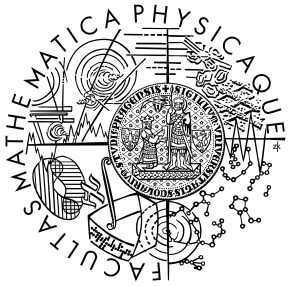

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

# HABILITATION THESIS

**Pavel Parízek**

# Finding Concurrency Errors in Software Systems Efficiently

*Computer Science, Software Engineering*

Prague, Czech Republic 2019

# Contents

# Preface

This habilitation thesis presents results of research in the fields of automated verification and systematic testing of software systems that involve multiple concurrent threads. All the research was performed within the context of program verification and testing approaches based on state space traversal. The specific results include (i) algorithms that improve performance and scalability of verification through more precise partial order reduction and (ii) techniques that enable fast search for concurrency errors by the usage of heuristics and randomization. Higher precision of partial order reduction was achieved by the combination of static analysis with dynamic analysis and state space traversal.

I would like to emphasize that all the research, whose results are presented in this thesis, has been always carried out as a team effort, with the help of my colleagues and collaborators. My special thanks go especially to František Plášil, my doctoral advisor at Charles University, and Ondřej Lhoták who was my supervisor during the post-doctoral stay at the University of Waterloo in Canada. František introduced me to the academic environment and to the world of scientific research. Ondřej guided me during the transition from a doctoral student into a researcher with an independent program, he was very supportive and taught me lot of things. My thanks also go to my other co-authors, students and colleagues with whom I discussed research questions, academic life, and other related topics. Namely: Ondřej Šerý, Tomáš Poch, Jan Kofroň, Pavel Jančík, Tomáš Kalibera, Michal Malohlava, Jakub Daniel, Vlastimil Dort, Martin Blicha, Petr Tůma, Tomáš Bureš, Petr Hnětynka, Tomáš Pop, Jiří Adámek, Pavel Ježek, Jaroslav Keznikl, Petra Novotná, Václav Pech, Corina Pasareanu, and Jan Vitek.

Our research included in the thesis was supported by many funding agencies, both national and international: Czech Science Foundation (projects 201/08/0266, 13-12121P, 14-11384S, and 18-17403S), Charles University institutional funding (project SVV-2016-260331), Grant Agency of the Charles University (project 203-10/253297), Ministry of Education of the Czech Republic (grant MSM0021620838), EU project ASCENS (257414), and Natural Sciences and Engineering Research Council of Canada (NSERC).

Pavel Parízek

# Chapter 1

# Introduction

Modern software systems often execute multiple threads concurrently in order to achieve better overall performance, increase responsiveness of the whole system to user actions, and to utilize multicore CPUs together with graphical processing units (GPUs) that are now widely available. Although concurrency has these benefits, the design and implementation of systems that involve multiple threads of execution and process asynchronous events is known to be a very complicated task mainly for the reasons given below. Developers usually find it difficult to reason about the behavior of concurrent systems and possible mutual interactions of threads. Concurrency errors, such as race conditions and deadlocks, are often caused by subtle interactions between threads and they appear rarely during the program execution — just in few of all the possible thread schedules. Specifically, the concurrency errors are hard to discover, reproduce, and fix properly. It is nevertheless imperative to reduce the number of such errors (and the related source code bugs) that are not fixed before deployment, since their cost may be very high especially in critical systems, including, for example, software components of transportation machinery and healthcare devices. A very high level of safety and reliability is required for such software components. Like manual inspection of source code and reasoning about program behavior, classic testing is not really applicable for this purpose, because it has limited coverage and thread scheduling is hard to control during execution of tests. Techniques and tools supporting automated verification and debugging of multithreaded programs are therefore needed in order to help developers create more reliable software.

A positive story is that lot of work has been done in this field over the last twenty years, and some great advances were achieved. While quite a large fraction of published results still has a mostly theoretical nature, many techniques and tools produced by researchers have been applied to realistic software systems and evaluated on them. Software companies even start to use methods of program verification in order to find concurrency bugs in the source code of their products — this refers both to organizations that develop safety-critical software (e.g., Airbus and NASA) and to major vendors of off-the-shelf software such as Microsoft and Google. Notable tools include Java Pathfinder[1] and CHESS[2]. We provide a brief overview of the current state-of-the-art in the next chapter.

