521-4.

[S99] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou, "Learning from mistakes - a comprehensive study on real world concurrency bug characteristics," *SIGPLAN Notices*, vol. 43, Mar. 2008, pp. 329-39.

[S100] A. Jannesari and W.F. Tichy, "On-the-fly race detection in multi-threaded programs," New York, NY 10036-5701, United States: Association for Computing Machinery, 2008, p. 1390847.

[S101] D.M. Carter, R. Aygun, G. Cox, M.E. Weisskopf, and L. Etzkorn, "The effect of uncontrolled concurrency on model checking," *Concurrency Computation Practice and Experience*, vol. 20, 2008, pp. 1419-1438.

[S102] Bin Lei, Linzhang Wang, and Xuandong Li, "UML activity diagram based testing of Java concurrent programs for data race and inconsistency," Piscataway, NJ, USA: IEEE, 2008, pp. 200-9.

[S103] Z. Letko, T. Vojnar, and B. Krena, "AtomRace: Data race and atomicity violation detector and healer," *6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging 2008, PADTAD'08, July 20, 2007 - July 21, 2007*, Seattle, WA, United states: Association for Computing Machinery, 2008.

[S104] S. Tallam, C. Tian, and R. Gupta, "Dynamic slicing of multithreaded programs for race detection," *24th IEEE International Conference on Software Maintenance, ICSM 2008, September 28, 2008 - October 4, 2008*, Beijing, China: IEEE Computer Society, 2008, pp. 97-106.

[S105] T. Ball, S. Burckhardt, J. de Halleux, M. Musuvathi, and S. Qadeer, "Deconstructing concurrency Heisenbugs," *2009 31st International Conference on Software Engineering - Companion Volume - ICSE-Companion, 16-24 May 2009*, Piscataway, NJ, USA: IEEE, 2009, pp. 403-4.

[S106] M. Naik, Chang-Seo Park, Koushik Sen, and D. Gay, "Effective static deadlock detection," *2009 31st International Conference on Software Engineering (ICSE 2009), 16-24 May 2009*, Piscataway, NJ, USA: IEEE, 2009, pp. 386-96.

[S107] N. Kidd, T. Reps, J. Dolby, and M. Vaziri, "Finding concurrency-related bugs using random isolation," *Verification, Model Checking, and Abstract Interpretation. 10th International Conference, VMCAI 2009, 18-20 Jan. 2009*, Berlin, Germany: Springer-Verlag, 2009, pp. 198-213.

[S108] Yang Zhao and J. Boyland, "Assuring lock usage in multithreaded programs with fractional permissions," *2009 Australian Software Engineering Conference (ASWEC 2009), 14-17 April 2009*, Piscataway, NJ, USA: IEEE, 2009, pp. 277-86.

[S109] N. Ben Rajeb, B. Nasraoui, R. Robbana, and T. Touili, "Verifying Multithreaded Recursive Programs with Integer Variables," *Electronic Notes in Theoretical Computer Science*, vol. 239, 2009, pp. 143-154.

Appendix B. Meta data and data extraction forms

Meta Data Form

Meta data item	Value
Article Title:	
Authors:	
Publication Year:	
Publisher:	
Article Type (Journal, Conference etc):	

Data Extraction Form

Data item	Value	Comment	Mapping to the research questions
Concurrency issues or characteristics	▪ Testing ▪ Automated testing ▪ Test case generation ▪ Defect detection ▪ Concurrent bug pattern ▪ Others		RQ1.1
Concurrency bugs	▪ Data race ▪ Deadlock ▪ Atomicity ▪ Synchronization ▪ Livelock ▪ Collected from articles		RQ1.1
Testing types	▪ Static ▪ Dynamic ▪ Model checking ▪ Coverage ▪ Replay ▪ Noise maker		RQ1.2
Testing techniques	▪ Static analysis ▪ Dynamic/on the fly ▪ Model checking ▪ Theorem prove ▪ Data race detection ▪ Deadlock detection ▪ State space exploration ▪ Coverage ▪ Replay ▪ Noise makers ▪ Collected from articles ▪ Not stated		RQ1.2
Testing tools	▪ Collected from articles		RQ1.2
Benchmarks	▪ Collected from articles		RQ1.2
Test case generation techniques	▪ Collected from articles		RQ1.3
Test case generation tools	▪ Collected from articles		RQ1.3
Programming language	▪ Java ▪ Ada ▪ C ▪ C++ ▪ Collected from articles ▪ Not stated		RQ1.2, RQ1.3
Recommendation/ Lessons learn	▪ Collected from articles		N/A

Appendix C. Distribution of studies by publication channel.

Publication channel	Publisher	Type	Count	In (%)
Asia Pacific Software Engineering Conference (ASPEC)	IEEE	Conference	11	10.09
IEEE Transactions on Software Engineering	IEEE	Journal	6	5.50
ACM SIGPLAN notices	ACM	Journal	5	4.59
International Computer Software and Applications Conference (COMPSAC)	IEEE	Conference	5	4.59
International Parallel and Distributed Processing Symposium (IPDPS)	IEEE	Conference	5	4.59
Electronic Notes in Theoretical Computer Science	Elseveir	Journal	4	3.67
ACM Operating Systems Review	ACM	Journal	3	2.75
Concurrency and Computation Practice & Experience	Wiley & Sons	Journal	3	2.75
International journal of software tools for technology transfer (STTT)	LNCS	Journal	3	2.75
International Conference on Software Engineering (ICSE)	IEEE	Conference	3	2.75
International Symposium on Software Reliability Engineering (ISSRE)	IEEE	Conference	3	2.75
ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP	ACM	Conference	3	2.75
Parallel and Distributed Systems: Testing, Analysis, and Debugging PADTAD	ACM	Conference	3	2.75
Australian Software Engineering Conference (ASWEC)	IEEE	Conference	2	1.83
International Conference on Quality Software (QSIC)	IEEE	Conference	2	1.83
Israeli Conference on Computer Systems and Software Engineering	IEEE	Conference	2	1.83
International Conference on Software Testing, Verification and Validation (ICST)	IEEE	Conference	2	1.83
ACM Transaction Computer System	ACM	Journal	1	0.92
Automated Software Engineering	LNCS	Journal	1	0.92
Empirical Software Engineering	LNCS	Journal	1	0.92
IBM Systems Journal	IBM	Journal	1	0.92
IEICE Transactions on Information and Systems	IEICE	Journal	1	0.92
Runtime Verification (in Connection with CAV)	Elseveir	Journal	1	0.92
SPIN Model Checking and Software Verification	Wiley & Sons	Journal	1	0.92
ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD	IEEE	Conference	1	0.92
Australian Computer Science Conference (ACSC)	IEEE	Conference	1	0.92
Engineering of Complex Computer Systems (ICECCS)	IEEE	Conference	1	0.92
IEEE International Conference on Systems, Man and Cybernetics	IEEE	Conference	1	0.92
IEEE International Workshop on Source Code Analysis and Manipulation	IEEE	Conference	1	0.92
International Conference on Application of Concurrency to System Design	IEEE	Conference	1	0.92
International Conference on Distributed Computing Systems (ICDCS)	IEEE	Conference	1	0.92
International Phoenix Conference on Computers and Communications	IEEE	Conference	1	0.92
International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (IsoLA)	IEEE	Conference	1	0.92
International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)	IEEE	Conference	1	0.92
International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE)	IEEE	Conference	1	0.92
International Test Conference	IEEE	Conference	1	0.92
International Workshop on Program Comprehension (IWPC)	IEEE	Conference	1	0.92
International Workshop on Software Engineering for Parallel and Distributed Systems	IEEE	Conference	1	0.92
Workshop on Program Analysis for Software Tools and Engineering	ACM	Conference	1	0.92
ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI)	ACM	Conference	1	0.92
ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)	ACM	Conference	1	0.92
ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)	ACM	Conference	1	0.92
The European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering	ACM	Conference	1	0.92
SIGMETRICS symposium on Parallel and distributed tools	ACM	Conference	1	0.92
Proceedings Supercomputing	ACM	Conference	1	0.92
Object-Oriented Programming Systems, Languages, and Applications, OOPSLA	ACM	Conference	1	0.92
Ada-Europe International Conference on Reliable Software Technologies	LNCS	Conference	1	0.92
Computer Aided Verification (CAV)	LNCS	Conference	1	0.92
International Euro-Par Conference	LNCS	Conference	1	0.92
Lecture Notes in Computer Science	LNCS	Conference	1	0.92
Logic-Based Program Synthesis and Transformation (LOPSTR)	LNCS	Conference	1	0.92
SPIN Model Checking and Software Verification	LNCS	Conference	1	0.92
Tools and Algorithms for the Construction and Analysis of Systems (TACAS)	LNCS	Conference	1	0.92
Verification, Model Checking, and Abstract Interpretation (VMCAI)	LNCS	Conference	1	0.92

International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL)	Elseveir	Conference	1	0.92
International Conference on Software Engineering Research and Practice and Conference on Programming Languages and Compilers SERP		Conference	1	0.92
International Conference Software Engineering and Applications (SEA)	IASTED-ActaPress	Conference	1	0.92
International Conference on Software Engineering (ICSE)	AVM, ICSE	Conference	1	0.92
International Conference on Software Maintenance (ICSM)	(ICSM)	Conference	1	0.92
PODC Workshop on Concurrency and Synchronization in Java Programs		Conference	1	0.92
Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin	MS thesis	Conference	1	0.92
Total articles			**109**	**100**

CHAPTER 3

Paper 2. An Evaluation of Static Analysis Tools for Java Multithreaded Bugs.

