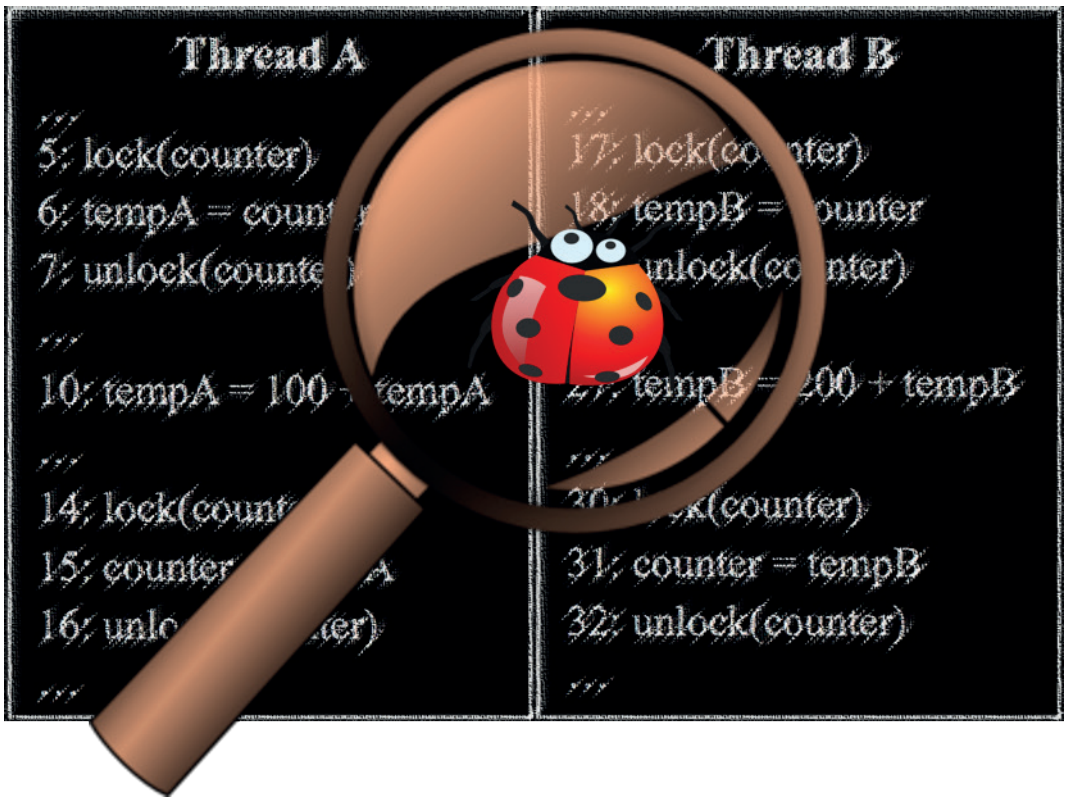
Mälardalen University Doctoral Dissertation 278

Concurrency Bugs

Characterization, Debugging and Runtime Verification

Sara Abbaspour Asadollah



MÄLARDALEN UNIVERSITY
SWEDEN

Mälardalen University Press Dissertations
No. 278

CONCURRENCY BUGS

CHARACTERIZATION, DEBUGGING AND RUNTIME VERIFICATION

Sara Abbaspour Asadollah

2018

School of Innovation, Design and Engineering

ISBN 978-91-7485-412-1
ISSN 1651-4238
Printed by E-Print AB, Stockholm, Sweden

Mälardalen University Press Dissertations
No. 278

CONCURRENCY BUGS
CHARACTERIZATION, DEBUGGING AND RUNTIME VERIFICATION

Sara Abbaspour Asadollah

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
tisdagen den 4 december 2018, 13.30 i Kappa, Mälardalens högskola, Västerås.

Fakultetsopponent: Associate Professor Tao Yue, University of Oslo

Akademin för innovation, design och teknik

Abstract

Concurrent software has been increasingly adopted in recent years, mainly due to the introduction of multicore platforms. However, concurrency bugs are still difficult to test and debug due to their complex interactions involving multiple threads (or tasks). Typically, real world concurrent software has huge state spaces. Thus, testing techniques and handling of concurrency bugs need to focus on exposing the bugs in this large space. However, existing solutions typically do not provide debugging information to developers (and testers) for understanding the bugs.

Our work focuses on improving concurrent software reliability via three contributions: 1) An investigation of concurrent software challenges with the aim to help developers (and testers) to better understand concurrency bugs. We propose a classification of concurrency bugs and discuss observable properties of each type of bug. In addition, we identify a number of gaps in the body of knowledge on concurrent software bugs and their debugging. 2) Exploring concurrency related bugs in real-world software with respect to the reproducibility of bugs, severity of their consequence and effort required to fix them. Our findings here is that concurrency bugs are different from other bugs in terms of their fixing time and severity, while they are similar in terms of reproducibility. 3) A model for monitoring concurrency bugs and the implementation and evaluation of a related runtime verification tool to detect the bugs. In general, runtime verification techniques are used to (a) dynamically verify that the observed behaviour matches specified properties and (b) explicitly recognize understandable behaviors in the considered software. Our implemented tool is used to detect concurrency bugs in embedded software and is in its current form tailored for the FreeRTOS operating system. It helps developers and testers to automatically identify concurrency bugs and subsequently helps to reduce their finding and fixing time.

ISBN 978-91-7485-412-1
ISSN 1651-4238

Sammanfattning

Parallellt exekverande mjukvara har blivit allt vanligare under senare år tack vare introduktionen av multicore-plattformar. Buggar relaterade till den parallella exekveringen är emellertid fortfarande svåra att testa och felsöka på grund av bl.a. komplexa och svårförutsägbara interaktionsmönster. Tillståndsrymden för system av parallella program är typiskt enorm. Således är en central utmaning att hitta buggarna i denna stora tillståndsrymd. Existerande lösningar tillhandahåller emellertid ofta inte tillräcklig felsökningsinformation till utvecklare (och testare) för att de ska kunna förstå buggarna och hur de kan åtgärdas.

Vårt arbete fokuserar på att förbättra tillförlitligheten i parallell mjukvara via i huvudsak tre bidrag: 1) Ökad förståelse av buggar relaterade till parallell exekvering genom en klassificering av sådana buggar baserad på deras observerbara egenskaper. Dessutom har vi identifierat ett antal luckor i kunskapsläget om programvarufel och felsökning av buggar relaterade till parallell exekvering. 2) En undersökning av parallellrelaterade buggar i verklig mjukvara med avseende på deras reproducerbarhet och hur svårt och kostsamt det är att åtgärda dem. Vi observerade här att de parallella buggarna skiljer sig från övriga buggar avseende såväl tid att åtgärda som hur allvarliga konsekvenser de kan leda till, samtidigt som de inte skiljer sig nämnvärt vad gäller reproducerbarhet. 3) En modell för övervakning av parallella buggar och ett tillhörande verktyg som utgående från programexekveringar kan upptäcka och klassificera buggarna. Verktyget används för att automatiskt upptäcka parallella buggar i inbyggd programvara och är skräddarsytt för operativsystemet FreeRTOS. Det hjälper utvecklare och testare att förstå parallella buggar som kan ha missats av befintliga verktyg för buggdetektering och mjukvarutestning. Därmed kan tiden för identifiering och åtgärd av dessa buggar kortas.

Abstract

Concurrent software has been increasingly adopted in recent years, mainly due to the introduction of multicore platforms. However, concurrency bugs are still difficult to test and debug due to their complex interactions involving multiple threads (or tasks). Typically, real world concurrent software has huge state spaces. Thus, testing techniques and handling of concurrency bugs need to focus on exposing the bugs in this large space. However, existing solutions typically do not provide debugging information to developers (and testers) for understanding the bugs.

Our work focuses on improving concurrent software reliability via three contributions: 1) An investigation of concurrent software challenges with the aim to help developers (and testers) to better understand concurrency bugs. We propose a classification of concurrency bugs and discuss observable properties of each type of bug. In addition, we identify a number of gaps in the body of knowledge on concurrent software bugs and their debugging. 2) Exploring concurrency related bugs in real-world software with respect to the reproducibility of bugs, severity of their consequence and effort required to fix them. Our findings here is that concurrency bugs are different from other bugs in terms of their fixing time and severity, while they are similar in terms of reproducibility. 3) A model for monitoring concurrency bugs and the implementation and evaluation of a related runtime verification tool to detect the bugs. In general, runtime verification techniques are used to (a) dynamically verify that the observed behaviour matches specified properties and (b) explicitly recognize understandable behaviors in the considered software. Our implemented tool is used to detect concurrency bugs in embedded software and is in its current form tailored for the FreeRTOS operating system. It helps developers and testers to automatically identify concurrency bugs and subsequently helps to reduce their finding and fixing time.

To my beloved Family,

My Soulmate

&

Whom it may read

Acknowledgments

My most earnest acknowledgment must go to my supervisor, Prof. Hans Hansson, for his extraordinary guidance, caring, and patience. As an excellent supervisor and researcher, he will be a great example throughout my professional life. This thesis would not exist without the contributions of my co-supervisors Prof. Daniel Sundmark and Dr. Sigrid Eldh for their continuous effort to support and encourage me. Their invaluable suggestions and discussions played an important role in improving this thesis.

I am very grateful to my colleagues and friends, Dr. Rafia Inam, Dr. Eduard Paul Enoiu, Dr. Adnan Čaušević and Docent Wasif Afzal for their supports, discussions and feedbacks as co-authors in my published papers. Also thanks to Prof. Elaine Weyuker and Prof. Thomas Ostrand for useful discussions. Thanks to all my friends and colleagues at Mälardalen University providing a fruitful environment and giving support when I have needed.

From the bottom of my heart, I would like to extend my deepest gratitude to my parents as well as my brother and sister for their unconditional support, love, and faith in many phases of my life and a big thanks to my husband for his unbounded support, tolerance and love for me. Without their support, I would not have been able to reach here.

Above all, I thank God for helping me and sending people who have been such strong influence in my life and giving confidence at my hard moments.

This research has been supported by Swedish Foundation for Strategic Research (SSF) via the SYNOPSIS project and the Swedish Research Council (VR) via the EXACT project.

Sara Abbaspour Asadollah
Västerås, October, 2018

List of publications

Papers included in the thesis[1]

Paper A *Towards Classification of Concurrency Bugs Based on Observable Properties*, Sara Abbaspour Asadollah, Hans Hansson, Daniel Sundmark, Sigrid Eldh. In the Proceedings of the 1st International Workshop on Complex faults and failures in large software systems (COUFLESS), ICSE 2015 Workshop, May 2015.

Paper B *10 Years of Research on Debugging Concurrent and Multicore Software: A Systematic Mapping Study*, Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson and Wasif Afzal. Software Quality Journal, January 2016.

Paper C *Concurrency Bugs in Open Source Software: A Case Study*, Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. Journal of Internet Services and Applications, December 2017.

