Towards Classification of Concurrency Bugs Based on Observable Properties

Sara Abbaspour Asadollah*, Hans Hansson*, Daniel Sundmark*, Sigrid Eldh†
*Mälardalen University, Västerås, Sweden
{sara.abbaspour, hans.hansson, daniel.sundmark}@mdh.se
†Ericsson AB, Kista, Sweden
sigrid.eldh@ericsson.com

***Abstract*—In software engineering, classification is a way to find an organized structure of knowledge about objects. Classification serves to investigate the relationship between the items to be classified, and can be used to identify the current gaps in the field. In many cases users are able to order and relate objects by fitting them in a category. This paper presents initial work on a taxonomy for classification of errors (bugs) related to concurrent execution of application level software threads. By classifying concurrency bugs based on their corresponding observable properties, this research aims to examine and structure the state of the art in this field, as well as to provide practitioner support for testing and debugging of concurrent software. We also show how the proposed classification, and the different classes of bugs, relates to the state of the art in the field by providing a mapping of the classification to a number of recently published papers in the software engineering field.**

I. Introduction

Concurrent programming puts demands on software development and testing. Concurrent software may exhibit problems, like deadlocks and race conditions, that may not occur in sequential software. There are a variety of challenges related to faults and errors in concurrent and multi-threaded application [1], [2], [3]. So far, there has been some research on bugs occurring in concurrent software (see e.g., [1], [2], [3]), but the efforts have been partially scattered, and no common terminology has been established. Since concurrent software bugs are treated separately in different papers, to the best of our knowledge this research is the first effort to provide a bug classification as a basis for extracting and categorizing the current knowledge in concurrent software bugs.

The purpose of the classification is to provide a common terminology for distinguishing between different types and classes of concurrency bugs. It will be useful in future research to use the same name and label for a specific concept, thereby facilitating communication among researchers and practitioners. Moreover, integrating and classifying concurrent software bugs will be helpful in order to find the interaction between separate elements and classes. The similarities and differences become more apparent, allowing for identification of gaps in the state of the art.

Another, more practical motivation for the classification is for the purpose of supporting software practitioners in finding the cause of different types of concurrency problems. Due to repeatability issues caused by the non-determinism inherent in most concurrent software, testing and debugging of such software is challenging. There is no guarantee that a repeated execution with the same input will yield the same behavior over different runs of execution. By providing an understanding the structure and distinguishing features of different concurrency bugs, a classification can provide benefits for testers as well as designers and developers. Moreover, the classification proposed in this paper is based on observable static and dynamic properties of the concurrent software under test. Thus, in case of erroneous program behavior, we seek a way to support the deduction of the cause of this behavior based on these observable properties.

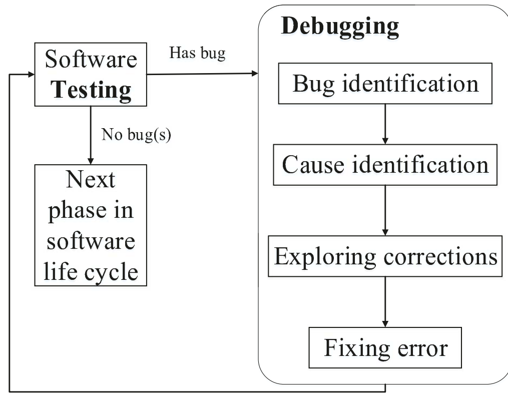
A. Intended Practical Use of the Classification

Software testing plays an important role in the software life cycle in order to produce high quality software with a low maintenance cost. Considering a basic process model in software testing like Fig. 1(a), bugs and defects are identified during testing, and corrected during debugging. In other words, the result of testing process will provide an indication of the answer to the question: ”Do we have bug(s) in the software under test?” In case indications of bugs are found, the common phase after testing phase would be debugging, with the purpose of identifying and removing the causes of the problems (i.e., the bugs).