An Evaluation of Static Analysis Tools for Java Multithreaded Bugs

Md. Abdullah Al Mamun and Aklima Khanam
School of Computing
Blekinge Institute of Technology
372 25, Ronneby, Sweden
to.mamun@yahoo.com, aklima.bth@gmail.com

Abstract

Static analysis (SA) tools are being used for early detection of software defects. The recent development of multicore processors has enabled software concurrency from servers to personal computers. By nature, concurrency bugs are different from bugs in sequential programs, and they are harder to detect. Available SA tools provide support for detecting concurrency bugs. However, the empirical evidence regarding the effectiveness of SA tools for detecting Java multithreaded bugs are not so strong. This paper presents the result of the evaluation of four static analysis tools detecting Java concurrency bugs and bug patterns. The tools are evaluated using concurrent benchmark programs and a collection of multithreaded bug patterns. In addition to the tools' evaluation, we have identified 87 unique bug patterns from the tools' detectors and literature. Finally, we have categorized the bug pattern detectors of the tools that would help the users to understand and use them more easily.

1. Introduction

Writing concurrent software is difficult; however, detecting concurrent defects is even more difficult. There has been a significant amount of work spent on developing suitable techniques and tools to detect concurrent software defects in the last few decades. As a result, we have various types of tools like static analyzers, model checkers, and dynamic analyzers [6,18]. Static analysis (aka automatic static analysis) tools are able to detect defects in the source code without executing it. Model checkers apply formal verification on source code using call graphs, finite state machines, etc. with a well-defined mathematical formulation [18]. Dynamic analyzers need to execute the code for detecting defects.

In general, the static analysis (SA) tools are more abstract in detecting bugs than dynamic tools. SA tools are scalable where model checking tools suffer from state explosion problems. Dynamic tools are not as easy to use as SA tools; because detecting bugs using dynamic tools needs to instrument source or object code [6].

The importance of early bug detection in software development is a well-established fact. Static analysis tools are capable of early detection of defects [17,19] and in reducing the cost of software development [3]. Empirical studies have shown that the use of SA tools with code inspection for concurrent software is cost effective [28]. Typically, SA tools are fast enough to deploy them on the developers' workstations where it can be used along with the coding. It can also be used as a batch processor for testing a huge amount of code. However, the important thing is that SA tools generally can be used to inspect the code before any other testing tools or techniques used.

A number of studies evaluate different static analysis tools for Java programs [2,5,20,21,23-26]. However, very few studies focus on the evaluation of the static analysis tools for Java multithreaded defects [1, 25]. Most of these attempts are partial in terms of multithreaded defects because they did not cover a wide range of concurrent defect types. More empirical evidence is necessary in order to evaluate different defect detection tools with a specialized focus on different types of bugs and bug patterns.

This study evaluates two commercial and two open source static analysis tools for the ability of detecting Java multithreaded bugs and bug patterns. To evaluate the SA tools, we used a benchmark suite containing Java programs with concurrency bugs [7] and a collection of bug patterns from a library of antipatterns [12] and the selected tools. This study addresses the following research questions:

RQ1: How effective are static analysis tools in detecting Java multithreaded bugs and bug patterns?

RQ2: Are commercial SA tools better than open source SA tools in detecting Java multithreaded defects?

We conducted an experiment to answer these questions where the SA tools acted as subjects and the benchmark suite and bug patterns worked as object. In addition to the evaluation of the tools, we have categorized the rules/checkers/bug patterns used by the tools to detect defects. This categorization would help the users to understand and choose an appropriate set of checkers/bug

patterns according to their need. We also studied bug pattern detectors implemented by the tools and an antipattern library [12], then unified them in a list of 87 unique Java multithreaded bug patterns.

The results of the study show that the commercial tool Jtest is better than other tools in detecting Java multithreaded bugs with the drawback that the false positive ratio reported by this tool is high. However, it was not possible to draw a clear distinction among the commercial and free 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.

Section 2 discusses about the concurrency bugs and bug patterns. Section 3 presents the selected tools and set of programs used to test the tools. Section 4 explains the identification of unique bug patterns and their categorization. The experiment is detailed in section 5. Section 6 discusses about the analysis and result. Discussion and related works are discussed in section 7 and 8 respectively and we conclude in section 9.

2. Concurrency bugs and bug 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, there arise a number of problems like data race, atomicity violation, synchronization defect, deadlock, livelock etc.

Concurrency problems can be divided into two types of basic properties, safety and liveness [18]. A safety property assures that nothing bad will happen during the program execution. On the other hand, the liveness property expresses that something good will eventually happen. The most known problems under these properties are race conditions (aka data race, interleaving problem), deadlock, livelock and starvation [18]. These problems must be absent in the program in order to fulfill the safety and liveness properties. These two basic properties of concurrent software are abstract and some concurrency problems overlap between them. For this reason, classifying the concurrency problems based on the basic properties is not fruitful.

In software engineering, patterns mean, some common techniques to document and reuse specific and important examples. There have been research regarding concurrent bug characteristics/patterns [9,12]. In a general sense, bug patterns describe common errors those can occur in the program. A bug pattern is a recurring correlation between a signaled error and the underlying bug in a program [29]. Design patterns are solutions to recurring problems. The solution of a design pattern that is decidedly attached with a negative consequence is called an antipattern. The terms, bug patterns and antipatterns are quite similar with a 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. For example, the FindBugs tool describes a number of design patterns in its bug pattern list [10]. In the same way, Hallal et el. [12] describe several bug patterns in an antipattern library. In this study, we used the term ‘bug pattern’ for both bug patterns and antipatterns.

3. Selection of tools and test programs

3.1. Selection of Java static analysis tools

We have selected 4 Java static analysis tools as shown in Table 1. Among the tools, FindBugs and Jlint are the most discussed tools in the literature. However, very few articles [3, 20] worked with the tools Coverity Prevent and Jtest and to the best of our knowledge no study evaluated the effectiveness of these two commercial tools for Java multithreaded bugs. Brief descriptions of the tools are given below.

Table 1. Selected static analysis tools

Tool name	License	Version
Coverity Prevent	Commercial	4.4.1
Jtest	Commercial	8.4.11
FindBugs	Open Source	1.3.9
Jlint	Open Source	3.1

Coverity Prevent [3,4] is a commercial static analysis tool combining statistical and inter-procedural analysis with Boolean Satisfiability (SAT) to detect defects. To infer correct behavior it uses statistical analysis based on the behavioral patterns identified in the code. Then inter-procedural (whole-program) analysis across method, file, and module boundaries is made 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 [15,20] 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. Jtest has a good collection of checkers detecting 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. It 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 [13,14,20,] 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 a category (e.g., multithreaded correctness, correctness, performance etc)

FindBugs provides both GUI and command line interfaces. In addition to its swing interface, it 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 it is known to run on Windows, GNU/Linux and MacOS X platforms. It is possible to expand FindBugs by defining custom detectors.

Jlint [1,20] is an open source static analysis tool that performs syntactic checks, flow analysis, and builds a lock graph for detecting defects like inheritance, synchronization. It can detect data race 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. Jlint detects about 36 different types of bugs. It has a component named AntiC which is a syntax checker for al C-family languages (i.e. C, C++, Objective C and Java).

Jlint has a simple command line interface. It runs on Windows, UNIX, Linux. Jlint is not easily expandable.

3.2. Selection of test programs

We selected programs containing both concurrency bugs and bug patterns. It is necessary to evaluate the tools with both bugs and bug patterns. Testing the tools for detecting bugs can reveal the effectiveness of the tools. A bug can occur due to many reasons. Collecting real life buggy programs with such a high variety of reasons is quite challenging, and it demands a huge amount of time. For this reason, testing the tools with respect to bug patterns is important because bug patterns can reflect a high variety of situations where a bug can potentially occur [12]. However, bug patterns are not always bugs hence it is meaningful to test the tools with both bugs and bug patterns.

