Information and Software Technology 105 (2019) 226–251

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Empirical research on concurrent software testing: A systematic mapping study

Silvana M. Melo[a,*], Jeffrey C. Carver[b], Paulo S.L. Souza[a], Simone R.S. Souza[a]

[a] Institute of Mathematics and Computer Sciences, University of Sao Paulo, Avenida Trabalhador Sao-carlense, 400 - Centro, Sao Carlos, SP CEP: 13566-590, Brazil
[b] Department of Computer Science, University of Alabama, 3441 SEC, Box 870290, Tuscaloosa, AL 35487, USA

A R T I C L E   I N F O

Keywords:
Empirical software engineering
Concurrent programming
Software testing
Systematic mapping

A B S T R A C T

Background: Concurrent software testing is a costly and difficult task, especially due to the exponential increase in the test sequences caused by non-determinism. Such an issue has motivated researchers to develop testing techniques that select a subset of the input domain that has a high probability of revealing faults. Academics and industrial practitioners rarely use most concurrent software testing techniques because of the lack of data about their applicability. Empirical evidence can provide an important scientific basis for the strengths and weaknesses of each technique to help researchers and practitioners choose concurrent testing techniques appropriate for their environments.

Aim: This paper gathers and synthesizes empirical research on concurrent software testing to characterize the field and the types of empirical studies performed.

Method: We performed a systematic mapping study to identify and analyze empirical research on concurrent software testing techniques. We provide a detailed analysis of the studies and their design choices.

Results: The primary findings are: (1) there is a general lack of empirical validation of concurrent software testing techniques, (2) the type of evaluation method varies with the type of technique, (3) there are some key challenges to empirical study design in concurrent software testing, and (4) there is a dearth of controlled experiments in concurrent software testing.

Conclusions: There is little empirical evidence available about some specific concurrent testing techniques like model-based testing and formal testing. Overall, researchers need to perform more empirical work, especially real-world case studies and controlled experiments, to validate properties of concurrent software testing techniques. In addition, researchers need to perform more analyses and synthesis of the existing evidence. This paper is a first step in that direction.

1. Introduction

The availability of multicore processors and inexpensive clusters has increased the demand both for concurrent applications and for testing techniques for their validation. Modern business applications use concurrency to improve overall system performance, consequently, researchers have developed a variety of testing techniques for concurrent software. However, testing teams generally rely on their own knowledge and experience when choosing technique(s) for each project, resulting in the repeated selection of same technique(s), whether or not they are most appropriate.

One of the difficulties in transfer of knowledge and research results from academia to industry is the lack of evidence of the applicability of results and techniques to specific software projects [1]. Such evidence should be gathered via empirical software engineering (ESE) methods that provide insights into the benefits and limits of each technique [2]. Secondary studies on concurrent software testing focus on the categorization of testing techniques, methodologies, and tools [3–6] (see Section 2 for more details). However, empirical validation for concurrent software testing techniques is still lacking [1,2].

Researchers use controlled experiments, case studies, and surveys as empirical methods to evaluate new techniques and new research. These methods produce data that helps researchers and practitioners decide whether a technique is appropriate for a given context. These empirical methods also help researchers identify specific factors that impact on the effectiveness of techniques and lead to empirically-based decisions about research and practice [7]. Therefore, they provide an important scientific basis for software engineering [8].

* Corresponding author.
E-mail address: morita@icmc.usp.br (S.M. Melo).

https://doi.org/10.1016/j.infsof.2018.08.017
Received 7 December 2017; Received in revised form 27 August 2018; Accepted 31 August 2018
Available online 1 September 2018

This paper discusses a systematic mapping study to understanding the current state-of-the-art of the empirical research on concurrent software testing research. The overall goal is to gather and synthesize empirical research and help developers evaluate the strength of evidence for the findings. The systematic mapping aims at:

1. providing an overview of the empirical studies about concurrent software testing;
2. identifying the concurrent software testing techniques that have empirical studies and the type of validation approach used;
3. analyzing the strength of the empirical studies and discussing the findings and limitations of the evidence;
4. discussing the design of the empirical studies along with the challenges and research opportunities;
5. identifying gaps in empirical research on concurrent software testing; and
6. providing guidance for the design of empirical studies about concurrent software testing.

The remainder of the paper is organized as follows. Section 2 provides an overview of the related work. Section 3 describes the systematic mapping protocol. Section 4 discusses the strengths of the empirical studies identified in the mapping study. Section 5 answers the research question trough the study results. Section 6 addresses the way researchers conduct empirical studies in for concurrent software testing and provides a guide for the planning of new studies. Section 7 discusses the limitations and validity threats of the mapping study. Finally, Section 8 provides the paper conclusions and suggests future work.

2. Related work

This section addresses some fundamentals about concurrent software testing and provides an overview of prior work. It also provides a brief overview of the types of empirical studies relevant to this review.

2.1. Concurrent software testing

Compared with sequential programming, concurrent programming provides features that (1) increase the efficiency of execution time, (2) avoid idle resources, and (3) reduce computational costs [9]. However, testing concurrent software raises some challenges. Concurrent (or parallel) execution of processes (or threads) leads to non-deterministic behavior, i.e, the same input may lead to multiple different (but correct) outputs. Therefore, the testing activity must analyze the outputs from the execution of different synchronization sequences. One such approach, the deterministic execution technique [10], forces the execution of a specific synchronization sequence for a given input to obtain the same output in a non-deterministic environment.

Other issues related to communication and synchronization between processes (or threads) also pose challenges for concurrent software testing [11]. These challenges include:

- development of techniques for static analysis;
- detection of errors related to synchronization, communication, data flow, deadlocks, livelocks, data races, and atomicity violation;
- adaptation of sequential programming testing techniques for concurrent software;
- definition of data flow criteria for message-passing and shared-memory;
- generation of automatic test data;
- efficient exploration of interleaving events;
- deterministic reproduction of a synchronization sequence; and
- representation of a concurrent program in a way that captures testing information.

Testing techniques can be grouped into families based on test case selection criteria. Techniques in the same family are similar regarding the information required for test case selection or generation (e.g. source code or specifications) or the aspect of the code exercised by the test cases (e.g., control flow, data flow, or typical errors) [2].

Researchers have adapted sequential testing techniques (e.g. structural, model-based, formal-method based, and fault-based) for use with concurrent software [12–15]. Researchers have also developed techniques that specifically take into account concurrent software features like non-determinism, synchronization, and communication [16–20].

2.2. Secondary studies on concurrent software testing

Researchers have conducted systematic reviews to identify and categorize the large number of proposed methods for concurrent software testing. Table 1 provides an overview of these literature reviews. The remainder of this section describes some details of the studies, their comparison and contrasts, and their limitations.

Mamun and Khanam [6] focused on concurrent software testing in shared-memory systems. The primary topics covered include concurrent software issues, bug types, testing techniques and tools, test case generation techniques and tools, and benchmarks of programs. The review classified each testing technique as either static or dynamic and identified the programming languages supported by the testing tools. The review identified 51 testing techniques, 38 testing tools, and 87 unique Java multithreaded bug patterns.

Souza et al. [4] focused on studies of concurrent software testing approaches, bug classification, and testing tools. They reused classifications from a previous review and added a classification based on programming paradigm (message-passing vs. shared-memory). The study reviewed the state-of-the-art of concurrent software and identified: 1) 166 papers that propose a new approach, mechanism, or framework for concurrent software testing, 2) 6 papers that address a taxonomy, classification, or discussion about concurrent bugs, and 3) 50 papers that present a tool or automated methodology that supports concurrent software testing.

Arora et al. [3] classified concurrent software testing approaches into eight categories, namely 1) reachability testing, 2) structural testing, 3) model-based testing, 4) mutation-based testing, 5) slicing-based testing, 6) formal methods, 7) random testing, and 8) search-based testing. They further classified the approaches as either algorithms (80), graphical representations for specific techniques (11), or tools that implement a specific methodology (83).

The previous systematic reviews focused on the state-of-the-art of concurrent software testing and categorized techniques, approaches, methods, types of defects revealed, and supporting tools. Their primary weakness, however, is the lack of discussion about technique evaluation (e.g. with empirical studies). This evaluation information is fundamental for analyzing the maturity of testing technique knowledge, including their strengths and weaknesses [2]. Although other studies have examined empiricism in software testing (e.g. [2,21–23]), their focus was not on concurrent software. This current review aims to remedy this deficiency by focusing on the empirical evidence about concurrent software testing techniques.

The results of our systematic mapping study are comparable to previous reviews because we include a similar set of primary studies and use the same classification of testing techniques. However, our study is different because it focuses on empirical studies and addresses how researchers have evaluated (or not) concurrent testing techniques. This focus helps to identify gaps that can be addressed by future research. This review also provides practitioners with empirical evidence about different concurrent testing approaches.

2.3. Background on empirical studies

Authors often use inconsistent terminology when describing empirical research methods. The literature reports several taxonomies for empirical evaluations (or research methods) [24–28]. In this paper, we used a classification appropriate for empirical software engineering [26]:

Table 1
Secondary studies on concurrent software testing research.

Author/Reference	Year	Period	# Papers		Focus
			Returned	Included	
Mamun and Khanam [6]	2009	1989 - 2009	4352	109	Mapping studies of testing in shared-memory programs. Evaluation of static analysis tools for detection of Java concurrent bugs.
Brito et al. [5] and Souza et al. [4]	2011	1988 - 2011	1166	175	Identification of studies related to concurrent testing techniques, bug classification, and testing tools.
Arora et al. [3]	2015	1983 - 2014	4175	179	Classification of testing approaches into eight categories, namely reachability testing, model-based testing, mutation-based testing, slicing-based testing, structural testing, search-based testing, formal testing, and random testing.
Our Study	**2016**	**1978 - 2016**	**7101**	**109**	**Collection and classification of empirical evidence in concurrent software testing.**

Table 2
Research questions.

Research questions	Motivational factor
RQ-1 What are the general trends of the empirical evaluation of concurrent testing approaches?	To provide an overview of empirical research in the concurrent software testing area.
RQ-2 What empirical methods have been used in the concurrent software testing area?	To identify the empirical methods adopted for evaluations in the concurrent software testing area.
RQ-3 What testing technique and approach were empirically evaluated?	To identify testing techniques that provide empirical evidence of their applicability.
RQ-4 What languages are supported by the testing techniques evaluated?	To classify the languages covered by the concurrent software testing techniques.
RQ-5 What aspects and response variables are analyzed for the empirical method?	To analyze the quality aspects of the testing technique evaluated by the empirical studies.
RQ-6 What studies provide comparisons among different approaches?	To obtain information on the effectiveness among different techniques.
RQ-7 What target programs are tested in the empirical evaluation?	To define the type of software used a subject in the empirical evaluations.
RQ-8 What types of statistical analysis are used in the empirical evaluation?	To identify the type of statistical analysis applied for the obtaining of the conclusions on the empirical studies.
RQ-9 What threats to validity are discussed in the empirical studies?	To evaluate the trustworthiness of the empirical study conclusions.
RQ-10 What are the most relevant publication channels in the context of concurrent software testing?	To list the most relevant publication channels in the field and provide a guide of references or publication alternatives.

- **Case study**: an empirical study that investigates a contemporary phenomenon in a realistic/representative real-life situation [29]. It answers a how or why question using data that associate the answer with the data sources and provides a deep understanding of the reasons and motivation behind phenomena of interest. We included exploratory studies (which lack a hypothesis and/or a real-world case, but otherwise have the same goal as a case study) as case studies, similar to previous SE research [30].
- **Controlled experiment and quasi-experiment**: an empirical study that investigates causal relationships and processes. Experiments involve at least a treatment, an outcome, a measure, units of assignment, and some comparison attributed to the treatment. The researcher manipulates a factor or a variable to test its effects on another factor or variable. An experiment is typically driven by a formal hypothesis. In a controlled experiment, the researcher randomly assigns subjects to treatments, whereas in a quasi-experiment, subjects are assigned to treatments in some other controlled way [28].
- **Literature Survey**: An examination of published literature for the characterization or analyses of research on a phenomenon of interest.

3. Systematic mapping plan

The following subsections describe the steps of the mapping protocol based on the model by Petersen et al. [31].

3.1. Research questions

Table 2 lists each research question along with its motivation. The PICO framework for this review is:

- Population: Concurrent software testing techniques.
- Intervention: Contributions of empirical studies to the validation of testing techniques for concurrent software.
- Comparison: Not applicable.

Table 3
Inclusion and exclusion criteria.

Inclusion criterion
IC-1 Studies that propose testing techniques for concurrent software and apply empirical studies for the evaluation of the proposed technique.
Exclusion criteria
EC-1 Studies that do not propose concurrent software testing techniques, studies that propose software testing techniques but are not related to concurrent applications, and studies that propose techniques only for hardware testing.
EC-1.1 Studies that propose concurrent program testing with no empirical evaluation.
EC-1.2 Studies whose author claims it includes empirical evaluation, however the process is not described in the paper.
EC-2 Studies not written in English and studies not accessible in full-text.

- Outcome: Empirical evidence on concurrent software testing.

3.2. Selection criteria

Table 3 provides the inclusion/exclusion criteria.

3.3. Control group of primary studies

We used a set of six primary studies to verify the completeness of the keyword search. To build this set, we chose highly-cited papers from previous reviews. Table 4 contains a list of these papers.

3.4. Definition of the search string

The most important terms for this review are *concurrent application* and *software testing*. Therefore, we built the search string using these terms and their synonyms. We applied the search string to the IEEExplore, ACM, Scopus, Web of Science, CiteseerX, and ScienceDirect databases because of their comprehensiveness and extensive use in software engineering secondary studies [37]. Table 5 lists the search strings

Table 4
Control group of primary studies.

Reference	Title
Hong et al. [32]	Are concurrency coverage metrics effective for testing? A comprehensive empirical investigation
Alowibdi and Stenneth [33]	An empirical study of data race detector tools
Souza et al. [13]	Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs
Gligoric et al. [34]	Efficient mutation testing of multithreaded code
Lu et al. [35]	Learning from mistakes: a comprehensive study on real world concurrency bug characteristics
Koppol et al. [36]	Incremental integration testing of concurrent programs

for each database. Since IEEExplore limits searches to 15 terms, we constructed two complementary strings for this database. The search includes papers published through the first half of 2016.

3.5. Review execution

Fig. 1 illustrates the four main phases of the selection process:

1. **Phase 1: Papers identification** – we executed the search string in each electronic database and retrieved the resulting studies.
2. **Phase 2: Title and abstract selection** – we analyzed the titles and abstracts of each study from Phase 1 to identification those that met the inclusion/exclusion criteria.
3. **Phase 3: Paper screening** – we scanned the studies selected in Phase 2 to identify studies that included empirical evaluations.
4. **Phase 4: Full text reading** – we carefully read each paper remaining after Phase 3 and kept only those relevant to the research questions.

For Phases 3 and 4, the first author performed the task. Then the second and third authors used the following validation process to reduce researcher bias. Using a randomly selected set of studies, the second and third authors applied the same selection criteria as used by the first author. We then compared the results and discussed any disagreements to arrive at a consensus. We used 5% of the papers that remained after Phase 2 for the validation of Phase 3. We used 10% of the papers that remained after Phase 3 for the validation of Phase 4. The result of Cohen's Kappa measuring the agreement between the raters was =0.93, which is classified as a "very good" agreement [38].