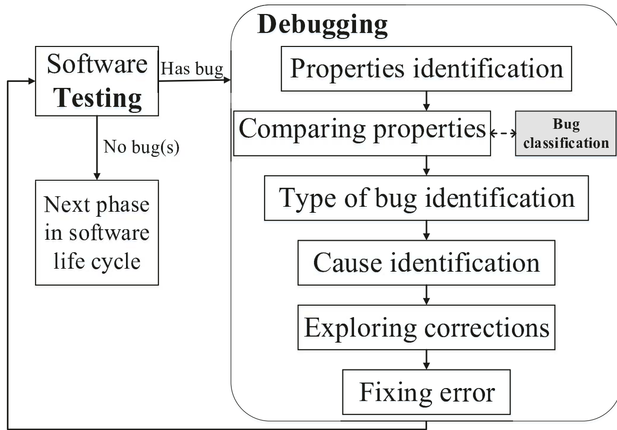
Following debugging, regression testing is commonly done to check if any new errors are introduced during debugging. Fig. 1(a) summarizes this process.

The intention of this research is for it to be used to clarify and simplify the explained process by guiding practitioners in testing and debugging of concurrent software. As shown in Fig. 1(b), a practitioner can check the properties of bug(s) found and compare them with the properties given for each class of the proposed classification. Thereby, he/she can figure out the potential types of concurrency bugs that may have been encountered (or at least reject some classes of bugs that could not possibly have occurred).

Since finding the cause of bugs is essential in exploring corrections and selecting the correct solution for fixing the errors therefor this classification can be useful as a guide to find real cause of errors and it can be helpful in understanding the type of bug by offering more information.



(a) A simple testing and debugging process



(b) A simple testing and debugging process with applying bug classification

Fig. 1: The testing and debugging process.

B. Contributions

In summary, this paper makes the following contributions:

- A summary of concurrency bugs addressed in software testing, analysis and debugging, based on a review of research literature over the last 10 years.
- A classification based on observable properties of these concurrency bugs, intended to support testing and debugging of concurrent software.

C. Paper Outline

This paper proceeds by presenting our research approach in Section II. The assumed system model and terminology is presented in in Section III. Concurrent software bugs are listed in Section IV, and Section V presents the proposed classification of concurrency bugs based on observable properties. In Section VI we map the focus of recently published papers to the proposed classification, and finally Section VII concludes the paper and proposes future work.

II. Research Approach

To propose a classification that fulfills the objective stated in the introduction, the following approach was used:

1) An exhaustive search for relevant articles on testing and debugging the parallel and concurrent applications in single and multicore platforms published in the period of 2005 - 2013 was conducted in the Web of Knowledge, Scopus, and IEEE Xplore databases. After exclusion of non-relevant articles, we ended up with a corpus of 282 relevant articles.
2) Information regarding concurrency bugs was extracted from each article in the above corpus.
3) The concurrency bug terminology was harmonized, and a common general list of concurrency bug types was compiled.
4) For each bug in the above list, observable properties were determined. Based on these observable properties, the bugs were ordered and a bug classification was established.

Regarding limitations in scope, note that we do not consider performance issues, hence this paper does not address bugs related to performance. In addition, we focus on application level bugs, and do not consider bugs related to the operation of the operating system.

III. Preliminaries

The below subsections provide details on the system model and the terminology used in this paper.

A. System Model

We consider concurrent, parallel, and multi-threaded programs running on a single- or multicore platform with:

a) A finite number of non-CPU resources, possibly partitioned into several resource types, such as memory, les, printers, drives, and some other shared resources.
b) A set of threads T $(T_1, T_2, T_3, \ldots, T_n)$; I.e., individual instructions in different threads can execute concurrently on different cores or be interleaved on the same one.
c) Access to resources can be protected by some mechanism, which can enforce atomic access to a shared resource.
d) The state of a thread in the system is defined by the current activity of the thread and each thread may at each point in time be in exactly one of the states shown in Fig. 2.

The different thread states are defined as described below:

- **Ready**: The thread is prepared (ready) to execute when given the opportunity.
- **Executing**: The thread is currently being executed.
- **Blocked/waiting**: The thread is waiting for a particular event (e.g. release of a lock, the passage of time, availability of a contested resource) and cannot execute until this event occurs.
- **Terminated**: The thread has completed its execution.