We selected two sets of programs, where the first set represents concurrency bugs and the second set represents bug patterns. The first set of programs are taken from a concurrency bug benchmark suite [6,7]. 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 [16]. However, benchmarks can be used if such limitations are considered [22].

The selected benchmarks are also used in other studies. An experience story [8] of the benchmark reports a list of 14 studies and research centers that used the benchmark. Experts in concurrent software testing and students of the course concurrent software testing wrote most of the benchmark programs. We selected 19 programs from this benchmark suite that provide a precise bug documentation and one program is written by us. Table 2 shows the selected benchmark programs. Detailed information about these programs is given in Appendix A and a further detail is available in the benchmark suite.

Table 2. Selected benchmark programs

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

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

The second set is a collection of Java multithreaded bug patterns and antipatterns. We have collected these patterns from the four tools discussed in section 3.1 and a

collection of antipatterns documented by Hallal et el [12]. We have categorized and identified 87 unique bug patterns from 141 bug patterns that are discussed in section 4. Then we collected/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. 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.

Table 3. Bug patterns and categories

Tools	Number of Checkers/bug-patterns	Concurrency checkers/rules/bug patterns by bug categories
Coverity	10 checkers	Concurrency: 4 regular checkers
		Preview: 6 non-regular checkers
Jtest	43 bug patterns	Thread safe programming: All 43 bug patterns
FindBugs	23 checkers with 40 bug patterns	Multithreaded correctness: All 23 checkers
Jlint	12 bug patterns	Deadlock: 7 bug patterns
		Race condition: 4 bug patterns
		waitNoSync: 1 bug pattern

Unfortunately, bug patterns categorized by the tools are not satisfactory. Only Jlint describes its bug patterns under a refined level of categories. Among the other tools, Jtest provides a further categorization of its bug patterns in the bug documentations provided with the tool. Jtest described 19 bug patterns in the category *Deadlocks and race conditions*, 6 bug patterns in the category *Concurrency* and 18 other bug patterns 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 major works [9,12] in the literature that worked with the concurrent bug patterns. Among them, Hallal et el [9] 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 of 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 [9]. The antipattern library documented 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 the selected 103 bug patterns and categorized them that are given in Table 4.

While unifying the bug patterns from different tools we found several cases where a bug pattern detector of a tool is partially described by another tool. For example, the five bug patterns shown in Figure 1 are identified as similar hence combined into a single group. The Jtest bug pattern describes both wait(), notify() and notifyAll() methods where FindBugs describes wait() and notify() in two different bug patterns. On the other hand, Jlint only describes the wait(). FindBugs describes these two bug patterns under a single checker. In this way, the bug pattern detectors implemented by different tools may vary from other to some extent, although they are listed as a common bug pattern. However, in some cases, FindBugs bug patterns described under a single checker are distributed into different groups.

Jtest: Ensure 'wait()', 'notify()' and 'notifyAll()' are invoked on an object that is clearly synchronized in its enclosing method scope [TRS.NSYN]
FindBugs: MWN: Mismatched notify()
FindBugs: MWN: Mismatched wait()
Jlint: Method name.wait() is called without synchronizing on name
Bug pattern library: Identifier.wait() method called without synchronizing on identifier

Figure 1. A group of similar bug patterns

Table 4. Categorized unique bug patterns

Category	Coverity	Jtest	FindBugs	Jlint	Antipattern library [12]	Total patterns	Total unique patterns
Deadlock	2	7	12	6	9	**36**	**17**
Livelock	0	0	1	0	2	**3**	**2**
Race Condition	6	7	8	6	8	**34**	**18**
Problems leading to unpredictable results	1	13	9	0	4	**28**	**15**
Quality and style problems	0	8	6	0	4	**18**	**15**
Efficiency/performance problems	1	3	0	0	11	**15**	**14**
General warnings/mistakes	0	5	2	0	0	**7**	**6**
Total	**10**	**43**	**38**	**12**	**38**	**141**	**87**

5. Experiment

We have followed the experiment process suggested by Wohlin et el. [27]. The experiment is defined as:

Analyze *"static analysis tools"* for the purpose of "*evaluation"* with respect to their *"bug detection ability"* from the point of the *"researchers and practitioners"* in the context of *"open source and commercial tools testing multithreaded Java programs"*

5.1. Experiment planning

5.1.1. Context selection. The experiment is *offline* since it is not executed in the real industry environment. 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 those will be analyzed by the testing tools. Since the result of the experiment would be helpful for the researchers and the practitioners, hence the experiment can be characterized as real.

5.1.2. Hypothesis formulation. We considered the following hypotheses for the experiment:

Null Hypothesis H_0: The selected static analysis tools are equally able to detect bugs and bug patterns in Java multithreaded programs.

Alternative Hypothesis H_1: The selected static analysis tools are not equally able to detect bugs and bug patterns in Java multithreaded programs.

Measures needed: Java multithreaded bugs, Java multithreaded bug patterns, bug detection ability.

5.1.3. Variable selection. The *independent variables* of the experiment are two "*sets of programs"* i.e. program set with *multithreaded bugs* and program set with multithreaded *bug patterns*. The dependent variable of the experiment is the *bug detection ability* of the selected tools. Two measures of the bug detection ability are *defect detection ratio* and *false positive ratio*, which are calculated as below:

$$\text{Defect detection ratio} = \frac{\text{Number of defects detected by SA tool}}{\text{Total number of defects}}$$

$$\text{False positive ratio} = \frac{\text{Number of false positive warnings by SA tool}}{\text{Total number of false positive warnings}}$$

The 'total number of false positive warnings' in calculating the false positive ratio is the summation of the false positive warnings reported by the tools.

5.1.4. Selection of subjects. Static analysis tools discussed in section 2 are the subjects of the study. In the open source software domain, the selected open source tools FindBugs [2,5,11-14,18,20,21,23-26,28] and Jlint [1,12,18,20,21,25,26,28] are most discussed tools in the academia. We strongly believe they represent the open source tools. Though, studies with the commercial tools are not very common, their industrial adoption [4,15] reflect that they sufficiently represent the commercial static analysis tools.

5.1.5. Experiment design. In the experiment, all *subjects* (tools) will test all *objects* (sets of programs). Here, the *independent* variable "*sets of programs"* is the only experiment *factor*. Since every *subject* will experience every possible *treatment* of the *factor* hence the experiment design is a complete block design. Table 5 illustrates the experiment design for the first set of programs. The same design is applied on the second set of programs.

Table 5. Complete block design

	Factor (sets of programs)			
	First program set (concurrency bugs)			
Subjects	Treatment 1	Treatment 2	…	Treatment N
Coverity	Program 1	Program 2	…	Program N
Jtest	Program 1	Program 2	…	Program N
FindBugs	Program 1	Program 2	…	Program N
Jlint	Program 1	Program 2	…	Program N

N indicates the number of treatments i.e. total test programs in a set.

5.1.6. Instrumentation. We will use Eclipse (version 3.4.2) environment for the tools Coverity Prevent, FindBugs, and Jtest since all of them provide plugins for Eclipse. Jlint will be used in the command line mode because it does not provide a graphical user interface.

In the static analysis tools, we will activate all multithread related checkers/rules and set the tools in full analysis mode. Coverity Prevent will be applied with both *concurrency* and *preview* (a collection of checkers still under development) checkers. Jtest combines a set of rules named as *thread safe programming* that will be used to 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 *+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 MS Excel is used to collect the experimental data.

5.1.7. Validity evaluation.

Conclusion validity. The selected programs were tested against the concurrency checkers/detectors of the tools. Warnings reported by the tools are documented and

inspected carefully. Hence, the conclusion validity is not considered to be vital.

Internal and construct validity. All the selected subjects are Java static analysis tools with concurrency bug detection capability. The classification of the selected subjects and the measurements of the experiment are straightforward. No notable threat is observed in these two validity categories.

External validity. The average size of the selected benchmark programs is 267 lines of code, which is small. In a real world, static analyzers deal with programs even with millions lines of code. We could use several medium to large sized open source Java projects instead of using these small benchmark programs. However, it would not be possible to find a variety of multithreaded bug in a couple projects that we have found in the benchmark programs. Moreover, classifying all the warnings generated from such programs would be so much time consuming hence practically not possible. A study [21] comparing Java bug finding tools, used 5 medium sized open source projects, which reports more than 9000 concurrency warnings generated from 4 SA tools. This study could not identify the false positive and false negative warnings due to the large number of warning generated, which is mentioned in the *threats to validity* section of the article.

Though the benchmark programs cover a variety of concurrency bugs, they are not evenly distributed in different categories. Table 6 shows the frequencies of bugs in deadlock and livelock categories are respectively four and two. Greater bug samples within these categories would make the result more generalized. The number of uniquely selected bug patterns in each category is quite satisfactory in terms of size and distribution except the category livelock that is shown in Table 4.

