Concurrent Software Testing in Practice: A Catalog of Tools

Silvana M. Melo, Simone R. S. Souza, Rodolfo A. Silva, and Paulo S. L. Souza
Institute of Mathematics and Computer Sciences, University of São Paulo
Avenida Trabalhador São-carlense, 400 - 13566-590
São Carlos, São Paulo, Brazil
{morita, srocio, adamshuk, pssouza}@icmc.usp.br

ABSTRACT

The testing of concurrent programs is very complex due to the non-determinism present in those programs. They must be subjected to a systematic testing process that assists in the identification of defects and guarantees quality. Although testing tools have been proposed to support the concurrent program testing, to the best of our knowledge, no study that concentrates all testing tools to be used as a catalog for testers is available in the literature. This paper proposes a new classification for a set of testing tools for concurrent programs, regarding attributes, such as testing technique supported, programming language, and paradigm of development. The purpose is to provide a useful categorization guide that helps testing practitioners and researchers in the selection of testing tools for concurrent programs. A systematic mapping was conducted so that studies on testing tools for concurrent programs could be identified. As a main result, we provide a catalog with 116 testing tools appropriately selected and classified, among which the following techniques were identified: functional testing, structural testing, mutation testing, model based testing, data race and deadlock detection, deterministic testing and symbolic execution. The programming languages with higher support were Java and C/C++. Although a large number of tools have been categorized, most of them are academic and only few are available on a commercial scale. The classification proposed here can contribute to the state-of-the-art of testing tools for concurrent programs and also provides information for the exchange of knowledge between academy and industry.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.1.3 [**Programming Techniques**]: Concurrent Programs

General Terms

Systematic review, Software Testing, Concurrent programs

A-TEST'15, August 30-31, 2015, Bergamo, Italy

http://dx.doi.org/10.1145/2804322.2804328

Keywords

Systematic mapping, Concurrent programs, Testing tools

1. INTRODUCTION

The activities of Verification, Validation, and Testing ensure quality of the software. Software testing is the process of executing a program for finding errors. Mistakes can occur in the software development process, therefore, the testing activity should be conducted throughout the software development cycle. Different testing phases, namely unit testing, integration testing, functional testing, system testing and acceptance testing should be performed. This study focuses on unit testing tools, in which each system module is tested separately so that logical and implementation faults can be found [71].

Testing techniques, such as structural, functional, and fault-based testing proposed to sequential programs have been adapted for use in concurrent programs. Other techniques have been developed specially for concurrent programs and consider features, as non-determinism, synchronization and communication of concurrent/parallel processes. They also look on common mistakes found in the concurrent software, such as race conditions, deadlocks, livelocks, and atomicity violation.

The use of concurrent software has increased, mainly because of the availability of multicore processors and computer clusters. Modern business applications use concurrency to improve the overall system performance, consequently, a variety of testing techniques (and their associated tools) have been proposed to test concurrent programs. However, no classification methodology of testing tools that helps the testing practitioner in the analysis and selection of a tool adequate to their needs has been designed. This paper proposes a new classification for a set of testing tools for concurrent programs regarding attributes, such as testing technique, programming language and paradigm of development. A useful categorization is provided to guide the tester during the selection of testing tools for concurrent programs.

The paper is organized as follows: Section 2 presents the concepts and challenges related to concurrent software testing; Section 3 provides a catalog with 116 testing tools for concurrent programs with some of their descriptions; finally, Section 4 addresses the conclusions and future work.

2. CONCURRENT SOFTWARE TESTING AND CHALLENGES

Concurrent programming enables a smart use of features

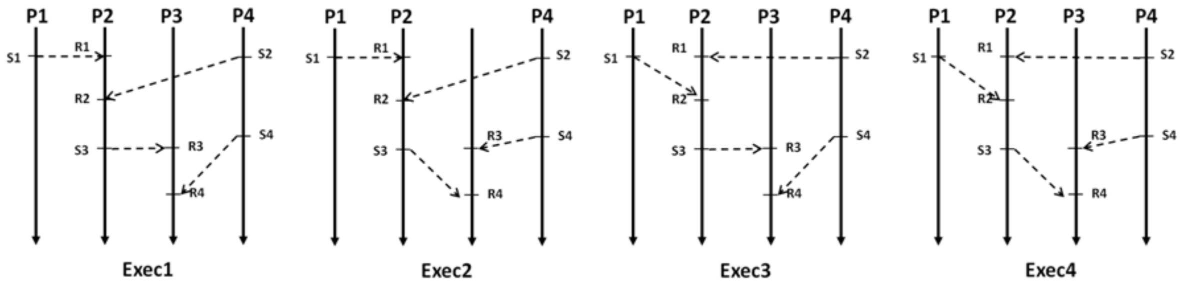

Figure 1: Example of non-determinism in concurrent programs.

for the increase in efficiency (in terms of time of execution), avoiding idleness of resources (as it occurs in the sequential process) and lowering computational costs [32]. However, some challenges may raise in the testing of such programs. The non-determinism enables different executions of a program with a single input and production of different and correct outputs. This non-deterministic behavior is due to communication and synchronization of concurrent (or parallel) processes (or threads). Figure 1 shows an example of non-determinism, in a program composed of four parallel processes. In *Exec1* a race condition occurs between *s1* and *s2*, related to *r1* and *r2*, and *s3* and *s4* related to *r3* and *r4*. Each execution represents a likely synchronization sequence in the concurrent program. The testing activity identifies all possible synchronization sequences and analyzes the outputs. The deterministic execution technique can be used to force the execution of a sequence for a given input in the presence of non-determinism [57].