Paper D *A Runtime Verification based Concurrency Bug Detector for FreeRTOS Embedded Software*, Sara Abbaspour Asadollah, Eduard Paul Enoiu, Adnan Čaušević, Daniel Sundmark and Hans Hansson. Submitted for a journal publication, September 2018.

[1] The included articles have been reformatted to comply with the thesis layout.

Additional papers, not included in the thesis

1. *A Runtime Verification Tool for Detecting Concurrency Bugs in FreeRTOS Embedded Software*, Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. In the Proceedings of the 17th IEEE International Symposium on Parallel and Distributed Computing (ISPDC-2018), August 2018.

2. *Management of Service Level Agreements for Cloud Services in IoT: A Systematic Mapping Study*, Saad Mubeen, Sara Abbaspour Asadollah, Alessandro Papadopoulos, Mohammad Ashjaei, Hongyu Pei-Breivold, and Moris Behnam. Journal of IEEE Access (ACCESS), June 2018.

3. *SLAs for Industrial IoT: Mind the Gap*, Alessandro Papadopoulos, Sara Abbaspour Asadollah, Mohammad Ashjaei, Saad Mubeen, Hongyu Pei-Breivold, and Moris Behnam. In the Proceedings of the 4th International Symposium on Inter-cloud and IoT (ICI 2017), August 2017.

4. *The pedagogical challenges of creating information literate librarians*, Maryam Derakhshan, Mohammad Hassanzadeh, Susan E. Higgins, and Sara Abbaspour Asadollah. Library Review Journal, Emerald Publishing, March 2017. **Best Paper award.**

5. *Runtime Verification for Detecting Suspension Bugs in Multicore and Parallel Software*, Sara Abbaspour Asadollah, Daniel Sundmark, and Hans Hansson. In the Proceedings of the 1st International Workshop on Testing Extra-Functional Properties and Quality Characteristics of Software Systems (ITEQS'17), ICST 2017 Workshop, March 2017.

6. *A Model for Systematic Monitoring and Debugging of Starvation Bugs in Multicore Software*, Sara Abbaspour Asadollah, Mehrdad Saadatmand, Sigrid Eldh, Daniel Sundmark, and Hans Hansson. The 1st International Workshop on Specification, Comprehension, Testing and Debugging of Concurrent Programs (SCTDCP2016), ASE 2016, August 2016.

7. *A Study on Concurrency Bugs in an Open Source Software*, Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson,

and Eduard Paul Enoiu. In the Proceedings of the 12th International Conference on Open Source Systems (OSS'16), June 2016.

8. *A Survey on Testing for Cyber Physical System*, Sara Abbaspour Asadollah, Rafia Inam, Hans Hansson. In the Proceedings of the 27th International Conference on Testing Software and Systems (ICTSS), Lecture Notes in Computer Science series, November 2015.

Contents

Chapter 1

Introduction

Concurrent software is getting increasingly popular due to the advancement of multicore processors. To obtain greater performance from multicore processors, developers need to implement parallel code, either by transforming sequential code or writing the code from scratch. From a software developer point of view, concurrent and parallel software introduce the possibility of a new type of software malfunctions, known as concurrency bugs [1]. The bugs typically appear under very specific (nondeterministic) thread interleavings between shared memory access. Their effects spread through the software until they cause the software to hang, crash or produce incorrect output. Unlike bugs in sequential programs, manifestation of concurrency bugs dependents not only on the program input but also on the scheduling and timing of different threads (or tasks). Concurrency bugs are hard to detect because multithreaded code can demonstrate different behavior based on the scheduling of threads, and the bugs may only be triggered by a small specific set of schedules. They are thus typically are considered to be problematic [2, 3, 4].

In real-world software, concurrency bugs in deployed systems have caused several disasters in the past and are generating increasingly severe problems in recent times with the growing popularity of multicore hardware. For instance, in the 1980s, a concurrency bug in the Therac-25 radiation therapy machine, caused radiation overdoses and killed at least five patients, with additional patients severely injured [1]. In 2003, ten million people were out of power due to a race condition in a monitoring software with multi-million lines of code (the

often cited 2003 Northeastern U.S. electricity blackout[5]). Facebook's initial public offering (IPO) was delayed by more than half an hour, leading to a loss of 350 million dollars due to a race condition in NASDAQ's IT systems [6].

It is extremely important for businesses to avoid these catastrophic losses. In 2007 a survey was conducted by Microsoft researchers to assess the state of the practice of concurrency in their products. The researchers indicated that in their company over 60% of respondents had to deal with concurrency issues while half of the concurrency issues occurred at least monthly [3].

Several formal approaches have been developed oriented to the analysis of computer software in order to diagnose and detect concurrency bugs during software development. Abstract interpretation, model checking, symbolic execution, and data-flow analysis are some of the most commonly used types of formal methods [7]. Similar to other formal verification techniques, the use of static analysis to discover and diagnose concurrency bugs during software development is costly. The main issues with these techniques is that analyzing all possible program executions takes a considerable amount of time.

Runtime verification, also referred to as dynamic analysis, can be used for many purposes [8], such as testing, debugging, validation, fault protection, profiling, verification, security or safety policy monitoring, and behavior modification. Runtime verification is concerned with checking a single trace of the program against properties described in some logic [9]. When a property is validated or violated, the program or runtime verification tool could take action in order to deal with the situation. Debugging can also benefit from runtime verification. Debugging, considered as a separate process and a key activity in software development, involves several steps i.e., identifying, localizing and fixing bugs.

Most experimental studies on concurrent and parallel applications provide information on application cost and efficiency, while there is still lack of knowledge to support the prevention and detection of concurrency bugs. There is a need for deeper knowledge on detecting and fixing concurrency bugs.

1.1 Concurrent Software Challenges

Concurrent software testing and debugging are, compared to that of sequential software, faced with a variety of challenges. The main challenges are:

- Concurrency bugs typically involve changes in program state due to particular interleavings of multiple threads (or tasks) of execution, which can make the bugs difficult to find and understand. Therefore, many concurrency bugs remain hidden in software until the software runs in a real environment and even then, it may take a long time before the bug manifests itself.

- The thread interleavings may vary a lot depending on the platform selected for software execution. As a consequence, the type of run-time environment which is selected for software execution largely affects the behavior, leading to the occurrence of different bugs on different platforms.

- Typically, concurrency bugs have a unique and notorious property which is non-determinism. In practice, every time an application executes, the background conditions are different. A problem may manifest only once every 10000 times during the application execution. Thus, repeated execution of the same concurrent source code will typically not guarantee the same result after each execution, even with the same input data. This non-determinism makes concurrency bugs difficult to identify, detect and fix, since developers might not be able to systematically reproduce the bug using traditional debugging methods. Moreover, non-determinism may introduce many troubles into testing. Thus, many concurrency bugs may sneak into production runs and manifest at user's site under special conditions. In general, reproducing the thread schedule, which led a specific bug, might be very difficult and the non-deterministic thread scenarios make concurrent software testing and debugging extremely difficult.

1.2 Research Method

This section includes an overview of the research methods used in the research presented in this thesis. The general research process and the overall validity of the studies are discussed in this section.

As shown in Figure 1.1, the research process of the research presented in this thesis consists of the following steps.

1. *Formulation of Main Research Goal.*

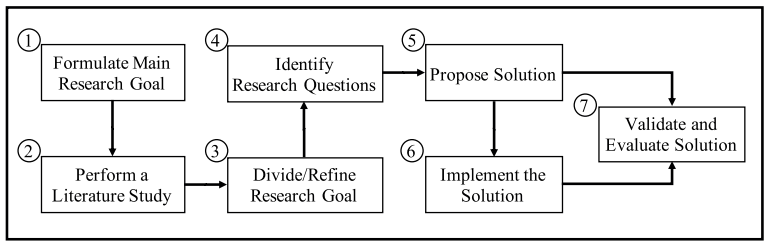


Figure 1.1: Overview of the research methods applied.

2. *A Literature Study* to better understand the common terminology and identify the gaps in the area based on guiding goals.

3. *Division/Refinement of the Research Goal* into smaller easily manageable subgoals and narrowing the scope of the study.

4. *Identify Research Questions* to approach the research subgoals. Several research questions are raised accordingly.

5. *Propose Solutions* to fulfill the main goal (or subgoals).

6. *Implement the Solution(s)*, Implementing the proposed solution was needed since only finding a solution would not constitute a sufficient remedy to the problem due to the nature of the goal and problem(s).

7. *Validate and Evaluate the Solution* to provide proof of concepts and evidence contributing to answering the research questions and fulfilling the research goal.

1.3 Motivation and Goal of Thesis

This research is carried out in the context of concurrent software debugging and runtime verification. The main goal of this research is:

To provide effective concurrency bug detection and related concurrent software runtime verification techniques applicable in real-world scenarios.

According to the literature study and research problem(s), we divide the main research goal into the following subgoals as a guideline for our research.

- **Subgoal 1**: To provide a common terminology for distinguishing between different types and classes of concurrency bugs and to identify the interrelation between separate classes.

- **Subgoal 2**: To identify the current gaps and less-explored areas in debugging of concurrency bugs.

- **Subgoal 3**: To identify the current state of concurrency related bugs in real-world software in terms of frequency, severity, resolving time and reproducibility.

- **Subgoal 4**: To propose a model and implement a tool for monitoring and detecting concurrency bugs.

- **Subgoal 5**: To evaluate the implemented tool in real-world concurrent software.

In general, to fulfill the main research goal and the subgoals, we started to define a theory by presenting a classification of bugs related to concurrent execution of application level software threads (or tasks). Then, we performed a systematic mapping study of the related existing literature by identifying the type of bug(s) and the addressed phase(s) in the debugging process. Next, we explored the nature and extent of concurrency bugs in real-world (open source) software by performing a case study. Finally, we proposed a runtime verification model for detecting concurrency bugs and based on the proposed model we implemented a tool for embedded software on open source real-time operating system. The tool can detect and diagnose some type of concurrency bugs such as deadlock, starvation and suspension-based locking. We have verified our implementation and presented our evaluation performed on software executing on an ARM Cortex-M-based micro-controller by injecting predefined concurrency bugs in Atmel Studio and detected the bugs by our implemented tool. Also, we experimentally evaluate the implemented tool using 21726 automatically generated logs using our own automated generator based on the UPPAAL model checker [10].

1.4 Research Contribution

To provide a common terminology for distinguishing between different types and classes of concurrency bugs and to identify the interrelation between separate elements and classes (Subgoal 1), we proposed a disjoint classification of concurrency bugs. We classified the bugs in a common structure considering their observable properties in Paper A [11].

In order to achieve Subgoal 2, we provided an overview of existing research on concurrent software debugging [12]. We undertook a systematic mapping study in order to clarify current research solutions and research gaps in the field. We highlighted the research gaps in the field based on attributes such as *types of concurrency bugs*, *types of debugging processes*, *types of research* and *research contributions*. The results of our mapping study indicate that the current body of knowledge concerning debugging concurrent software does not report studies on many of the types of bugs or on the debugging process. Thus, there are still quite a number of issues and aspects that have not been sufficiently covered in the field.