This selection process resulted in the inclusion of 109 papers out of the initial set of 7262 returned by the database search in Phase 1.

3.6. Data extraction

An existing data extraction form defined by Dybå and Dingsøyr [39] included all of the data items necessary to answer the research questions. Therefore, we used this form (see Appendix A).

4. Strength of empirical evidence

This section discusses the strengths of the empirical studies conducted to validate concurrent software testing techniques. We use the grouping in Table 6 [40], which differ based upon the information used for test data selection, to organize of the discussion.

The following subsections provide an overview of each type of technique, the main conclusions drawn from the studies about the techniques, and the limitations of those studies. Appendix B provides the details from each included study. First, Section 4.1 describes the limitations common across the studies.

4.1. Common limitations

The following list contains the limitations common across the empirical studies on concurrent software testing techniques:

1. The programs and benchmarks used are smaller than 1000 LOC and may not be representative of real industrial applications, which reduces the generalization of the results.
2. The studies are primarily individual case studies. Confounding factors hamper the researcher's ability to draw cause-effect conclusions from individual case studies.
3. Most papers use *random testing* as the comparison for determining the strengths and weaknesses of the studied techniques. Although these results were expected, they would be more robust if researchers used other techniques for comparison.
4. The conclusions are based primarily upon descriptive rather than inferential statistics. Descriptive statistics usually provide a starting point and not an end point for statistical analyses. The use of inferential statistics enables researchers to draw stronger conclusions about properties of the population and to investigate hypotheses.

Table 5
Search strings.

Database	String
ACM	(Abstract:(("concurrent code") OR ("concurrent software") OR ("concurrent program") OR ("concurrent application") OR ("concurrent algorithm") OR ("parallel code") OR ("parallel software") OR ("parallel program") OR ("parallel application") OR ("parallel algorithm") OR ("multi-thread*") OR ("multithread*")) AND ("test*")))

IEEE	1(("Abstract":"concurrent code" OR "Abstract":"concurrent software" OR "Abstract":"concurrent program" OR "Abstract":"concurrent application" OR "Abstract":"concurrent algorithm" OR "Abstract":"parallel code") AND ("Abstract":"test*"))
	2(("Abstract":"parallel software" OR "Abstract":"parallel program" OR "Abstract":"parallel application" OR "Abstract":"parallel algorithm" OR "Abstract":"multi-thread*" OR "Abstract":"multithread*") AND ("Abstract":"test*"))

CiteseerX	((abstract:("concurrent code") OR abstract:("concurrent software") OR abstract:("concurrent program") OR abstract:("concurrent application") OR abstract:("concurrent algorithm") OR abstract:("parallel code") OR abstract:("parallel software") OR abstract:("parallel program") OR abstract:("parallel application") OR abstract:("parallel algorithm") OR abstract:("multi-threaded") OR abstract:("multithreaded")) AND (abstract:("test") OR abstract:("testing")))
Web of Science	TS=((("concurrent code" OR "concurrent software" OR "concurrent program" OR "concurrent application" OR "concurrent algorithm" OR "parallel code" OR "parallel software" OR "parallel program" OR "parallel application" OR "parallel algorithm" OR "multi-thread*" OR "multithread*") AND ("test*")) NOT ("simulation" OR "extensive test*" OR "hardware test*" OR "performance test*" OR "GPU"))
Scopus	ABS ((("concurrent code" OR "concurrent software" OR "concurrent program" OR "concurrent application" OR "concurrent algorithm" OR "parallel code" OR "parallel software" OR "parallel program" OR "parallel application" OR "parallel algorithm" OR "multi-thread*" OR "multithread*") AND ("test*")) AND NOT ("simulation" OR "extensive test*" OR "hardware test*" OR "performance test*" OR "GPU")) AND (LIMIT-TO (SUBJAREA , "COMP"))
ScienceDirect	Abstract(("concurrent code" OR "concurrent software" OR "concurrent program" OR "concurrent application" OR "concurrent algorithm" OR "parallel code" OR "parallel software" OR "parallel program" OR "parallel application" OR "parallel algorithm" OR "multi-thread*" OR "multithread*") AND ("test*"))

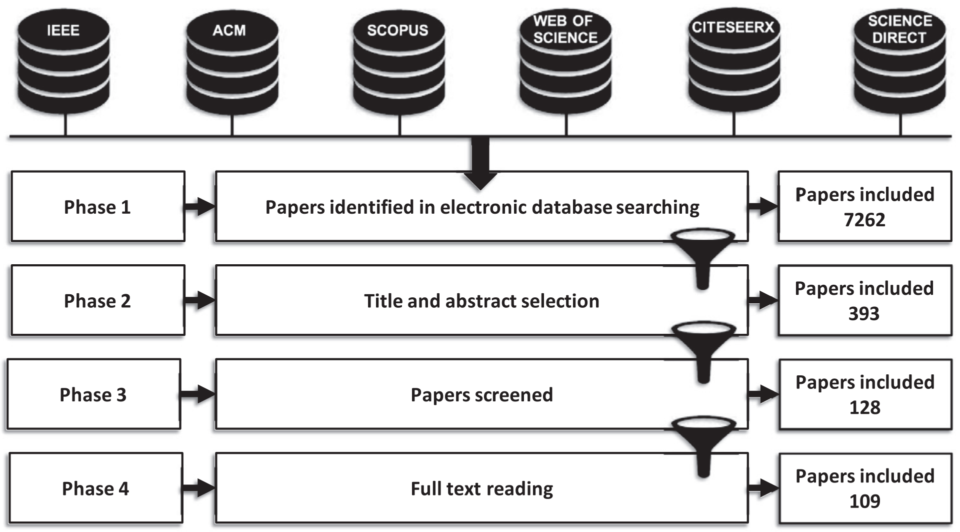

Fig. 1. Study selection procedure.

Table 6
Classification of testing techniques.

Technique	Definition
Structural testing	Uses knowledge of source code internal structures to derive test cases for testing. Some testing criteria include node, edge and path coverage.
Model-based testing	Uses an abstract model to specify the behavior of the program under test and extract information to assist software testing.
Fault-based testing	Uses knowledge of programmers' typical errors to guide the generation of test cases. Its basic objective is the selection of test cases to distinguish the program under test from the derived programs that contain hypothetical faults.
Formal method-based testing	Uses mathematical techniques for specification, development, and verification of software and hardware systems.

5. The dependent variables and their measures differ across studies, which hinders the comparison of results.
6. Several papers do not describe threats or aspects that may negatively affect the validity of results. A discussion of validity threats is important to establish the trustworthiness and quality of the results.

4.2. *Structural testing*

Structural (white-box) testing uses knowledge of the internal structure of the software to derive test cases. Researchers have adapted control and data-flow testing criteria from sequential programming for use in concurrent programming [12,13,41]. Researchers also developed criteria specifically for concurrent software that focus on communication and synchronization [16,32,42–44]. Table B.13 lists the papers that contain empirical studies on structural testing techniques. Our analysis of these studies leads to the following conclusions.

- Similar to sequential software, testing concurrent software based on control and communication flows is generally cheaper than the testing based on data-flow or message passing [12].
- A composite approach combining structural and reachability testing requires lower computational cost than structural testing alone and achieves maximum coverage and higher defect detection with fewer test cases [13].
- Concurrent testing coverage metrics often have a moderate to strong ability to reveal faults. Furthermore, classical sequential coverage metrics are not useful for improving the testing of concurrent programs [44]. Combinations of coverage metrics provide more reliable performance across different programs, specifically for data test generation, fault detection, and difficult-to-cover test requirements [43].

Common limitations 2, 4, and 5 (Section 7) apply to these conclusions.

In summary, half of the empirical studies on structural testing are controlled experiments. Few studies compare different techniques and the lack of common metrics or studies designs make comparison of results of different testing approaches difficult.

4.3. *Fault-based testing*

The non-determinism of concurrent applications introduces faults not found in sequential applications, e.g. atomicity violation, deadlock, and data races. Researchers have developed testing approaches based on communication and synchronization among threads to identify these faults [45–51]. Atomicity-violation bugs occur when concurrent execution violates the atomicity of certain code regions [52–56]. Table B.14 lists the papers that describe empirical studies on fault-based testing techniques. Our analysis of these studies leads to the following conclusions.

- The Ctrigger [52–54], Kivati [55], and Intruder [56] tools detect concurrency bugs and the empirical results show they can find atomicity violation bugs with reasonable overhead;
- Researchers have developed tools based on the *locking discipline* or the *happens-before* relations for detection of races in multithreaded applications. Examples of these tools include RACEZ [45], DrFinder [46], ThreadSanitizer [47], Portend + [48], rccjava [49], NARADA [57], SAM [58], Deva [59], DROIDRACER [60], Dialyzer [61], TRADE [62] and CLAP [63]. Empirical evaluations show the tools effectively detect data race bugs.
- Tools, such as ConLock [50], OMEN1 [51], ASN [64], analyze multithreaded programs and predict deadlocks. The empirical results show an increasing effectiveness in detecting deadlocks and a decreasing number of false positives [35].

Mutation testing, a fault-based technique, uses information about common mistakes made by developers in the generation of mutant programs. The mutants are similar to the original program, with minor syntactical changes produced by the mutation operators. Concurrent software requires additional mutation operators to handle communication and synchronization mistakes and seek alternatives for deterministic re-execution [15,34,65,66]. Our analysis of these studies leads to the following conclusions.

- Operator-based selection (ConMAn operators) is slightly better than random mutant selection, since it reduces the number of mutants generated for testing tools [66].
- Selective mutation can optimize the execution of mutants by reducing the state space explored by each mutant [15].
- Sequential and concurrent mutation operators are independent, which highlights the importance of concurrent mutation operators [15]. MuTMuT [34,65], an optimization for generating mutants for multithreaded code, reduces the state space and the amount of time required to execute the mutants by up to 77%.

Common limitations 2, 4, and 6 apply to these conclusions. Nondeterminism increases the cost of mutation testing, therefore, the results show the main concern of researchers regarding mutation testing is execution time, rather than effectiveness.

4.4. *Formal method-based testing*

Formal methods can automate the testing process, and in particular, the derivation of test cases from formal specifications [67]. Formal methods are important and useful in computer science, as evidenced by the variety of theoretical research in the technical literature. Researchers have developed a number of formal languages and verification techniques, along with supporting tools, to check properties of formal system descriptions.

Researchers have applied formal methods to concurrent testing [47,49,68–80]. Table B.15 lists the papers that describe empirical studies on formal method-based testing techniques. Our analysis of these studies leads to the following conclusions.

- The state-space coverage increases at an even faster rate when partial-order reduction is performed through the iterative context bounding approach than with other search methods, since it limits the number of scheduler choices without limiting the depth of the execution [77].
- The schedule bounding approach is superior to depth-first search because many, but not all, defects can be found by a small schedule bound [77].
- Schedule bounding is more effective at finding bugs than unbounded depth-first search [69,74,76,79].
- Preemption sealing uses a scheduling strategy to increase the efficiency and efficacy of run-time tools and detect concurrency errors [80].
- The dynamical partial order reduction (TransDpor) technique leverages transitivity to speed-up exploration by up to two orders of magnitude over the existing partial order reduction (POR) techniques [77].
- Compared with state-based or event-based frameworks, a state/event formalism facilitates the formulation of appropriate specifications and substantially improves verification time and memory usage [73].
- Use of transactional memory schedules for unit testing or use of a synthetic description written in a domain-specific language reduces the runtime overhead by dynamically executing the workload [75].
- Artificial intelligence automated planning techniques [68] collect information that can be successfully used to predict program runs that violate generic correctness specifications and new runs that contain bugs with violation patterns defined for concurrent software.

Common limitations 2, 3, 4, and 6 apply to these conclusions. In summary, the results show formal techniques are useful and achieve good performance relative to effectiveness.

4.5. *Model-based testing*

Testers use models (normally formal models), which specify program behaviors, to generate test sequences [81]. Model checking is the most common model-based testing approach for concurrent software [18,50,81–90]. However, it may lead to a state-space explosion problem. Researchers have explored state-space reduction to avoid this issue. Table B.16 lists the papers that describe empirical studies on formal method-based testing techniques. Our analysis of these studies leads to the following conclusions.

- Labeled transition systems (LTS) with a modular approach can considerably reduce the number of test sequences generated and executed during model-based testing. Modular testing may require significantly fewer test sequences than non-modular testing [81].
- Techniques that analyze a predictive run and generate scheduling constraints with respect to a given cycle (constraint-based approach) [50] for a dynamic checking of deadlocks can be both effective and efficient.
- The value-schedule-based technique [84] can achieve a higher coverage of program behaviors while testing thread schedules.
- Reachability testing [86] ensures every partially ordered synchronization sequence of a program with a given input is exercised exactly once. Empirical studies show a good speedup in the number of sequences exercised during reachability testing and a high fault detection ability.
- Dynamic symbolic execution with unfoldings typically requires fewer test executions. However, it is slower per test execution when compared to DPOR [90].
- The Estimation of Distribution Algorithms (EDA) [89] saves computational and manual effort when building and debugging concurrent systems. Such savings are achieved through the reuse of information from earlier constructed models.
- The model reusing approach, which uses information from earlier constructed models, can save computational effort and improve fault detection [18].

Common limitations 2, 3, 4, and 5 apply to these conclusions.

Model checkers systematically control the thread scheduler to explore all possible program behaviors. Because they are systematic and exhaustive in nature, they can often prove a concurrent system satisfies its desired properties. A key problem with model checking is it does not scale with program size because the number of possible interleavings of a concurrent program often grows exponentially as a function of the length of execution (the well-known state-explosion problem).

In summary, the empirical results show the main concern of researchers is state-space explosion and solutions for avoiding it. Researchers need to conduct more empirical research to compare the effectiveness, in terms of revealing faults, of model-based testing with the effectiveness of other testing techniques [83].

4.6. *Summary*

The results in this section show a general need for additional empirical studies about concurrent software testing, specifically those that compare techniques. Researchers must also replicate studies, so that the community can draw broader and deeper conclusions on the approaches. To help the establishment of stronger causal relationships, researchers must conduct more controlled experiments, with stronger internal validity [91]. Such experiments can complement the findings of more frequent case studies. Finally, a full discussion of validity threats is fundamental for ensuring reasonable and trustworthy results. The lack of information on validity threats hinders the replication of studies and aggregation of data across this topic.

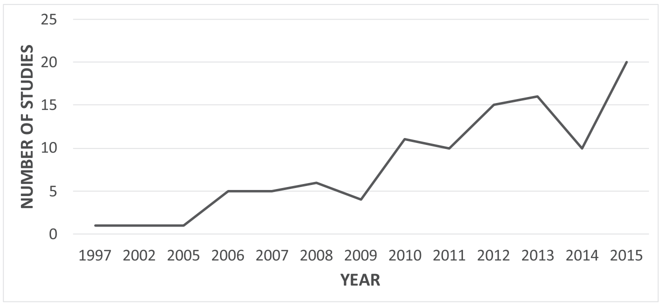

Fig. 2. Publications per year.