Other features related to communication and synchronization between processes (or threads) impose challenges on concurrent program testing, such as development of techniques for static analysis, detection of errors related to synchronization, communication, data flow, deadlocks, livelocks, data race, and atomicity violation, adaptation of testing techniques for sequential programming to concurrent programs, definition of a data flow criterion that considers message passing and shared variables, automatic test data generation, efficient exploration of *interleaving* events, reduction of costs in testing activities, deterministic reproduction for a given synchronization sequence, and representation of a concurrent program that captures relevant information to the test.

Studies in the domain of software testing for concurrent programs have proposed solutions for such problems and some testing tools have been developed to support the utilization of the techniques. The need for the execution and testing of different synchronization sequences and the deterministic execution of the program are solutions to this issue. However, they impose high costs on the testing activity. Regarding of this, we consider the building of tools to automatize this activity very promising.

Li et al. [41] propose a taxonomical overview of software testing tools for both sequential and concurrent programs. The classification is based on testing activities and testing stages. The considered activities were test planning/designing, test generation, test execution, test adequacy, test feedback/fault localization, assess readiness and test process management. In relation to testing stages, the following stages are covered: static checking, unit testing, integration testing, system testing/ maintenance testing. In relation to concurrent testing, the authors cite just one model checking tool. Differently, in this paper we present several testing tools for concurrent programs, mainly for the unit testing stage.

Muhammad and Labiche [97] conducted and described a systematic review on state-based testing tools. They proposed a classification of the tools found. The authors highlight that just a few commercial tools were found in the review. The authors argue that this happened due the use of only academic databases for selection of studies. In our study we face with the same problem, but nevertheless, we believe that the academic databases are the most reliable bases for systematic mapping.

3. A CATALOG OF TESTING TOOLS FOR CONCURRENT PROGRAMS

We conducted a systematic mapping (following the process defined by Petersen et al. [80]) to identify tools proposed for testing concurrent programs. The focus of this paper is not the systematic mapping and, therefore, details about the mapping are not shown due to space restrictions The conducted mapping was more extensive, including other research questions (out of scope of this paper). Thus, only the necessary information to understand how the catalog was generated is shown here. A search string was defined with the words “testing”, “concurrent software” and their synonyms. The search was performed in 5 research databases and 6316 papers were returned, of which 334 were selected. We identified 116 different testing tools for concurrent programs. Figure 2 shows the number of testing tools developed from 1992 to 2014.

We can observe a continuous increase in the number of papers in this research area. The bubble chart in Figure 3 illustrates the current state-of-the-art of the concurrent software testing domain in relation to the total number of tools available for each testing technique proposed and programming language supported.

Although a large number of supporting tools for concurrent program testing has been proposed, their maturity level should be analyzed. Most tools represent *concepts proof* of academic proposals, which may be a threat to the validity of this study that considered only academic data bases to conduct the search of primary studies. Finding commercial tools is hard because the vendors offer only user’s manuals

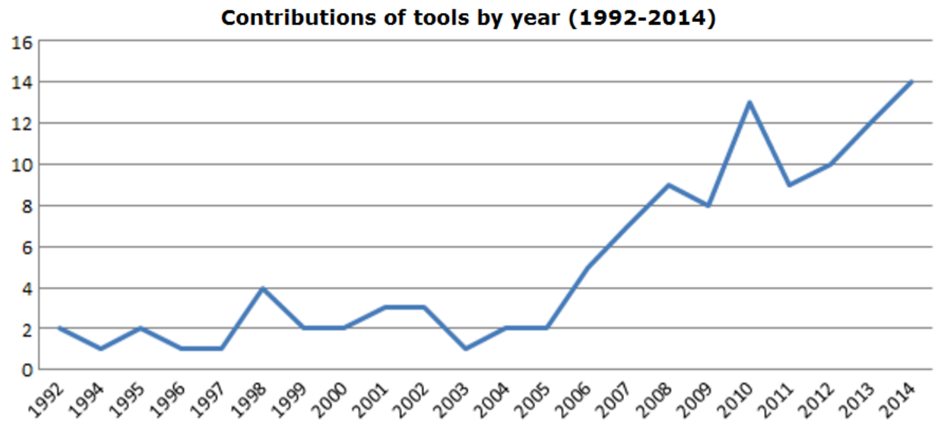

Figure 2: Proposition of concurrent testing tools over the years (1992-2014).