[1] https://github.com/javapathfinder/jpf-core/

[2] https://www.microsoft.com/en-us/research/project/chess-find-and-reproduce-heisenbugs-in-concurrent-programs/

However, despite the recent progress, usage of verification tools by software developers in their day-to-day practice is still limited because the respective techniques do not scale to large systems. The primary cause is that, for any non-trivial system, the number of possible thread schedules that have to be analyzed is really huge. Verification of such systems might run for days (weeks or months), if it finishes at all. While some techniques can be successfully applied to large systems, then they have other important limitations — for example, they report many false warnings or miss some errors due to imprecise approximation of program behavior. The approximation is a consequence of abstraction performed in order to achieve better scalability. In the case of verification tools that use abstraction, developers often have to manually inspect each warning and decide whether the corresponding error can truly occur during program execution in a given setting (and therefore is worth fixing) or it does not. Overall, the main challenges in automated verification and search for concurrency errors in multithreaded systems include: scalability, performance, soundness, and precision.

The goal of our research is to address some of the challenges and limitations described above. In particular, we focus on the construction of automated methods that improve scalability of verification and performance of error detection, and help developers to fix the corresponding bugs in the program code. The primary target domain for methods that we develop consists of software written in mainstream object-oriented programming languages. We have implemented all our methods within already existing third-party verification frameworks (e.g., Java Pathfinder) that directly process the actual code (source or binary) — that means without explicit prior transformation of the concrete program code into a high-level abstract model of some kind. Systematic analysis of abstract models is useful especially in the design phase of a software system, but in our research we focus mainly on checking the implementation in some programming language (such as Java) against low-level properties such as correct thread synchronization (including, e.g., the absence of race conditions and deadlocks).

**Outline of the habilitation thesis.** We begin with the general motivation and context above in this chapter, and then we introduce basic terminology and provide necessary background on the main approaches to program verification in Chapter 2. The main body of this thesis consists of 9 publications in the proceedings of peer-reviewed international conferences and international journals (Chapter 4-12), together with a brief summary of each publication and an overview that connects together results and contribution described in the individual publications (Chapter 3). Each of the chapters 4-12 starts with a full reference to the respective publication that follows in its original preprint form. We conclude and present a brief outlook on the future research directions in Chapter 13.

# Chapter 2

# Background: Automated Verification of Multithreaded Software

In the context of software systems that involve multiple concurrent threads, the goal of both verification and search for bugs is to analyze the observable behavior of a given system under different thread schedules that may occur at runtime. There exist two principal directions: (1) analysis of high-level abstract models of the system behavior and (2) checking the implementation that has the form of actual code written in some concrete programming language. Classic algorithms of model checking [10] are used in the first case. An abstract model is a mathematical structure that captures the possible behavior of a given system. For example, a popular approach is to encode the system behavior using the structure known as Labeled Transition System, where nodes and transitions are annotated by atomic propositions, and to express the desired property as a temporal logic formula. More details can be found in literature [10]. We focus on the second direction in this chapter, since it is much more related to the contribution that we present in the thesis.

Techniques and tools for verification of systems implemented in mainstream programming languages, such as Java and C++, are typically based on the following approaches: systematic traversal of program state space and execution traces [42, 53], constraint solving [34], static data-flow and pointer analysis [19, 43, 44], dynamic analysis [6, 21], and concurrency testing [38, 55, 58]. Each of the approaches has certain benefits and limitations. Systematic traversal of state space is exhaustive and precise, but also very time consuming and it has limited scalability. On the other hand, static analysis scales much better due to approximation that, however, is the cause of imprecision and spurious warnings. Dynamic techniques, as well as concurrency testing, very precisely analyze just selected few thread schedules, and therefore have a small coverage. A user always has to consider the trade-off between performance, precision, scalability and coverage. Nevertheless, while only a single basic approach has been used traditionally in most of the published techniques, there were also some attempts to combine multiple approaches in specific ways [7, 9] in order to take advantage of their complementary strengths and compensate for their respective limitations. We provide more details about each of the main approaches in this chapter. Please note that when using terms like *state*, *execution trace*, and *thread*, we refer to entities at the level of the subject program (its source code and state space), unless explicitly specified otherwise.