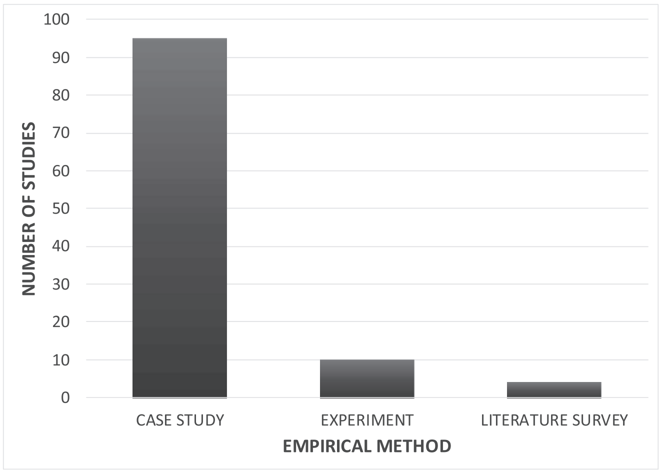

Fig. 3. Distribution of studies per type of empirical method.

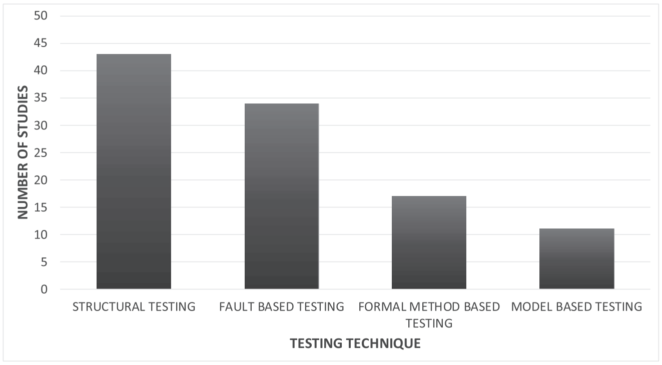

Fig. 4. Distribution of empirical studies per type of testing technique.

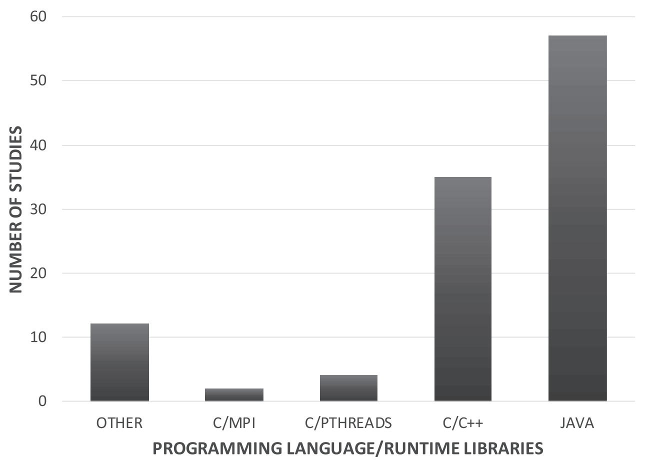

Fig. 5. Distribution of empirical studies per programming language supported.

5. Results and synthesis

We organize this section around the research questions from Table 2 using the evidence from Section 4.6.

5.1. General trends in empirical research on concurrent testing (RQ-1)

Out of 393 papers that propose concurrent software testing techniques, 109 (28%) include an empirical evaluation. This percentage is higher than the percentage for all of software engineering, where only 17%–20% of papers contain empirical validation [92,93]. Fig. 2 shows the number of empirical studies published each year. The increasing trend beginning in 2006 can be attributed to the importance of concurrent testing after the availability of dual core and hyper-threaded processors on personal computers, which started in 2006 [94]. Because our search completed in March 2016, we exclude 2016 from the chart.

5.2. Empirical methods used in concurrent software testing (RQ-2)

RQ-2 identified frequency of empirical evaluation approaches. We classified each empirical study as either a case study, an experiment, or a survey, according to the definitions given in Section 2.3. The first author performed the initial classification. Then the second and third authors reviewed a random sample of approximately 20% of the papers to validate the classification. We performed this two-step validation due to the lack of consistent terminology.

Fig. 3 shows case studies are the most common empirical method researchers use to evaluate or demonstrate the functionality of concurrent software testing techniques. Although case studies provide insights into potential cause-effect relations, controlled experiments can help the establishment of causal relationships that support or refute hypotheses.

5.3. Testing techniques and approaches empirically evaluated (RQ-3)

Fig. 4 shows the distribution of empirical studies for each concurrent software testing technique. *Structural testing* techniques have the largest number of empirical studies. This outcome could be the result of wide use of structural testing in sequential programming along with the significant results showing usefulness of structural testing in sequential programming.

Next, *fault-based testing* has the second largest number of empirical studies. This group of techniques includes approaches for detecting concurrency errors (e.g. data race, deadlock, and atomicity violation). Fault-based testing primarily focuses on finding certain structural interleaving patterns that are common triggers of concurrency problems. Although these techniques can effectively detect many concurrency bugs, they still suffer from the same state explosion problem as other testing techniques. The presence of false negatives and false positives also increases the costs associated with these techniques. Therefore, researchers use case studies most often for empirical evaluation of fault-based techniques.

Researchers have not empirically evaluated the *formal-method based testing* and *model-based testing* approaches as frequently as the others. This lack of evaluation may result from the complexity of the formal methods and concurrent applications, which result in a state explosion in model-based testing because of non-determinism.

5.4. Programming languages supported by empirically evaluated testing techniques (RQ-4)

Fig. 5 shows that Java and C/C+ + are the programming languages most supported by the testing techniques, which have empirical studies.

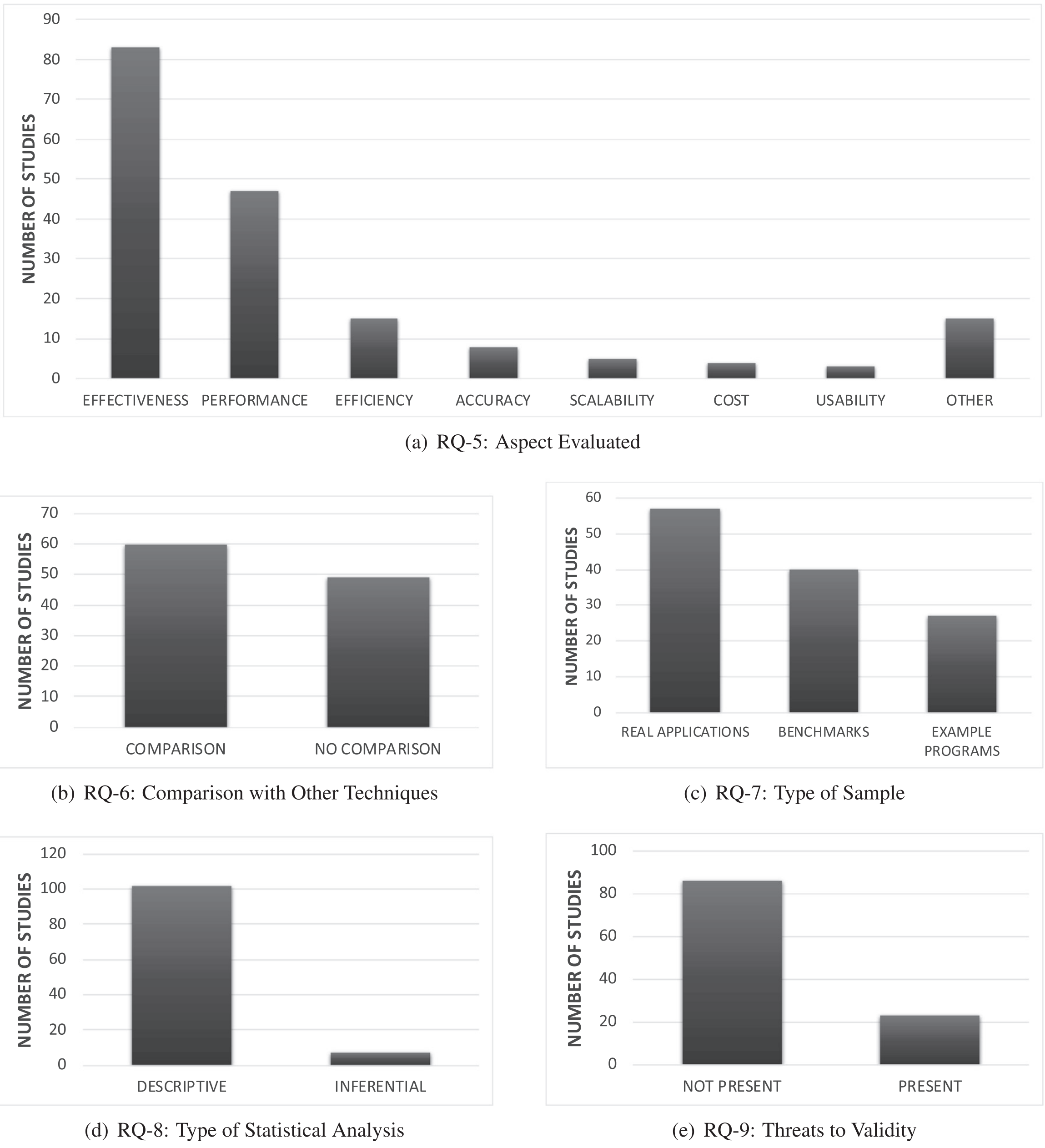

Fig. 6. Characteristics of the selected empirical studies.

5.5. *Characteristics of the empirical studies (**RQ-5–RQ-9**)*

Fig. 6 shows the distribution of studies according to each aspect of their definition:

(a) **Aspect Evaluated**: Fig. 6(a) shows the most evaluated dependent variable is *effectiveness in revealing faults*. It measures whether a technique can identify the most important and critical faults, which, if fixed, result in higher quality software [2].

(b) **Comparison with other techniques**: Fig. 6(b) shows just over half of the empirical studies include comparisons among testing approaches. Although this number is positive, most studies compare only against techniques from the same family. This type of comparison does not provide the information needed to analyze the cost of aggregating results between testing techniques to improve the testing coverage.

(c) **Type of sample**: Fig. 6(c) shows that most authors claim they use either real examples or benchmark programs for analysis. However, a careful analysis of these studies shows that they actually use only examples programs or parts of real applications.

(d) **Type of statistical analysis**: Fig. 6(d) shows most studies use only descriptive analysis. The use of inferential statistics would provide more insight into the results.

(e) **Discussion about threats to validity**: Fig. 6(e) shows only a few discuss the validity of the empirical study. The omission of this information reduces the reader's ability to assess the quality of the results and conclusions. A proper validity evaluation helps the reader understand the results and enables replications with higher confidence.

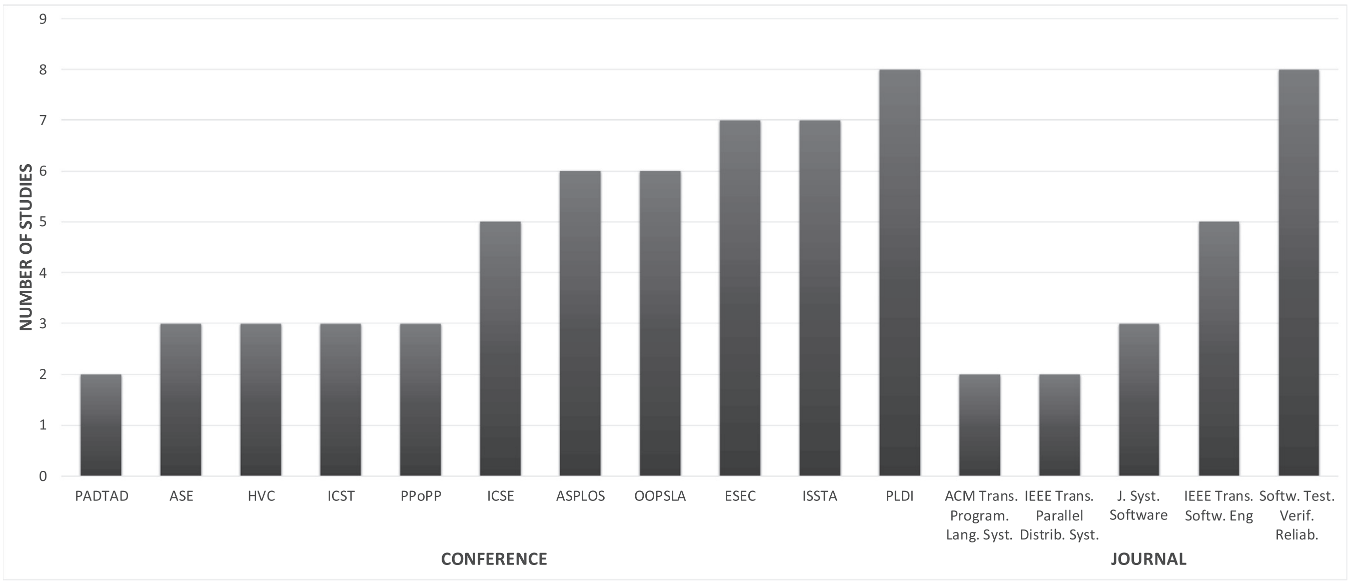

Fig. 7. Main channels with published papers about empirical studies on concurrent software testing.

5.6. *Publication channels (**RQ-10**)*

Fig. 7 shows the publication venues for the empirical studies. The papers appear in a variety of journals and conferences.

The foci of the journals in which the papers appear include: software testing, verification and validation (*Software Testing, Verification and Reliability*), general software engineering (*IEEE Transactions on Software Engineering*), systems and programming (*Journal of Systems and Software* and *ACM Transactions on Programming Languages and Systems*), and parallel programming and distributed systems (*IEEE Transactions on Parallel and Distributed Systems*).

The foci of the conferences in which the papers appear include: programming languages and systems implementation (*PLDI, OOPSLA*, and *ASPLOS*), software testing, verification and validation (*ISSTA, ICST, HVC*), general software engineering (*ESEC, ICSE, ASE*), parallel programming (*PPoPP*), and parallel and distributed system testing and debugging (*PADTAD*).

Overall, most papers appear in conferences and journals focused on general software testing, programming languages, and software engineering. A smaller number of papers appear in venues specifically targeting concurrent software, possibly because of the lack of such venues.

5.7. State-of-the-art *of empirical research on concurrency software testing*

The bubble chart in Fig. 8 shows a summary of the *state-of-the-art* of empirical research on concurrent software testing. The size of the bubble represents the number of studies. According to the figure, *fault-based testing* techniques use more realistic applications, therefore, they involve a large number of case studies, whereas *structural testing* techniques use most case studies based on real applications and example programs.

Regarding gaps, only a few empirical studies focused on *model-based testing* techniques and lack controlled experiments about *formal method-based testing* and *model-based testing* techniques. Moreover, most studies are case studies used as a “proof-of-concept” for a new technique or tool.

6. Designing empirical studies in concurrent software testing

The guidance in this section comes from our literature review and from our own experiences [95]. Throughout this section, we use examples from the literature review to suggest experimental content including study goals, study designs, subject programs, variables, and metrics.

Table 7
GQM template for concurrent software testing.

Goal definition	
Object of study	Concurrent software testing technique (Table 6)
Purpose	Evaluate, analyze, compare, measure, study
Quality focus	Effectiveness, efficiency, applicability, usability, feasibility, reproducibility, correctness, cost, performance (Table 8)
Perspective	Researcher, testing practitioner, students
Context	Subjects, objects, environment

Therefore, this section provides an outline for designing studies about concurrent software testing techniques.

The following subsections describe the steps in the experimental study design process. We believe by providing a template for experiments, researchers can more easily perform replications. These studies will then serve as the content for secondary studies.