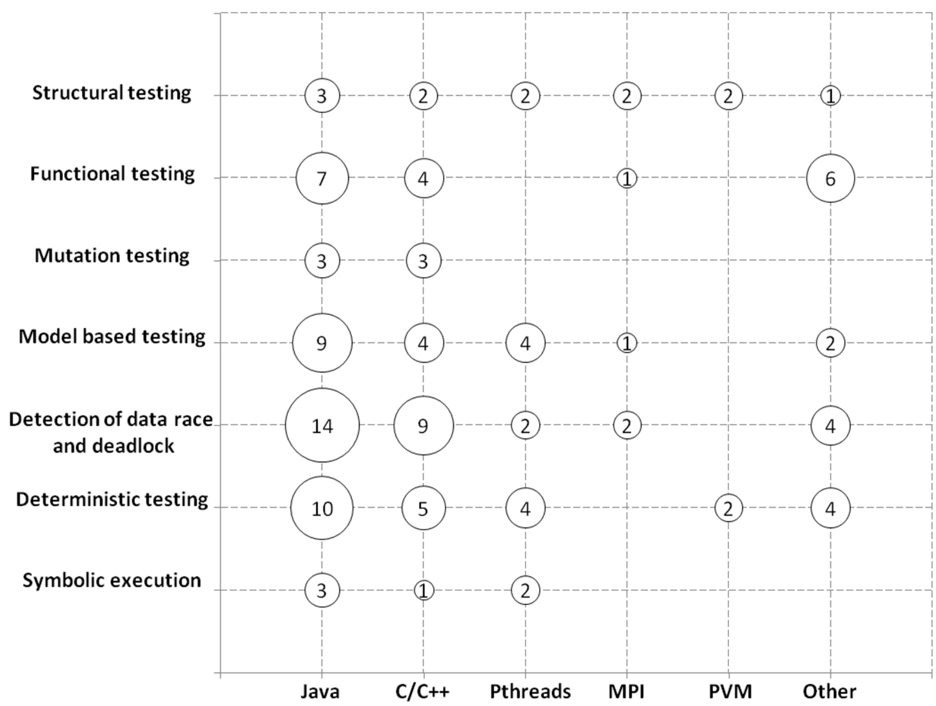

Figure 3: Testing tools by testing approach and implementation language.

and case studies with no technique information in scientific paper for proprietary reasons. The transference of technology from the academy to the industry still remains a challenge in the concurrent software testing domain. Therefore, a closer interaction between the interests of academy and industry is required so that a feedback loop can be created between them.

We have defined a set of relevant attributes to classify the concurrent testing tool selected from the systematic mapping. The definition was based on features of the concurrent programs and information considered relevant for the tester to select the desired testing tool. The following attributes were defined: testing technique, paradigm programming, and language supported. Based on such attributes, we have developed a catalog of tools for testing concurrent programs, shown in Table 1. Subsections 3.1 and 3.2 address some most important tools divided into two groups: one containing tools that apply testing techniques (functional, structural, and mutation testing) and another with tools that test specific characteristics of concurrent programs (model checking, deadlock and data race detection, deterministic testing, and symbolic execution).

3.1 Structural Testing Tools

For the structural testing technique, **ValiPar** [105] supports the application of control flow and data flow criteria for concurrent programs in different programming languages and using different paradigms of development. For programs that use the message-passing paradigm, **ValiPVM** [103] supports the testing of programs in PVM (*Parallel Virtual*

Table 1: A testing tools catalog for concurrent programs

Technique	Paradigm	Language	Tools
Structural testing	Shared memory	Pthread	ValiPthread [88], DellaPasta [118]
	Message passing	MPI	ValiMPI [35]
		C	Monitoring tool [40], Maple [121]
		Pascal	Steps [51], Pet [33]
		PVM	ValiPVM [103]
	Both	Ada	CATS [120]
		Java	ValiJava [104], New JLint [5], JML toolset [4]
		C	Valipar [105]
Functional testing	Shared memory	Java	Oshajava [116], Tiddle [86], Ndetermin [10], Race-Fuzzer [93], Rstest [107]
		C	TMUnit [34], Storm [83], Relaxer[11]
	Message passing	MPI	ISP-GEM [38]
		Ada	TSG [13]
	Both	UML	TCaseUML [2]
		PLINQ	SLUG [108]
		Ada	TCgen [47]
		C/C++	ATEMES [49]
Mutation testing	Shared memory	Java	Javalanche [91], MutMut [30], ConMan [8]
		C, C++	Comutation [31], CCmutator [54]
	Message passing	MPI	ValiMut [100]
Model Based testing	Shared memory	Java	Vyrdmc [25], Cute [94], Fusion [113], Bandera [21], TJT [1], TIE [65], SearchBestie [55]
		C, C#, Java	Chess [70]
		C, C++	CDSchecker [74], Inspect [119]
		C, Pthread	Concurrit [9], C2Petri [48], RegressionMaple [110]
		.Net	Gambit [20]
		C#, Java, D	DemonL [115]
	Message passing	C	Magic [14]
		C, MPI	MPI-SPIN [99]
	Both	C, C++	VIP [23]
		LISP	Spin [36]
		Java, LTL	EDA [106]
Data race and deadlock detection	Shared memory	Java	Droidracer [66], ConEE [76], Carisma [123], Jcute [95], Concrash [64], Contest [53], Epaj/Eprfj [90], Have [17], Javapathfinder [112], Omen [87], Penelope [102], RccJava [27], Enforcer [6], Calfuzzer [92], ConcJunit [85], Kivati [18]
		C, C++	ConMem [3], Ctrigger [79], Light64 [72], Pike [28], SPin [12], Racez [98], MultiRace [81], ThreadSanitizer [96], Gadara [114]
		C, Pthread	MDAT [56]
		.Net	Colfinder [117], AutoRT/CorrRT [43]
		UPC	UPC-Check [22]
		Fortran	Eraser [68]
	Message passing	C, MPI	Marmot [50], MPIRace-Check [78]
Deterministic Testing	Shared memory	C, C++	Dthreads [59], InstantCheck [73], DeSTM [84]
		Pthreads	Kendo [77], FPDet [124], Synctester [122], DetLock [69]
		Java,C, C++	RichTest [58]
		Java	Conan [60], IMunit [42], Dejavu [19], SAM [16], Cooperari [67], Java PathExplorer [37], TransDPOR [109]
		C	Direct [15]
		Titanium	Titanium [46]
		C++, Pthreads	RFDet [62]
		STM,C,C++	DeTrans [101]
		Ruby	DPR/TARDIS [63]
	Message passing	PVM	Viper [75]
		C, PVM	DEIPA [61]
		Ada	SpyLayer [7], AIDA [24]
Symbolic execution	Shared memory	C	Concrest [26]
		Java	SPF [82], Z3 [44], LCT [45]
		C/C++/Java	BEST [29]
		C/Pthread	MultiOtter [111], CDT-Eclipce [39]