5.2. Experiment operation

As the preparation of the experiment, we collected and organized the test programs. The first set of programs containing bugs were collected from the benchmark. The programs in the second set (bug patterns) were collected from different sources. We wrote/refined several programs for the second set, since they were not either available or complete.

The experiment was executed over 450 person-hours. For each program, first, we read the program descriptions provided with each benchmark program when it was available. Then we inspect the code to identify the bugs in the programs. During the inspection, we found few cases where the bug documented by the benchmark program is not exactly a bug. We exclude such false documented bugs from the list of bugs. In addition, we found a number of bugs not described in the benchmark documentations. We documented these new bugs. Then we tested the programs with the tools. We documented the warnings generated by tools and inspected each warning to classify them as true or false warning. We identified that the tools report a huge number of warnings those are like general warnings concerning different coding styles and standards. We classified them as general warnings. One important fact is that all the bugs in the benchmark programs are related to multithreaded correctness. We found several warnings reported by the tools those are related to software performance issues. We have documented these warnings as general warnings. For the limitation of time, we could not decide few warnings as true or false hence we marked them as undecided.

For evaluating the tools in terms of bug patterns, we tested the tools against 87 unique bug patterns derived from the unification of the bug patterns mentioned in section 4. For the bug pattern programs, we only checked whether the tools are able to detect it or not. We have not analyzed the warnings further to identify the false positive warnings. This is because, the bug pattern programs are very small and most of the cases they are in lack of proper context where we can say an identified pattern as a bug or not. Hence, trying to identify the false positive warnings for the bug patterns are meaningless.

6. Analysis and result

6.1. Testing concurrency bugs

We tested the SA tools against 20 Java programs containing 32 multithreaded bugs. All these 32 bugs are related to software correctness. However, the tools are able to detect performance related defects that are explored by the bug patterns' testing. Table 6 shows the bugs detected by the selected SA tools. The selected benchmark programs contain 11 different types of bugs. Detail descriptions of these bug types are available in a study [9] carried by the researchers who developed the benchmark suite. This article provides a taxonomy of concurrency bugs as a form of bug patterns. However, these bug patterns are generally more high level than the bug patterns actually implemented by the tools. Even though the benchmark programs describe the defects as bug patterns, certainly they are multithreaded bugs.

There are 26 bugs in the data race and atomicity violation category, 4 bugs in the deadlock and 2 bugs in the livelock category. From Table 6, clearly Jtest is the winner among the tools in detecting data race and atomicity bugs. In the deadlock category, both Jtest and Jlint detect 2 defects out of 4. None of the tools could detect weak-reality (two stage access), blocking critical

section and orphaned-thread bugs. All 5 bugs detected by Coverity Prevent falls under 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. Jlint detects 8 bugs, which is more than both Coverity Prevent and FindBugs.

Table 6. Bug detection

Bug Category	Bug type (described by benchmark programs)	No. of Bugs	Bug detected			
			Coverity	JTest	FindBugs	Jlint
Data race and Atomicity violation	Wrong Lock/No Lock	11	3	7	2	3
	Non-atomic	9	-	3	-	3
	Weak-reality (two stage access)	2	-	-	-	-
	Lazy Initialization	1	-	1	-	-
	Double checked locking	1	-	-	1	-
	Condition for Wait	1	-	1	1	-
	Use of deprecated methods	1	1	1	-	-
	Total	26	4	13	4	6
	Defect detection ratio		0.15	0.50	0.15	0.23
Deadlock	Blocking-Critical-Section	1	-	-	-	-
	Hold and wait	2	-	1	-	2
	Notify instead of notifyAll	1	-	1	1	-
	Total	4	0	2	1	2
	Defect detection ratio		0	0.50	0.25	0.50
Livelock	Orphaned-Thread	2	-	-	-	-
	Total	2	0	0	0	0
	Defect detection ratio		0	0	0	0
Sub total		**32**	**4**	**15**	**5**	**8**
Overall Defect detection ratio			**0.13**	**0.47**	**0.16**	**0.25**

We have documented and inspected each warning generated by the tools. An overview of the warnings is shown in Table 7. The general warning category contains the warnings those are not exactly related to the correctness of the program. The general warnings are related with quality and standards that might or might not be followed by a software development organization. The importance of such warnings may vary from organization to organization.

Table 7. Warnings

Warning Type	Coverity	Jtest	FindBugs	Jlint
General	5	136	2	0
True	4	21	8	11
False positive	4	16	5	20
Undetermined	3	8	1	3
Total	**16**	**181**	**16**	**34**

Jtest reports a huge number of warnings in comparison to the other tools where more than 75% are general warnings. Among 136 general warnings, 50 warnings are generated from a single Jtest rule named TRS.NAME that checks whether a thread initializes its name. Jtest rule TRS.SOUF reports 23 warnings that checks whether a non-final field is used to synchronize a code block. The rule TRS.CIET generated another 22 warnings that says not to catch *InterruptedException* in a class that does not extend the Java Thread class. Another 22 warnings come from the rule TRS.STR that checks whether '*this*' reference is used to synchronize a code block. These four Jtest rules reported 117 warnings that we categorized as general.

Jlint does not have quality and styles related concurrency rules hence it did not produce any general warnings. Coverity Prevent has a preview checker named UNEXPECTED_SYNC which is related to performance issues. This checker reported 5 warnings, and we documented them as general warnings. It is interesting that the FindBugs reported only 2 general warnings having 23 checkers with 40 bug patterns where several bug patterns address quality and style problems.

The numbers of unique false positive warnings are shown in Table 8 along with the false positive ratio of the tools. Though, FindBugs and Coverity Prevent are almost same in the false positive ratio, FindBugs can be considered as better, because it checks for quite a large number of bug patterns in comparison to Coverity Prevent. In the same way, the false positive ratio of Jtest and Jlint are same, however Jtest can be seen as more powerful as it checks for more bug patterns than Jlint.

Table 8. False positive ratio of unique false warnings

Tool	Coverity	Jtest	FindBugs	Jlint
False positive warnings	4	16	5	16
False positive ratio	0.10	0.39	0.12	0.39

6.2. Testing concurrent bug patterns

We tested 87 unique bug patterns, shown in Table 3, with the tools. It is expected that every tool should be able to detect a bug pattern that it claims to detect and vice versa. The tools were almost able to detect the bug patterns, as promised. Few cases were observed where the tools could not detect a bug pattern that it claims to detect. Few cases are observed where the strength of the bug pattern detectors differs though they are described in a similar way.

Five cases were documented where Coverity Prevent and Jlint failed to detect bug patterns, though they have detectors for these bug patterns. The checkers of Coverity Prevent are 'guarded by violation', 'double check lock', and 'Unsafe laze init' where 'guarded by violation' is a regular production checker. Jlint's failure bug pattern detectors are 'Method name can be called from different threads and is not synchronized', and 'Field name of class'. In contrast with these cases, the Jlint bug pattern 'Loop id: invocation of synchronized method name can cause deadlock' detects the bug pattern TRS.CSFS described by Jtest. However, these two bug patterns are not exactly equivalent. The Jlint's bug pattern can detect a *cycle of lock graph*, which is much stronger than the Jtest's TRS.CSFS rule. Table 9 shows the number of bug patterns detected by the tools in different categories.

Table 9. Bug patterns detected by the tools

Category	Coverity	Jtest	FindBugs	Jlint
Deadlock	2	7	10	7
Livelock	0	0	1	0
Race Condition	3	7	8	4
Problems leading to unpredictable results	1	13	5	0
Quality and style problems	0	7	6	0
Efficiency/performance problems	1	3	0	0
General warnings/mistakes	0	5	2	0
Total	7	42	32	11
Detection ratio	0.08	0.48	0.37	0.13

Jtest is quite better than the other tools in detecting bugs. However, it is also leading in producing warnings than other tools. FindBugs and Coverity Prevent are almost same in detecting bugs and reporting false positive warnings. Jlint detected more bugs than FindBugs and Coverity Prevent. However, it also reports a maximum number of false positive warnings like Jtest. Jtest detects maximum bug patterns followed by FindBugs and Jlint detects more bugs than Coverity Prevent.

In overall, the commercial tools are better than the open source tools in detecting Java concurrency bugs and bug patterns. However, the distinction is not very significance if the number of false positive warnings is considered. The users need to tradeoff between the bug detection ratio and false positive ratio of the tools to select an appropriate tool. If the cost of inspecting a false positive warning is less than the amount of money saved by early bug detection then Jtest should be the best choice. If inspecting a false positive warning cost more, then FindBugs or Coverity Prevent can be selected.