Next, we investigated 11860 fixed bug reports from a widely used open source storage designed for big-data applications [13]. We started by selecting a proper open source software for our study. We considered five open source applications viz., Apache Hadoop project[1], Apache ZooKeeper project[2], Apache Oozie project[3], Apache Accumulo project[4] and Apache Spark project[5]. The projects coordinate distributed processes with significant number of releases and an issue management platform for managing, configuring and testing. We identified the set of concurrency bug reports in the issue tracking database of the selected projects through a keyword search. We automatically filtered reports that are not likely to be relevant by performing a search query on the bug report databases. Then we manually analyzed the full set of identified bug reports in order to exclude those which were not concurrency-related. We determined the relevance of the bugs by checking if they describe a concurrency bug, and if they do, what type of concurrency bug it is. Two aspects of these reports were examined: fixing time and

[1] https://issues.apache.org/jira/browse/HADOOP
[2] https://issues.apache.org/jira/browse/ZOOKEEPER
[3] https://issues.apache.org/jira/browse/OOZIE
[4] https://issues.apache.org/jira/browse/ACCUMULO
[5] https://issues.apache.org/jira/browse/SPARK/

severity. Finally, we collected data for the concurrency bugs, and classified the bug reports using the classification scheme described in Section 4.1. Each bug report contains several types of information, which were valuable in recognizing and filtering the concurrency bugs with other types of bugs to aid us understand the characteristics of bugs. Finally, we analyzed the result of the study and discussed the frequencies of concurrency and non-concurrency bugs. The study is useful to recognize the most common types of concurrency bugs in terms of severity, fixing time and reproducibility. By this we address Subgoal 3.

As explained in Section 1.1, due to the complexity of concurrent and parallel software, detecting potential concurrency bugs in early stages of the software life-cycle might be difficult as they usually arise during system execution. Thus, software monitoring may address and alleviate this challenge by collecting, processing and measuring significant data at execution time. This led us to propose a runtime verification model for detecting and identifying three types of concurrency bugs (*Deadlock*, *Starvation*, and *Suspension*). We have considered these three bug types based on our prior study, Paper B [12], where further motivation is provided. The model is proposed for these three types of bugs and it can be extended for other types of concurrency bugs. We also implemented a tool to find the concurrency bugs at runtime without debugging and tracing the source code [14] (Paper D). The models and the tool address Subgoal 4.

To achieve Subgoal 6, we experimentally evaluated the implemented tool using realistic FreeRTOS test scenarios. Three types of concurrency bug examples running on an AVR 32-bit board SAM4S were implemented and injected to a bug-free embedded software. We used the real log files collected during system execution as an input to implemented tool in order to detect the injected bugs and shows a proof-of-concept evaluation using realistic FreeRTOS logs. Moreover, we evaluated the implemented tool by generating log files using the UPPAAL model checker. We automatically generated 21726 log files by using our own trace generator[6] based on the UPPAAL model checker and used them as an input to our tool in order to evaluate its bug detecting capability.

[6]Trace Generator is a transformation tool which we implemented for translating the Uppaal traces into log files supported by the Tracealyzer file template [14] (Paper D).

1.4.1 Publications Included in the Thesis

Paper A

Towards Classification of Concurrency Bugs Based on Observable Properties [11], co-authored by Sara Abbaspour Asadollah, Hans Hansson, Daniel Sundmark, Sigrid Eldh.
Status: Published in the Proceedings of the 1st International Workshop on Complex faults and failures in large software systems (COUFLESS), ICSE 2015 Workshop, IEEE, May 2015.

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.

Personal contribution: I am the initiator, main driver and author of all parts in this paper. All other co-authors have contributed with valuable discussion and reviews, in their role as supervisors.

Paper B

10 Years of Research on Debugging Concurrent and Multicore Software: A Systematic Mapping Study [12], co-authored by Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson and Wasif Afzal.
Status: Published in the Software Quality Journal, January 2016.

Abstract Debugging – the process of identifying, localizing and fixing

bugs – is a key activity in software development. Due to issues such as non-determinism and difficulties of reproducing failures, debugging concurrent software is significantly more challenging than debugging sequential software. A number of methods, models and tools for debugging concurrent and multicore software have been proposed, but the body of work partially lacks a common terminology and a more recent view of the problems to solve. This suggests the need for a classification, and an up-to-date comprehensive overview of the area.

This paper presents the results of a systematic mapping study in the field of debugging of concurrent and multicore software in the last decade (2005–2014). The study is guided by two objectives: (1) to summarize the recent publication trends and (2) to clarify current research gaps in the field.

Through a multi-stage selection process, we identified 145 relevant papers. Based on these, we summarize the publication trend in the field by showing distribution of publications with respect to *year*, *publication venues*, *representation of academia and industry*, and *active research institutes*. We also identify research gaps in the field based on attributes such as *types of concurrency bugs*, *types of debugging processes*, *types of research* and *research contributions*.

The main observations from the study are that during the years 2005–2014: (1) there is no focal conference or venue to publish papers in this area, hence a large variety of conferences and journal venues (90) are used to publish relevant papers in this area; (2) in terms of publication contribution, academia was more active in this area than industry; (3) most publications in the field address the data race bug; (4) bug identification is the most common stage of debugging addressed by articles in the period; (5) there are six types of research approaches found, with solution proposals being the most common one; and (6) the published papers essentially focus on four different types of contributions, with ”methods” being the type most common one.

We can further conclude that there is still quite a number of aspects that are not sufficiently covered in the field, most notably including (1) *exploring correction* and *fixing bugs* in terms of debugging process; (2) *order violation*, *suspension* and *starvation* in terms of concurrency bugs; (3) *validation* and *evaluation research* in the matter of research type; (4) *metric* in terms of research contribution. It is clear that the concurrent, parallel and multicore software community needs broader studies in debugging. This systematic mapping study can help direct such efforts.

Personal contribution: I am the main driver and author of this paper. My supervisors contributed in their supervisory capacity. Wasif Afzal has contributed by valuable discussions and reviewing of the whole paper.

Paper C

Concurrency Bugs in Open Source Software: A Case Study [13], co-authored by Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson.
Status: Published in Journal of Internet Services and Applications (JISA), April 2017. This paper is the extended version of the paper: A Study on Concurrency Bugs in an Open Source Software [15] which is one of the selected papers by the conference committee for submitting to the JISA journal.

Abstract Concurrent programming puts demands on software debugging and testing, as concurrent software may exhibit problems not present in sequential software, e.g., deadlocks and race conditions. In aiming to increase efficiency and effectiveness of debugging and bug-fixing for concurrent software, a deep understanding of concurrency bugs, their frequency and fixing-times would be helpful. Similarly, to design effective tools and techniques for testing and debugging concurrent software, understanding the differences between non-concurrency and concurrency bugs in real-word software would be useful.

This paper presents an empirical study focusing on understanding the differences and similarities between concurrency bugs and other bugs, as well as the differences among various concurrency bug types in terms of their severity and their fixing time, and reproducibility. Our basis is a comprehensive analysis of bug reports covering several generations of five open source software projects. The analysis involves a total of 11860 bug reports from the last decade, including 351 reports related to concurrency bugs. We found that concurrency bugs are different from other bugs in terms of their fixing time and severity while they are similar in terms of reproducibility. Our findings shed light on concurrency bugs and could thereby influence future design and development of concurrent software, their debugging and testing, as well as related tools.

Personal contribution: I am the main driver and author of all parts in this paper. My supervisors contributed with valuable discussions, useful ideas and review of the whole paper.

Paper D

A Runtime Verification based Concurrency Bug Detector for FreeRTOS Embedded Software [14], co-authored by Sara Abbaspour Asadollah, Eduard Paul Enoiu, Adnan Čaušević, Daniel Sundmark and Hans Hansson.
Status: Is submitted for a journal publication in September 2018. This paper is the extended version of the paper: A Runtime Verification Tool for Detecting Concurrency Bugs in FreeRTOS Embedded Software [16].

Abstract When developing embedded software, detecting bugs as early as possible is important. Concurrency bugs is a particularly problematic class of bugs. Several methods have been proposed to detect such bugs, but few of these methods have been implemented in tools and even fewer have been evaluated systematically using realistic software logs. In this paper we present a novel method and tool called DeCoB, which uses runtime verification to detect concurrency bugs in embedded software. DeCoB is tailored for the open source real-time operating system FreeRTOS, and detects and diagnoses concurrency bugs, such as deadlock, starvation, and suspension-based-locking, by analyzing runtime traces provided by the Tracealyzer tool, i.e., without debugging and tracing the source code.

This paper presents the implementation of the tool in detail, as well as its functional architecture, together with illustrations of its use in practice. The DeCoB tool can be used during program testing for identifying concurrency bugs using information about the software executions. We experimentally evaluate the DeCoB tool using realistic FreeRTOS test scenarios and 21726 automatically generated logs using our own generator based on the UPPAAL model checker. Our results suggest that the DeCoB tool is effective at detecting whether a diverse set of logs contains concurrency bugs.

Personal contribution: I am the main driver and author of all parts in this paper except the work on the UPPAAL model checker and related

sectiones which were joint work with Eduard Paul Enoiu. My co-authors contributed with valuable discussions, ideas and review of the whole paper.

1.5 Thesis Outline

This thesis is organized in 9 chapters. Chapter 2 introduces the required background of the thesis. In Chapter 3, we present related work relevant to this thesis. Chapter 4 presents the results according to the respective research goals, introduced in Section 1.3. Finally, in Chapter 5, we present a discussion based on our results, a list of conclusions from development of this thesis as well as possible future work, followed by the included papers in Chapter 6 to 9.

Chapter 2

Background

In this chapter, we provide background information needed for understanding the context of the thesis and the work itself.

2.1 Types of Concurrency Bugs

Concurrent programming puts demands on software development and testing. Concurrent software may exhibit problems that may not occur in sequential software. There is a variety of challenges related to faults and errors in concurrent, multicore and multi-threaded applications [17, 18, 19]. A well-known concurrency bug is *Data race*. *Data race* requires that at least two threads access the same data and at least one of them write the data [20]. It occurs when concurrent threads perform conflicting accesses by trying to update the same memory location or shared variable [17, 21]. Figure 2.1 shows an example of a *Data race*.

In the example, the following sequential actions will happen when executing the indicated code in each thread:

1. Load the value of counter in memory.
2. Add 1 to the value.
3. Save the new value to counter.

Thread A	Thread B
… counter = counter + 1; …	… counter = counter + 1; …

Figure 2.1: Data race example.