6.1. *Goal definition*

Clearly defining the goal(s) is essential for experimental design. The goal focuses the researcher on the appropriate data and corresponding analysis process.

The Goal, Question, Metric (GQM) [96] approach helps researchers define appropriate goals. The GQM goal template consists of five parameters: 1) Object of study, 2) Purpose of the study, 3) Focus of the study (which aspect of the object of study is of interest), 4) Perspective (the viewpoint from which the experiment results are interpreted), and 5) Context (the subjects and software artifacts (objects) used in the experiment) [97].

Table 7 shows an example of a GQM template for research questions on concurrent software testing. Researchers can use the values as a reference to define specific goals. An example goal is:

Analyze *the concurrent software testing technique* (object of study) **for the purpose of** *evaluation* (purpose) **with respect to its** *effectiveness* (quality focus) **from the point of view of the** *researcher* (perspective) **in the context of** *testing specialists finding defects in concurrent software* (context).

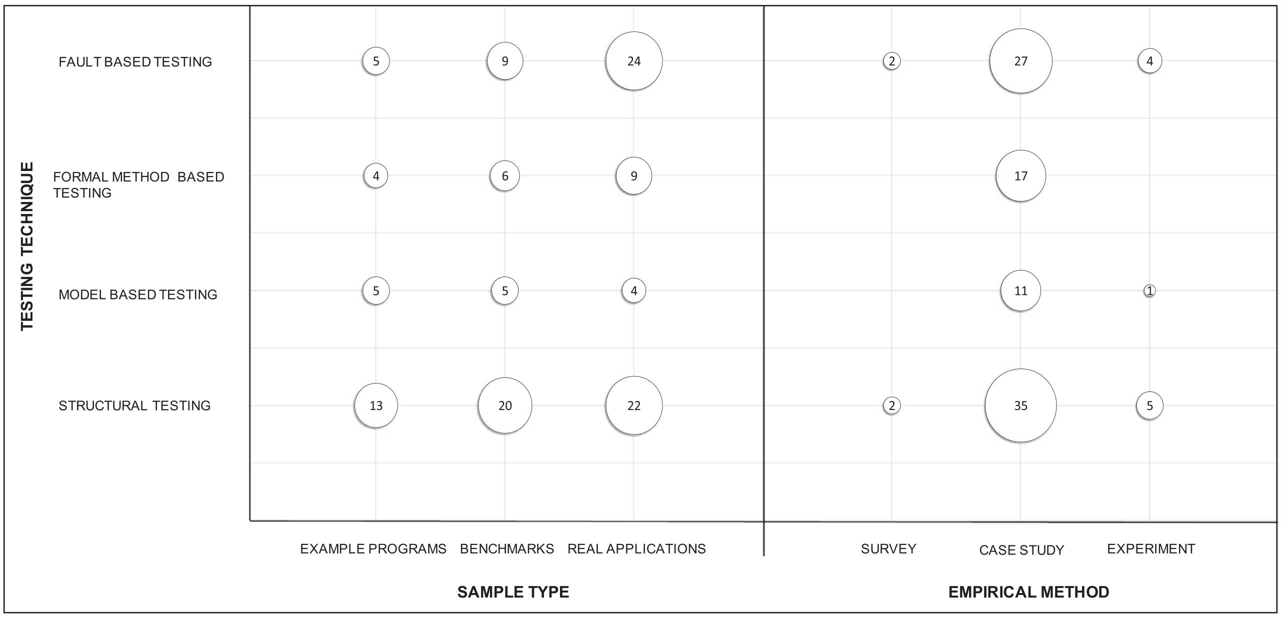

Fig. 8. State-of-the-art of empirical studies on concurrent software testing.

6.2. *Experiment planning*

According to Wohlin et al. [8], the design of an experiment includes seven key steps. The following subsections provide details on those steps to help guide the design of experiments on concurrent software testing.

6.2.1. *Context selection*

The experiment context, i.e. the environment, can be characterized on four dimensions based upon the number of subjects and objects involved: 1) off-line or on-line, 2) students or professionals, 3) academic examples or real problems, and 4) specific or general. Researchers have conducted most concurrent software testing studies off-line using either benchmark programs or real programs in a specific controlled context.

Regarding the use of students vs professionals as experimental subjects, each subject type has strengths and weaknesses. Researchers argue that laboratory studies performed by students have lower quality and relevance than those performed by professionals. Studies with professionals are expensive, difficult to coordinate, and lack strong control. Conversely, studies with students are easier to control, but often lack external validity because of the participants' experience or lack of realism.

6.2.1.1. Subjects. The subjects are the people or software systems that participate in an experiment. People can be students or testing professionals. When researchers employ testing techniques in software systems, the latter, called experimental units, are the subjects. In either case, researchers must describe participant characteristics that may affect the results. For example, for human subjects the characteristics include experience, schooling, and knowledge of the technique. If the subjects are software systems the characteristics include language, domain, system size, and system complexity. Our literature review did not find any studies that used human subjects.

6.2.1.2. Objects. Experimental objects are the units in which an experiment is executed. In the context of software testing, an object can be characterized as the SUT (Software Under Test), which includes didactic examples with less than 1000 LOC (example program), industrial application (real application), or a set of programs from a benchmark. Several benchmarks can be used by researchers as objects for empirical evaluations of concurrent software testing techniques [98–102].

6.2.2. *Formulation of hypotheses*

Hypotheses formalize the researchers expectations about the study results. Typically, researchers pose the following types of hypotheses [8].

6.2.2.1. Null Hypothesis. The null hypothesis states that there is no relationship between the A null hypothesis is a statement with no co-relation among the variables in a study. Typically, the researcher aims to reject the null hypothesis at a chosen significance level (i.e. $\alpha < 0.05$). An example of a null hypothesis for concurrent software testing is testing technique *A* helps developers find the same number of faults as testing technique *B*. Formally, it is described as:

Null hypothesis 1: *Technique A and Technique B yield the same number of faults.*

6.2.2.2. Alternate Hypothesis. An alternate hypothesis is the opposite of the null hypothesis, i.e., a statement with a co-relation among the variables in a study. An example for concurrent software testing is testing technique *A* helps developers find more faults than testing technique *B*. Formally, it is described as:

Alternative hypothesis 1.1: *Developers find more faults using testing technique A than testing technique B.*

6.2.3. *Selection of variables*

Variables define the causes (independent variables) and effects (dependent variables) examined in an experiment.

6.2.3.1. Independent variables. One type of independent variable is the factors used by the researcher to predict the outcomes of a study and their impact on dependent variables. They include treatments, materials, and some context factors. For concurrent testing studies, the treatments can be the testing techniques. The first column in Table 6, which lists the testing techniques, shows some of the possible values this variable can take.

Another type of independent variable is related to the characteristics of the subjects and objects (programs under test). Program characteristics include number of threads/processes, synchronizations and communication primitives, and LOC. The source of faults is another program characteristic. Faults can be naturally-occurring, randomly seeded, or systematically seeded.

Table 8
Dependent variables for concurrent software testing evaluation.

Dependent variable	Description	Metrics (possible values)
Effectiveness	Ability of each technique to detect faults	Defects detected, sequences exercised, number of mutants killed, number of preemptions, number of states visited, number of interleaving tested, number of executions explored, coverage archived, number of program events and data flow explored
Efficiency	Speed at which the technique finds faults	computation time, time and storage overhead, speedup, number of path explored, response time, time required for prediction
Usability	User's satisfaction	quality analysis by questionnaires
Cost	Effort required for the application of a technique	number of test cases for the coverage of a criterion, time for the execution of schedules, number of interleaving explored, number of program executions
Performance	Resources expended in the application of a technique	execution time, running time, memory overhead, disk space overhead, speedup, compilation time, memory usage, overhead time, response time
Accuracy	Number of false positives/false negatives generated	number of false positives, number of false negatives, number of infeasible elements, number of equivalent mutants, number of non executable paths/ interleavings
Scalability	Ability of a testing technique to work properly with increasing demands	relation between number of threads and runtime overhead, number of executions per time, number of threads used

A third set of independent variables relates to the subjects and includes the number of times tests are executed and the time limit for test execution (since concurrent software faces the state-space explosion problem, which prevents exhaustive testing).

6.2.3.2. Dependent variables. Dependent variables are the outcome variables that measure the effects of independent variables. In the above hypotheses, a dependent variable would be number of faults detected. Table 8 summarizes the most common dependent variables in concurrent software testing, based on the studies presented in this review.

6.2.4. Subject selection

Subjects are responsible for the execution of experiments. Ideally, subjects would be a random sample of the population of computer science professionals. However, it is difficult to obtain such a sample. Instead, researchers often use convenience samples of the population, i.e., students enrolled in a particular university or industrial course [103]. Most experiments identified in the literature review used students as subjects.

6.2.5. Choice of design type

The study design depends on factors including the experimental goal, the number of factors and their alternatives, and the number of undesired variations [104]. Subjects apply different treatments (control or experimental) while keeping other variables constant. Researchers measure the effect of the treatments on the dependent variables. In human-subjects experiments, subjects (humans) apply different treatments (techniques) to objects (programs). Conversely, in technology-based experiments, different technical treatments (testing techniques) are applied to different objects (programs) automatically (by the computer) [8].

The experiments identified in the literature review are all technology-based experiments. However, the researchers do not provide details about the study design. Papers only describe the procedure for the testing applications and do not discuss the assignment of the treatments to the objects. The lack of this information hampers replication and can yield undesired sources of variation.

Table 9 summarizes common software engineering experimental designs that can help researchers choose the appropriate design based upon their objectives and variables. Other publications provide more detailed information about experiment design types and definitions [8,104].

6.2.6. Instrumentation

In the instrumentation phase, the researcher prepares the instruments used in the experiment. Instrumentation consists of:

- Experiment object: object to which the experiment is applied (e.g., program code).
- Guidelines: guidance for participants in the experiment (e.g., experiment descriptions, tool user's guide, and checklists).
- Measurements: data collection (via automatic tool's output or manually by the tester).

The overall goal of the instrumentation is to assist in execution and monitoring of the experiment without impacting the results. In case where the instrumentation affects the outcomes, researchers have introduce a threat to validity [8].

6.2.7. Validity evaluation

Validity is a measure of the trustworthiness of results. During the planning phase, researchers must identify the threats to validity and assess their impact. Table 10 summarizes the most common threats to validity found in the literature on concurrent software testing. The following subsections define these threats in more detail.

6.2.7.1. Construct validity. Refers to how closely the operationalization of the measures matches real-world constructs. Construct threats in concurrent software testing include how well a tool implements the testing technique, the types of faults inserted in the software, the specific test cases chosen, and the metrics chosen for evaluation of effectiveness.

6.2.7.2. Internal validity. Refers to the extent to which the independent variables are responsible for the outcomes on the dependent variables. Examples include (1) incorrect tool settings, (2) defects injected into example programs, (3) testing context and instrumentation process, and (4) experimenter bias.

6.2.7.3. External validity. Refers to the degree to which the findings of a study can be generalized for other participant populations or settings. In concurrent software testing, examples of external validity threats include (1) representativeness of the programs used in the experiments (complexity and concurrency), (2) representativeness of the faults (through taxonomies, pre-existent bugs, faults seeding), (3) representativeness of the test suites (size and complexity), (4) size of the test sequences executions (due to non-deterministic behavior), (5) context

Table 9
Standard design types.

Type of design	Conditions of the experiment	Application
One factor and two treatments	Complete randomized design	Compare two treatments with each subject using only one treatment on each object
	Paired comparison	Compares pairs of experiment material and each subject uses both treatments on the same object
One factor and more than two treatments	Randomized complete block design	Subjects form a more homogeneous experiment unit (block). Each subject uses one object for all treatments and the order in which the subjects use the treatments is randomly assigned
	Completely randomized design	Uses one object to all treatments and the subjects are assigned randomly to the treatments
Two factors	2*2 factorial design	Subjects are randomly assigned to each combination of treatments
	Two-stage nested design	Has two factors and, each with two or more treatments. The experiment design and analysis are the same as for the 2*2 factorial design
More than two factors	2^k factorial design	The subjects are randomly assigned to the 2^k combinations of treatments
	2^k fractional factorial design	Runs only a fraction of the complete factorial experiment

Table 10
Threats to validity in concurrent software testing.

Conclusion validity	Internal validity
Method used is not correct	Setting of tools used
Measure used is not correct	Selection of defects
Technique implementation is not correct	Configuration of the context
Process for fault insertion is not correct	Instrumentation process
Test suite size is not representative	
Incorrect statistical analysis	
Construct validity	**External validity**
Researcher's expectancy affects the result	Representativeness of the object programs
Setting of faults affects the results	Representativeness of the faults seeded
Choice of test case design	Representativeness of the test suites used
Random heterogeneity of test cases	Size of test sequences
Incorrect measure metric	Different behaviours
	Randomization process used
	Configuration of the tools used
	Generalization to other types of software

in which the experiment is conducted, and (6) randomization process that can affect the results.

6.2.7.4. Conclusion validity. Refers to the correctness of the conclusions. In concurrent software testing, conclusion validity problems result from the misapplication of statistical tests.

7. Threats to validity

This section describes the threats of our research.

Construct Validity threats result from the specific set of papers included in the mapping study. Our search string, choice of databases, and paper selection process may have inadvertently omitted relevant papers. We mitigated this threat by using a systematic process and periodic checking of results by a second author.

Internal Validity threats relate to the accuracy of conclusions on cause and effect data extracted from each study. We used a predefined data extraction form (Appendix A) to reduce this threat and drew conclusions from the data collected in the extraction form. Despite the process, we could have missed information about a study.

8. Conclusions

This paper describes a systematic mapping that identifies and classifies empirical studies on concurrent software testing techniques. None of the existing secondary studies explicitly focused on the empirical validation of the proposed techniques. Therefore, this paper fills a gap in the literature. Our systematic mapping study includes 109 studies that contain empirical validation of concurrent software testing techniques. Based on those studies, we can draw the following conclusions:

1. **The literature lacks empirical validation of concurrent software testing techniques**. The 109 papers included in this study represent only 28% of the concurrent software testing. The lack of empirical validation in almost 3/4 of the papers results in a large number of techniques for which practitioners have no evidence to judge their viability making them unsafe for use in practice. Therefore, researchers need to perform more empirical studies, reproductions, replications and re-analyses of studies [21,105].
2. **The frequency of empirical validation varies according to the type of concurrent software testing technique**. The largest number of studies focused on *Structural testing* techniques, followed by *fault-based testing* techniques, and *formal method-based testing* techniques. We also identified different kinds of empirical validation in all families of concurrent software testing techniques. This result indicates that researchers are attempting to employ empirical validation methods to such techniques.
3. **The empirical validation of concurrent software testing techniques poses challenges, including:**

- The subject programs used in empirical studies are small and may not be representative of real industrial applications, decreasing the generality of results.
- Case studies are the most commonly used empirical method. Confounding factors hamper researchers' ability to draw cause-effect conclusions from individual case studies.
- Studies primarily compare the proposed technique against *random testing*, rather than against other concurrent software testing techniques, reducing the ability to draw conclusions across techniques or families of techniques.
- Conclusions are based primarily upon descriptive rather than inferential statistics, reducing the trustworthiness of the results.
- Dependent variables and their metrics differ across studies, reducing the ability to compare results across studies.
- Most papers do not describe the study's threats to validity. Therefore, readers cannot assess the trustworthiness of the conclusions or perform more reliable replications.