The users can be choosy in selecting specific bug detectors from different tools. For example, the deadlock detectors provided by Jlint are quite strong. However, it does not mean that Jlint is sufficient for detecting deadlock bugs.

An interesting observation of the result reflects a relationship between the number of bug detection and the number of false positive warnings reported by the tools. A tool detecting more bugs generates more false positive warnings. However, further analysis toward is not made whether this observation is always true because it does not cover the scope of this study.

7. Discussion

The tool Jtest detected about half of the multithreaded bugs with the fact that it produces a high number of false positive warnings. The Coverity Prevent tool is promising from the point of false positive warnings but this tool is quite weak in detecting faults. One reason might be the lowest number of concurrency bug detectors presented by this tool among the selected tools. We have found that the open source tools FindBugs and Jlint detected different defects hence they should be used in combined.

Though Jlint produces a high number of false positive warnings and the command line user interface of Jlint is not user friendly, it is highly recommended to use the deadlock detectors of Jlint in addition to any other tools. We found that the deadlock detectors of Jlint are quite strong in detecting bugs, and they report fewer false positive warnings than the race condition detectors. We have identified that Jlint repeatedly reports similar warnings. Grouping such warnings into a single warning would decrease the number of reported warnings that would both save the time and increase usability of this tool.

FindBugs provides an easy and flexible GUI interface both as an eclipse plugin and standalone swing interface. FindBugs has a rich set of bug pattern detectors, and it reports very few false positive warnings. Though FindBugs identified few concurrency bugs, Coverity Prevent users should use it because it identified different bugs than Coverity Prevent.

Jtest users should review the detectors before using them specially the warnings in the category *'general warning/mistakes'* and *'quality and style problems'* since some of the detectors could produce a huge number of general warnings.

FindBugs and Coverity Prevent show an event list for certain detectors that are very useful in inspecting the respective warnings. Jlint repeatedly generates warnings for such detectors where Jtest only reports a single warning about the defect without further details. It would be easy to understand it with an example. Let's assume that a shared field is used in three places of a method without being locked. Both FindBugs and Coverity Prevent will report a warning with the details of three events where the field is accessed without being locked. Jlint will report 3 different warnings and Jtest will report a single warning without the details of the three accesses.

The swing interface of FindBugs provides a very flexible interface to view the reported warnings in a customized way. Jtest also provides a good view of the reported warnings. It groups the warnings by category then by priority and then by the rules. Hence, the user can only see all warnings generated by a certain rule or all rules warned under a specific priority.

We have identified a large number of general warnings reported by the tools which are shown in Table 7. It is possible to skip these general warnings if they are used when necessary. For this the rules/checkers/bug-patterns need to be reviewed before use according to the standard of a certain project.

A considerable amount of time is given in inspecting the benchmark programs. The selected benchmark programs implement certain concurrency bugs. However, while inspecting the source code, we have identified several concurrent defects that are not discussed in the available benchmark documentations. However, the available bug documentations do not mention the fact that there might have additional concurrency bugs in the programs. If a benchmark user completely relies on the documented bugs available in the benchmark programs, an incorrect bug detection ratio and false positive ratio might result. We identified 11 new concurrency bugs in the selected programs from the benchmark suite, which is shown in Appendix A.

We strongly suggest that the bug pattern detectors of the SA tools should be categorized more rigorously. This would help the users to learn and review the bug pattern detectors more easily. The analysis of the warnings reported by the tools shows that a huge number of general warnings is generated from a few numbers of bug pattern detectors. A refined taxonomy would help the users to identify such detectors that are not much necessary or crucial for the standard of a certain software project.

Although static analysis tools report false positive warnings, they can be used for detecting concurrency defects. The average defect detection ratio of the tools is 0.25, which is not bad indeed with the fact that early detection of defects is a vital issue in cost reduction of software development.

8. Related work

C. Artho [1] evaluated three dynamic (MaC, Rivet, Visualthreads) and two SA analysis tools (Jlint and ESC/Java) for finding multithreaded bugs in Java program. The result of the study said that none of the tools is a clear winner. A major part of this study is about extending the Jlint tool.

A study [21] 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.

Among the two studies discussed above, the first study worked to detect concurrency bugs and the second study used concurrency as an issue. However, we found several studies evaluating static analysis tools from different perspectives other than detecting concurrency bugs. These studies are discussed below.

C. N. Christopher [5] evaluated four SA frameworks with an ideal framework. The results of the evaluation say that no framework is ideal. However, FindBugs and Soot are comparatively better than PMD and Crystal. The evaluation focus was on the usability of the framework, not on the tools.

A study [20] evaluated four commercial and seven open source SA tools. This study also recommended six steps methodology to assess the quality of the software.

Two industrial case study results are described in [23] where two SA bug pattern tools, FindBugs and PMD are applied for evaluation. However, this study did not discuss any concurrency issues. Another industrial case study [24] analyzed the interrelationships of the bug finding tools testing and inspection. The study shown that testing tools detect more defects but tools do not detect some defects those are found by inspection. Therefore, it is good to combine both testing.

Another study [25] compared eight SA tools to detect secure coding heuristics violation in Java. The tools identified 115 distinct violations of secure coding heuristics and developed a taxonomy that would make the understanding, applying them easier.

F. Wedyan et al. [26] evaluated the usefulness of automated SA tools for Java program. They evaluate the effectiveness of SA tools for detecting defects and identifying code refactoring modifications.

9. Conclusion

We have evaluated static analysis tools for detecting Java multithreaded bugs and bug patterns. An experiment

is executed with a set of benchmark programs containing concurrency bugs and a set of bug patterns. We inspected each warning reported by the tools, testing benchmark programs and classified them as true or false positive warnings. A total number of 141 bug patterns is collected from the tools and a library of antipatterns. We identified 87 unique bug patterns and tested the tools against them. Moreover, we classified all the bug patterns, which would help the users to quickly learn and review the bug patterns detectors implemented by the tools.

The defect detection ratio of the tool Jtest is 0.47 where 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 reported false positive warnings, which is about the same as the number of defect detected. However, it would be a good idea to use the tools for early detection of defects. The experiment with the bug patterns shows that the selected tools are able to detect a wide range of bug patterns. In overall, the commercial tools are better than the open source tools. 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 advantage of several tools in detecting bugs in different categories.

The documented bug patterns might have conflicts and dependencies among themselves. Some of the tools ranked the bug checkers with different levels of priority. However, this study has not considered the bug pattern priorities. Future work is possible toward the identification of conflicts, dependencies, and priorities of the bug patterns.

10. References

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

[2] N. Ayewah, W. Pugh, J.D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, June 13, 2007 - June 14, 2007*, San Diego, CA, United states: Association for Computing Machinery, 2007, pp. 1-7.

[3] D. Baca, B. Carlsson, and L. Lundberg, "Evaluating the cost reduction of static code analysis for software security," *3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security 2008, PLAS'08, June 8, 2008 - June 8, 2008*, Tucson, AZ, United states: Association for Computing Machinery, 2008, pp. 79-88.

[4] Coverity Prevent. [online]. Viewed 2009 august 13. Available: http://www.coverity.com/products/coverity-prevent.html

[5] C.N. Christopher, "Evaluating Static Analysis Frameworks," Carnegie Mellon University Analysis of Software Artifacts, 2006.

[6] Y. Eytani, K. Havelund, S. Stoller, and S. Ur, "Towards a framework and a benchmark for testing tools for multi-threaded programs," *Concurrency and Computation Practice & Experience*, vol. 19, Mar. 2007, pp. 267-79.

[7] Y. Eytani, K. Havelund, S. Stoller, and S. Ur, "Toward a benchmark for multi-threaded testing tools," *Concurrency and Computation: Practice and Experience*, 2005.

[8] Y. Eytani, R. Tzoref, and S. Ur, "Experience with a concurrency bugs benchmark," *2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW), 9-11 April 2008*, Piscataway, NJ, USA: IEEE, 2008, pp. 379-84.

[9] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," Los Alamitos, CA, USA: IEEE Comput. Soc, 2003, p. 7 pp.

[10] FindBugs Bug Description. [online]. Viewed 2009 august 13. Available: http://findbugs.sourceforge.net/bugDescriptions.html

[11] J.S. Foster, M.W. Hicks, and W. Pugh, "Improving software quality with static analysis," *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, June 13, 2007 - June 14, 2007*, San Diego, CA, United states: Association for Computing Machinery, 2007, pp. 83-84.

[12] H. Hallal, E. Alikacem, W. Tunney, S. Boroday, and A. Petrenko, "Antipattern-based detection of deficiencies in Java multithreaded software," *Proceedings. Fourth International Conference on Quality Software, 8-9 Sept. 2004*, Los Alamitos, CA, USA: IEEE Comput. Soc, 2004, pp. 258-67.