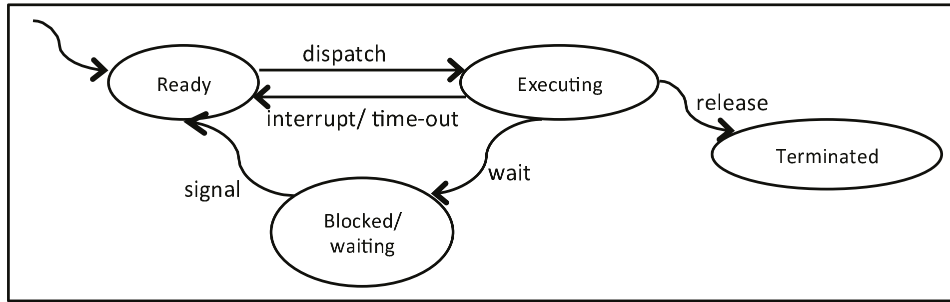


Fig. 2: Thread states

B. Bugs, Faults, Errors, and Failures

It should be noted that the terminology concerning software problems is not entirely consistent. In software testing, debugging and troubleshooting, different terms like fault, error, bug, failure, and defect exist and are sometimes used interchangeably. Fault, error and failure are the most common terms [4]. Swebok [5] describes a fault as the cause of a malfunction, and Eldh et al. [4] define it as the static origin of the problem in the source code.

Error is typically defined as an intermediate infection of the code [4], or as a problem detectable during execution or at run time that causes the program to perform incorrectly [6].

A failure will happen when a system or component cannot perform its required functions as defined by the specified requirements [7]. In other words, if an error propagates into output and becomes visible during execution it has caused a failure [4]. In this research we use the term *bug* to refer to an observable malfunction in the concurrent program under test. While this may not be entirely in line with the above terminology, it is consistent with the terminology used in related work on concurrency bugs.

IV. Concurrent Software Bugs

In order to avoid omission of relevant bugs, we conducted a literature review to identify faults, errors and bugs relevant to parallel and concurrent application testing and debugging, that have been covered in textbooks and in the scientific literature over the last 10 years. The common properties of bugs presented below are primarily extracted from references [1], [3], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18].

The explanation of each concurrent bug with their observable properties are listed as follows:

A **Data race** occurs when at least two threads access the same data and at least one of them write the data [9]. It occurs when concurrent threads perform conflicting accesses by trying to update the same memory location or shared variable [1] [10]. If the memory accesses satisfy the four following conditions, a data race bug has happened: (1) The access were from different threads, (2) At least one of the accesses was a *write* access, (3) The accesses were targeting the same memory location and (4) The accesses were NOT protected by a synchronization mechanism e.g. lock. There are however a few subtle subcategories of data races, as described below:

- **Memory inconsistency** is when different threads have inconsistent views of shared variables [3]. In this case the results of a write operation by one thread are not guaranteed to be visible to a read operation by another thread. If the memory accesses satisfy the five following conditions then a memory inconsistency bug has occurred: (1) The access were from different threads, (2) The accesses were related to the same memory location, (3) The accesses were NOT protected by a synchronization mechanism e.g. lock, (4) There were at least two accesses one of them was write and the other was read and (5) The read access has happened too early.
- **Write-Write race** is a data corruption caused by accessing a shared variable via at least two threads, which one of them overwrites the data before any reads. If the memory accesses satisfy the five following conditions then a write-write race bug has occurred: (1) The access were from different threads, (2) The accesses were related to the same memory location, (3) The accesses were NOT protected by a synchronization mechanism e.g. lock, (4) There were at least two Write accesses and (5) The Write accesses have happened without any Read in-between.

Deadlock is “a condition in a system where a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs” [11]. It occurs when two or more threads attempts to access shared resources held by other threads, and neither is willing to give them up [1] [8]. The common properties for this type of bugs are: (1) None of the threads are able to proceed and progress, (2) All threads involved hold a lock, (3) All threads involved are waiting for a lock held by another involved thread and (4) At least one thread is in waiting for an unacceptably long time.

Livelock is “a situation where a thread is waiting for a resource that will never become available. It is similar to deadlock except that the state of the process involved in the livelock constantly changes with regards to each other, non progressing” [12]. The common properties for this type of bugs are: (1) At least one of the threads is executing on one of the processor cores, (2) None of the threads are able to

proceed and progress and (3) All threads involved have read and written to a spinlock variable (i.e., a shared variable used to enforce synchronization between threads).