Consider that this example is a small part of an application which is executing on a Symmetric Multiprocessing (SMP) architecture. In SMP, all CPU cores are identical and have two levels of cache. If a programmer writes a code to run on one core, then the code can run on any of the cores. The memory and I/O devices are shared equally among all of the processors in the SMP [22]. Each core at least has a private level cache[1](L1 cache), while the last level cache (LLC) is shared among all cores. In this example, suppose that threads A and B execute in parallel on Core1 and Core2 and the value of *counter* is 100 initially. After execution, the value of *counter* could be 101 while the expected (correct) result is 102. Both cores execute the indicated line of code, but due to the parallel execution the second load is in this scenario performed before the first save. Hence, the value saved by both threads will be 101. This scenario shows that the result of parallel execution of the example could be incorrect. Thus a concurrency bug (*Data dace*) has occurred.

Atomicity violation is another type of concurrency bug. It 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 threads are executed without being overlapped with any other code block. Figure 2.2 shows an example of single variable atomicity, and Table 2.1 displays the values of shared and local variables after each interleaving execution.

Suppose Thread A is executing on Core1 and Thread B on Core2. Both of them use a shared variable *counter* and each has its local variable (*tempA* and *tempB*). The initial value of *counter* is 0. Since both threads are using the lock mechanism to protect from data corruption, only one core at a time can access the *counter*. If Core1 reaches line 5 before Core2 reaches line 17

[1]Cache is “an area of memory that holds recent used data and instruction” [23].

Thread A	Thread B
…	…
5: lock(counter)	17: lock(counter)
6: tempA = counter	18: tempB = counter
7: unlock(counter)	19: unlock(counter)
…	…
10: tempA = 100 + tempA	27: tempB = 200 + tempB
…	…
14: lock(counter)	30: lock(counter)
15: counter = tempA	31: counter = tempB
16: unlock(counter)	32: unlock(counter)
…	…

Figure 2.2: Atomicity violation example.

	L1 cache of Core1		L1 cache of Core2		LLC			DRAM		
	counter	tempA	counter	tempB	counter	Core1 tempA	Core2 tempB	counter	Core1 tempA	Core2 tempB
1	0	0			0	0	0	0	0	0
2				0	0	0	0	0	0	0
3	0	100	0	200	0	100	200	0	100	200
4	100	100			100	100	1000	100	100	1000
5			200	200	200	100	200	200	100	200

Table 2.1: Shared and local variables' value after interleaving execution.

then the *counter* will be fetched from DRAM (system memory) to LLC and L1 Cache of Core1. *tempA* will be fetched similarly. The value of *tempA* will be 0 after executing line 6 and 7. Meanwhile if Core2 reaches line 17 then Thread B will wait in the waiting queue. By releasing the lock by Core1, Thread B will wait in the ready queue. Since Core2 is free and no more threads is waiting in the ready queue then Core2 will continue to execute Thread B from line 17, 18 and 19. The value of *counter* will be fetched to L1 Cache of Core2 and the *tempB* value of Thread B will be 0. During Core2 execution Core1 is executing Thread A. The *tempA* value of Thread A will be 100 while

the *tempB* value of Thread B becomes 200. If we suppose Core1 reaches line 14 before Core2 reaches line 30 then 100 will be stored in LLC and DRAM as *counter* value, and then Core2 will continue (line 30, 31 and 32) and store 200 in LLC and DRAM. This scenario shows that a concurrency bug (*Single variable atomicity violation*) occurred because the updated *counter* by Core1 is corrupted by Core2.

From the above examples it should be clear that concurrent and parallel executions of threads may lead to bugs that are only possible when executing threads concurrently. Investigating, understanding and detecting such bugs is the main motivation and focus of this thesis.

2.2 Debugging Techniques and Process for Concurrent Software

Debugging is a key activity in the software development life-cycle. Debugging is a methodical process of identifying, localizing, reducing and fixing bugs in a computer program. There are a number of tricks (methods) that can be used in the daily software development activity to facilitate the hunt for software problems (bugs). Some of these methods are as follows:

- ***Exploiting compiler features***: programmers can obtain static analysis of the code provided e.g. by the compiler. Static code analysis is the analysis of software that is performed without actually executing it. Such analysis helps programmers detect a number of basic semantic problems, e.g. type mismatch or dead code.
- ***Abused cout debugging***: the cout technique[2] consists of adding print statements in the code to track the control flow and data values during code execution (also known as Print debugging or Echo Debugging). This technique is the favorite technique of beginners and has been the most common method for debugging [24].
- ***Logging***: logging is another common technique for debugging. This technique automatically record information messages or events to monitor the status of the program in order to diagnose problems.

[2] cout technique's name is taken from the C++ statement for printing on terminal screen (or any standard output stream).

- ***Assertions and defensive programming***: assertions are expressions, which should evaluate to true at a specific point in the code. If an assertion fails, a bug is found. The bug could possibly be in the assertion, but more likely it will be in the code. In this method after an assertion fails it makes no sense to re-execute the program.

- ***Debugger***: a debugger works through the code line-by-line in order to make the execution visible to the developer, thereby helping to find bugs, the location of bugs and the cause of bugs. It can work interactively by controlling the execution of the program and stopping it at various times, inspecting variables, changing code flow whilst running, etc. Trace debugging, Omniscient debugging techniques [24] and Deterministic Replay Debugging (DRD) [25] can be considered as subgroups of this technique.

In addition to traditional debugging techniques, concurrent and parallel programs have specific debugging techniques to support tracing and debugging of multithreaded software. These techniques include:

- ***Event-based debugging***: regards the execution of parallel programs as a series of events and records and analyzes the events when a program is executing. Instant Replay [26] can be considered as a type of this group.

- ***Control information analysis***: this technique can analyze the control information in execution and the global data.

- ***Data-flow-based static analysis***: this technique can detect and analyze the bugs when a program does not execute.

In this section, we present the concepts of the different phases in the debugging process. We discuss the stages that follow after a software failure has been observed, when its root cause is determined and corrected.

From an industrial perspective, a simple life cycle of a software problem is defined by Zeller [27] to include the following phases: (1) A user reports a problem to the software provider; (2) A developer at the software provider reproduces the problem; (3) The developer isolates the circumstances of the problem; (4) The developer fixes the problem locally; (5) The developer delivers the fix(es) to the user.

The debugging process is handled differently in different types of organizations and teams. In a small team with few developers, it is normally clear what part of the code is in question when a program executes unsuccessfully or a test case fails. Here, typically, the developer has to find the bug [28]. In larger organizations, usually the first sign of any bug is the failure of the software or system. The bug fixing process then starts with the submission of an anomaly report. The following list discusses the stages that follow after a software failure has been observed, and its root cause should be determined and corrected.

- ***Bug identification*** is the process of finding the approximate location of a bug (in terms of source code unit, sub-system or even organizational unit), such that the remainder of the debugging process can be assigned to the appropriate stakeholder. It is to be noted that the scope of our definition of bug identification covers terms such as bug localization and bug detection. In case the failure was detected during testing, bug identification is usually performed by the testing team and is followed by a team review to prioritize fixes [29].

- ***Type of bug identification*** is a process to help developers in finding the real cause of a bug by understanding the type of bug. In [11] we extended the common debugging process by adding a sub-process that suggests that before the type of bug is identified, developers could check the properties of identified bug(s) and compare them with the properties given for each class of concurrency bugs. Thus, developer(s) can thereby identify the type of the bug at hand.

- In ***cause identification***, the root cause of a bug is identified. Since the root cause refers to the most basic reason(s) for the occurrence of a bug, during this process a bug can reasonably be identified by a developer or the debugger (e.g., unexpected value of variable A was the root cause of a bug related to variable B or an erroneous lock was the root cause of bug number 5).

- The process of ***exploring corrections*** can be applicable when we have more than one possible solution for fixing the bug. Typically, the potential solutions are compared and the best solution for the current bug is selected.

- ***Fixing bug*** is the process for repairing and fixing the current bug. It is the last stage of the debugging process in order to remove the bug.
- Finally, after debugging is completed the fixed system needs to be tested to ensure that the fix did not introduce new bugs in the system (***regression testing***).

2.3 Runtime Verification

Monitoring the behavior of a software, either on the fly as it executes, or post-mortem after its execution (analyzing log files) is considered as a runtime verification component [30]. Runtime verification is concerned with checking a program trace against properties described in some logic [9]. When a property is violated, the technique makes it possible to act in order to deal with the situation. However, since the check is done during the program execution, only states that are actually reached are considered. Runtime verification is useful in both testing and monitoring. For instance, if a user comes up with a test case and wants to check a possible bug, then a runtime verification tool can be considered to automate the creation of oracles for revealing the bug [31]. Further, if the user is interested to act in response to the violation of a property, then she/he can consider runtime verification as a monitor. The monitor can guide the program reaction to bugs and steer it to the correct behavior [32].

Runtime verification is mainly used to detect unexpected or even expected behaviors of a software during execution. It helps the underlying program to observe relevant events and feed them to a decision procedure (a monitor). Then the monitor states a decision property fulfilment or violation. Runtime verification is a useful technique in verifying the user-provided specifications by checking if the software satisfies a given specification or not. There are some approaches typically proposed for specification-based runtime monitoring. We can classify these approaches in four main categories: rule-based approaches [30, 33], automaton based approaches [34, 35, 36, 9], temporal logic-based approaches [37, 38, 39, 40, 41], and regular expression and grammar-based approaches [42].

Runtime verification is a useful technique for detecting concurrency bugs in multi-threaded and concurrent software. It can extract the information during the software execution in order to determine if a concurrency bug is happening (or happened) on any execution.

Chapter 3

Related Work

This chapter presents a cross-section of related work relevant to this thesis.

3.1 Literature Reviews and Classification Studies on Concurrent Software

There are some SLR, surveys, and state of art review studies related to concurrent software testing and debugging. These reviews provide a list of relevant studies in the area. A systematic review on concurrent software testing was published by Brito et al. [43] in 2010. Their main goal was to obtain evidence of current state-of-the-art related to testing criteria, testing tools and to find bug taxonomies for concurrent and parallel programs. They further provided a list of relevant studies as a foundation for new research in the area. The authors concluded that there is a lack of testing criteria and tools for concurrent programs. They notice that most experimental studies are providing information on application cost, efficiency and complementary aspects, while there is lack of knowledge on bug taxonomy and on evaluating testing criteria. We use a similar study methodology (Systematic Mapping Study) with focus on current state of research related to debugging criteria rather than testing. However, our study is based on different classifications compared to Brito et al.'s study.

A state of the art review on deterministic replay debugging in multithread programming was performed by Wang et al. [44] in 2012. They categorize

replay-based debugging techniques for parallel and multithread programs and divided them into three types: hardware-based, software-based and hybrid methods. Furthermore, software-based methods are classified into two groups: virtual machine based methods and pure software-based methods. Further, they present some classical software-based systems for multithread deterministic replay debugging. Related to this, we provide a state of the art overview with focus on the processes that may occur during concurrent software debugging.

Hong and Kim present a survey of race bug detection techniques for multithreaded software [45]. They classify 43 race bug and corresponding race bug detection techniques. In addition, they describe and compare the mechanisms of race bug detection techniques. Further, the authors present some examples of race bugs, with the aim to help software developers to avoid race bugs in their code.