Machine) and **ValiMPI** [35] for programs in MPI (*Message Passing Interface*). For programs that use the shared memory paradigm, **ValiPthread** [89] tests programs using Posix standard for *threads* (PThreads) and **ValiJava** [104] supports the testing of Java concurrent programs. Other tools, such as **STEPS** [52] and **Dellapasta** [118] use a graphical representation of the program to derive test cases and apply coverage testing criteria to evaluate the testing activity. **MonitoringTool** [40] the coverage of concurrent programs according to the testing criterion *k-tuples of concurrent commands*, proposed by the same authors. This criterion requires implementation of all sequences of k length concurrent commands. This tool can be applied to concurrent C programs and the coverage analysis is achieved by monitoring of the testing execution. Mechanisms to force the execution of concurrent commands are implemented on tool.

3.2 Functional Testing Tools

For functional testing technique, **OSHAJAVA** [116] uses dynamic analysis to test the specification of concurrent programs written in Java annotations. The instrumentation of the bytecode is used to set each "write" operation with the state of the communication updated and the "read" operation to check if a method violated or not its specification. The semantic formalism is used to indicate when a dynamic operation has violated the specification of an inter-thread communication, so that the safety properties of multithreaded programs can be checked. Other tools, such as **SLUG** [108] and **Ndetermin** [10] also use a program specification to derive test cases and evaluate the testing results.

3.3 Mutation Testing Tools

For mutation testing, **MutMut** [30] proposes an approach for an efficient execution of mutants in multithreaded programs. It uses a technique for the selection of mutants to be executed. When the original program is executed, the technique selects points in the code for mutation considering relevant aspects of the concurrent programs. The approach also enables the tester to select a *thread* to be executed, forcing the mutation introduced to be executed. **ConMan** [8] implements a set of mutation operators for concurrent programs in Java (J2SE 5.0). The mutation operators are classified into operators that modify critical regions, keywords, and calls for concurrent methods and operators that replace concurrent objects. **CCmutator** [54] implements those operators as well as new specific mutation operators for concurrent programs in *PThreads*. It utilizes the High Order Mutation technique, in which two or more mutations are inserted in the program for the creation of strong mutants and improvements in the quality of the testing case set. **Comutation** [31] uses selective mutation based on the mutation operators for concurrent Java programs. Selective mutation selects a subset of mutation operators in which test cases that have a high mutation score for this subset also feature for the other operators. The objective is reduce the mutation testing cost.

3.4 Model Checking Testing Tools

The model checking technique has been widely used in concurrent software testing and enables the analysis of system properties by a formal model. It can also be used to explore the state space of a system. Techniques for state space reduction are used to limit the testing search space. **Inspect** [119] uses model checking for concurrent programs in C language. The exploration of relevant interleavings is facilitated by the use of an executable model of the instrumented version of the program and enables the tool to communicate with the scheduler. **CHESS** [70] implements a model checker to analyze the correctness of concurrent programs in relation to the expected properties (e.g. *interleavings*) derived from a test scenario. Testing scenarios are defined by the tester and explore all possible synchronizations among threads. **Magic** [14] analyzes events and states of the operating system. The temporal logic language LTL (*Linear Temporal Logic*) is used to instantiate finite state machines. Also considering a concurrent system formalized in LTL, it is proposed **SPIN** [36] which implements a model checker to analyze the correctness of concurrent systems in relation to the properties formally defined. This tool is instantiated for the MPI pattern, **MPISpin** [99] and later used as the basis for verification of concurrent code in Java, **Bandera** [21].

3.5 Deadlock and Data Race Detection Tools

Carisma [123] implements a data race detector based on statistic sampling. A program, in a single site of the code, can perform multiple accesses to the memory, therefore, the tool uses an analysis of the trace of execution to estimate and distribute sampling between such locations and collects a fraction of all memory accesses. The information assists the tool in detecting data races. In an attempt to prevent data races, programmers generally write a code that will result in a deadlock when executed with some inputs, due to the misuse of synchronization primitives. Some tools, such as **Gadara** [114], **Marmot** [50], and **UPC-Check** [22] address the problem of deadlock detection. They analyze the code and insert delays into it to force the execution of a given synchronization sequence and then detect the presence of deadlocks, or monitor the execution through a scheduler of processes. **Javapathfinder (JPF)** [112] was developed by NASA Research Center. It uses model checking to detect deadlock and data race in Java programs (bytecode). The user can also define the property classes to be analyzed. JPF monitors the execution, extracts events (synchronization and communication) that occur and analyzes them through an observer process. The observer performs a verification based on the information of the monitoring and information of an analysis of error pattern. JPF is especially useful for the verification of concurrent Java programs due its systematic exploration of scheduling sequences of threads, which is a difficult task in traditional testing tools. **MPIRace Check** [78] performs data race detection for programs in MPI by checking the communication messages between the processes.