4. **The literature lacks controlled experiments**: The snapshot of the empirical studies in concurrent software testing indicates the need for more controlled experiments, especially related to *formal method-based testing* techniques and *model-based testing* techniques.

The primary contributions of this research include:

1. **Identification of the main characteristics of empirical studies**. We described the main factors (empirical method, dependent variable, type of sample, type of statistical analysis, and threats to validity) that compose empirical studies for concurrent software testing and can be used as a basis for comparisons and strength evaluation.
2. **Presentation of an empirical design**. We developed an empirical study design for experiments that evaluate concurrent software testing techniques. This template can help reduce the effort required to design an experiment and facilitate the comparison of results. We also provide a guide that assists the development of more controlled and replicated studies in concurrent software testing.
3. **Collection of empirical evidence about concurrent software testing**. The findings of this review can be used as a foundation for constructing a body of evidence about concurrent software testing. We identified and extracted relevant information about empirical evidence from the studies.
4. **Identification of the need for more empirical validation**. Based on our classification of the types of empirical studies about concurrent software testing, there is a need for more controlled experiments to establish stronger cause-effect relationships and a need for more aggregation of case studies, so researchers can draw deeper conclusions. The review also identified gaps in empirical research results, which provides a starting point for future research.

Because there is a need for more empirical studies about concurrent software testing, we propose the following topics for further research:

1. **Development of more real-world case studies**. Although the mapping study found case studies as the most common evaluation method, very few of those studies are real-world case studies, compared with academic case studies and proof-of-concept experiments. There is a need for additional practical research to understand the behaviour of techniques in specific contexts, where testers have different needs and restrictions that would impact on empirical evaluations. More research on evaluation of concurrent software testing techniques in real-life settings is, therefore, required.
2. **Increase in the number of empirical studies related to formal method-based testing techniques and model-based testing techniques**. Despite their relevance, these families of techniques were not well-represented in our systematic mapping. While many types of techniques are used in concurrent testing, only a few use empirical studies to validate their practical applicability. We suggest using the experimental design described in Section 4.6 as a guide for the running of controlled experiments.
3. **Investigation, by means of replication, of the empirical evidence detected in our mapping**. Although we identified some studies on testing techniques, more empirical research, reproduction, replication, and re-analysis of studies is necessary. New studies would complement existing studies and enable researchers to compare data and report differences among them. Researchers can also increase the statistical confidence in the conclusions.

As complementary research, we plan to use the results of this mapping to define criteria, based on empirical evidence, that guide developers in choosing testing techniques. The results will be coded in a tool that automatically recommends the most appropriate techniques according to the characteristics of a given project.

Acknowledgment

The authors acknowledge São Paulo Research Funding, FAPESP - Sao Paulo Research Funding for the financial support under process no. 2015/23653-5 and 2013/05046-9.

Appendix A. Data extraction form

Study description	Study identifier	Unique numeric id for the study.
	Study title	Article title.
	Date of data extraction	Date of the study analysis.
	Bibliography reference	Bibliography reference.
	Type of article	Type of publication (journal article, conference paper, PhD thesis, technical report, among others).
	Publication channel	Name of the conference, journal or institution where the study was published.
	Study aims	General objectives of the paper (purpose of the research).
Empirical evaluation	Objectives	Goals of the empirical evaluation.
	Design of study	Research method adopted by the author to conduct the empirical study.
	Research question	Describe a research question, if any.
	Research hypothesis	Describe the hypothesis of the empirical study, if any.
	Testing technique	Type of testing technique evaluated by the empirical study.
	Comparison	Verify if the empirical study presents a comparison of the proposed testing technique/tool with other similar technique.
	Sample description	Information on the sample used in the study.
	Language	Programming language supported by the testing technique evaluated in the empirical study.
	Control group	If the study presents a control group and the number of groups and sample size.
	Data analysis	If the data analysis is qualitative or quantitative.
	Metrics	Metrics that measured the dependent variable.
	Statistical analysis	If the study presents a statistical analysis.
	Aspects evaluated	Aspect or technique evaluated by the empirical study.
	Data collection	The way data were obtained.
	Response variable	Dependent variable of the empirical study.
Study findings	Findings and conclusions	Main findings and conclusions of the empirical study.
	Validity	If the study discuss threats to validity.
	Relevance	If the empirical study has useful results for research or industry.

Appendix B. Empirical evidence on concurrent testing

Table B1
Empirical evidence on structural testing.

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Chen et al. [58]	Data race detection, detecting data race and atomicity violations, dynamic testing	Case study	Performance overhead and effectiveness	Run time, number of redundant contexts, number of total context checks performed, atomicity violation coverage	9 real world programs Java	Compare SAM with Artemis	Descriptive	N
Deniz et al. [106]	Mutation-based coverage metrics	Case study	Performance	Running time (seconds) and memory usage (megabytes)	5 benchmarks, 6 author examples MCAPI	N	Descriptive	N
Steenbuck and Fraser [107]	GA algorithm, coverage criterion	Case study	Effectiveness	Number of data races found	12 example programs Java	N	Descriptive	Y
Rungta and Mercer [108]	Greedy depth-first search	Case study	Effectiveness	Density of error discovery	3 benchmarks, 3 real world programs Java	Compare meta heuristic to randomized depth-first search	Descriptive	N
Krena et al. [109]	Searchbestie (concurrency search-based testing)	Case study	Capability to find bugs	Percentage of code coverage	4 example, 1 real world Java	Compare a simple hill climbing algorithm with a random search algorithm	Descriptive	N

(continued on next page)

Table B1 (*continued*)

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Krena et al. [110]	Coverage metrics (ConcurPairs, DUPair, Sync)	Case study	Accuracy	Percentage of code coverage	5 real world programs Java	Compare the proposed metrics with previously proposed metrics gathered by several dynamic analyses: Eraser, GoldiLocks, Avio, GoodLock	Descriptive	N
Brito et al. [12]	Data and control flow criteria	Experiment	Cost, efficiency, effectiveness	Number of test cases that cover a criterion, probability to satisfy a testing criterion through a test set adequate to another testing criterion, percentage of defects detected	8 example programs C/MPI	N	Inferential	Y
Joisha et al. [41]	Data flow criteria	Case study	Performance	Compilation time	8 benchmarks C/C++	N	Descriptive	N
Souza et al. [13]	Data, control and communication flow criteria	Experiment	Efficiency, effectiveness	Number of faults found, number of faults injected, number of SYNsequences (or paths) executed for each approach	9 example programs C/MPI	Compare composite approach to both structural testing and reachability testing	Inferential	Y
Ganai et al. [42]	Dynamic taint analysis for control flow	Case study	Relevancy	Ratio of relevant inputs to the total number of inputs	9 benchmarks C/C++	Compare the result of relevancy analysis for different DTAM approaches	Descriptive	N
Sherman et al. [16]	Synchronization, wait-notify, inter-thread and data flow	Case study	Cost and effectiveness	Coverage values	8 example programs Java	N	Descriptive	N
Hong et al. [44]	Blocked, Blocked-Pair, Blocking, Def-Use, and S-FF	Experiment	Effectiveness, coverage achieved, test suite size	Coverage achieved, test suite size, and fault detection	9 example programs Java	N	Inferential	Y
Hong et al. [32]	Blocked, Blocked-Pair, Blocking, Def-Use, and S-FF	Experiment	Effectiveness, coverage achieved, test suite size	Coverage achieved, test suite size, and fault detection	9 example programs Java	Compare Blocked, Blocked-Pair, Blocking, Def-Use, and S-FF coverage metrics	Inferential	Y
Tasiran et al. [43]	Location-Pairs	Case study	Effectiveness, coverage achieved, test suite size	Percentage of iterations for which a metric was successful	9 example programs Java	Compare concurrency bug detection approach (CFP) with the traditional approach (Full)	Descriptive	N

(*continued on next page*)

Table B1 (*continued*)

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Deng et al. [111]	Concurrent Function Pairs CFP	Case study	Effectiveness, performance, false negatives, bug-report duplication rate, and trace size	Number of inputs that report each bug, number of selected inputs and functions, number of races and single-variable atomicity violations overlap across inputs, trace size and trace-analysis time	5 real applications C/C++	Compare concurrency bug detection approach (CFP) with the traditional approach that applies full-blown detection to every input (Full)	Descriptive	N
Tasharofi et al. [112]	Schedule coverage criteria	Case study	Effectiveness	Number of bugs detected, number of schedules generated, number of executions until the detection of the bug, average time for finding a bug	5 real world Scala Actor programs	Compare Bita with random scheduling	Descriptive	N
Koong et al. [113]	Multi-round coverage ATEMES	Survey and Case study	Performance, reliability, usability	line coverage, response time, branch coverage, execution time, and failed test case data	13 example programs C/C++	N	Descriptive	N
Artho et al. [114]	Failure injection	Case study	Usefulness	Exception coverage after Enforcer was used	9 real world programs Java	Compare to manual approaches	Descriptive	N
Yu et al. [115]	Coverage-adequate criteria and test oracle	Experiment	Effectiveness	Number of faults in object programs detected by the different test suites	3 real world programs C	N	Descriptive	Y
Koppol et al. [36]	ALTS-based coverage criteria all-int-T-synchronizations and all-int-Lsynchronizations	Case study	Effectiveness, effort, state space reduction	Number of transitions and states of a particular reduced ALTS, number of states in the reduced ALTS as a percentage of the number of states in the unreduced ALTS, average sizes of the program slices produced by the algorithm, number of int-T-synchronizations and int-L-synchronizations in the reduced models compared with the level of effort required to satisfy the all-T-synchronization and all-L-synchronization coverage criteria, number of test paths	3 example programs CSS	N	Descriptive	N
Yu et al. [17]	Thread interleaving coverage-driven testing	Case study	Efficiency, effectiveness	Average performance overhead, number of successfully exposed iRoots/ number, total predicted iRoots	13 benchmarks C/C++	Compare Maple with: PCT, PCTLarge, RandDelay and CHESS	Descriptive	N
Guo et al. [116]	Assertion guided, symbolic execution	Case study	Performance, effectiveness, scalability	Number of explored executions, run-time	50 benchmark and real world programs C/C++	N	Descriptive	N
Lu et al. [18]	Deterministic lazy release consistency (DLRC)	Case study	Accuracy, execution time	Number of synchronization operations, number of memory operations and memory footprint	16 benchmark C/C++/ Pthreads	Compare RFDet with DThreads	Descriptive	N
Lu et al. [20]	Deterministic multithreading (DMT)	Case study	Performance	Execution time, memory overhead, disk space overhead	5 benchmarks Pthread	Compare DTM with a state-of-the-art DMT system — DThreads	Descriptive	N

(continued on next page)

Table B1 (*continued*)

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Terragni et al. [117]	Recontest, dynamic CIA	Case study	Effectiveness, efficiency, correctness	Number of potential violations detected by the number or violations predicted, computation time, number of violations reported by Recontest	13 real world programs Java	Compare Recontest with CAPP (stress testing)	Descriptive	Y
Tsui et al. [118]	Similarity-based active testing strategy	Case study	Performance	Testing time	3 benchmarks Pthreads	Compare Maple and RegressionMaple	Descriptive	N
Lei et al. [19]	SYN-sequences	Case study	Effectiveness, adequacy	Number of synchronization sequences exercised during reachability testing	5 example programs Java	Compare the performance of RichTest and VeriSoft	Descriptive	N
Lei and Carver [119]	SYN-sequences	Case study	Performance, effectiveness	Number of sequences exercised, number of times the programs are executed and number of transitions explored	4 example programs C++	Compare Richtest and VeriSoft	Descriptive	N
Hwang et al. [120]	SYN-sequences	Case study	Feasibility	Deadlocks detected, time, max memory usage, number of states, number of threads used	7 benchmarks Java	Compare depth-first search and breadth-first search	Descriptive	N
Sen and Agha [121]	Concolic testing	Case study	Effectiveness	Run time in seconds, number of Paths, number of races/deadlocks/infinite loops/exceptions, number of threads, number of functions tested	13 real world benchmarks Java	Compare Jcute and JPF	Descriptive	N
Farzan et al. [122]	Con2colic testing	Case study	Effectiveness, scalability	Diagnosis time, number of iterations required for the completion of the diagnosis, average size of the root cause returned by the method	14 example programs Benchmarks Java	Compared Concrest with Poirot	Descriptive	N
Setiadi et al. [123]	Thread-Pair-Interleaving (TPAIR) criterion	Case study	Effectiveness	Number of test cases, number of interleavings	5 benchmarks Java	Compare the number of test cases against an existing test case reduction method based on the TPAIR criterion	Descriptive	N
Olszewski et al. [124]	Deterministic multithreading	Case study	Performance	Percentage of nondeterministic executions (pthreads), execution time	9 real world benchmarks Pthreads	Compare the performance of the proposed lazy read API to using deterministic locks	Descriptive	N
Luo et al. [125]	Replay-based	Case study	Effectiveness, efficiency, performance overhead, crash reproducibility	Effectiveness: number of test cases generated and reproducible test cases; overhead: execution time, reproducible failure	6 benchmarks Java	N	Descriptive	Y
Nistor et al. [126]	On-the-fly incremental hashing	Case study	Performance, determinism checking	Overhead, detected bug, number of determinism checks	17 real world programs C++	N	Descriptive	N

(continued on next page)

Table B1 (*continued*)

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Dhok et al. [127]	Automatic Barrier Inference	Case study	Effectiveness, performance	Execution time	5 real world programs C/C++	Compare the ability of Pegasus to address performance problems across versions of manually inserted barriers	Descriptive	N
Yuan et al. [128]	Data race detection, synchronization identification	Case study	Effectiveness, efficiency, running time	Identified sync pairs, number of false positives, number of testing threads, testing points, running time of SyncTester, throughput	15 real world benchmarks Pthreads	Compare SyncTesters with two existing dynamic test schemes, ISSTA08 and Helgrind+	Descriptive	N
Xin et al. [129]	Synchronization coverage	Experiment	Lock manifestation, lock pattern and lock usage evolution	Lock manifestation, lock pattern and lock usage evolution	4 real world programs Java	N	Descriptive	N
Rungta and Mercer [130]	Stateless random walk, randomized depth-first search, guided search	Case study	Effectiveness	Hardness, bound, trials	3 benchmarks Java/C#	Compare tools: CalFuzzer, ConTest, CHESS, and Java Pathfinder	Descriptive	N
Nagarakatte et al. [131]	PPCT randomized scheduler	Case study	Effectiveness, performance	Runs with the bug found, execution time overhead	9 real world programs C++	Compare PCCT with PCT	Descriptive	N
Burckhardt et al. [132]	Randomized Scheduler	Case study	Effectiveness	Percentage of bugs detected	8 real world programs C/C++	Compare PCT with stress, synchronization based random sleeps	Descriptive	N
Kim et al. [133]	Synchronization coverage, active scheduling	Case study	Effectiveness	Time spent on the achievement of a full coverage	6 real world Pthread	Compare with basic random testing	Descriptive	N
Park et al. [134]	Active testing	Case study	Data race detection, scalability	Runtime, overhead of the program with race prediction enabled and average runtime and overhead for program re-execution with race confirmation, total number of potential races, speedup	14 benchmarks C/UPC	N	Descriptive	N
Burnim et al. [135]	Active testing	Case study	Feasibility	Average runtime (seconds), cycles predicted, cycles confirmed, number of bugs identified, probability of confirmation of a cycle	7 benchmarks C	N	Descriptive	N
Fiedor et al. [136]	Noise-based testing	Survey	Efficiency in detecting concurrency-related errors	Time, number of test cases	8 Java, 4 C benchmarks	N	Descriptive	N

(*continued on next page*)

Table B1 (*continued*)

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Zhai et al. [137]	Context sensitive	Case study	Effectiveness	Detection rate, time overhead: slowdown of the strategy and space overhead: memory consumption	9 benchmarks Java	N	Descriptive	Y
Wang et al. [138]	Coverage-guided selective search	Case study	Performance, effective-ness: bug detection capability	Number of interleavings tested, run time in seconds, number of bugs detected	6 real-world programs C/C++/ Pthreads	Compare HaPSet, DPOR, and PCB	Descriptive	N
Jha et al. [139]	Value-deterministic search-based replay	Case study	Efficiency	Compilation Time (CT) in seconds. Number of Lines of Code (LOC). Number of Classes (NOC). Number of Reported Data Race (RDR). Number of Actual Data Race (ADR). Type of Data Race (TDR). Output Result (OR). Tool Usage (TU)	8 real world applications Java	N	Descriptive	N
Christakis and Sagonas [61]	Message passing errors in Erlang	Case study	Effectiveness, perfor-mance	Number of message-passing problems identified by the analysis, elapsed wall clock time (in seconds), and memory requirements (in MB)	15 real-world programs Erlang	N	Descriptive	N

Table B2
Empirical evidence on fault based testing.