Starvation is "a condition in which a process indefinitely delayed because other processes are always given preference" [13]. Starvation typically occurs when high priority threads are monopolising the CPU resources. The common properties for this type of bugs are: (1) At least one of the threads is executing on one of the processor cores, (2) At least one of the threads is in the Ready state, (3) The number of involved threads is larger than the number of free core and (4) At least one thread is in the ready queue for an unacceptably long time.

A **Suspension-based locking or Blocking suspension** occurs when a calling thread waits for an unacceptably long time in a queue to acquire a lock for accessing a shared resource [14]. The common properties for this type of bugs are: (1) At least one of the threads is executing on one of the processor cores, (2) The number of requests to a specific resource is larger than the number of available resources of that type, (3) At least one of the threads has acquired a lock and (4) At least one thread is in waiting for an unacceptably long time.

Order violation is defined as the violation of the desired order between at least two memory accesses [15]. It occurs when the expected order of interleavings does not appear [18]. If a program fails to enforce the programmer's intended order of execution then an order violation bug will happen [16]. If the memory accesses satisfy the four following conditions then an order violation bug has happened. (1) The access were from different threads, (2) At least one of the accesses was Write, (3) The accesses were related to the same memory location and (4) A specific (desired) execution ordering between the access was required.

Atomicity violation refers to the situation when the execution of two code blocks (sequences of statements) in one thread is concurrently overlapping with the execution of one or more code blocks of other threads in such a way that the result is not consistent with any execution where the blocks of the first thread are executed without being overlapping with any other code block. Atomicity violation can be further subcategorized into *single variable atomicity violation* and *multi-variable atomicity violation*, where:

- **Single variable atomicity violation** is when there is a sequence of concurrent memory access to a single variable, yields different result from the state of sequential memory accesses [17]. If the memory accesses are satisfied by the five following conditions then a single variable atomicity violation bug has happened. (1) The access were from different threads, (2) At least one of the accesses was Write, (3) The accesses were related to the same memory location, (4) An atomic execution of statements was required and (5) The accesses targeted only one memory location.
- **Multi-variable atomicity violation** occurs when multiple variables are involved in an unserializable interleaving pattern [17]. If the memory accesses are satisfied by the four following conditions then a multi-variable atomicity violation bug has happened. (1) The access were from different threads, (2) At least one of the accesses was Write, (3) An atomic execution of statements was required and (4) The accesses targeted more than one memory locations.

V. A Classification for Concurrent Software Bugs

In order to propose this classification, we first gathered the common system states and symptoms properties of bugs based on a literature review. In the following lists, when we refer to threads t, we are referring to the threads in the set $T_b \subseteq T$, where T_b is the set of threads directly involved in the bug. Similarly, when we refer to a shared resource r, we are referring to a resource in the set $R_b \subseteq R$, where R_b is the set of resources directly involved in the bug.

Further, our conceptual standing point is that we have identified a concurrency bug and based on observable properties directly related to the bugs we want to uniquely classify it into one the introduced classes.

We divide the observable properties in properties related to the system state, and properties related to the symptoms of the concurrent program under test. All properties used for the classification are listed in the below subsections:

A. System State Properties

The below list collects the properties related to the system state at the time of the bug. We refer to the thread execution states (shown in Fig. 2) in the properties list to present the state of threads when the bugs occur. Most of these properties are related to operations of the operating system and they can be observable by available data structure in operating system kernel such as Thread Control Block (TCB) or using the suitable method(s) in source code to observe these properties during debugging or tracing the software code.

1) At least one thread $t \in T_b$ is in the Waiting state.
2) At least one thread $t \in T_b$ is in the Executing state.
3) At least one thread $t \in T_b$ is in the Ready state.
4) All threads in T_b have read and written to a spinlock variable (spinlock is "mutual execution mechanism in which a process executes in an infinite loop waiting for the value of lock variable to indicate availability" [13]).
5) All threads in T_b are waiting for a lock held by another involved thread.
6) At least one thread $t \in T_b$ is in the ready queue for an unacceptably long time.
7) At least one thread $t \in T_b$ is in Waiting state for an unacceptably long time.
8) All threads in T_b are in Executing state.