3.6 Deterministic Testing Tools

Tools are developed for provide threads control and deterministic execution/re-execution in a non-deterministic environment. They usually store information about a preliminary execution (traces) to enable its re-execution, performing the same synchronization sequence. **Dejavu** [19] records thread schedules and the reproduction of a schedule in a controlled execution. **Dthreads** [59] ensures deterministic execution, even in the presence of data race, forcing the program to produce the same output for each input sequence. **SPY-Layer** [7] records and re-runs concurrent or distributed Java programs, verifying and validating synchronization sequences. The re-execution is used for error detection.

3.7 Symbolic Execution Tools

Symbolic execution is a powerful technique for the exploration of systematic paths of a program with symbolic values as inputs. **MultiOtter** [111] uses a symbolic executor to trace values following the control flow of the program and conceptually changes the execution if it finds a conditional dependence of a symbolic value. **LCT** [45] uses a combination of dynamic and symbolic executions, known as *Concolic testing*, in which the program under testing is executed in a hybrid way with real test data and symbolic values for the exploration of different behaviors of the program.

4. CONCLUSIONS

This paper presents a catalog that has addressed the state-of-the-art of concurrent software testing area. The study covered the period from 1992 to 2014 and 116 testing tools were identified and classified into different testing techniques and programming languages. We strongly believe the catalog of tools and the other results provided in this study will be useful for future research and also to help practitioners of the area in the selection of testing techniques and tools.

The results also show concurrent software testing is still a domain for new studies and a research trend. In recent years, researchers have concentrated their efforts mainly on the C/C ++ and Java languages and on techniques for concurrent context, such as: formal verification techniques, model checking, static and dynamic analysis and deterministic execution. Many tools implement a testing approach that combines different testing techniques for increases in the quality of testing.

In future studies, we aim at the development of an online iterative catalog with information on all tools identified by each technique, paradigm, language and others important attributes. Additional research will be focus on analyses of the benefits of the catalog to different stakeholders (testing practitioners, enterprises and researchers) and how such techniques and tools can be employed to improve higher software quality.

5. ACKNOWLEDGMENT

This research is sponsored by FAPESP under process no. 2013/05046-9 and 2013/01818-7.

6. REFERENCES

[1] D. Adalid, A. Salmerón, M. D. M. Gallardo, and P. Merino. Using spin for automated debugging of infinite executions of java programs. *J. Syst. Softw.*, 90:61–75, Apr. 2014.

[2] C. ai Sun. A transformation-based approach to generating scenario-oriented test cases from uml activity diagrams for concurrent applications. In *COMPSAC*, pages 160–167. IEEE Computer Society, 2008.

[3] B. Aichernig, A. Griesmayer, E. Johnsen, R. Schlatte, and A. Stam. Conformance testing of distributed concurrent systems with executable designs. In F. de Boer, M. Bonsangue, and E. Madelaine, editors, *Formal Methods for Components and Objects*, volume 5751 of *Lecture Notes in Computer Science*, pages 61–81. Springer Berlin Heidelberg, 2009.

[4] W. Araujo, L. Briand, and Y. Labiche. On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software. *Software Engineering, IEEE Transactions on*, 40(10):971–992, Oct 2014.

[5] C. Artho. Finding faults in multi-threaded programs. Master's thesis, Swiss Federal Institute of Technology ETH Zŭrich, Zŭrich, 2001.

[6] C. Artho, A. Biere, and S. Honiden. Enforcer - efficient failure injection. In *Proceedings of the 14th International Conference on Formal Methods*, FM'06, pages 412–427, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] A. Bechini, J. Cutajar, and C. Prete. A tool for testing of parallel and distributed programs in message-passing environments. In *Electrotechnical Conference, 1998. MELECON 98., 9th Mediterranean*, volume 2, pages 1308–1312 vol.2, May 1998.

[8] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation operators for concurrent java (J2SE 5.0). *Workshop on Mutation Analysis*, page 11, 2006.

[9] J. Burnim, T. Elmas, G. Necula, and K. Sen. Concurrit: Testing concurrent programs with programmable state-space exploration. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, pages 16–16, Berkeley, CA, USA, 2012. USENIX Association.

[10] J. Burnim, T. Elmas, G. Necula, and K. Sen. Ndetermin: Inferring nondeterministic sequential specifications for parallelism correctness. *SIGPLAN Not.*, 47(8):329–330, Feb. 2012.

[11] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 122–132, New York, NY, USA, 2011. ACM.

[12] Y. Cai, W. Chan, and Y. Yu. Taming deadlocks in multithreaded programs. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 276–279, July 2013.

[13] R. Carver and R. Durham. Integrating formal methods and testing for concurrent programs. In *Proceedings of the Tenth Annual Conference on Computer Assurance, 1995. COMPASS '95. Systems Integrity, Software Safety and Process Security.*, pages 25–33, Jun 1995.

[14] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, V17(4):461–483, December 2005.

[15] K. Chatterjee, L. de Alfaro, V. Raman, and C. Sánchez. Analyzing the impact of change in multi-threaded programs. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, FASE'10, pages 293–307, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] Q. Chen, L. Wang, and 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*, HVC'11, pages 115–129, Berlin, Heidelberg, 2012. Springer-Verlag.

[17] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. Have: Detecting atomicity violations via integrated dynamic and static analysis. In M. Chechik and M. Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2009.

[18] L. Chew and D. Lie. Kivati: Fast detection and prevention of atomicity violations. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 307–320, New York, NY, USA, 2010. ACM.