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Bradbury et al. [66]	ConMan operators	Experiment	Effectiveness	Number of faults detected	4 example programs Java	N	Descriptive	Y
Gligoric et al. [34]	MutMut, selective mutation	Experiment	Performance	Execution time	12 example programs Java	N	Descriptive	Y
Kester et al. [14]	Static analysis	Survey	Effectiveness	Number of bugs detected and percentage of spurious results produced by the tool	12 benchmarks Java	N	Descriptive	Y
Alowibdi and Stenneth [33]	Data race detection	Survey	Correctness, perfor-mances,and effectiveness	Compilation Time (CT) in seconds. Number of Lines of Code (LOC). Number of Classes (NOC). Number of Reported Data Race (RDR). Number of Actual Data Race (ADR). Type of Data Race (TDR). Output Result (OR). Tool Usage (TU)	5 benchmarks Java	Compare tools - active,testing, static analysis, aspect-oriented approach, static and dynamic, program analysis, model checking	Descriptive	N
Wu et al. [140]	Partition-based	Experiment	Cost-effectiveness	Percentage of bugs detected versus percentage of suspicious instances confirmed	10 benchmarks Java	Comparisons among untreated ordering (U), random ordering (R), and our family of techniques (F)	Inferential	Y
Kasikci et al. [48]	Data race detection	Case study	Effectiveness, Accuracy and Precision, Performance (Time to Classify)	Number of bugs found and states covered, classification accuracy, number of executions per time, running time, memory overhead	11 real world programs C/C++	Compare Portend+ to the Record/Replay Analyzer technique	Descriptive	N
Safi et al. [59]	Data race detection, Event-based interaction	Case study	Accuracy and execution time	Execution time, number of false positives, number of false negatives	20 real world programs Java	Compare DEvA with CAFA and Chord.	Descriptive	N

(*continued on next page*)

Table B2 (*continued*)

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Lu et al. [35]	Deadlock detection, transactional memory	Survey	Bug pattern, manifestation, repair strategy	Number of bugs detected	4 real world programs, 105 bugs C/C++	Compare strategy to deadlock detection and transactional memory	Descriptive	Y
Sheng et al. [45]	Dynamic data race detection	Case study	Effectiveness, efficiency	Overhead and false positives	2 real world programs C/C++	N	Descriptive	N
Zhang and Wang [141]	Prevention of low-level bugs	Case study	Effectiveness, performance, scalability	Effectiveness: suppression of type-state violations performance, overhead of runtime failure mitigation scalability: increasing the number threads and runtime overhead	11 real world programs C/C++/ Pthreads	Compare the proposed approach to other methods for runtime prevention of low-level concurrency bugs	Descriptive	N
Gligoric et al. [15]	MutMut, selective mutation	Experiment	Effectiveness and cost	Execution time	14 real world programs Java	Compare concurrent mutation operators and sequential mutation operators	Descriptive	Y
Sen [142]	Randomized dynamic analysis	Case study	Effectiveness	Average runtime, number of potential races detected, number of real races reported, number of exceptions	14 real world programs Java	N	Descriptive	N
Kestor et al. [62]	Data race detection, relaxed transactional data race	Case study	Effectiveness, performance	Number of existing and injected data races detected by tools, ratio of the execution time, run-time overhead	8 real world programs C/C++	Compare TRADE with s-TRADE	Descriptive	N
Wang and Stoller [143]	Atomicity violation detection	Case study	Performance, accuracy	Running time	12 example programs Java	Compare the proposed algorithm for view-atomicity with the off-line reduction-based algorithm and pairwise block-based algorithm	Descriptive	N
Schimmel et al. [144]	Data race detection	Case study	Efficiency	Bugs detected	25 real world bugs Java	Compare four data races detectors, three dynamic: MTRAT, ConTest, Jinx, and one static: Jchord	Descriptive	N
Zhai et al. [64]	Deadlock detection	Case study	Effectiveness	Deadlocks detected	15 real world bugs C/C++/Java	Compare ASN to PCT, MagicScheduler and DeadlockFuzzer	Descriptive	N
Huang et al. [63]	Reproduction of concurrent bugs (CLAP)	Case study	Effectiveness in the reproduction of bugs, performance	Runtime and space overhead, number of schedules, number of bugs reproduced	20 real world programs Java	Compare CLAP and LEAP	Descriptive	N
Zhang et al. [145]	Data race detection, consequence-oriented approach ConSeq	Case study	Effectiveness, accuracy, performance	Number of bugs detected,number of false positives, runtime overhead	7 real world programs C/C++	Compare race detector and atomicity-violation detector	Descriptive	N
Park et al. [52]	Atomicity violation detection	Case study	Effectiveness, efficiency, re-producibility	Time spent on the reproduction of an exposed bug	7 real world programs C/C++	Compare Ctrigger with stress testing	Descriptive	N
Cai and Cao [46]	Data race detection, detection of hidden races	Case study	Effectiveness, performance,	Thrashing Rate Number of races detected, overhead time, Thrashing Rate	1 real world, 7 benchmarks Java	Compare DrFinder, FastTrack, ConTest, and PCT	Descriptive	N

(*continued on next page*)

Table B2 (*continued*)

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Lu et al. [53]	Atomicity violation detection	Case study	Effectiveness, efficiency, reproducibility	Bug-exposing time, CTrigger bug-exposing time breakdown, unserializable interleaving coverage	7 real world benchmarks C/C++	Compare CTrigger with four other bug-exposing mechanisms on the same platform: Stress, Pure-Pin, Race-based	Descriptive	N
Chew and Lie [55]	Atomicity violation detection	Case study	Effectiveness, performance	Number of bugs detected, number of false positives, run-time overhead	5 example programs Java	N	Descriptive	N
Samak and Ramanathan [51]	Deadlock detection	Case study	Effectiveness	Number of deadlocks synthesized, numbers corresponding to synthesized multithreaded tests and deadlocks detected	8 real world classes Java	N	Descriptive	N
Maiya et al. [60]	Data race detection, happens before reasoning	Case study	Performance	Number of data races reported, number of reports generated by DROIDRACER, number of true and false positives	10 real world open-source applications, 5 proprietary applications Java	N	Descriptive	N
Wang and Stoller [146]	Atomicity violation detection	Case study	Usability, accuracy, performance	Atomicity violations reported, number of missed errors for the online reduction algorithm, running time	12 example programs Java	Compare only the three algorithms proposed: online reduction-based, offline reduction based, and block-based	Descriptive	N
Li et al. [147]	Data race detection	Case study	Performance, effectiveness	Runtime, race detected, overhead	6 benchmarks Java	Compare the effectiveness of approaches for detecting races in deployed environments with Pacer (a sampling based race detector based on FastTrack)	Descriptive	N
Lu et al. [54]	Atomicity violation detection	Case study	Effectiveness	Number of bugs detected, number of false positives	3 real world, 5 benchmark programs C/C++	N	Descriptive	N
Park et al. (2011) [134]	Active testing	Case study	Data race detection, scalability	Runtime, overhead of the program with race prediction enabled and average runtime and overhead for program re-execution with race confirmation, total number of potential races, speedup	14 benchmarks C/UPC	N	Descriptive	N
Gligoric et al. [65]	MutMut, selective mutation	Experiment	Performance	Execution time	8 benchmarks Java	N	Descriptive	N
Smaragdakis et al. [148]	Data race detection, causally-precedes (CP)	Case study	Performance, prediction capability	Number of races detected, running time	12 real world programs Java	N	Descriptive	N
Samak et al. [57]	Data race detection	Case study	Effectiveness, efficacy	Races detected, synthesized test count, synthesis time	7 real world Java library	N	Descriptive	N
Samak and Ramanathan [56]	Atomicity violation detection	Case study	Effectiveness	Number of synthesized tests, number of true atomicity violations, time	7 real world java libraries Java	Compare Intruder with ConTeGe, Omen and Narada	Descriptive	N

Table B3
Empirical evidence on formal method-based testing.

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Razavi et al. [68]	AI planning	Case study	Effectiveness, performance, predictive	Number of predicted runs, time required for each prediction, number of bugs found during the tests for each violation type	8 real world programs Java	N	Descriptive	N

(*continued on next page*)

Table B3 (*continued*)

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Thomson et al. [69]	Scheduling bounding	Case study	Effectiveness	Number of schedules, time, bugs found	52 benchmarks C/C++/Java	Compare straightforward depth-first search (DFS), iterative preemption bounding (IPB), iterative delay bounding (IDB) and use of a naive random scheduler (Rand)	Descriptive	Y
Demsky and Lam [70]	Model checking	Case study	Performance	Execution time	5 real world benchmarks C	Compare SAT-Directed Stateless Model Checking performance to the original DPOR stateless model checking algorithm implemented in CDSChecker, source DPOR algorithm implemented in Nidhugg, and CheckFence.	Descriptive	N
Wood et al. [71]	Composable Specifications	Case study	Precision and conciseness of annotations and tool performance	Size of annotation, number of specified or exercised communication, runtime overhead (time and space), number or annotation	3 real world, 8 example programs Java	N	Descriptive	Y
Khoshnood et al. [72]	Logical constraint-based symbolic analysis	Case study	Effectiveness	Reduction ratio, i.e., number of constraints in the root cause divided by the average number of constraints in a bad schedule	34 benchmarks C	N	Descriptive	N
Chaki et al. [73]	Model checking	Case study	Effectiveness	Deadlock: maximum number of states; number of reachable states; number of iterations; time in seconds; memory usage in MB; effectiveness: number of states and transitions in the, number of states in the model; model construction time; model checking time; total verification time	3 real world programs C	N	Descriptive	N
Dolz et al. [47]	Data race detection	Case study	Effectiveness	Number of data races detected	39 real world programs C/C++/Pthreads	N	Descriptive	N
Musuvathi and Qadeer [74]	Model checking, iterative context-bounding	Case study	Efficacy	Number of states visited for the context-bounded and depth-first strategies both with and without fairness.	5 benchmarks C/C++/C#	N	Descriptive	N
Aziz and Shah [149]	GA Algorithms, test data generation	Case study	Performance and scalability	Execution time	1 real world benchmark C/Pthreads	Compare to randomly generated input data	Descriptive	N
Harmanci et al. [75]	Model checking, variable and thread bounding	Case study	Performance	Overhead of the dynamic execution	6 example programs C	Compare hand-written C code and TMUNIT	Descriptive	N
Bindal et al. [76]	Model checking	Case study	Performance, effectiveness	Running time, average number of executions and time required for the capture of a bug	9 benchmarks Java/C#	N	Descriptive	N
Abadi et al. [49]	Data race detection	Case study	Effectiveness	Number of annotations and time required for the annotation of each program, number of race conditions found in each program, time spent by the programmer inserting annotations and the time to run the tool	5 example programs Java	N	Descriptive	N
Tasharofi et al. [77]	Dynamic partial order reduction	Case study	Effectiveness	Number of paths and transitions, exploration time in seconds, and memory usage in MB	8 example programs ActorFoundry	Compare TransDPOR and DPOR	Descriptive	N
Adalid et al. [78]	Model checking	Case study	Efficiency	Hash projection, number or transitions, time, number of executions, number of states	4 real world servers Java applications	Compare TJT and the LTL extension for JPF (Java Path Finder)	Descriptive	N

(*continued on next page*)

Table B3 (*continued*)

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Musuvathi and Qadeer [79]	Fair Stateless Model Checking	Case study	Efficacy, performance, effectiveness	Number of states visited for the context-bounded and depth-first strategies both with and without fairness	7 real world C++	N	Descriptive	N
Ball et al. [80]	Preemption sealing	Case study	Efficacy and efficiency of preemption sealing	Number of executions explored during testing, time, speedup	5 .NET libraries .Net	N	Descriptive	Y
Araujo et al. [150]	Designing contracts	Case study	Effectiveness	Probability of oracles detecting faults, average effort for the diagnosis of faults normalized with the size of an execution thread	1 real world application Java	compare to manual analysis	Inferential	Y

Table B4
Empirical evidence on model based testing.