## 2.1 Systematic Traversal of Program State Space

A state space of a multithreaded program is encoded as a directed graph where nodes represent states and edges represent transitions in the form of finite sequences of executed statements. Each path in the graph then corresponds to an execution trace, respectively to a particular thread schedule (interleaving of threads). The basic principle of this approach is to create non-deterministic thread scheduling choices at certain program states, and systematically explore all the options at each choice to cover the behavior of a given program under every possible thread schedule. Systematic traversal can be performed in an explicit manner [42, 53] or using symbolic methods [12, 54]. In the first case, the state space traversal procedure represents each relevant program state explicitly and inspects all the execution traces one-by-one. Various techniques built on explicit state space traversal are used especially (1) for systematic testing of multithreaded programs [3] and (2) to detect concurrency errors very fast with the help of optimizations and heuristics that we discuss below. On the other hand, symbolic methods use logic formulas to represent sets of program states, execution traces, and the transition relation. Actual verification is then performed, for example, by constraint solving and satisfiability checking. More details about verification techniques based on constraint solving are provided below.

We already indicated that systematic traversal does not scale well to complex software systems with many concurrent threads because of the very large number of possible thread schedules that have to be fully explored. The main cause is the high number of non-deterministic thread scheduling choices that are created in the program state space during the traversal. In practice, the prevailing approaches used to cope with this challenge are the following three: partial order reduction (POR) [25], bounded search, and directed search with heuristics. All of them are complementary and therefore can be easily combined in order to achieve better performance and scalability.

Partial order reduction exploits the fact that only some actions (statements) represent communication between threads, for example by reading or modifying the global shared state. We call them interfering actions. The set of interfering actions includes, for example, accesses to fields of shared heap objects and thread synchronization. Following upon this, the key idea behind POR is that just all the interleavings of interfering actions (by different threads) have to be explored during the state space traversal in order to cover all observable behaviors of a given system. A common implementation strategy is to create thread scheduling choices only at interfering actions, such that (i) the first option corresponds to the scenario where the current thread continues further by executing the respective interfering action and (ii) other options represent the cases where another runnable thread will execute instead. In this way, all possible sequences of concurrent actions from different threads are covered, even though much less thread choices are made in the state space. The main challenge is to determine precisely (i) which statements may be interfering and (ii) which are provably thread-local. Algorithms of POR differ in how they address this challenge. For example, there is POR strategy based on a static escape analysis [18] that is run in advance, dynamic concrete POR that is performed on-the-fly during the state space traversal [1, 22], and symbolic methods of POR [56]. Dynamic POR is more precise than static POR and symbolic POR, as it recognizes each dynamic heap object, but its performance may be relatively poor for systems with large state space and

long transitions. We provide more details about the benefits and limitations of dynamic POR in Chapter 3 and then in Chapter 5 and 8. Each of the POR strategies mentioned above is, in general, naturally suitable for a different flavor of a state space traversal procedure, along the lines of exhaustive versus symbolic.

Nevertheless, even when state space traversal is used together with POR, it is not practically feasible to explore all thread schedules for large software systems. A popular way of avoiding this problem is to give up on complete verification and focus on the search for errors. The respective techniques aim to find as many concurrency errors as possible in a limited time, using some variant of bounded (incomplete) search, randomization, or directed search with heuristics. For example, the CHESS toolset [42] implements bounding of the number of thread preemptions [41] on each state space path and also preemption sealing [2]. The key idea of preemption bounding is to prune every path suffix where the bound is exceeded, while preemption sealing disables thread scheduling choices in specific parts of the program (e.g., within library methods or previously verified fragments). Heuristics navigate the search towards error states, for example by (i) changing the order in which states and transitions are explored or (ii) skipping parts of the state space that likely do not contain errors, thus helping to detect errors much faster. One useful heuristic function maximizes thread preemption on each trace [28], while another heuristic gives preference to transitions that may interfere with some of the previous transitions on the current trace [57]. Usage of randomized search [13, 17] also helps to achieve significant performance improvements of concurrency error detection. Many techniques from this category exploit the observation that errors can often be found quickly in a particular small part of the state space [48].

Bounded model checking [5] explores all state space paths up to a certain depth that corresponds, for example, to the user-defined bound on the number of loop unrollings. It belongs to the group of techniques based on constraint solving, because the program behavior and the property are together encoded into a logic formula (a set of constraints) whose satisfiability is then checked by an automated SAT or SMT solver. Recently, bounded model checking has been successfully applied also to multithreaded software [14, 35].

Symbolic execution [33], too, represents a special case of systematic traversal of program state space that involves constraint solving. While techniques based on symbolic execution have been designed mostly for single-threaded programs and used for automated generation of tests, there exist some recently published extensions towards concurrency [4, 20].