[19] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 210–220, New York, NY, USA, 2002. ACM.

[20] K. E. Coons, S. Burckhardt, and M. Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 15–24, New York, NY, USA, 2010. ACM.

[21] J. Corbett, M. Dwyer, and J. Hatcliff. Bandera: a source-level interface for model checking java programs. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 762–765, 2000.

[22] J. Coyle, I. Roy, M. Kraeva, and G. Luecke. Upc-check: a scalable tool for detecting run-time errors in unified parallel c. *Computer Science - Research and Development*, 28(2-3):203–209, 2013.

[23] J. Dingel and H. Liang. Automating comprehensive safety analysis of concurrent programs using verisoft and txl. In *In Proceedings of the International Symposium on Foundations of Software Engineering ACM SIGSOFT 2004/FSE12*, 2004.

[24] F. E. Eassa, L. J. Osterweil, and M. Z. Abdel Mageed. Aida: A dynamic analyzer for ada programs. *J. Syst. Softw.*, 31(3):239–255, Dec. 1995.

[25] T. Elmas and S. Tasiran. Vyrdmc: Driving runtime refinement checking with model checkers. *Electr. Notes Theor. Comput. Sci.*, 144(4):41–56, 2006.

[26] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *ESEC/FSE 2013*, pages 37–47, New York, NY, USA, 2013. ACM.

[27] C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI '00*, pages 219–232, New York, NY, USA, 2000.

[28] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *EuroSys '11*, pages 215–228, New York, NY, USA, 2011.

[29] M. Ganai, N. Arora, C. Wang, A. Gupta, and G. Balakrishnan. Best: A symbolic testing tool for predicting multi-threaded program failures. In *ASE 2011*, pages 596–599, 2011.

[30] M. Gligoric, V. Jagannath, and D. Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *ICST '10*, pages 55–64, Washington, DC, USA, 2010.

[31] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *ISSTA 2013*, pages 224–234, New York, NY, USA, 2013. ACM.

[32] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing.* Addison Wesley, 2^o edition, January 2003.

[33] E. L. Gunter and D. Peled. Path exploration tool. In *TACAS '99*, pages 405–419, London, UK, UK, 1999.

[34] D. Harmanci, P. Felber, V. Gramoli, and C. Fetzer. C.: TMunit: Testing software transactional memories. In *TRANSACT 2009*, 2009.

[35] A. C. Hausen, S. R. Vergilio, S. Souza, P. Souza, and A. Simao. Valimpi: Uma ferramenta para teste de programas paralelos. In *XX SBES*, pages 1–6, Florianopolis, SC, 2006.

[36] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space-craft controller using spin. *IEEE Trans. Softw. Eng.*, 27(8):749–765, Aug. 2001.

[37] K. Havelund and G. Rosu. Monitoring java programs with java pathexplorer. Technical report, 2001.

[38] A. Humphrey, C. Derrick, G. Gopalakrishnan, and B. Tibbitts. Gem: Graphical explorer of mpi programs. In *ICPPW 2010*, pages 161–168, Sept 2010.

[39] A. Ibing. Path-sensitive race detection with partial order reduced symbolic execution. In *Software Engineering and Formal Methods*, volume 8938 of *Lecture Notes in Computer Science*, pages 311–322. 2015.

[40] E. Itoh, Z. Furukawa, and K. Ushijima. A prototype of a concurrent behavior monitoring tool for testing of concurrent programs. In *APSEC 1996*, pages 345–354, Dec 1996.

[41] E. M. J. Jenny Li and D. M. Weiss. *Software Testing: Tools*, pages 1178–1187. In: Encyclopedia of Software Engineering Two-Volume Set (Print). Auerbach Publications, 2010.

[42] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In T. Gyimothy and A. Zeller, editors, *SIGSOFT FSE*, pages 223–233, 2011.

[43] A. Jannesari and F. Wolf. Automatic generation of unit tests for correlated variables in parallel programs. *International Journal of Parallel Programming*, pages 1–19, 2015.

[44] K. Kahkonen and K. Heljanko. Testing multithreaded programs with contextual unfoldings and dynamic symbolic execution. In *ACSD 2014*, pages 142–151, June 2014.

[45] K. Kähkönen, O. Saarikivi, and K. Heljanko. Lct: A parallel distributed testing tool for multithreaded java programs. *Electronic Notes in Theoretical Computer Science*, 296:253 – 259, 2013.

[46] A. Kamil and K. Yelick. Enforcing textual alignment of collectives using dynamic checks. In *LCPC'09*, pages 368–382, Berlin, Heidelberg, 2010.

[47] T. Katayama, Z. Furukawa, and K. Ushijima. Design and implementation of test-case generation for concurrent programs. In *Software Engineering Conference, 1998. Proceedings. 1998 Asia Pacific*, pages 262–269, Dec 1998.

[48] K. M. Kavi, A. Moshtaghi, and D.-J. Chen. Modeling multithreaded applications using petri nets. *Int. J. Parallel Program.*, 30(5):353–371, Oct. 2002.

[49] C.-S. Koong, C. Shih, P.-A. Hsiung, H.-J. Lai, C.-H. Chang, W. C. Chu, N.-L. Hsueh, and C.-T. Yang. Automatic testing environment for multi-core embedded software—atemes. *Journal of Systems and Software*, 85(1):43 – 60, 2012.