Reference	Testing approach	Empirical study	Aspect studied	Response variable	Sample description	Comparison	Statistical analysis	Threats to validity
Carver and Lei [81]	LTS MODELS	Case study	Effectiveness	Number of partially-ordered sequences, totally-ordered sequences, modular sequences	3 example programs Java	N	Descriptive	Y
Nistor et al. [82]	Clustering, bug detection	Experiment	Effectiveness	Number of transitions	14 real world bugs Java	N	Descriptive	Y
Cai et al. [50]	Constraint-based	Case study	Effectiveness, performance	Performance: time, and effectiveness on real deadlocks: number of real deadlocks confirmed, effectiveness on false positives: number of false positives reported	3 benchmarks C/C++/Java	Compare PCT, MagicScheduler (MS), DeadlockFuzzer (DF), and ConLock (CL)	Descriptive	Y
Sen [83]	Random partial order sampling & Randomized dynamic analysis	Case study	Effectiveness	Number of executions required to find a defect, number of partial orders sampled	12 example programs Java	Compare simple randomized algorithm and RAPOS (random partial order sampling)	Descriptive	Y
Chen and Macdonald [84]	Value-schedule-based technique	Case study	Functionality, applicability	Number of transitions and intermediate states required to cover all relevant behaviors of the program or discover a bug	9 example programs Java	N	Descriptive	N
Lu et al. [18]	Deterministic lazy release consistency (DLRC)	Case study	Accuracy, execution time	Number of synchronization operations, number of memory operations and memory footprint	16 benchmark C/C++/ Pthreads	Compare RFDet with DThreads	Descriptive	N
Machado et al. [85]	SMT solver, differential schedule projection	Case study	Efficiency, efficacy	Efficiency: execution time, efficacy: number of program events and data-flow edges in the full execution	4 real world, 3 example programs C/C++/Java	N	Descriptive	N
Carver and Lei [86]	Interleaving free concurrency	Case study	Performance	Speedup	4 benchmarks, 1 Lotus specification Java/Lotus	N	Descriptive	N
Zhou et al. [87]	FPDet deterministic runtime	Case study	Determinism, performance, effectiveness of ABA, stability	Time, scalability, redistribution granularity and maximum execution budget	15 real world benchmark Java	Compare FPDet with three different deterministic runtime strategies: (1) Nondet strategy, which does not provide determinism of programs, (2) DMP, which uses serial execution to eliminate data races, and (3) CoreDet	Descriptive	N
Turpie et al. [88]	Multiprocess symbolic execution	Case study	Performance	Line coverage, number of paths explored	2 real world programs C	N	Descriptive	N
Staunton and Clark [89]	Model reuse, estimation distributed algorithm EDA	Case study	Efficiency	Time to find an error	3 example programs LTL	N	Inferential	N
Kähkönen et al. [90]	Dynamic symbolic execution with unfolding	Case study	Effectiveness, performance	Number of tests for a full coverage of the benchmarks and execution time	10 benchmarks Java	Compare unfolding approach to DPOR	Descriptive	N

References

[1] N. Juristo, A.M. Moreno, S. Vegas, Towards building a solid empirical body of knowledge in testing techniques, ACM SIGSOFT Softw. Eng. Notes 29 (5) (2004) 1–4.
[2] N. Juristo, A.M. Moreno, S. Vegas, Reviewing 25 years of testing technique experiments, Empir. Softw. Eng. 9 (1–2) (2004) 7–44.
[3] V. Arora, R. Bhatia, M. Singh, A systematic review of approaches for testing concurrent programs, Concurrency Comput. (2015) 1572–1611.
[4] S.R.S. Souza, M.A.S. Brito, R.A. Silva, P.S.L. Souza, E. Zaluska, Research in concurrent software testing: a systematic review, in: Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, in: PADTAD '11, ACM, New York, NY, USA, 2011, pp. 1–5.
[5] M.A.S. Brito, K. Felizardo, P.S.L. Souza, S.R.S. Souza, Concurrent software testing: a systematic review, in: 22nd IFIP International Conference on Testing Software and Systems, 2010, pp. 79–84. Short paper
[6] A.A. Mamun, A. Khanam, Concurrent software testing: a systematic review and an evaluation of static analysis tool., School of Computing, Blekinge Institute of Technology, Sweden, 2009 Master's Thesis.
[7] D.E. Perry, A.A. Porter, L.G. Votta, Empirical studies of software engineering: a roadmap, in: Proceedings of the Conference on The Future of Software Engineering, in: ICSE '00, ACM, New York, NY, USA, 2000, pp. 345–355.
[8] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.
[9] A. Grama, G. Karypis, V. Kumar, A. Gupta, Introduction to Parallel Computing, Second, Addison Wesley, 2003.
[10] Y. Lei, R.H. Carver, Reachability testing of concurrent programs, IEEE Trans. Softw. Eng. 32 (6) (2006) 382–403.
[11] C.-S.D. Yang, Program-based, Structural Testing of Shared Memory Parallel Programs, University of Delaware, Newark, DE, USA, 1999 Ph.D. Thesis.
[12] M.A.S. Brito, S.R.S. Souza, P.S.L. Souza, An empirical evaluation of the cost and effectiveness of structural testing criteria for concurrent programs., in: International Conference on Computational Science, ICCS, in: Procedia Computer Science, 18, Elsevier, 2013, pp. 250–259.
[13] S.R.S. Souza, P.S.L. Souza, M.A.S. Brito, A.S. Simao, E.J. Zaluska, Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs, Softw. Testing, Verif. Reliab. 25 (3) (2015) 310–332.
[14] D. Kester, M. Mwebesa, J.S. Bradbury, How good is static analysis at finding concurrency bugs? in: SCAM, IEEE Computer Society, 2010, pp. 115–124.
[15] M. Gligoric, L. Zhang, C. Pereira, G. Pokam, Selective mutation testing for concurrent code., in: M. Pezzé, M. Harman (Eds.), International Symposium on Software Testing and Analysis, ISSTA, ACM, 2013, pp. 224–234.
[16] E. Sherman, M.B. Dwyer, S.G. Elbaum, Saturation-based testing of concurrent programs., in: H. van Vliet, V. Issarny (Eds.), Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineerin ESEC/SIGSOFT FSE, ACM, 2009, pp. 53–62.
[17] J. Yu, S. Narayanasamy, C. Pereira, G. Pokam, Maple: a coverage-driven testing tool for multithreaded programs, SIGPLAN Not. 47 (10) (2012) 485–502.
[18] K. Lu, X. Zhou, T. Bergan, X. Wang, Efficient deterministic multithreading without global barriers, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, in: PPoPP '14, ACM, New York, NY, USA, 2014, pp. 287–300.
[19] Y. Lei, R.H. Carver, R. Kacker, D. Kung, A combinatorial testing strategy for concurrent programs, Softw. Testing Verif. Reliab. 17 (4) (2007) 207–225.
[20] K. Lu, X. Zhou, X. Wang, T. Bergan, C. Chen, An efficient and flexible deterministic framework for multithreaded programs, J. Comput. Sci. Technol. 30 (1) (2015) 42–56.
[21] N. Juristo, A.M. Moreno, Basics of Software Engineering Experimentation, 1st, Springer Publishing Company, Incorporated, 2010.
[22] L. Briand, Y. Labiche, Empirical studies of software testing techniques: challenges, practical strategies, and future research, SIGSOFT Softw. Eng. Notes 29 (5) (2004) 1–3.
[23] M. Tyagi, S. Malhotra, A review of empirical evaluation of software testing techniques with subjects, Int. J. Adv. Res. Comput. Eng. Technol. 3 (2014) 519–523.
[24] R.L. Van Horn, Empirical studies of management information systems, ACM Spec. Interest Group Manag. Inf. Syst. SIGMIS 5 (2-3-4) (1973) 172–182.
[25] M.J. Cheon, V. Grover, R. Sabherwal, The evolution of empirical research in IS: a study in IS maturity, Inf. Manag. 24 (3) (1993) 107–119.
[26] F. Shull, J. Singer, D.I. Sjøberg, Guide to Advanced Empirical Software Engineering, Springer-Verlag London, London, UK, 2008.
[27] R.L. Glass, I. Vessey, V. Ramesh, The evolution of empirical research in is: a study in is maturity, Inf. Softw. Technol. 44 (2002) 491–506.
[28] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.
[29] R.K. Yin, Case Study Research: Design and Methods, 3rd, SAGE Publications, 2002.
[30] C. Zannier, G. Melnik, F. Maurer, On the success of empirical studies in the international conference on software engineering, in: Proceedings of the 28th International Conference on Software Engineering, in: ICSE '06, ACM, New York, NY, USA, 2006, pp. 341–350.
[31] K. Petersen, S. Vakkalanka, L. Kuzniarz, Guidelines for conducting systematic mapping studies in software engineering : an update, Inf. Softw. Technol. 64 (2015) 1–18.
[32] S. Hong, M. Staats, J. Ahn, M. Kim, G. Rothermel, Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation, Softw. Testing Verif. Reliab. 25 (4) (2015) 334–370.
[33] J.S. Alowibdi, L. Stenneth, An empirical study of data race detector tools, in: 2013 25th Chinese Control and Decision Conference (CCDC), 2013, pp. 3951–3955.
[34] M. Gligoric, V. Jagannath, Q. Luo, D. Marinov, Efficient mutation testing of multithreaded code, Softw. Testing, Verif. Reliab. 23 (5) (2013) 375–403.
[35] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, in: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS XIII, ACM, New York, NY, USA, 2008, pp. 329–339.
[36] P.V. Koppol, R.H. Carver, K.-C. Tai, Incremental integration testing of concurrent programs, IEEE Trans. Softw. Eng. 28 (6) (2002) 607–623.
[37] B. Kitchenham, S. Charters, Guidelines for Performing Systematic Literature Reviews in Software Engineering, Technical Report, Keele University and Durham University Joint Report, 2007.
[38] J. Cohen, A coefficient of agreement for nominal scales, Educ. Psychol. Meas. 20 (1960) 37–46.
[39] T. Dybå, T. Dingsøyr, Empirical studies of agile software development: a systematic review, Inf. Softw. Technol. 50 (9–10) (2008) 833–859.
[40] I.C. Society, P. Bourque, R.E. Fairley, Guide to the software engineering body of knowledge (SWEBOK(r)): Version 3.0, 3rd, IEEE Computer Society Press, Los Alamitos, CA, USA, 2014.
[41] P.G. Joisha, R.S. Schreiber, P. Banerjee, H.-J. Boehm, D.R. Chakrabarti, On a technique for transparently empowering classical compiler optimizations on multithreaded code, ACM Trans. Program. Lang. Syst. 34 (2) (2012) 9:1–9:42.
[42] M. Ganai, D. Lee, A. Gupta, Dtam: dynamic taint analysis of multi-threaded programs for relevancy, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, in: FSE '12, ACM, New York, NY, USA, 2012, pp. 46:1–46:11.
[43] S. Tasiran, M.E. Keremoglu, K. Muslu, Location pairs: a test coverage metric for shared-memory concurrent programs, Empir. Softw. Eng. 17 (3) (2012) 129–165.
[44] S. Hong, M. Staats, J. Ahn, M. Kim, G. Rothermel, The impact of concurrent coverage metrics on testing effectiveness, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 232–241.
[45] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, Racez: a lightweight and non-invasive race detection tool for production applications, in: International Conference on Software Engineering ICSE, 2011, pp. 401–410.
[46] Y. Cai, L. Cao, Effective and precise dynamic detection of hidden races for java programs, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2015, ACM, NY, USA, 2015, pp. 450–461.
[47] M.F. Dolz, D. del Rio Astorga, J. Fernández, J.D. García, F. García-Carballeira, M. Danelutto, M. Torquati, Embedding semantics of the single-producer/single-consumer lock-free queue into a race detection tool, in: Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, in: PMAM'16, ACM, New York, NY, USA, 2016, pp. 20–29.
[48] B. Kasikci, C. Zamfir, G. Candea, Automated classification of data races under both strong and weak memory models, ACM Trans. Program. Lang. Syst. 37 (3) (2015) 8:1–8:44.
[49] M. Abadi, C. Flanagan, S.N. Freund, Types for safe locking: static race detection for java, ACM Trans. Program. Lang. Syst. 28 (2) (2006) 207–255.
[50] Y. Cai, S. Wu, W.K. Chan, Conlock: a constraint-based approach to dynamic checking on deadlocks in multithreaded programs, in: Proceedings of the 36th International Conference on Software Engineering, in: ICSE 2014, ACM, New York, NY, USA, 2014, pp. 491–502.
[51] M. Samak, M.K. Ramanathan, Multithreaded test synthesis for deadlock detection, in: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, in: OOPSLA '14, ACM, New York, NY, USA, 2014, pp. 473–489.
[52] S. Park, S. Lu, Y. Zhou, Ctrigger: exposing atomicity violation bugs from their hiding places, in: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS XIV, ACM, New York, NY, USA, 2009, pp. 25–36.
[53] S. Lu, S. Park, Y. Zhou, Finding atomicity-violation bugs through unserializable interleaving testing., IEEE Trans. Softw. Eng. 38 (4) (2012) 844–860.
[54] S. Lu, J. Tucek, F. Qin, Y. Zhou, Avio: detecting atomicity violations via access interleaving invariants, ACM Spec. Interest Group Oper. Syst. SIGOPS 40 (5) (2006) 37–48.
[55] L. Chew, D. Lie, Kivati: fast detection and prevention of atomicity violations, in: Proceedings of the 5th European Conference on Computer Systems, in: EuroSys '10, ACM, New York, NY, USA, 2010, pp. 307–320.
[56] M. Samak, M.K. Ramanathan, Synthesizing tests for detecting atomicity violations, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 131–142.
[57] M. Samak, M.K. Ramanathan, S. Jagannathan, Synthesizing racy tests, SIGPLAN Notes 50 (6) (2015) 175–185.
[58] Q. Chen, L. Wang, Z. Yang, Sam: self-adaptive dynamic analysis for multithreaded programs, in: Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing, in: HVC'11, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 115–129.
[59] G. Safi, A. Shahbazian, W.G.J. Halfond, N. Medvidovic, Detecting event anomalies in event-based systems, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 25–37.
[60] P. Maiya, A. Kanade, R. Majumdar, Race detection for android applications, SIGPLAN Notes 49 (6) (2014) 316–325.