[13] D. Hovemeyer and W. Pugh, "Finding concurrency bugs in Java," *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs, St. John's, Newfoundland, Canada*, 2004.

[14] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, 2004, pp. 92-106.

[15] Java testing tools: Static code analysis, code review, unit testing. [online]. Viewed 2009 august 13. Available: http://www.parasoft.com/jsp/products/home.jsp?product=Jtest

[16] B.A. Kitchenham, "The case against software benchmarking, keynote lecture," *Proceedings of The European Software Measurement Conference (FESMA-DASMA 2001)*, Heidelberg, May 2001, pp 1–9.

[17] S. Lipner, The Trustworthy Computing Security Development Life Cycle, Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'o4), p.2-13, 2004.

[18] 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 & Experience*, vol. 19, Mar. 2007, pp. 281-94.

[19] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," *27th International Conference on Software Engineering, ICSE 2005, May 15, 2005 - May 21, 2005*, Saint Louis, MO, United states: Institute of Electrical and Electronics Engineers Computer Society, 2005, pp. 580-586.

[20] F. Painchaud and R. Carbone, *Java software verification tools: Evaluation and recommended methodology,* Technical Memorandum. Defence R&D Canada. Document No. TM 2005-226. March 2006. http://cradpdf.drdc.gc.ca/PDFS/unc57/p527369.pdf

[21] N. Rutar, C.B. Almazan, and J.S. Foster, "A comparison of bug finding tools for Java," *ISSRE 2004 Proceedings; 15th International Symposium on Software Reliability Engineering, November 2, 2004 - November 5, 2004*, Saint-Malo, France: IEEE Computer Society, 2004, pp. 245-256.

[22] W.F. Tichy, "Should computer scientists experiment more?," *Computer*, vol. 31, 1998, pp. 32-40.

[23] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb, "An evaluation of two bug pattern tools for Java," *2008 First IEEE International Conference on Software Testing, Verification and Validation (ICST '08), 9-11 April 2008*, Piscataway, NJ, USA: IEEE, 2008, pp. 248-57.

[24] S. Wagner, J. Jurjens, C. Roller, and P. Trischberger, "Comparing Bug finding tools with reviews and tests," *Testing of Communicating Systems. 17th IFIP TC6/WG 6.1 International Conference TestCom 2005. Proceedings, 31 May-2 June 2005*, Berlin, Germany: Springer-Verlag, 2005, pp. 40-55.

[25] M.S. Ware and C.J. Fox, "Securing Java code: heuristics and an evaluation of static analysis tools," *Proceedings of the 2008 workshop on Static analysis*, Tucson, Arizona: ACM, 2008, pp. 12-21.

[26] F. Wedyan, D. Alrmuny, and J. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," *2009 2nd International Conference on Software Testing Verification and Validation (ICST 2009), 1-4 April 2009*, Piscataway, NJ, USA: IEEE, 2009, pp. 141-50.

[27] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell and A. Wesslen, *Experimentation in Software Engineering:An Introduction*, Kluwer Academic Publishers, 2000.

[28] M.A. Wojcicki and P. Strooper, "Maximising the information gained from a study of static analysis technologies for concurrent software," *Empirical Software Engineering*, vol. 12, 2007, pp. 617-645.

[29] L. Yu, J. Zhou, Y. Yi, P. Li, and Q. Wang, "Ontology Model-Based Static Analysis on Java Programs," *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference-Volume 00*, IEEE Computer Society Washington, DC, USA, 2008, pp. 92-99.

Appendix A. Selected benchmark programs

Program name	Functionality	LOC	Bug type
Producer Consumer	Producer–consumer	203	Orphaned-Thread
			Wrong lock or No lock*
SoftWare Verification	Bubble-sort	362	Orphaned-Thread
			Not-atomic
			Lazy Initialization*
BuggedProgram	thread pingPong	272	Not-atomic
SoftTestProject	Selling airline tickets via agents	95	Not-atomic : Interleaving
			Wrong lock or No lock*
BugTester	Linked list	416	Non-atomic
MergeSortBug	Mergesort	257	Not-atomic
Manager	Memory controller	236	Not-atomic
Critical	Wrong use of critical section	73	Not-atomic
Suns account program	Bank simulation	141	Not-atomic
Buggy Program	Generates random numbers	359	Wrong-Lock or No-Lock
			Blocking-Critical-Section
			Wrong lock or No lock*
Bufwriter	Buffer read-write	255	Wrong lock or No lock
			Wring lock or No lock*
			Data Race*
			Data Race*
Account	Bank simulation	155	Wrong Lock/No Lock
bug1	Copying files	121	Deadlock
GarageManager	Workers tasks	584	Wrong lock/No Lock*
TicketsOrderSim	Selling airline tickets via agents	183	Double checked locking
Shop	Simulates the operation of shop	273	weak reality (two stage lock)
			Wrong lock or No lock*
BoundedBuffer	Producer–consumer	536	Notify instead of notifyAll
			Data race (static field in non static method)*
Test	Allocation vector	286	Weak-reality (two stage access)
IBM_Airlines	Producer–consumer	243	[Condition-For-Wait]
			Wrong lock or No lock*
Deadlock	Deadlock example	54	Hold and wait **
Total	20 Programs	5335	32 bugs

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

Appendix B. Categorization of the bug patterns/checkers of the tools

	Coverity Prevent
Category	**Bug pattern/checker**
Deadlock	LOCK_INVERSION
	LOCK_ORDERING
Race condition	DOUBLE_CHECK_LOCK (preview)
	GUARDED_BY_VIOLATION
	INDIRECT_GUARDED_BY_VIOLATION
	NON_STATIC_GUARDING_STATIC (preview)
	UNSAFE_LAZY_INIT (preview)
	ATOMICITY (preview)
Problems leading to unpredictable results	DC.EXPLICIT_DEPRECATION defect
Efficiency problems/Performance	UNEXPECTED_SYNC (preview)
	JTest
Category	**Bug pattern/checker**
Deadlock	Do not synchronize on constant Strings [TRS.SCS-1]
	Ensure 'wait()', 'notify()' and 'notifyAll()' are invoked on an object that is clearly synchronized in its enclosing method scope [TRS.NSYN-2]
	Release Locks in a "finally" block [TRS.RLF-2]
	Do not synchronize on "public" fields since doing so may cause deadlocks [TRS.SOPF-2]
	Do not call 'Thread.sleep()' while holding a lock since doing so can cause poor performance and deadlocks [TRS.TSHL-2]
	Do not use 'notify()'; use 'notifyAll()' instead so that all waiting threads will be notified [TRS.ANF-3]
	Do not cause deadlocks by calling a "synchronized" method from a "synchronized" method [TRS.CSFS-3]
Race condition	Inspect accesses to "static" fields which may require synchronization [TRS.IASF-2]
	Use the same locking object to access variables [TRS.USL-2]
	Avoid unsafe implementations of the "double-checked locking" pattern [TRS.DCL-3]
	Do not synchronize on non-"final" fields since doing so makes it difficult to guarantee mutual exclusion [TRS.SOUF-3]
	Make the get method for a field synchronized if the set method is synchronized [TRS.SSUG-3]
	Make lazy initializations thread-safe [TRS.ILI-4]
	Access related Atomic variables in a synchronized block [TRS.MRAV-4]
	Use synchronization on methods that implement 'Runnable.run()' [TRS.RUN-5]
Problems leading to unpredictable results	Avoid calling unsafe deprecated methods of 'Thread' and 'Runtime' [TRS.THRD-1]
	Call 'wait()' and 'await()' only inside a loop that tests the liveness condition [TRS.UWIL-1]
	Do not use variables of the unsafe type 'java.lang.ThreadGroup' [TRS.AUTG-3]
	Do not use 'Thread.yield()' because it may behave differently under different Virtual Machines [TRS.AUTY-3]
	Avoid invoking methods using "static" 'java.text.DateFormat' and 'java.util.Calendar' [TRS.CDF-3]
	Avoid compound synchronized collection accesses which violate atomicity [TRS.CMA-3]
	Do not call the "start" method of threads from inside a constructor [TRS.CSTART-3]
	Do not let "this" reference escape during construction [TRS.CTRE-3]
	Do not catch 'IllegalMonitorStateException' since this exception likely indicates a design flaw [TRS.IMSE-3]
	Do not call the 'run()' method directly on classes extending 'java.lang.Thread' or implementing 'java.lang.Runnable' [TRS.IRUN-3]
	Do not perform synchronization using the "synchronized" keyword on implementations of "java.util.concurrent.locks.Lock" [TRS.SOL-3]
	Avoid unsynchronized accesses of "Collections.synchronized" wrapped Collections [TRS.UACS-3]
	Use unsynchronized Collections/Maps only when safe [TRS.UCM-3]
Quality and Style Problems	Give subclasses of Thread a 'run()' method so they can run as separate threads [TRS.MRUN-2]
	Do not start a thread without specifying a 'run()' method [TRS.UT-2]
	Do not use DiscardOldestPolicy with PriorityBlockingQueue [TRS.DOPQ-3]
	Do not perform synchronization nor call semaphore methods on an Object's 'this' reference [TRS.STR-3]
	Do not catch InterruptedException except in classes extending Thread [TRS.CIET-4]
	Do not make the "writeObject()" method synchronized if no other method is synchronized [TRS.WOS-4]
	Use ConcurrentHashMap instead of Hashtable and "synchronizedMap" wrapped HashMap when possible [TRS.CHM-5]
Efficiency problems/Performance	Use 'wait()' and 'notifyAll()' instead of polling loops [TRS.UWNA-2]
	Do not use Atomic variables when always accessed in synchronized manner [TRS.AIL-4]
	Use "synchronized" blocks instead of making the whole method declaration "synchronized" [TRS.NSM-5]
General warnings/mistakes	Avoid accidental use of "Thread.interrupted()" [TRS.ATI-3]
	Ensure threads are named [TRS.NAME-3]
	Use the correct method calls on "java.util.concurrent.locks.Condition" objects [TRS.WOC-3]
	Do not use "getState" except for debugging purposes [TRS.GSD-4]
	Do not mix "static" and non-"static" "synchronized" methods [TRS.SNSM-4]