B. Symptom Properties

The below list collects the properties related to the observable output at the time of the bug. Based on the bug's symptoms one may recognize the cause of the problem and the nature of the bugs. The following list thus shows some of the typical symptoms that can be used to categorize bugs.

1) No thread $t \in T_b$ is able to proceed and progress.
2) The number of threads in T_b is larger than the number of free processor cores.
3) There are incorrect or unexpected results.
4) The number of requests to a resource r is larger than the number of available resources of that type.
5) All threads in T_b hold a lock.
6) At least one of the threads $t \in T_b$ holds a lock.
7) Accesses to shared memory were made from different threads in T_b.
8) At least one of the accesses to the shared memory was a Write.
9) Accesses to shared memory were targeted the same memory location.
10) Accesses to shared memory were NOT protected by a synchronization mechanism.
11) Accesses to shared memory targeted just one memory location.
12) Accesses to shared memory targeted more than one memory locations.
13) There were at least two accesses to the same shared memory location, a Write and a Read, where the Read has happened too early.
14) There were at least two Write accesses to shared memory, and they occurred without any Read in-between.
15) There was at least one correct execution ordering between the accesses to shared memory which the program failed to enforce.
16) An atomic execution of statements was required.

C. Combination of System State and Symptom Properties

Based on the above lists of observable properties, we have derived a classification of concurrency bugs. The resulting classification is shown in Table I.

As shown in the table, the first column illustrates the observable properties while the first row displays the different types of concurrency bugs. The mapping between bugs and observable properties should be interpreted as $Bug \rightarrow property$. Thus, an "X" in the column of bug B and the row of property p would mean that if you have come across bug B, then you would inevitably be able to observe property p. Note that the opposite implication (i.e., $property \rightarrow Bug$) does not hold.

Different execution scenarios cause different sub-types of bugs. For instance, "Order violation 1" happens if threads execute on multicore platform and there are enough free cores for executing threads on each core, but the order of access to shared memory between the threads is incorrect.

In "Order violation 2" the order of execution is not as same as desired order because of reasons of a lock mechanism. The locking patterns force at least one of the involved threads to stay in Waiting state when the bug has happened.

Moreover, "Order violation 3" happens when the number of involved threads is larger than the number of free cores, and at least one thread was forced to stay in the Ready state.

"Single variable atomicity violation 1" happens when involved threads execute in two separate cores but the blocks intended to be atomic are not adequately protected. In this case the last state of at least one involved thread was Waiting when the bug occurred. "Single variable atomicity violation 2" happens when the number of involved threads is larger than the number of free cores, and at least one thread is forced to change from Waiting to Ready state. The difference between "Multi-variable atomicity violation 1" and "Multi-variable atomicity violation 2" is equivalent to the difference between "Single variable atomicity violation 1" and "Single variable atomicity violation 2".

VI. Mapping the Classification to the State of the Art

In order to evaluate how the proposed classification relates to recent work in the field, we selected fifteen relevant papers from the original literature review corpus of 282 articles (retrieved as explained in Section II). For selection criteria, we focused on papers published between 2005 to 2013 in the *IEEE Transactions on Software Engineering (TSE)* journal or the proceedings of the *International Conference on Software Engineering (ICSE)*. These publication venues were chosen as they are commonly viewed as the premier publication venues in software engineering.

We then mapped each article onto our proposed concurrency bug classification, based on the type of bug(s) focused on. As can be seen from the resulting Table II, most recent work has focused on data races and atomicity violations. Also, many papers focus both on data races and atomicity violations. However, contributions focusing on suspension, starvation and deadlock are sparse, and livelock bugs are not regarded as an open issue in development of concurrent software. This may of course be due to the fact that management of these types of bugs are considered to a resolved issue, but the question of to what extent the managing techniques has spread to software engineering professionals is still open. For example, in their recent study on real-world bugs, Lu et al. [16] found that over 30% of the sample of bugs they studied were in fact deadlock bugs.

It should also be noted that since papers [19], [20] and [21] did not investigate any specific bugs and their topics was mostly related on analyzing the system testing result, establishing a framework for achieving good performance, and testing strategy, they are not mapped to any particular bug type in the classification.