[50] B. Krammer, M. S. Müller, and M. M. Resch. Mpi application development using the analysis tool marmot. In *ICCS 2004*, volume 3038 of *Lecture Notes in Computer Science*, pages 464–471, 2004.

[51] H. Krawczyk and B. Wiszniewski. Systematic testing of parallel programs. Technical report, Massachusetts Institute of Technology, 1999.

[52] H. Krawczyk, B. Wiszniewski, P. Kuzora, M. Neyman, and J. Proficz. Integrated static and dynamic analysis of pvm programs with steps. *Computers and Artificial Intelligence*, 17(5), 1998.

[53] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *PADTAD '07*, pages 54–64, New York, NY, USA, 2007.

[54] M. Kusano and C. Wang. Ccmutator: A mutation generator for concurrency constructs in multithreaded c/c++ applications. In *ASE*, pages 722–725, 2013.

[55] B. Křena, Z. Letko, T. Vojnar, and S. Ur. A platform for search-based testing of concurrent software. In *8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 48–58, 2010.

[56] E. Larson and R. Palting. Mdat: A multithreading debugging and testing tool. In *SIGCSE '13*, pages 403–408, New York, NY, USA, 2013.

[57] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Software Eng.*, 32(6):382–403, 2006.

[58] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 32(6):382–403, June 2006.

[59] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *SOSP '11*, pages 327–336, New York, NY, USA, 2011.

[60] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent java components. *Software Engineering, IEEE Transactions on*, 29(6):555–566, June 2003.

[61] J. Lourenço, J. C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewski. An integrated testing and debugging environment for parallel and distributed programs. In *EUROMICRO 97*, page 291, Budapest, Hungary, 1997.

[62] K. Lu, X. Zhou, T. Bergan, and X. Wang. Efficient deterministic multithreading without global barriers. In *PPoPP '14*, pages 287–300, New York, NY, USA, 2014.

[63] L. Lu, W. Ji, and M. L. Scott. Dynamic enforcement of determinism in a parallel scripting language. In *PLDI '14*, pages 519–529, New York, NY, USA, 2014.

[64] Q. Luo, S. Zhang, J. Zhao, and M. Hu. A lightweight and portable approach to making concurrent failures reproducible. In *FASE'10*, pages 323–337, Berlin, Heidelberg, 2010.

[65] G. Maheswara, J. S. Bradbury, and C. Collins. Tie: An interactive visualization of thread interleavings. In *SOFTVIS '10*, pages 215–216, New York, NY, USA, 2010.

[66] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. *SIGPLAN Not.*, 49(6):316–325, June 2014.

[67] E. R. B. Marques, F. Martins, and M. Simões. Cooperari: A tool for cooperative testing of multithreaded java programs. In *PPPJ '14*, pages 200–206, New York, NY, USA, 2014. ACM.

[68] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *PADD '93*, pages 129–139, New York, NY, USA, 1993.

[69] H. Mushtaq, Z. Al-Ars, and K. Bertels. Efficent and highly portable deterministic multithreading (detlock). *Computing*, 96(12):1131–1147, 2014.

[70] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI'08*, pages 267–280, Berkeley, CA, USA, 2008.

[71] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, New York, 2 edition, 2004.

[72] A. Nistor, D. Marinov, and J. Torrellas. Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, pages 541–552, 2009.

[73] A. Nistor, D. Marinov, and J. Torrellas. Instantcheck: Checking the determinism of parallel programs using on-the-fly incremental hashing. In *MICRO-43 2010*, pages 251–262, Dec 2010.

[74] B. Norris and B. Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. *SIGPLAN Not.*, 48(10):131–150, Oct. 2013.

[75] M. Oberhuber, S. Rathmayer, and A. Bode. Tuning parallel programs with computational steering and controlled execution. In *HICSS 1998*, volume 7, pages 157–166 vol.7, Jan 1998.

[76] A. Offenwanger and Y. Lucet. Conee: An exhaustive testing tool to support learning concurrent programming synchronization challenges. In *WCCCE '14*, pages 11:1–11:6, New York, NY, USA, 2014.

[77] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108, Mar. 2009.

[78] M.-Y. Park, S. J. Shim, Y.-K. Jun, and H.-R. Park. Mpirace-check: Detection of message races in mpi programs. In *GPC'07*, pages 322–333, 2007.

[79] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS XIV*, pages 25–36, New York, NY, USA, 2009.

[80] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic mapping studies in software engineering. In *EASE'08*, pages 68–77, 2008.

[81] E. Pozniansky and A. Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, Mar. 2007.

[82] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *ASE '10*, pages 179–180, New York, NY, USA, 2010.

[83] Z. Rakamaric. Storm: static unit checking of concurrent programs. In *32nd International Conference on Software Engineering, 2010 ACM/IEEE*, volume 2, pages 519–520, May 2010.

[84] K. Ravichandran, A. Gavrilovska, and S. Pande. Destm: Harnessing determinism in stms for application development. In *PACT '14*, pages 213–224, New York, NY, USA, 2014.

[85] M. Ricken and R. Cartwright. Concjunit: unit testing for concurrent programs. In B. Stephenson and C. W. Probst, editors, *PPPJ*, pages 129–132. ACM, 2009.

[86] C. Sadowski and J. Yi. Tiddle: A trace description language for generating concurrent benchmarks to test dynamic analyses. In *WODA 2009*, 2009.

[87] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. *SIGPLAN Not.*, 49(10):473–489, Oct. 2014.