	FindBugs
Category	**Bug pattern/checker**
Deadlock	ML: Synchronization on field in futile attempt to guard that field
	MWN: Mismatched notify()
	MWN: Mismatched wait()
	DL: Synchronization on Boolean could lead to deadlock
	No: Using notify() rather than notifyAll()
	DL: Synchronization on boxed primitive could lead to deadlock
	SWL: Method calls Thread.sleep() with a lock held
	TLW: Wait with two locks held
	UL: Method does not release lock on all paths
	UL: Method does not release lock on all exception paths
	UW: Unconditional wait
	DL: Synchronization on interned String could lead to deadlock
Livelock	SP: Method spins on field
Race condition	DC: Possible double check of field
	LI: Incorrect lazy initialization of static field
	LI: Incorrect lazy initialization and update of static field
	MSF: Mutable servlet field
	UG: Unsynchronized get method, synchronized set method
	WL: Sychronization on getClass rather than class literal
	IS: Inconsistent synchronization
Problems leading to unpredictable results	JLM: Synchronization performed on java.util.concurrent Lock
	Ru: Invokes run on a thread (did you mean to start it instead?)
	SC: Constructor invokes Thread.start()
	STCAL: Call to static Calendar
	STCAL: Call to static DateFormat
	STCAL: Static Calendar
	STCAL: Static DateFormat
	Wa: Condition.await() not in loop
	Wa: Wait not in loop
	DL: Synchronization on boxed primitive values
Quality and Style Problems	NN: Naked notify
	NP: Synchronize and null check on the same field.
	RS: Class's readObject() method is synchronized
	WS: Class's writeObject() method is synchronized but nothing else is
	Dm: A thread was created using the default empty run method
	ESync: Empty synchronized block
General warnings/mistakes	VO: A volatile reference to an array doesn't treat the array elements as volatile
	Dm: Monitor wait() called on Condition
	Jlint
Category	**Bug pattern/checker**
Deadlock	Loop id: invocation of synchronized method name can cause deadlock
	Value of lock name is changed outside synchronization or constructor
	Value of lock name is changed while (potentially) owning it
	Method name.wait() is called without synchronizing on name
	Loop LoopId/PathId: invocation of method name forms the loop in class dependency graph
	Lock a is requested while holding lock b, with other thread holding a and requesting lock b
	Method wait() can be invoked with monitor of other object locked
	Call sequence to method name can cause deadlock in wait()
Race condition	Synchronized method name is overridden by non-synchronized method of derived class name
	Method name can be called from different threads and is not synchronized
	Field name of class
	Method name implementing 'Runnable' interface is not synchronized

Appendix C. Unified categorized bug patterns

SL	Bug pattern/checker	Tool
	Deadlock	
1	Do not synchronize on constant Strings [TRS.SCS-1]	JT
	DL: Synchronization on interned String could lead to deadlock	FB
2	Ensure 'wait()', 'notify()' and 'notifyAll()' are invoked on an object that is clearly synchronized in its enclosing method scope [TRS.NSYN-2]	JT
	MWN: Mismatched notify()	FB
	MWN: Mismatched wait()	FB
	Method name.wait() is called without synchronizing on name	JL
	Identifier.wait() method called without synchronizing on identifier	AL
	Ref: Cigital Java Security Rulepack # 5: http://www.cigital.com/securitypack/view/index.html	
3	Release Locks in a "finally" block [TRS.RLF-2]	JT
	UL: Method does not release lock on all exception paths	FB
	UL: Method does not release lock on all paths	FB
4	Do not synchronize on "public" fields since doing so may cause deadlocks [TRS.SOPF-2]	JT
5	Do not call 'Thread.sleep()' while holding a lock since doing so can cause poor performance and deadlocks [TRS.TSHL-2]	JT
	SWL: Method calls Thread.sleep() with a lock held	FB
	Ref: Cigital Java Security Rulepack # 8: http://www.cigital.com/securitypack/view/index.html	
6	Do not use 'notify()'; use 'notifyAll()' instead so that all waiting threads will be notified [TRS.ANF-3]	JT
	No: Using notify() rather than notifyAll()	FB
	Ref: Ken Arnold, James Gosling: "The Java Programming Language Second Edition" Addison Wesley, 1997, pp.188-190.	
7	Do not cause deadlocks by calling a "synchronized" method from a "synchronized" method [TRS.CSFS-3]	JT
	Ref: Michael Daconta, Eric Monk, J Keller, Keith Bohnenberger: "Java Pitfalls" John Wiley & Sons, ISBN: 0-471-36174-7 pp.50-60.	
8	Loop id: invocation of synchronized method name can cause deadlock	JL
	Synchronized method call in cycle of lock graph	AL
9	Loop LoopId/PathId: invocation of method name forms the loop in class dependency graph	JL
	Method call sequence leads to a cycle in lock graph	AL
	LOCK_ORDERING	CP
10	Unconditional wait(): not testing the condition for which a wait is needed	AL
	UW: Unconditional wait	FB
11	Cross Synchronization	AL
	Lock a is requested while holding lock b, with other thread holding a and requesting lock b	JL
	LOCK_INVERSION	CP
12	Method wait() invoked with another object locked	AL
	Method wait() can be invoked with monitor of other object locked	JL
	TLW: Wait with two locks held	FB
13	Call sequence can cause deadlock in wait()	AL
	Call sequence to method name can cause deadlock in wait()	JL
14	Unconditional notify() or notifyAll()	AL
15	Waiting forever: a thread executes wait() but is never notified.	AL
16	DL: Synchronization on Boolean could lead to deadlock	FB
17	DL: Synchronization on boxed primitive could lead to deadlock	FB
	DL: Synchronization on boxed primitive values	FB
	Livelock	
18	Unsynchronized spin-wait	AL
	SP: Method spins on field	FB
19	Wait stall: a thread calls wait()without specifying a timeout, (Jprobe Threadalyzer).	AL
	Race Condition	
20	Reference value is changed when it is used in synchronization block	AL
	ML: Synchronization on field in futile attempt to guard that field	FB
	Value of lock name is changed while (potentially) owning it	JL
21	Value of lock name is changed outside synchronization or constructor	JL
22	Inspect accesses to "static" fields which may require synchronization [TRS.IASF-2]	JT
	Ref: OWASP WebGoat Project "Thread Safety Problem" lesson: http://www.owasp.org/index.php/Race_condition_within_a_thread	
23	Use the same locking object to access variables [TRS.USL-2]	JT
	IS: Inconsistent synchronization	FB
	Inconsistent synchronization: fields that are used both with and without synchronization in the application class	AL
	Guarded_By_Violation	CP
	Ref: Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 27-29.	