Additionally, by mapping the result of an empirical study [16] with our classification we found out all concurrent bugs that they investigated in their study is mapped to our classification.

VII. Conclusion and Future Work

In this paper we propose a bug classification for concurrent and multithreaded applications. The classification has been derived by first searching for relevant papers and extracting bug information from them, making a list of bugs and determining the observable properties for each bug, and then classifying

TABLE I: Concurrent software bugs properties

					Data race		Order violation			Atomicity violation			
										Single variable		Multi variable	
Property	Deadlock	Livelock	Starvation	Suspension	Memory inconsistency	Write-Write race	Order violation 1	Order violation 2	Order violation 3	Single variable-AV 1	Single variable-AV 2	Multi variable-AV 1	Multi variable-AV 2
At least one thread $t \in T_b$ is in the Waiting state	X			X				X		X		X	
At least one thread $t \in T_b$ is the Executing state		X	X	X	X	X		X	X	X	X	X	X
At least one thread $t \in T_b$ is in the Ready state			X						X		X		X
All threads in T_b have read and written to a spinlock variable		X											
All threads in T_b are waiting for a lock held by another involved thread	X												
At least one thread $t \in T_b$ is in the ready queue for an unacceptably long time			X										
At least one thread $t \in T_b$ is in Waiting state for an unacceptably long time	X			X									
All threads in T_b are in Executing state					X	X	X						
No thread $t \in T_b$ is able to proceed and progress	X	X											
There are incorrect or unexpected results					X	X	X	X	X	X	X	X	X
The number of threads in T_b is larger than the number of free processor cores			X						X		X		X
Potential request to a resource is larger than the number of available resources of that type				X									
All threads in T_b hold a lock	X												
At least one of the threads $t \in T_b$ holds a lock	X			X				X	X	X	X	X	X
Accesses to shared memory were made from different threads in T_b					X	X	X	X	X	X	X	X	X
At least one of the memory accesses was Write					X	X	X	X	X	X	X	X	X
Accesses to shared memory were targeted the same memory location					X	X	X	X	X	X	X	X	X
The memory accesses were NOT protected by a synchronization mechanism					X	X							
Accesses to shared memory targeted just one memory location										X	X		
Accesses to shared memory targeted more than one memory locations												X	X
There were at least two accesses to the same shared memory location, a Write and a Read, where the Read has happened too early					X								
There were at least two Write accesses to shared memory, and they occurred without any Read in-between						X							
There was at least one correct execution ordering between the memory accesses which the program failed to enforce							X	X	X				
An atomic execution of statements was required										X	X	X	X

these bugs in a common structure using these observable properties.

The grouping and classification of concurrency bugs presented in this paper is structured based on properties that are commonly observable in concurrent systems. In its design, it is intended to serve as an aid for software developers during debugging and testing of concurrent applications. By using the knowledge on the connection between bug types and observable properties, the bug classification helps users to make appropriate decisions when they encounter problems. The classification may also serve as a structure in which the current body of knowledge can be arranged, thereby allowing for identification of gaps in this knowledge.

As for future work, we intend to elaborate this classification, adding more rigour to the definitions of the different bug types. Further, we seek to empirically investigate the occurrence and frequency of concurrency bugs in real-world software, as well as what is done to prevent and remedy such bugs.

Acknowledgment

We acknowledge the Swedish Foundation for Strategic Research (SSF) SYNOPSIS Project for supporting this work.

References

[1] K. Henningsson and C. Wohlin, “Assuring fault classification agreement - an empirical evaluation,” in *2004 International Symposium on Empirical Software Engineering, 2004. ISESE ’04. Proceedings*, Aug. 2004, pp. 95–104.