[88] F. S. Sarmanho. Teste de programas concorrentes com memória compartilhada. Master's thesis, ICMC/USP, São Carlos, SP, 2009.

[89] F. S. Sarmanho, P. S. L. Souza, S. R. S. Souza, and A. S. S. ao. Structural testing for semaphore-based multithread programs. In *ICCS (1)*, pages 337–346, 2008.

[90] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05*, pages 83–94, New York, NY, USA, 2005. ACM.

[91] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In *ESEC/FSE '09*, pages 297–298, New York, NY, USA, 2009. ACM.

[92] K. Sen. Effective random testing of concurrent programs. In *ASE '07*, pages 323–332, New York, NY, USA, 2007.

[93] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 08, pages 11–21, New York, NY, USA, 2008. ACM.

[94] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *In CAV*, pages 419–423. Springer, 2006.

[95] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In E. Bin, A. Ziv, and S. Ur, editors, *Haifa Verification Conference*, pages 166–182, 2006.

[96] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *WBIA '09*, pages 62–71, New York, NY, USA, 2009. ACM.

[97] M. Shafique and Y. Labiche. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, 17(1):59–76, 2015.

[98] T. Sheng, N. Vachharajani, S. Eranian, and R. Hundt. Racez: A lightweight and non-invasive race detection tool for production applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 401–410. ACM, 2011.

[99] S. F. Siegel. Verifying parallel programs with mpi-spin. In F. Cappello, T. Hérault, and J. Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 13–14. Springer, 2007.

[100] R. A. Silva, S. d. R. S. de Souza, and P. S. L. de Souza. Mutation operators for concurrent programs in mpi. In *Test Workshop (LATW), 2012 13th Latin American*, pages 1–6, April 2012.

[101] V. Smiljkovic, S. Stipic, C. Fetzer, O. Unsal, A. Cristal, and M. Valero. Detrans: Deterministic and parallel execution of transactions. In *26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2014*, SBAC-PAD 2014, pages 152–159, Oct 2014.

[102] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 37–46, New York, NY, USA, 2010. ACM.

[103] P. S. L. Souza, E. Sawabe, A. S. Simao, S. R. Vergilio, and S. R. S. Souza. ValiPVM - a graphical tool for structural testing of PVM programs. In A. Lastovetsky, T. Kechadi, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science (LNCS)*, pages 257–264. Springer Berlin, Heidelberg, 2008.

[104] P. S. L. Souza, S. R. S. Souza, M. G. Rocha, R. R. Prado, and R. N. Batista. Data flow testing in concurrent programs with message-passing and shared-memory paradigms. In *ICCS*, pages 149–158, 2013.

[105] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simão, T. B. Goncalves, A. M. Lima, and A. C. Hausen. Valipar: A testing tool for message-passing parallel programs. In *SEKE*, pages 386–391, 2005.

[106] J. Staunton and J. A. Clark. Applications of model reuse when using estimation of distribution algorithms to test concurrent software. In *SSBSE'11*, pages 97–111, Berlin, Heidelberg, 2011.

[107] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Second Workshop on Runtime Verification (RV)*, volume 70(4), July 2002.

[108] R. Tan, P. Nagpal, and S. Miller. Automated black box testing tool for a parallel programming library. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 307–316, April 2009.

[109] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In *FMOODS/FORTE*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234, 2012.

[110] P. Thomson, A. F. Donaldson, and 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*, PPoPP '14, pages 15–28, New York, NY, USA, 2014. ACM.

[111] J. Turpie, E. Reisner, J. S. Foster, and M. Hicks. Multiotter: Multiprocess symbolic execution. Technical report, 2011.

[112] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[113] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 221–230, New York, NY, USA, 2011. ACM.

[114] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 281–294, Berkeley, CA, USA, 2008. USENIX Association.

[115] S. West, S. Nanz, and B. Meyer. Demonic testing of concurrent programs. In T. Aoki and K. T. 0001, editors, *ICFEM*, volume 7635 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2012.

[116] B. P. Wood, A. Sampson, L. Ceze, and D. Grossman. Composable specifications for structured shared-memory communication. In *OOPSLA*, pages 140–159. ACM, 2010.

[117] Z. Wu, K. Lu, X. Wang, and X. Zhou. Collaborative technique for concurrency bug detection. *International Journal of Parallel Programming*, 43(2):260–285, 2015.

[118] C. S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. *ACM SIGSOFT Software Engineering Notes*, 23:153–162, March 1998.

[119] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded c programs. Technical report, 2008.

[120] M. Young, R. N. Taylor, D. L. Levine, K. A. Nies, and D. Brodbeck. A concurrency analysis tool suite for ada programs: Rationale, design, and preliminary experience. *ACM Trans. Softw. Eng. Methodol.*, 4(1):65–106, Jan. 1995.

[121] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. *ACM SIGPLAN Notices*, 47(10):485–502, Oct. 2012.

[122] X. Yuan, Z. Wang, C. Wu, P.-C. Yew, W. Wang, J. Li, and D. Xu. Synchronization identification through on-the-fly test. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 4–15, Berlin, Heidelberg, 2013. Springer-Verlag.

[123] K. Zhai, B. Xu, W. K. Chan, and T. H. Tse. Carisma: A context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In *ISSTA 2012*, pages 221–231, New York, NY, USA, 2012.

[124] X. Zhou, K. Lu, X. Wang, and X. Li. Exploiting parallelism in deterministic shared memory multiprocessing. *Journal of Parallel and Distributed Computing*, 72(5):716–727, May 2012.