24	Avoid unsafe implementations of the "double-checked locking" pattern [TRS.DCL-3]	JT
	DC: Possible double check of field	FB
	The double-check locking for synchronized initialization	AL
	Double_Check_Lock (preview)	CP
	Ref: 1. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html 2. http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-double.html 3. "Secure Programming with Static Analysis" by Brian Chess and Jacob West. pg. 32, 2007 Edition.	
25	Make the get method for a field synchronized if the set method is synchronized [TRS.SSUG-3]	JT
	UG: Unsynchronized get method, synchronized set method	FB
	Get-Set methods with opposite declarations	AL
26	Make lazy initializations thread-safe [TRS.ILI-4]	JT
	LI: Incorrect lazy initialization of static field	FB
	Unsafe_Lazy_Init (preview)	CP
	Ref: http://www.cs.umd.edu/~pugh/java/memoryModel/	
27	Use synchronization on methods that implement 'Runnable.run()' [TRS.RUN-5]	JT
	Non synchronized run() method	AL
	Method name implementing 'Runnable' interface is not synchronized	JL
	Ref: http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm	
28	Overriding a synchronized method	AL
	Synchronized method name is overridden by non-synchronized method of derived class name	JL
29	Non synchronized method called by more than one thread	AL
	Method name can be called from different threads and is not synchronized	JL
30	Unprotected non volatile field used by several threads	AL
	Field name of class	JL
31	LI: Incorrect lazy initialization and update of static field	FB
32	MSF: Mutable servlet field	FB
33	WL: Sychronization on getClass rather than class literal	FB
34	Indirect_Guarded_By_Violation	CP
35	Access related Atomic variables in a synchronized block [TRS.MRAV-4]	JT
	Ref: Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 23-25.	
36	Non_Static_Guarding_Static (preview)	CP
37	Atomicity (preview)	CP
	Problems leading to unpredictable results	
38	Avoid calling unsafe deprecated methods of 'Thread' and 'Runtime' [TRS.THRD-1]	JT
	DC.Explicit_Deprecation defect	CP
	Improper method calls: e.g. deprecated methods, use of Thread unsafe methods	AL
39	Do not call the 'run()' method directly on classes extending 'java.lang.Thread' or implementing 'java.lang.Runnable' [TRS.IRUN-3]	JT
	Ru: Invokes run on a thread (did you mean to start it instead?)	FB
	Improper method calls: e.g. deprecated methods, use of Thread unsafe methods	AL
40	Call 'wait()' and 'await()' only inside a loop that tests the liveness condition [TRS.UWIL-1]	JT
	Wa: Wait not in loop	FB
	Wa: Condition.await() not in loop	FB
	wait() is not in loop	AL
	Ref: Joshua Bloch: "Effective Java - Programming Language Guide" Addison Wesley, 2001, pp. 201 - 203	
41	Avoid invoking methods using "static" 'java.text.DateFormat' and 'java.util.Calendar' [TRS.CDF-3]	JT
	STCAL: Call to static Calendar	FB
	STCAL: Call to static DateFormat	FB
	STCAL: Static Calendar	FB
	STCAL: Static DateFormat	FB
	Ref: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6178997	
42	Do not call the "start" method of threads from inside a constructor [TRS.CSTART-3]	JT
	SC: Constructor invokes Thread.start()	FB
	Ref: Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 39-42.	
43	Do not let "this" reference escape during construction [TRS.CTRE-3]	JT
	Ref: Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 39-42.	
44	Do not catch 'IllegalMonitorStateException' since this exception likely indicates a design flaw [TRS.IMSE-3]	JT
45	Do not perform synchronization using the "synchronized" keyword on implementations of "java.util.concurrent.locks.Lock" [TRS.SOL-3]	JT
	JLM: Synchronization performed on java.util.concurrent Lock	FB
46	Avoid unsynchronized accesses of "Collections.synchronized" wrapped Collections [TRS.UACS-3]	JT
	Ref: 1. Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 58-60.	

	2. Collections, http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html	
47	Use unsynchronized Collections/Maps only when safe [TRS.UCM-3]	JT
48	Double call of the start() method of a thread	AL
49	Start() method call in constructor	AL
50	Do not use 'Thread.yield()' because it may behave differently under different Virtual Machines [TRS.AUTY-3]	JT
	Ref: 1. Joshua Bloch: "Effective Java - Programming Language Guide" Addison Wesley, 2001, pp. 211 2. https://developer.opencloud.com/devportal/display/ENGBLOG/2009/03/11/Avoid+ThreadGroup+in+Resource+Adaptors	
51	Do not use variables of the unsafe type 'java.lang.ThreadGroup' [TRS.AUTG-3]	JT
	Ref: 1. Joshua Bloch: "Effective Java - Programming Language Guide" Addison Wesley, 2001, pp. 204-207 2. Cigital Java Security Rulepack # 4: http://www.cigital.com/securitypack/view/index.html	
52	Avoid compound synchronized collection accesses which violate atomicity [TRS.CMA-3]	JT
	Ref: 1. Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 71-74. 2. Collections, http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html	
Quality and Style Problems		
53	Do not synchronize on non-"final" fields since doing so makes it difficult to guarantee mutual exclusion [TRS.SOUF-3]	JT
54	Blob thread: a thread that implements most of program functionality.	AL
55	Unnecessary notification: when no threads are waiting.	AL
56	Premature join() call: joining a thread which has not started yet, (FLAVERS).	AL
57	Dead interactions: calling already terminated threads, (FLAVERS).	AL
58	ESync: Empty synchronized block	FB
59	NN: Naked notify	FB
60	NP: Synchronize and null check on the same field.	FB
61	RS: Class's readObject() method is synchronized	FB
62	Give subclasses of Thread a 'run()' method so they can run as separate threads [TRS.MRUN-2]	JT
	Do not start a thread without specifying a 'run()' method [TRS.UT-2]	JT
	Dm: A thread was created using the default empty run method	FB
63	Do not use DiscardOldestPolicy with PriorityBlockingQueue [TRS.DOPQ-3]	JT
	Ref: Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 174-175.	
64	Do not perform synchronization nor call semaphore methods on an Object's 'this' reference [TRS.STR-3]	JT
65	Use ConcurrentHashMap instead of Hashtable and "synchronizedMap" wrapped HashMap when possible [TRS.CHM-5]	JT
	Ref: 1. ConcurrentHashMap, http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ConcurrentHashMap.html 2. Hashtable, http://java.sun.com/j2se/1.5.0/docs/api/java/util/Hashtable.html 3. HashMap, http://java.sun.com/j2se/1.5.0/docs/api/java/util/HashMap.html 4. Collections, http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html 5. Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 242-243.	
66	Do not make the "writeObject()" method synchronized if no other method is synchronized [TRS.WOS-4]	JT
	WS: Class's writeObject() method is synchronized but nothing else is	FB
67	Do not catch InterruptedException except in classes extending Thread [TRS.CIET-4]	JT
	Ref: 1. Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 92-94. 2. PCI Data Security Standard: https://www.pcisecuritystandards.org/security_standards/pci_dss.shtml	
Efficiency problems/Performance		
68	Use 'wait()' and 'notifyAll()' instead of polling loops [TRS.UWNA-2]	JT
	Ref: Peter Haggar: "Practical Java - Programming Language Guide". Addison Wesley, 2000, pp.191 - 193	
69	Do not use Atomic variables when always accessed in synchronized manner [TRS.AIL-4]	JT
	Ref: Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 29-32.	
70	Use "synchronized" blocks instead of making the whole method declaration "synchronized" [TRS.NSM-5]	JT
	Excessive synchronization: a synchronized method contains several operations that do not need synchronization.	AL
	Ref: Peter Haggar: "Practical Java - Programming Language Guide". Addison Wesley, 2000, pp.116-122 and pp.173-176	
71	Overuse of synchronized methods: a method is synchronized even if used by only one thread.	AL
72	Synchronized read only methods while there are no methods with a write access.	AL
73	Internal call of a method	AL
74	Object locked but not used: it is likely that synchronization is not needed	AL
75	Overthreading: defining too many threads.	AL
76	Irresponsive or slow interface (Complex computation in an AWI/Swing thread.)	AL
77	Misuse of notifyAll(): notify() could be safely used, e.g., in the case of an object shared by only two threads.	AL
78	Synchronized immutable object	AL
79	Synchronized atomic operations.	AL
80	Calling join() to an immortal thread,	AL

81	UNEXPECTED_SYNC (preview)	CP
	General warnings/mistakes	
82	Use the correct method calls on "java.util.concurrent.locks.Condition" objects [TRS.WOC-3]	JT
	Dm: Monitor wait() called on Condition	FB
83	VO: A volatile reference to an array doesn't treat the array elements as volatile	FB
84	Avoid accidental use of "Thread.interrupted()" [TRS.ATI-3]	JT
	Ref: Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 138-141.	
85	Ensure threads are named [TRS.NAME-3]	JT
86	Do not use "getState" except for debugging purposes [TRS.GSD-4]	JT
	Ref: Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency In Practice. Stoughton: Addison-Wesley, 2007. pp. 250-252.	
87	Do not mix "static" and non-"static" "synchronized" methods [TRS.SNSM-4]	JT

CP=Coverity Prevent, JT=Jtest, FB=FindBugs, JL= Jlint, AL=Antipattern Library