Moreover, related to this thesis there are some other studies that propose taxonomies covering concurrency bug types. Long et al. [46] present a classification of Java concurrency bugs by using a Petri-net model diagram. The transitions in the model represent changes in the concurrent state of a thread. The classification is used to justify the construction of concurrency flow graphs for each method in a concurrent component. The authors believe that the concurrency flow graphs can be used in the construction of test sequences for testing concurrent components to ensure coverage of concurrency primitives.

Tchamgoue et al. [47] classify event-driven program models into low and high level based on event types. They categorize concurrency bug patterns in event-driven programs. In addition to the taxonomy, they survey tools for detecting concurrency bugs in these programs. In contrast, our classification of concurrency bugs is based on symptom and system state bug properties.

Helmbold et al. [48] summarize the concepts of race bug detection techniques for parallel software, and present a taxonomy with respect to the characteristics of the target program structure. Their race taxonomy separates races into categories based on the error types that cause that kind of race (e.g. loop, synchronization operations).

3.2 Tools for Debugging Concurrent Software

To our knowledge, there are few related studies on debugging concurrent programs. One of these is done by Lönnberg et al. to investigate how students

understand concurrency bugs [49]. The authors performed an empirical study on students, by providing an assignment to students (to write concurrent programs). They suggested several ways to help students debug their assignments. For instance, they guided students to use software visualization tools. Further, the authors interviewed the students and analyzed their responses. The authors claim that since students usually have different understanding of concurrent programs from teachers, software visualization tools will help both teachers and students to get the same view of the programs and bugs. Another study is done by Sadowski and Yi to show how developers use a new concurrency notation called *cooperability* [50]. They posted three concurrency bugs on an internet-based survey form, divided participants into two groups, where one group of people have the aid of *cooperability* and the others do not. In evaluating the responses, they scored the correctness of the responses with a ranking scheme and statistically showed that developers can understand concurrency bugs better with the aid of *cooperability*.

In order to help developers to debug concurrent software and trace the thread interactions some visualization tools such as CHESS [51], JPF [52], TIE [53], JIVE [54, 55], JOVE [54, 55], FALCON [5], UNICORN [56], GRIFFIN [57] and Concurrency Explorer [51] are proposed.

Most of these tools are evaluated with toy programs and not with real concurrent software, except the Concurrency Explorer, which is used internally at Microsoft.

In addition, there are some tools proposed by researchers for detecting concurrency bugs, including data race detectors, serializability violation detectors, atomicity violation detectors and other bug detectors. Data race detectors can typically be of three different types based on the algorithms that are used. The first type relies on the lockset algorithm [58] to check whether the software developer protected all accesses to a specific shared variable with a common lock. The second type relies on the happens-before algorithm [59, 60] and the third type relies on sampling and the use of breakpoints [61] instead of relying on any of these algorithms. Typically, race detectors operate at the lower-level of individual memory accesses. However, Artho et al. [62] investigate data races on a higher abstraction layer. The authors developed a runtime analysis algorithm to detect high-level data races. They introduce a concept of view consistency and utilize it to detect high-level data races. A view is the entire set of shared variables accessed in a synchronized block. According to the

authors, by their algorithms it is possible to detect inconsistent uses of shared variables, even if no classical race condition occurs.

Xu et al. [63] propose a serializability violation detector to detect erroneous executions of shared-memory programs without requiring a priori program annotations. Their tool can report some dynamic false positives, which makes it particularly suitable to be used in avoiding erroneous executions caused by unknown bugs. The authors validate their proposed method by conducting an empirical case study and claim that the experimental results show that the method is effective on real server programs.

Lu et al. propose a tool that detects atomicity violation at the level of individual memory accesses (low-level) [64]. It relies on training and can detect atomicity violation bugs by learning from a large set of runs of valid memory access patterns.

A bug detector tool is proposed by Huang et al. [65]. Their tool relies on detecting whether critical sections are commutative. The authors achieve this by identifying pairs of critical sections that non-deterministically change the contents of shared memory due to execution order.

Other researchers have addressed the problem of detecting concurrency bugs in different types of event-based frameworks [66, 67, 68]. In our study we present and classify relevant papers that propose concurrency debugging tool(s).

3.3 Case Studies of Concurrency Bugs

There are some case studies on real-world concurrency bugs such as propagation [69], [70] and even prediction [71], [72], [73] of bugs in source code. Some of these case studies consider the components or source code files that are most prone to errors in order to understand the software reliability. This thesis is focused on a specific class of bugs i.e., concurrency bugs and different classes of concurrency bugs. In most of the previous work, the authors analyzed the consequences of bugs and did not distinguish between concurrency and non-concurrency bugs.

Chandra and Chen [74] investigated the reported bugs from three open source software database, i.e., MySQL database, Apache web server and Gnome. They analyzed all bugs with focus on the effectiveness of generic recovery techniques in tolerating the bugs. They found 12 concurrency bugs in

their study. The concurrency bug type was one of the possible types of bug in their study while the scope of our study is more narrow (i.e., on concurrency bugs only). However, we provide a broader analysis taking into consideration several characteristics of concurrency bugs.

Real concurrency bugs were investigated in [75]. Lu et al. analyzed 105 concurrency bugs reported in four open-source applications, i.e., MySQL, Apache, Mozilla and OpenOffice. They studied the causes of concurrency bugs with focus of determining whether they caused deadlocks or not. We use a similar study methodology to find relevant bug reports for our analysis but we provide a complementary angle by studying the effects of recent concurrency bugs with a more fine-grained classification than mapping bugs into deadlock and not-deadlock bug classes.

Schimmel et al. [76] present an empirical evaluation of bug detection capabilities of two data race bug detection tools on real-world concurrent software. The authors tracked 25 data races in bug repositories, created parallel unit tests and executed 4 different data race detectors. They conclude that with a combination of all detectors 92% of the contained data races can be found, whereas the best data race detector only finds about 50%.

More recently, Lin et. al. analyzed reported bugs from the Apache web server, Mozilla browser and Linux kernel [77]. They found that the Linux kernel has a higher fraction of concurrency bugs i.e., 13.6%. While the Apache web server has 5.2% and Mozilla browser 1.2% of bugs being of concurrency type. They also recognized that 10.2% of Linux kernel bugs are associated to interrupt handling (missing instructions to enable or disable interrupts at the appropriate locations). The focus of Lin et. al.'s study is mostly on the distribution of concurrency bugs from the three application while we analyzed the concurrency bugs not only to understand the differences between concurrency and non-concurrency bugs distribution and to recognize the most common type of concurrency bugs, but also to recognize the most common type of concurrency bugs in terms of fixing time, severity and reproducibility.

The study by Gu et al. [78] look at the change history for thread synchronization. The authors investigate code repositories of open-source multithreaded software projects to understand synchronization challenges encountered by real-world developers. They reviewed over 250,000 revisions of four representative open source software projects to distinguish how developers handle synchronizations. Further, the authors conduct case studies to better

understand how concurrency bugs are introduced by code changes and how developers handle synchronization problems. Gu et al. conclude that it is necessary to have tool support to help developers who tackle synchronization problems.

3.4 Runtime Verification Tools for Concurrency Bugs

Although many frameworks have been proposed for runtime monitoring as explained in Chapter 2, just a few runtime verification tools are available for use with most of them focusing on Java programs. For instance, Java PathExplorer (JPAX) is a runtime verification tool proposed by Havelund and Rosu [79, 80] for monitoring the execution of sequential and concurrent Java programs. The prototype of Java PathExplorer has been applied to the executive module of the NASA Ames planetary Rover K9 [79]. The general concept of the tool concerns extracting events while the program is executing and then analysing these events with a remote observer process. JPAX instruments Java byte code to send a set of relevant events to the observation module that performs two kinds of verifications: 1) logic-based monitoring and 2) error pattern analysis. Logic-based monitoring is a kind of specification based monitoring which counts upon an underlying logic and the user can express any application dependent, logical requirements. Error pattern analysis implements more or less standard programming language dependent algorithms, e.g., exploring execution trace to detect potential concurrency errors, including deadlocks and data races, even they do not explicitly occur in the trace.

Falcon is another tool for on-line monitoring and steering of large-scale parallel programs [81]. Falcon's architecture has a monitoring component which consists of a high-level view specification and a low level sensor specification. Programmers define application-specific sensors for capturing the program behavior and runtime attributes. Falcon has another component that allow users to implement on-line display system to graphically display data structures, runtime program behaviors, and performance information. Falcon is designed for distributed systems and its implementation relies on the C threads library on several hardware platforms.

Java with Assertions (Jass) is a monitoring approach developed for sequen-

tial and concurrent systems written in Java [82]. Jass translates annotations to programs written in Java into pure Java code. Compliance with the specified annotations is dynamically tested during runtime. It checks specification violations dynamically at runtime by adding assertions which provide the specification of the program. Assertions are boolean expressions of Java with certain keywords and quantifications over finite sets. They are in the form of class invariants, loop invariants, method post and pre-conditions and additional checks which can be inserted into every part of the code. Jass is able to detect possible interferences in a parallel program by having the thread in Jass classes which start in the main method. Jass is able to detect when an assertion in one thread becomes invalid through statements in another thread.

As briefly surveyed in this section, there are a few runtime verification tools available, but none is a runtime verification tool for embedded software to detect concurrency bugs. For instance, JPAX is a runtime verification tool for monitoring and detecting potential concurrency errors in Java programs. From our understanding, it cannot detect these concurrency bugs for embedded software running under FreeRTOS. In addition, the proposed tool cannot detect other types of concurrency bugs such as *Starvation* and *Suspension* bugs. Similarly, Jass is a monitoring approach considering Java applications and is not able to detect if the interferences are protected by synchronization methods [82]. Our tool does not have this limitation. Moreover, the other tool (Falcon) is adapted to C and based on static analysis while our tool is based on dynamic analysis.

Chapter 4

Research Results

This chapter presents the results of our research in relation to the respective following research goals:

Subgoal 1: To provide a common terminology for distinguishing between different types and classes of concurrency bugs and to identify the interrelation between separate classes.

Subgoal 2: To identify the current gaps and less-explored areas in debugging of concurrency bugs.

Subgoal 3: To identify the current state of concurrency related bugs in real-world software in terms of frequency, severity, resolving time and reproducibility.

Subgoal 4: To propose a model and implement a tool for monitoring and detecting concurrency bugs.

Subgoal 5: To evaluate the implemented tool in real-world concurrent software.

4.1 Research Results Related to Subgoal 1

In order to achieve the first subgoal of this thesis (*To provide a common terminology for distinguishing between different types and classes of concur-*

rency bugs and to identify the interrelation between separate classes.), in Paper A [11], we propose a classification for concurrency bugs. We classify the bugs in a common structure in which each bug-type is characterized by a unique set of observable properties.