[61] M. Christakis, K. Sagonas, Detection of asynchronous message passing errors using static analysis, in: Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages, in: PADL'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 5–18.
[62] G. Kestor, O.S. Unsal, A. Cristal, S. Tasiran, Trade: precise dynamic race detection for scalable transactional memory systems, ACM Trans. Program. Lang. Syst. 2 (2) (2015) 11:1–11:23.
[63] J. Huang, C. Zhang, J. Dolby, Clap: recording local executions to reproduce concurrency failures, SIGPLAN Not. 48 (6) (2013) 141–152.
[64] K. Zhai, Y. Cai, S. Wu, C. Jia, W. Chan, Asn: a dynamic barrier-based approach to confirmation of deadlocks from warnings for large-scale multithreaded programs, IEEE Trans. Parallel Distrib. Syst. 99 (PrePrints) (2014) 1.
[65] M. Gligoric, V. Jagannath, D. Marinov, Mutmut: Efficient exploration for mutation testing of multithreaded code, in: 2010 Third International Conference on Software Testing, Verification and Validation, 2010, pp. 55–64.
[66] J.S. Bradbury, J.R. Cordy, J. Dingel, Comparative assessment of testing and model checking using program mutation, in: Proc. of the 3rd Workshop on Mutation Analysis (Mutation 2007), 2007, pp. 210–219.
[67] G. Tretmans, CONCUR'99 - Concurrency Theory: 10th International Conference, Lecture Notes in Computer Science, Springer Verlag, pp. 46–65.
[68] N. Razavi, A. Farzan, S.A. McIlraith, Generating effective tests for concurrent programs via AI automated planning techniques, Int. J. Softw. Tools Technol. Transfer 16 (1) (2014) 49–65.
[69] P. Thomson, A.F. Donaldson, A. Betts, Concurrency testing using schedule bounding: an empirical study, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, in: PPoPP '14, ACM, New York, NY, USA, 2014, pp. 15–28.
[70] B. Demsky, P. Lam, SATCheck: SAT-directed stateless model checking for SC and TSO, SIGPLAN Notes 50 (10) (2015) 20–36.
[71] B.P. Wood, A. Sampson, L. Ceze, D. Grossman, Composable specifications for structured shared-memory communication, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, in: OOPSLA '10, ACM, New York, NY, USA, 2010, pp. 140–159.
[72] S. Khoshnood, M. Kusano, C. Wang, Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, in: ISSTA 2015, ACM, New York, NY, USA, 2015, pp. 165–176.
[73] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, N. Sinha, Concurrent software verification with states, events, and deadlocks, Formal Aspects Comput. V17 (4) (2005) 461–483.
[74] M. Musuvathi, S. Qadeer, Iterative context bounding for systematic testing of multithreaded programs, SIGPLAN Notes 42 (6) (2007) 446–455.
[75] D. Harmanci, P. Felber, V. Gramoli, M. SuBkraut, C. Fetzer, TMUNIT: A Transactional Memory Unit Testing and Workload Generation Tool, Technical Report, Informatics Institute, University of de Neuchâtel, Switzerland, 2008.
[76] S. Bindal, S. Bansal, A. Lal, Variable and thread bounding for systematic testing of multithreaded programs, in: Proceedings of the 22nd International Symposium on Software Testing and Analysis, ACM, 2013, pp. 145–155.
[77] S. Tasharofi, R.K. Karmani, S. Lauterburg, A. Legay, D. Marinov, G. Agha, Transdpor: a novel dynamic partial-order reduction technique for testing actor programs, in: Proceedings of the International Conference on Formal Techniques for Distributed Systems, in: FMOODS'12/FORTE'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 219–234.
[78] D. Adalid, A. Salmerón, M.D.M. Gallardo, P. Merino, Using spin for automated debugging of infinite executions of java programs, J. Syst. Softw. 90 (2014) 61–75.
[79] M. Musuvathi, S. Qadeer, Fair stateless model checking, SIGPLAN Notes 43 (6) (2008) 362–371.
[80] T. Ball, S. Burckhardt, K.E. Coons, M. Musuvathi, S. Qadeer, Preemption sealing for efficient concurrency testing, in: Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, in: TACAS'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 420–434.
[81] R. Carver, Y. Lei, Multicore software engineering, performance, and tools. MUSEPAT, c11 2013, St. Petersburg, Russia., in: J.M. Lourenço, E. Farchi (Eds.), Multicore Software Engineering, Performance, and Tools, Springer, Berlin, Heidelberg, 2013, pp. 85–96.
[82] A. Nistor, Q. Luo, M. Pradel, T.R. Gross, D. Marinov, Ballerina: automatic generation and clustering of efficient random unit tests for multithreaded code, in: Proceedings of the 34th International Conference on Software Engineering, in: ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 727–737.
[83] K. Sen, Effective random testing of concurrent programs, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, in: ASE '07, ACM, NY, USA, 2007, pp. 323–332.
[84] J. Chen, S. Macdonald, Incremental testing of concurrent programs using value schedules, in: IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 313–322.
[85] N. Machado, B. Lucia, L. Rodrigues, Concurrency debugging with differential schedule projections, SIGPLAN Notes 50 (6) (2015) 586–595.
[86] R.H. Carver, Y. Lei, Distributed reachability testing of concurrent programs., Concurrency Comput. 22 (18) (2010) 2445–2466.
[87] X. Zhou, K. Lu, X. Wang, X. Li, Exploiting parallelism in deterministic shared memory multiprocessing., J. Parallel Distrib. Comput. 72 (5) (2012) 716–727.
[88] J. Turpie, E. Reisner, J.S. Foster, M. Hicks, MultiOtter: Multiprocess Symbolic Execution, Technical Report, Department of Computer Science, University of Maryland, College Park, MD, 2011.
[89] J. Staunton, J.A. Clark, Applications of model reuse when using estimation of distribution algorithms to test concurrent software, in: Proceedings of the Third International Conference on Search Based Software Engineering, in: SSBSE'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 97–111.
[90] K. Kähkönen, O. Saarikivi, K. Heljanko, Unfolding based automated testing of multithreaded programs, Autom. Softw. Eng. 22 (4) (2015) 475–515.
[91] S. Easterbrook, Empirical research methods for software engineering, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, in: ASE '07, ACM, New York, NY, USA, 2007, p. 574.
[92] D.I.K. Sjoberg, T. Dyba, M. Jorgensen, The future of empirical methods in software engineering research, Future Softw. Eng. 0 (2007) 358–378.
[93] W.F. Tichy, P. Lukowicz, L. Prechelt, E.A. Heinz, Experimental evaluation in computer science: a quantitative study, J. Syst. Softw. 28 (1) (1995) 9–18.
[94] Y. Eytani, K. Havelund, S.D. Stoller, S. Ur, Towards a framework and a benchmark for testing tools for multi-threaded programs., Concurrency Comput. 19 (3) (2007) 267–279.
[95] S.M. Melo, P.S.L. Souza, S.R.S. Souza, Towards an empirical study design for concurrent software testing, in: Proceedings of the Fourth International Workshop on Software Engineering for HPC in Computational Science and Engineering, in: SE-HPCCSE '16, IEEE Press, Piscataway, NJ, USA, 2016, p. 1.
[96] V.R. Basili, Software Modeling and Measurement: The Goal/Question/Metric Paradigm, Technical Report, University of Maryland, College Park, MD, USA, 1992.
[97] V.R. Basili, F. Shull, F. Lanubile, Building knowledge through families of experiments, IEEE Trans. Softw. Eng. 25 (4) (1999) 456–473.
[98] G.G.M. Dourado, P.S.L. de Souza, R.R. Prado, R.N. Batista, S.R.S. Souza, J.C. Estrella, S.M. Bruschi, J. Lourenço, A suite of java message-passing benchmarks to support the validation of testing models, criteria and tools, in: International Conference on Computational Science, ICCS, 2016, pp. 2226–2230.
[99] N. Rungta, E.G. Mercer, Clash of the titans: tools and techniques for hunting bugs in concurrent programs, in: 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, 2009, pp. 9:1–9:10.
[100] Y. Eytani, S. Ur, Compiling a benchmark of documented multi-threaded bugs., International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, 2004. 0
[101] Y. Yang, Inspect: a framework for dynamic verification of multithreaded C programs, accessed July URL http://www.cs.utah.edu/~yuyang/inspect/, 2016.
[102] Valgrind-Developers, Valgrind-3.6.1, accessed July, URL http://valgrind.org/, 2016.
[103] C.M. Lott, H.D. Rombach, Repeatable software engineering experiments for comparing defect-detection techniques, Empir. Software Eng. 1 (3) (1996) 241–277.
[104] N. Juristo, A.M. Moreno, Basics of Software Engineering Experimentation, 1st, Springer Publishing Company, Incorporated, 2010.
[105] I. da Costa Araújo, W.O. da Silva, J.B. de Sousa Nunes, F.O. Neto, Arrestt: a framework to create reproducible experiments to evaluate software testing techniques, in: Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing, in: SAST, ACM, New York, NY, USA, 2016, pp. 1:1–1:10.
[106] E. Deniz, A. Sen, J. Holt, Verification and coverage of message passing multicore applications, ACM Trans. Des. Autom. Electron. Syst. 17 (3) (2012) 23:1–23:31.
[107] S. Steenbuck, G. Fraser, Generating unit tests for concurrent classes, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 144–153.
[108] N. Rungta, E.G. Mercer, Hardware and Software: Verification and Testing: 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27–30, 2008. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 23–37.
[109] B. Krena, Z. Letko, T. Vojnar, S. Ur, A platform for search-based testing of concurrent software, in: Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, in: PADTAD '10, ACM, New York, NY, USA, 2010, pp. 48–58.
[110] B. Krena, Z. Letko, T. Vojnar, Coverage metrics for saturation-based and search-based testing of concurrent software, in: Proceedings of the Second International Conference on Runtime Verification, in: RV'11, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 177–192.
[111] D. Deng, W. Zhang, S. Lu, Efficient concurrency-bug detection across inputs, SIGPLAN Notes 48 (10) (2013) 785–802.
[112] S. Tasharofi, M. Pradel, Y. Lin, R.E. Johnson, Bita: coverage-guided, automatic testing of actor programs, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, 2013, pp. 114–124.
[113] C.-S. Koong, C. Shih, P.-A. Hsiung, H.-J. Lai, C.-H. Chang, W.C. Chu, N.-L. Hsueh, C.-T. Yang, Automatic testing environment for multi-core embedded software-atemes, J. Syst. Softw. 85 (1) (2012) 43–60.
[114] C. Artho, A. Biere, S. Honiden, Enforcer - efficient failure injection, in: FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21–27, 2006, Proceedings, 2006, pp. 412–427.
[115] T. Yu, A. Sung, W. Srisa-An, G. Rothermel, An approach to testing commercial embedded systems, J. Syst. Softw. 88 (2014) 207–230.
[116] S. Guo, M. Kusano, C. Wang, Z. Yang, A. Gupta, Assertion guided symbolic execution of multithreaded programs, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 854–865.
[117] V. Terragni, S.-C. Cheung, C. Zhang, Recontest: Effective regression testing of concurrent programs, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, in: ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 246–256.
[118] T. Tsui, S. Wu, W.K. Chan, Toward a methodology to expose partially fixed concurrency bugs in modified multithreaded programs, in: Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices, ACM, NY, USA, 2014, pp. 49–56.

[119] Y. Lei, R.H. Carver, Reachability testing of concurrent programs, IEEE Trans. Softw. Eng. 32 (6) (2006) 382–403.
[120] G.-H. Hwang, C.-S. Lin, T.-S. Lee, C. Wu-Lee, A model-free and state-cover testing scheme for semaphore-based and shared-memory concurrent programs, Softw. Testing, Verif. Reliab. 24 (8) (2014) 706–737.
[121] K. Sen, G. Agha, A race-detection and flipping algorithm for automated testing of multi-threaded programs, in: Proceedings of the 2Nd International Haifa Verification Conference on Hardware and Software, Verification and Testing, in: HVC'06, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 166–182.
[122] A. Farzan, A. Holzer, N. Razavi, H. Veith, Con2colic testing, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 37–47.
[123] T.E. Setiadi, A. Ohsuga, M. Maekawa, Efficient execution path exploration for detecting races in concurrent programs., Int. J. Comput. Sci. 40 (3) (2013) 15–34.
[124] M. Olszewski, J. Ansel, S. Amarasinghe, Kendo: efficient deterministic multithreading in software, SIGPLAN Notes 44 (3) (2009) 97–108.
[125] Q. Luo, S. Zhang, J. Zhao, M. Hu, A lightweight and portable approach to making concurrent failures reproducible, in: Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering, in: FASE'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 323–337.
[126] A. Nistor, D. Marinov, J. Torrellas, Instantcheck: checking the determinism of parallel programs using on-the-fly incremental hashing, in: IEEE/ACM International Symposium on Microarchitecture, 2010, pp. 251–262.
[127] M. Dhok, R. Mudduluru, M.K. Ramanathan, Pegasus: automatic barrier inference for stable multithreaded systems, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, in: ISSTA 2015, ACM, New York, NY, USA, 2015, pp. 153–164.
[128] X. Yuan, Z. Wang, C. Wu, P.-C. Yew, W. Wang, J. Li, D. Xu, Synchronization identification through on-the-fly test, in: Proceedings of the 19th International Conference on Parallel Processing, in: Euro-Par'13, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 4–15.
[129] R. Xin, Z. Qi, S. Huang, C. Xiang, Y. Zheng, Y. Wang, H. Guan, An automation-assisted empirical study on lock usage for concurrent programs, in: IEEE International Conference on Software Maintenance (ICSM), 2013, pp. 100–109.
[130] N. Rungta, E.G. Mercer, Clash of the titans: tools and techniques for hunting bugs in concurrent programs, in: Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, in: PADTAD '09, ACM, New York, NY, USA, 2009, pp. 9:1–9:10.
[131] S. Nagarakatte, S. Burckhardt, M.M. Martin, M. Musuvathi, Multicore acceleration of priority-based schedulers for concurrency bug detection, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI, ACM, NY, USA, 2012, pp. 543–554.
[132] S. Burckhardt, P. Kothari, M. Musuvathi, S. Nagarakatte, A randomized scheduler with probabilistic guarantees of finding bugs, SIGPLAN Notes 45 (3) (2010) 167–178.
[133] H. Kim, D. Cho, S. Moon, Maximizing synchronization coverage via controlling thread schedule, in: 2011 IEEE Consumer Communications and Networking Conference (CCNC), 2011, pp. 1186–1191.
[134] C.-S. Park, K. Sen, P. Hargrove, C. Iancu, Efficient data race detection for distributed memory parallel programs, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, in: SC '11, ACM, New York, NY, USA, 2011, pp. 51:1–51:12.
[135] J. Burnim, K. Sen, C. Stergiou, Testing concurrent programs on relaxed memory models, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, in: ISSTA '11, ACM, New York, NY, USA, 2011, pp. 122–132.
[136] J. Fiedor, V. Hruba, B. Krena, Z. Letko, S. Ur, T. Vojnar, Advances in noise-based testing of concurrent software., Softw. Test., Verif. Reliab. 25 (3) (2015) 272–309.
[137] K. Zhai, B. Xu, W.K. Chan, T.H. Tse, Carisma: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications., in: M.P.E. Heimdahl, Z. Su (Eds.), International Symposium on Software Testing and Analysis, ISSTA, ACM, 2012, pp. 221–231.
[138] C. Wang, M. Said, A. Gupta, Coverage guided systematic concurrency testing, in: Proceedings of the 33rd International Conference on Software Engineering, in: ICSE '11, ACM, New York, NY, USA, 2011, pp. 221–230.
[139] A.K. Jha, S. Jeong, W.J. Lee, Value-deterministic search-based replay for android multithreaded applications, in: Proceedings of the 2013 Research in Adaptive and Convergent Systems, in: RACS '13, ACM, New York, NY, USA, 2013, pp. 381–386.
[140] S. Wu, C. Yang, C. Jia, W.K. Chan, Asp: abstraction subspace partitioning for detection of atomicity violations with an empirical study, IEEE Trans. Parallel Distrib. Syst. 27 (3) (2016) 724–734.
[141] L. Zhang, C. Wang, Runtime prevention of concurrency related type-state violations in multithreaded applications, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, in: ISSTA 2014, ACM, New York, NY, USA, 2014, pp. 1–12.
[142] K. Sen, Race directed random testing of concurrent programs, in: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI '08, ACM, NY, USA, 2008, pp. 11–21.
[143] L. Wang, S.D. Stoller, Accurate and efficient runtime detection of atomicity errors in concurrent programs, in: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, in: PPoPP '06, ACM, New York, NY, USA, 2006, pp. 137–146.
[144] J. Schimmel, K. Molitorisz, W.F. Tichy, An evaluation of data race detectors using bug repositories, in: M. Ganzha, L.A. Maciaszek, M. Paprzycki (Eds.), FedCSIS, 2013, pp. 1349–1352.
[145] W. Zhang, C. Sun, S. Lu, Conmem: detecting severe concurrency bugs through an effect-oriented approach, SIGPLAN Notes 45 (3) (2010) 179–192.
[146] L. Wang, S.D. Stoller, Runtime analysis of atomicity for multithreaded programs, IEEE Trans. Softw. Eng. 32 (2) (2006) 93–110.
[147] D. Li, W. Srisa-an, M.B. Dwyer, Sos: saving time in dynamic race detection with stationary analysis, in: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, in: OOPSLA '11, ACM, NY, USA, 2011, pp. 35–50.
[148] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, C. Flanagan, Sound predictive race detection in polynomial time, SIGPLAN Not. 47 (1) (2012) 387–400.
[149] M.W. Aziz, S.A.B. Shah, Test-data generation for testing parallel real-time systems, in: International Conference Testing Software and Systems, ICTSS 2015, Sharjah and Dubai, United Arab Emirates, 2015, pp. 211–223.
[150] W. Araujo, L.C. Briand, Y. Labiche, On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software, IEEE Trans. Softw. Eng. 40 (10) (2014) 971–992.