TABLE II: Mapping of the classification to the state of the art

Property	Deadlock	Livelock	Starvation	Suspension	Data race	Order violation	Atomicity violation
Park, S. et al. [18]					X	X	X
Araujo, W. et al. [19]	-	-	-	-	-	-	-
Oh, J. et al. [20]	-	-	-	-	-	-	-
Rungta, N. and Mercer, E. [21]	-	-	-	-	-	-	-
Ball, T. et al. [22]			X	X	X	X	X
Bodden, E. and Havelund, K. [23]					X		
Chen, F. et al. [24]					X		X
Hammer, C. et al. [25]					X		X
Lai, Z. et al. [26]					X		X
Lei, Y. and Carver, R.H. [27]					X		
Liu, P. and Zhang, C. [28]	X						X
Lu, S. et al. [29]							X
Pankratius, V. et al. [30]					X		
Sheng, T. et al. [31]					X		
Wang, L. and Stoller, S.D. [32]					X		X

[2] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 135–145.

[3] L. L. Wu and G. E. Kaiser, "Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators," Tech. Rep., 2011.

[4] S. Eldh, S. Punnekkat, H. Hansson, and P. Jnsson, "Component testing is not enough-a study of software faults in telecom middleware," in *Testing of Software and Communicating Systems*. Springer, 2007, pp. 74–89.

[5] P. Bourque, R. Fairley, and eds., "Guide to the software engineering body of knowledge, version 3.0," 2014.

[6] V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[7] "Systems and software engineering vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, Dec. 2010.

[8] D. Gove, *Multicore Application Programming: For Windows, Linux, and Oracle Solaris*. Addison-Wesley Professional, 2010.

[9] N. Yoshiura and W. Wei, "Static data race detection for java programs with dynamic class loading," in *Internet and Distributed Computing Systems*. Springer, 2014, pp. 161–173.

[10] S. Akhter and J. Roberts, *Multi-core programming*. Intel press Hillsboro, 2006, vol. 33.

[11] Y. Bhatia and S. Verma, "Deadlocks in distributed systems," *International Journal of Research*, vol. 1, no. 9, pp. 1249–1252, 2014.

[12] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008, vol. 10.

[13] W. Stallings, *Operating Systems- internals and design principles*. Prentice Hall Englewood Cliffs, 2012, vol. 7th.

[14] S. Lin, A. Wellings, and A. Burns, "Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 16, pp. 2227–2251, 2013.

[15] D. Jayasinghe and P. Xiong, "CORE: Visualization tool for fault localization in concurrent programs."

[16] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ACM Sigplan Notices*, vol. 43, no. 3. ACM, 2008, pp. 329–339.

[17] S. Park, R. Vuduc, and M. J. Harrold, "A unified approach for localizing non-deadlock concurrency bugs," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 51–60.

[18] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 245–254.

[19] W. Araujo, L. C. Briand, and Y. Labiche, "Enabling the runtime assertion checking of concurrent contracts for the java modeling language," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 786–795.

[20] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic, "LIME: A framework for debugging load imbalance in multi-threaded execution," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 201–210.

[21] N. Rungta and E. Mercer, "Slicing and dicing bugs in concurrent programs," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 195–198.

[22] T. Ball, S. Burckhardt, J. de Halleux, M. Musuvathi, and S. Qadeer, "Deconstructing concurrency heisenbugs," in *31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009*, May 2009, pp. 403–404.

[23] E. Bodden and K. Havelund, "Aspect-oriented race detection in java," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 509–527, Jul. 2010.

[24] F. Chen, T. Serbanuta, and G. Rosu, "jPredictor," in *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE '08*, May 2008, pp. 221–230.

[25] C. Hammer, J. Dolby, M. Vaziri, and F. Tip, "Dynamic detection of atomic-set-serializability violations," in *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE '08*, May 2008, pp. 231–240.

[26] Z. Lai, S. C. Cheung, and W. K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 235–244.

[27] Y. Lei and R. Carver, "Reachability testing of concurrent programs," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 382–403, Jun. 2006.

[28] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 299–309.

[29] S. Lu, S. Park, and Y. Zhou, "Finding atomicity-violation bugs through unserializable interleaving testing," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 844–860, Jul. 2012.

[30] V. Pankratius, F. Schmidt, and G. Garreton, "Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java," Jun. 2012, pp. 123–133.

[31] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng, "RACEZ: A lightweight and non-invasive race detection tool for production applications," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 401–410.

[32] L. Wang and S. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Transactions on Software Engineering*, vol. 32, no. 2, pp. 93–110, Feb. 2006.