We first gathered the common system states and symptoms (observable properties) of bugs based on a literature review. We divide the observable properties in properties related to the system state, and properties related to the symptoms of the concurrent program under test. The resulting classification is shown in Table 4.1. In the table, when we refer to a thread t, we are referring to threads in the set $T_b \subseteq T$, where among all threads T, 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 among all resources R, R_b is the set of resources directly involved in the bug. 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 "✓" in the column of bug B and the row of property p would mean that if you have come across bug B, then property p will invariably hold. Note that the reverse implication (i.e., $property \rightarrow Bug$) does not necessarily hold.

Table 4.1: Concurrent software bugs classes and the properties for each class (from Paper A).

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

In order to avoid omission of relevant bugs, we conducted a literature review to identify faults, errors and bugs relevant to parallel, concurrent and multicore software testing and debugging. The common properties of bugs presented above are primarily extracted from relevant references identified in the literature review.

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 [20]. It occurs when concurrent threads perform conflicting accesses by trying to update the same memory location or shared variable [17, 21].
  - **Memory inconsistency** is when different threads have inconsistent views of shared variables [19]. 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.
  - **Write-Write race** is a data corruption caused by accessing a shared variable via at least two threads, in which one of them overwrites the data before any reads.
- **Deadlock** is "a condition in a system where a process cannot proceed because it needs to obtain a resource held by another process but is itself holding a resource that the other process needs" [83]. More generally, it occurs when two or more threads attempts to access shared resources held by other threads, and none of the threads can give them up [17, 23].
- **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" [84].
- **Starvation** is "a condition in which a process is indefinitely delayed because other processes are always given preference" [85]. Starvation typically occurs when high priority threads are monopolising the CPU resources.

- 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 [86].

- **Order violation** is defined as the violation of the desired order between at least two memory accesses [87]. It occurs when the expected order of interleavings does not appear [5]. If a program fails to enforce the programmer's intended order of execution then an order violation bug could happen [75].

- **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 inconsistent 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, which yields different result from the state of sequential memory accesses [88].
  - **Multi-variable atomicity violation** occurs when multiple variables are involved in an unserializable interleaving pattern [88].

4.2 Research Results Related to Subgoal 2

In order to achieve the the second subgoal of this thesis (*To identify the current gaps and less-explored areas in debugging of concurrency bugs.*), we present in Paper B [12] the results of a systematic mapping study in the field of concurrent software debugging in the period 2005–2014.

In terms of publication trends on debugging of concurrent software from 2005 to 2014, we found that the topic has gained increasing interest, with the highest number of published papers in 2013. Our investigation indicates that the number of publications in the field increase from 4 in 2005 to 24 in 2013.

In order to investigate the current gaps in debugging concurrency bugs we explored the addressed concurrency bugs, different type of debugging processes, types of research and research contributions.

Regarding concurrency bugs, we found that six specific types of concurrency bugs (viz., Deadlock, Livelock, Starvation, Data race, Order violation, and Atomicity violation) were addressed by articles from 2005 to 2014.

Figure 4.1, presents the identified research gaps related to concurrency bugs and concurrent software debugging. It is evident from the figure that *bug identification* is the most widely studied process with 92 papers (63%) across different types of bugs. Among this amount, about 45% of papers focus on *data race* while no paper was about *suspension* and *livelock*. Moreover, very few papers focus on *starvation* and *order violation* bugs (3%). More details are presented in Chapter 7.

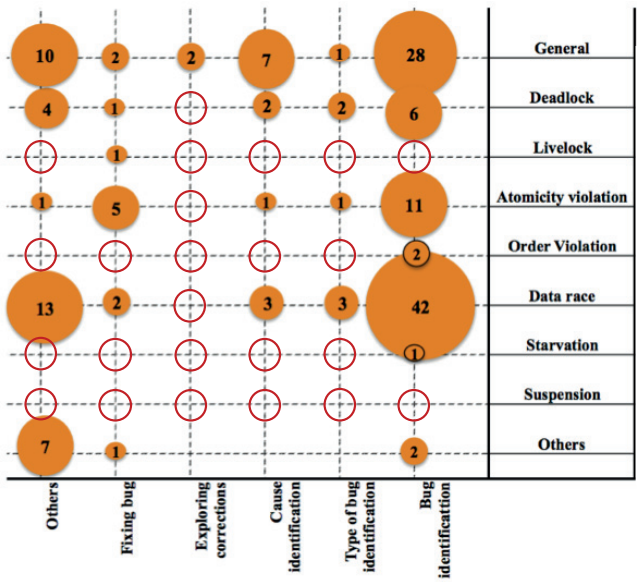


Figure 4.1: Identified research gaps related to concurrent software based on *types of concurrency bugs* and *types of debugging processes*

.

4.3 Research Results Related to Subgoal 3

In order to achieve the third subgoal of this thesis (*To identify the current state of concurrency related bugs in real-world software in terms of frequency, severity, resolving time and reproducibility.*), we provide a comprehensive study of 11860 fixed bug reports from five widely used open source storage designed for big-data applications (viz., the Apache Hadoop project, the Apache ZooKeeper project, the Apache Oozie project, the Apache Accumulo project and the Apache Spark project) in Paper C [13]. The study covers the reports of fixed bugs from 2006 to 2015.

Our comparative study of concurrency bugs and non-concurrency bugs revealed that only 4% of the total set of bugs are related to concurrency issues, while the majority of bugs (i.e., 96%) are of non-concurrency type. The distribution of non-concurrency and concurrency bug types is shown in Figure 4.2. This Venn chart also illustrates the obtained results of our investigation in terms of reproducibility[1] from the all five projects' repository.

In Figure 4.2, a bug which reported during 2006 to 2015 is categorized as *All*, a fixed and closed bug is categorized as *Fixed & Closed*. If a report is tagged as "Cannot reproduce" then it is categorized as *unreproducible*. A bug with at least one keyword related to concurrency issues is categorized as *Concurrency keywords matched*, some of these bugs are fixed and closed and others are unreproducible. We are not considering the bugs that are under investigation and the reports related to these are excluded from our study although they are included in the *All* category. Finally, if a bug falls into one of the concurrency classification types then it is categorized as a *Concurrency bug*. This Venn chart illustrates that the fraction of unreproducible bugs from the total set of bugs is only 4%, while 2% of the total set are unreproducible and related to concurrency issues.

We also compared the time required to fix concurrency bugs and non-concurrency bugs. Our results show that concurrency bugs require longer fixing time than non-concurrency bugs, but the difference is not very large. Figure 4.3 shows the results of comparing the fixing time for concurrency and non-concurrency bugs in the form of box-plots. Boxes span from 1st to 3rd quartile, black middle lines are marking the median and the whiskers extend

[1]Bug reproducibility indicates the success in reproduction of software failure(s) caused by bug(s).

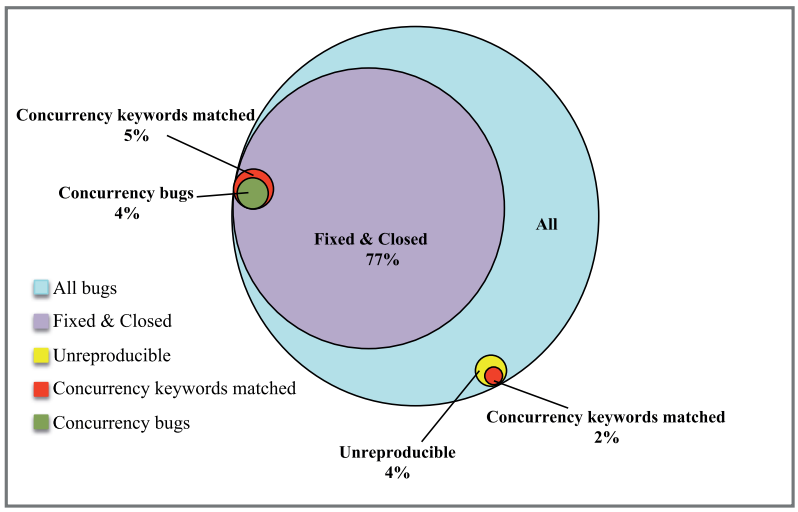


Figure 4.2: Distribution of non-concurrency and concurrency bug types together with distribution of reproducible and unreproducible concurrency bugs (from Paper C).

up to 1.5 times the inter-quartile range while the circles represent the outliers.

Further, our study on severity of concurrency bugs and non-concurrency bugs indicates that concurrency bugs are considered to be more severe than non-concurrency bugs, but the difference is not that large. Figure 4.4 shows the severity distributions.

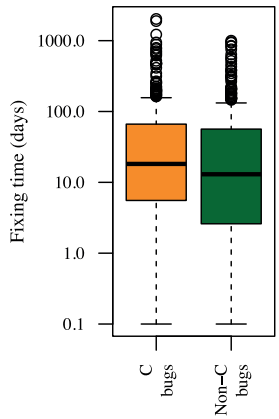


Figure 4.3: Fixing time comparison for concurrency (C) and non-concurrency (Non-C) bugs (from Paper C).

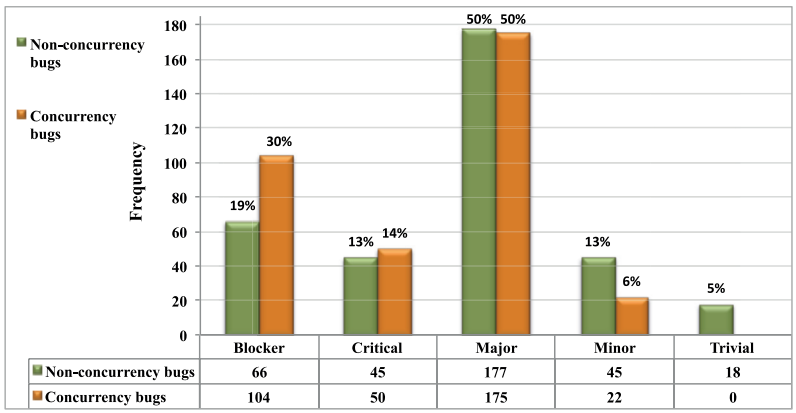


Figure 4.4: Concurrency and non-concurrency bug severity (from Paper C).

4.4 Research Results Related to Subgoal 4

In order to achieve the forth subgoal of this thesis (*To propose a model and implement a tool for monitoring and detecting concurrency bugs.*), we present

a novel method and a tool called DeCoB (Detecting Concurrency Bugs), which use runtime verification to detect concurrency bugs in embedded software in Paper D [14].

The method and tool can cover the less-explored concurrency bugs based on our obtained result in Paper B [12] and Paper C [13]. The logical architecture for detecting concurrency bugs in embedded software is shown in Figur 4.5. It is decomposed into four layers viz., Logging, Monitoring, Concurrency Bug Diagnosis, and Mitigation. The detailed description of the architecture is given in Section 9.3.1.

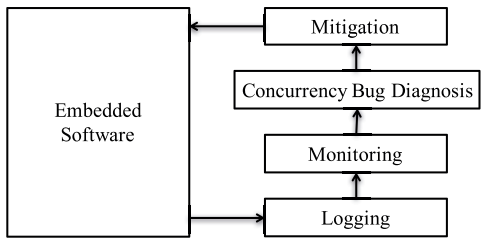


Figure 4.5: Architecture of the runtime verification framework for detecting concurrency bugs in embedded software (from Paper D).

Our implemented tool (DeCoB) is tailored for the open source real-time operating system (FreeRTOS), and detects and diagnoses concurrency bugs, such as deadlock, starvation, and suspension-based-locking, by analysing runtime traces provided by the Tracealyzer tool[2], i.e., without debugging and tracing the source code. Figure 4.6 shows our proposed architecture of the DeCoB tool. The proposed architecture is comprised of five separate modules, viz., *Parser Module*, *Starvation Bug Diagnosis Module*, *Deadlock Bug Diagnosis Module*, *Suspension Bug Diagnosis Module*, and *Data Visualization Module*. The detailed description of the architecture is given in Section 9.3.2.

[2]Tracealyzer is a stand-alone application for visualizing and tracing embedded software executions and is developed by Percepio AB since 2004 [89].

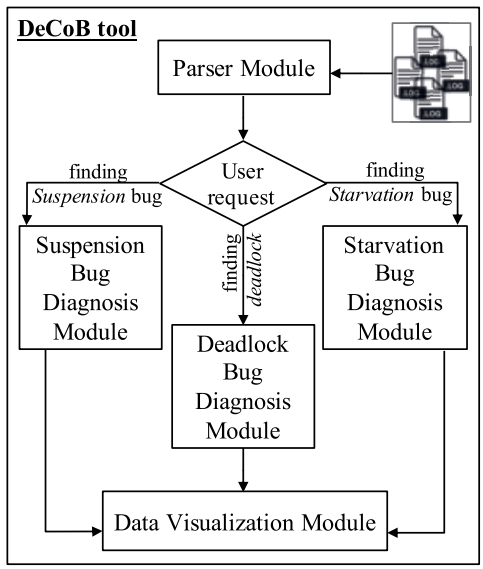


Figure 4.6: The architecture of the DeCoB tool (from Paper D).

4.5 Research Results Related to Subgoal 5

The method and tool presented in Paper D [14] is also designed to achieve the fifth subgoal of this thesis (*To evaluate the implemented tool in real-world concurrent software.*). Thus, an experimental evaluation of the DeCoB tool is designed and applied in Paper D [14]. Figure 4.7 illustrates an overview of the experimental evaluation. The evaluation includes two distinct approaches: (1) a proof-of-concept evaluation using realistic FreeRTOS logs and (2) a systematic evaluation using automatically generated logs by the UPPAAL model checker and our Trace generator.

We verified our implementation and performed an evaluation on software executing on an ARM Cortex-M-based micro-controller according to the approach to the left in Figure 4.7. We evaluated the tool by injecting three predefined types of concurrency bugs in Atmel Studio and used real log files collected during system execution using the Tracealyzer tool. Then we used the log file as input to the DeCoB tool in order to detect the injected bugs. The im-

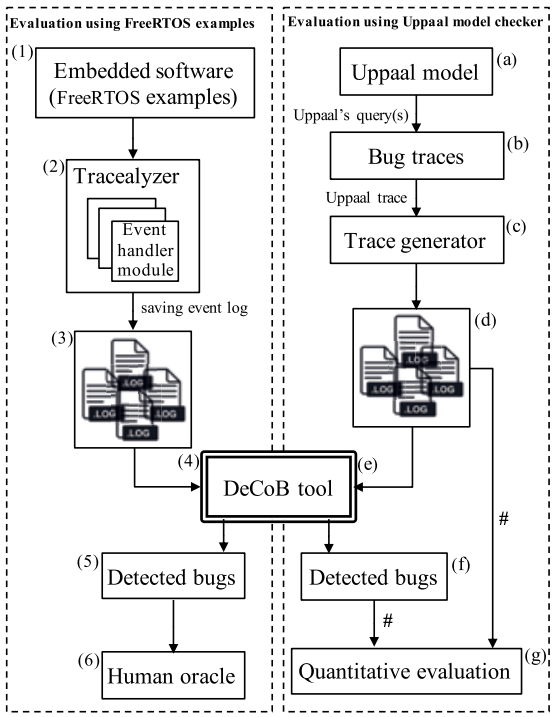


Figure 4.7: Overview of the experimental evaluation of the DeCoB tool (from Paper D).

plemented tool was able to detect all the injected bugs. In the second approach to the right in Figure 4.7, we evaluated the implemented tool using 21726 synthetic logs generated using the UPPAAL model checker and our Trace generator. As a result, we could show that DeCoB was capable of detecting all concurrency bugs in the 21726 created traces, which shows that DeCoB is effective at detecting concurrency bugs from a diverse set of logs. Detailed description of the evaluation and the results are given in Section 9.4, 9.5 and 9.6.

Chapter 5

Discussion, Conclusion and Future Work

In this chapter, we present a discussion based on our results, a list of conclusions, as well as a set of potential directions for future work.

5.1 Discussion and Threats to Validity

According to our literature review, we found that existing taxonomies for concurrent and multi-threaded software debugging properties are lacking coverage of some aspects, specifically the ones related to the debugging process. The existing knowledge gaps in different types of bugs may be due to the fact that some specific types of bugs are not well-known yet, or recognizing them is not easy. Another reason for these gaps could be related to the debugging processes which are not well defined and not applicable in all software development projects, thus they are not easy to apply.

On the other hand, according to our case study results, the distribution of concurrency bugs reported in the bug repositories is not large in comparison to non-concurrency bugs. This is not very surprising, since it has long been believed that concurrency bugs are hard to detect and reproduce. There are three main possible reasons for this belief: (1) when users are faced with the bug a single time they may not even be sure that it is a problem with the soft-

ware and might not report it; (2) it might not be possible to reproduce the bug in the developer's environment due to small differences in the environments even when users are able to reproduce the bug on their machines; (3) software developers might not be able to systematically reproduce the bug using traditional debugging methods since some debugging tools and methods might affect the reproducibility of the bug. In our study, we found a much smaller share of concurrency bugs than the one found by other similar studies. This could possibly be due to one of the three mentioned reasons or due to a different time span of our study and that of other similar studies. Based on our investigation, about half of the concurrency bugs are of *Data race* type. Our investigation also shows that 48% of the bugs that we observed were reported in the five-year interval of 2006-2010, and the remaining 52% were reported in the five-year interval of 2011-2015. Possible reasons could be that either the current approaches and tools still need progress in detecting and fixing bugs, or the software developers and testers are not aware of the proposed methods and the implemented tools.

Due to increasing software system complexity, there is renewed interest in implementing tools for detecting faults and managing recovery from them at runtime. Concurrent programming also increases the complexity of different types of software. Automating concurrency bug detection typically provides an overview of the concurrency bugs properties which can lead to simplified fix and reproduction of the concurrency bugs. Improvements we are seeking are easier, faster and more reliable discovery of concurrency bugs during software execution. Using our proposed runtime tool (DeCoB) can give an opportunity for the test managers, testers and developers to detect the concurrency bugs (even in cases when the bugs do not lead to an observable failure). Our two-fold evaluation demonstrate that DeCoB is able to successfully detect bugs in the examined logs. In total, DeCoB is able to correctly detect whether the 21726 automatically generated logs containing concurrency bugs.

In the design and execution of this thesis, there are several issues that need to be considered as they can potentially limit the validity of the obtained results.

We limited the search for studies and bugs in the systematic study and the case study within the time span of 2005-2014 and 2006-2015, respectively. This was done for two reasons: (1) to limit the volume of search results for practical reasons; (2) to present more *recent* trends (i.e., in the last decade). This limitation of years obviously excludes papers published before the year

2005 and excludes bug reported before the year 2006, including highly cited papers and important bugs. Thus, our systematic mapping study and our case study are not complete with respect to all research papers and reported bugs on the topic, but instead presents the more recent development in the field.

Another threat is related to the classification schema for mapping included papers in our systematic mapping study and included bug reports in our case study. Since authors and bug reporters cannot be expected to follow any standard concurrency bug terminology, partially based on our proposed classification, we categorized the papers and bug reports. We believe that the process of classification would have been more reliable if consistent terminologies would have been used in the primary studies and bug reports. However, some papers and bug reports were difficult to categorize due to unclear boundaries between some classification scheme categories.

As stated before, most of the runtime verification tools for concurrency bugs detection focusing on Java programs. The body of knowledge in runtime verification tools are for embedded software to detect concurrency bugs is limited. The implemented DeCoB tool is able to detect the concurrency bugs for embedded system. We have selected FreeRTOS as the target environment for DeCoB since it is a widely used open source operating system that offers support for different hardware architectures in the embedded system domain.

5.2 Conclusions

We propose a taxonomy of different types of concurrency bugs by classifying the bugs based on their observable properties. The grouping and classification of concurrency bugs presented is structured based on properties that are commonly observable in concurrent systems. The aim of the proposed taxonomy is to aid software developers during the debugging and testing of their concurrent applications. The taxonomy also helps users to make appropriate decisions when they encounter problems.

In addition, we provide an overview of existing research on concurrent and multi-threaded software debugging. We pinpoint current gaps in the research area that may represent opportunities for further research on debugging concurrent and multi-threaded software.

In particular, we provide a case study on concurrency bugs. This study analyzed bugs reported from a widely used open source storage designed for

big-data applications and classified the bugs into two classes of bugs: non-concurrency and concurrency bugs. The case study also helped us to recognize the severity, fixing time and reproducibility of the most common types of concurrency bugs in terms of . The findings from our case study could help software designers and developers to understand how to address concurrency bugs, estimate the most time-consuming ones, and prioritize them to speed up the debugging and bug-fixing processes.

Apart from our theoretical and experimental outcomes, we propose a runtime verification method and implement a tool (DeCoB) based on the method in order to automate the concurrency bug detection process. DeCoB can be utilized as a supportive tool for making decisions on finding, localizing and fixing concurrency bugs.

In general, despite all the mentioned limitations, this thesis improves our understanding of the characteristics of the different types of concurrency bugs, the less-explored areas in debugging concurrency bugs and the current state of concurrency related bugs in real-world software. Besides, it introduced an effective concurrency bug detector and a concurrent software runtime verification technique applicable on real-world scenarios.

5.3 Future Work

This thesis raises a number of questions, which we strongly believe can form the basis for future work, as outlined below.

As stated before, the DeCoB tool supports detecting deadlock, starvation and suspension type of bugs. An interesting agenda for future work would be to expand the DeCoB tool and the proposed method behind it for detecting other types of concurrency bugs during runtime (e.g., Data race, Atomicity violation and Order violation).

Moreover, our classification is focusing on shared memory concurrency. There are additional types of concurrency bugs that are specific for message passing systems (e.g., message races). Considering these type of concurrency bugs could be another direction for future work.

Nevertheless, the case study in Chapter 4 provides basis for many research directions, one noticeable such direction is to apply other case studies with other projects (e.g., implemented in other programming languages) in order to generalize the results to other projects. Besides, we argue that having access to

real industrial data is important for the advancement of detecting concurrency bugs and more industrial case studies targeting concurrency bugs are needed to generalize the results of this thesis to other systems and to increase the body of knowledge in general.

Bibliography

[1] David A. Weiser. *Hybrid Analysis of Multi-threaded Java Programs*. ProQuest, 2007.

[2] Jayant Desouza, Bob Kuhn, Bronis R. De Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 78–82. ACM, 2005.

[3] Patrice Godefroid and Nachiappan Nagappan. Concurrency at Microsoft: An exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[4] Michael Süßand Claudia Leopold. Common mistakes in OpenMP and how to avoid them. In *OpenMP Shared Memory Parallel Programming*, pages 312–323. Springer, 2008.

[5] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 245–254. ACM, 2010.

[6] Joab Jackson. Nasdaq's Facebook Glitch Came From Race Conditions, May 2012. preprint (2011), available at `http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html`.

[7] Jeffrey JP Tsai and Kuang Xu. A comparative study of formal verification techniques for software architecture specifications. *Annals of Software Engineering*, 10(1-4):207–223, 2000.

[8] V Altukhov, V Podymov, V Zakharov, and E Chemeritskiy. Vermont-a toolset for checking sdn packet forwarding policies on-line. In *Science and Technology Conference (Modern Networking Technologies)(MoNeTeC), 2014 First International*, pages 1–6. IEEE, 2014.

[9] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[10] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.

[11] Sara Abbaspour A, Hans Hansson, Daniel Sundmark, and Sigrid Eldh. Towards Classification of Concurrency Bugs Based on Observable Properties. In *Workshop on Complex faUlts and Failures in LargE Software Systems (COUFLESS)*, 2015.

[12] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, and Wasif Afzal. 10 years of research on debugging concurrent and multicore software: a systematic mapping study. *Software Quality Journal*, pages 1–34, 2016.

[13] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. Concurrency bugs in open source software: a case study. *Journal of Internet Services and Applications*, 8(1):4, 2017.

[14] Sara Abbaspour Asadollah, Eduard Paul Enoiu, Adnan Čaušević, Daniel Sundmark, and Hans Hansson. A runtime verification based concurrency bug detector for freertos embedded software. *Is submitted to a journal*, 2018.

[15] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, and Eduard Paul Enoiu. A study on concurrency bugs in an open source software. In IFIP, editor, *12th International Conference on Open Source Systems*, June 2016.

[16] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. A runtime verification tool for detecting concurrency bugs in freertos embedded software. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 172–179. IEEE, 2018.

[17] K. Henningsson and C. Wohlin. Assuring fault classification agreement - an empirical evaluation. In *2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04. Proceedings*, pages 95–104, August 2004.

[18] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145. ACM, 2008.

[19] Leon Li Wu and Gail E. Kaiser. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators. Technical report, Columbia University, 2011.

[20] Noriaki Yoshiura and Wei Wei. Static data race detection for java programs with dynamic class loading. In *Internet and Distributed Computing Systems*, pages 161–173. Springer, 2014.

[21] Shameen Akhter and Jason Roberts. *Multi-core programming*, volume 33. Intel press Hillsboro, 2006.

[22] R.W. Brown. Method and apparatus for processing requests for video presentations of interactive applications in which vod functionality is provided during nvod presentations, June 23 1998. US Patent 5,771,435.

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

[24] Juan Gonzalez, David Insa, and Josep Silva. A new hybrid debugging architecture for eclipse. In *Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation(LOPSTR)*, pages 183–201. Springer International Publishing, 2014.

[25] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 284–295. IEEE Computer Society, 2005.

[26] T. J. Leblanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[27] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[28] Wenwen Wang, Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Xipeng Shen, Xiang Yuan, Jianjun Li, Xiaobing Feng, and Yong Guan. Localization of concurrency bugs using shared memory access pairs. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 611–622. ACM, 2014.

[29] Ghazia Zaineb and Irfan Anjum Manarvi. Identification And Analysis Of Causes For Software Bug Rejection With Their Impact Over Testing Efficiency. *International Journal of Software Engineering & Applications (IJSEA)*, 2(4), 2011.

[30] Klaus Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):143–170, 2015.

[31] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Roşu, and Willem Visser. Experiments with test case generation and runtime analysis. pages 87–108. Springer Berlin Heidelberg, 2003.

[32] Feng Chen and Grigore Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2):108–127, 2003.

[33] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 44–57. Springer, 2004.

[34] D. Drusinsky. *Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*. Elsevier Science, 2011.

[35] Klaus Havelund. Havelund verification of c programs. pages 7–22. Springer Berlin Heidelberg, 2008.

[36] Christian Colombo, Gordon J Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 135–149. Springer, 2008.

[37] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. *Departmental Papers (CIS)*, page 294, 1999.

[38] David Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In *International Conference on Computer Aided Verification*, pages 1–18. Springer, 2010.

[39] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In *International Conference on Runtime Verification*, pages 59–75. Springer, 2013.

[40] Volker Stolz and Frank Huch. Runtime verification of concurrent haskell programs. *Electronic Notes in Theoretical Computer Science*, 113:201–216, 2005.

[41] S. Halle and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Transactions on Services Computing*, 5(2):192–206, April 2012.

[42] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 345–364. ACM, 2005.

[43] Maria Brito, Katia R. Felizardo, Paulo Souza, and Simone Souza. Concurrent Software Testing: A Systematic Review? *on Testing Software and Systems: Short Papers*, page 79, 2010.

[44] Peng Wang, Xiaofang Qi, Xiaoyu Zhou, and Xiang Zhang. Multithread Deterministic Replay Debugging: The State of The Art. *International Journal of Advancements in Computing Technology*, 4(23), 2012.

[45] Shin Hong and Moonzoo Kim. A survey of race bug detection techniques for multithreaded programmes. *Software Testing, Verification and Reliability*, 25(3):191–217, 2015.

[46] B. Long and P. Strooper. A classification of concurrency failures in java components. In *Proceedings of the 13th International Conference on Parallel and Distributed Processing Symposium*, pages 8–pp. IEEE, April 2003.

[47] G.M. Tchamgoue, O.-K. Ha, K.-H. Kim, and Y.-K. Jun. A taxonomy of concurrency bugs in event-driven programs. In *Communications in Computer and Information Science*, volume 257 CCIS, pages 437–450, 2011.

[48] D.P. Helmbold and C.E. McDowell. A taxonomy of race conditions. *J. Parallel Distrib. Comput.*, 33(2):159–164, March 1996.

[49] Jan Lönnberg, Lauri Malmi, and Anders Berglund. Helping Students Debug Concurrent Programs. In *Proceedings of the 8th International Conference on Computing Education Research*, Koli '08, pages 76–79, New York, NY, USA, 2008. ACM.

[50] Caitlin Sadowski and Jaeheon Yi. User Evaluation of Correctness Conditions: A Case Study of Cooperability. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 2:1–2:6, New York, NY, USA, 2010. ACM.

[51] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software. *Microsoft Research*, 38:39, 2007.

[52] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[53] Gowritharan Maheswara, Jeremy S. Bradbury, and Christopher Collins. Tie: An interactive visualization of thread interleavings. In *Proceedings of the 5th international symposium on Software visualization*, pages 215–216. ACM, 2010.

[54] Steven P. Reiss and Manos Renieris. Demonstration of JIVE and JOVE: Java as it happens. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 662–663. IEEE, 2005.

[55] Steven P. Reiss and Suman Karumuri. Visualizing threads, transactions and tasks. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–16. ACM, 2010.

[56] Sangmin Park, Richard Vuduc, and Mary Jean Harrold. UNICORN: a unified approach for localizing non-deadlock concurrency bugs. *Software Testing, Verification and Reliability*, 25(3):167–190, 2015.

[57] Sangmin Park, Mary Jean Harrold, and Richard Vuduc. Griffin: grouping suspicious memory-access patterns to improve understanding of concurrency bugs. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 134–144. ACM, 2013.

[58] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[59] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[60] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of*

the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, pages 134–143, New York, NY, USA, 2009. ACM.

[61] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *OSDI*, volume 10, pages 1–16, 2010.

[62] Armin Biere Cyrille Artho, Klaus Havelund. High-level data races. In *journal on software testing, verification and reliability (STVR)*, pages 1–12, 2003.

[63] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 1–14, 2005.

[64] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48. ACM, 2006.

[65] Ruirui Huang, Erik Halberg, and G. Edward Suh. Non-race concurrency bug detection through order-sensitive critical sections. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 655–666, New York, NY, USA, 2013. ACM.

[66] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336. ACM, 2014.

[67] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262. ACM, 2012.

[68] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages; Applications*, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.

[69] Lucian Voinea and Alexandru Telea. How do changes in buggy mozilla files propagate? In *Proceedings of the 2006 ACM symposium on Software visualization*, pages 147–148. ACM, 2006.

[70] F PanW, MaYT LiB, et al. Measuring structural quality of object-oriented softwares via bug propagation analysis on weighted software networks. *JournalofComputerScience andTechnology*, 25(6):1202–1213, 2010.

[71] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.

[72] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 372–381. IEEE Press, 2013.

[73] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 424–434. ACM, 2014.

[74] Subhachandra Chandra and Peter M Chen. Whither generic recovery from application faults? a fault study using open-source software. In *Proceedings International Conference on Dependable Systems and Networks*, pages 97–106. IEEE, 2000.

[75] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

[76] J. Schimmel, K. Molitorisz, and W.F. Tichy. An evaluation of data race detectors using bug repositories. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pages 1361–1364, Sept 2013.

[77] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[78] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. What change history tells us about thread synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 426–438. ACM, 2015.

[79] Klaus Havelund and Grigore Rosu. Java pathexplorer-a runtime verification tool. 2001.

[80] Klaus Havelund and Grigore Roşu. Monitoring java programs with java pathexplorer. *Electronic Notes in Theoretical Computer Science*, 55(2):200–217, 2001.

[81] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers' 95., Fifth Symposium on the*, pages 422–429. IEEE, 1995.

[82] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass: Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

[83] Yogesh Bhatia and Sanjeev Verma. Deadlocks in distributed systems. *International Journal of Research*, 1(9):1249–1252, 2014.

[84] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[85] William Stallings. *Operating Systems- internals and design principles*, volume 7th. Prentice Hall Englewood Cliffs, 2012.

[86] Shiyao Lin, Andy Wellings, and Alan Burns. Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages. *Concurrency and Computation: Practice and Experience*, 25(16):2227–2251, 2013.

[87] Deepal Jayasinghe and Pengcheng Xiong. CORE: Visualization tool for fault localization in concurrent programs. 2010.

[88] Sangmin Park, Richard Vuduc, and Mary Jean Harrold. A unified approach for localizing non-deadlock concurrency bugs. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 51–60. IEEE, 2012.

[89] Tracealyzer for freertos. `https://percepio.com/tz/freertostrace/`. Accessed: 2018-09-18.