## 2.2 Static Data-Flow and Pointer Analysis

Many verification and bug finding techniques involve static analysis that computes over-approximate information about concurrency-related behavior of a given program, such as reachability of heap objects from multiple threads and sets of locks held at specific code locations [19]. The information can be directly used to recognize possible concurrency bugs or to prove the absence of specific errors. For example, if we consider a pair of accesses to the same field of a shared heap object, such that each of the accesses is performed by a different thread and within the scope of a different lock, then analysis reports a data race condition. However, the consequence of analysis imprecision (caused by over-approximation) is that spurious warnings may be reported, because it is often

not possible to statically decide whether some error can really occur during the program execution. Several analyses are typically chained together in order to get more precise results and, in particular, to eliminate some of the spurious warning. The basic component is, in most cases, the aliasing analysis that determines whether two variables may point to the same heap object. For example, the static detection of races proposed in [43] combines aliasing analysis with thread-escape and lock analyses. Escape analysis [49, 50] identifies heap objects that are shared between threads, i.e. globally reachable. May-happen-in-parallel (MHP) [40, 46] is another example of static analysis that is often used in practice, also as one component of verification tools. The MHP analysis determines whether a given pair of actions, each from a different thread, may be executed concurrently when considering thread synchronization that is in place.

## 2.3 Dynamic Analysis

The key characteristic of dynamic analysis is that, contrary to static analysis, it computes very precise information but only for a small number of execution traces. It considers just traces (i.e., thread schedules) that were actually executed at runtime. Dynamic analysis techniques compute and use some representation of happens-before ordering [36] to identify possible conflicts among threads. For example, if a pair of dynamically recorded accesses to the same field of a heap object cannot actually happen concurrently, i.e. if one of the accesses must happen before the other due to thread synchronization, then the pair does not form a race condition. FastTrack [21] and DoubleChecker [6] are two very efficient and precise approaches based on the happens-before ordering. An advantage of dynamic analysis is the ability to find real errors, but it also has important limitations: (1) it cannot prove the absence of errors because of its low coverage, and (2) it usually has a significant overhead caused by the need to record many events during the program execution. In particular, the second limitation applies also to dynamic techniques of POR, as we indicated above.

Specific combinations of dynamic analysis with static techniques of program analysis and verification have been proposed in recent years (see, e.g., [26] and [45]). Information computed by static analyses in advance is used to control the dynamic analysis (for example, to choose which concrete execution traces should be inspected further), while the results of dynamic analysis refine the precision of static over-approximation. The whole process is repeated until the given program is fully verified or a real error is found.

Even though dynamic analysis checks only a small number of execution traces, additional concurrency errors may be discovered based on its results using so-called predictive analysis [32, 54]. The key idea is to take one of the fully analyzed traces and change the ordering of instructions while preserving the happens-before ordering. In this way, additional feasible execution traces can be generated without actually running the program, and they are subsequently inspected for possible errors. Tools implement the predictive analysis by constraint solving over symbolic representation of execution traces [8, 24].

## 2.4 Concurrency Testing

While classic testing is not applicable for detecting concurrency errors, as we discussed in the previous chapter, some advanced testing techniques for multithreaded programs have been developed [38, 55, 58]. They are motivated by practical infeasibility of exhaustive state space traversal for large software systems due to the huge number of possible thread interleavings. Most techniques of concurrency testing are based on a specific combination of state space traversal with dynamic analysis. Since only a relatively small subset of possible thread interleavings can be analyzed in a limited time, concurrency testing frameworks carefully select the interleavings to be inspected — a common approach is to pick interleavings that were not yet analyzed in order to increase coverage [55, 58]. A particular interleaving is enforced during the program execution through a custom scheduler that orchestrates all threads according to the desired plan.

## 2.5 Modular Verification

From the practical perspective, a highly important aspect of verification techniques is the level of modularity. This aspect is common to all the approaches that we described above. Specifically, verification algorithms can be designed either as modular or as whole-program (inter-procedural). Modularity is typically used in one of the following two ways: (1) at the granularity of procedures or (2) at the level of individual threads. In the first case, a modular algorithm checks one procedure at a time, using some kind of summaries for other procedures called within it. Similarly, thread-modular verification [23] checks the behavior of one thread at a time, using abstractions of possibly interfering actions of other concurrent threads [30, 52], such as writes to shared variables and heap objects. Lot of work has been done both on theoretical foundations [15, 16] and practical frameworks [39].

# Chapter 3

# Overview: Contribution and Included Publications

This chapter provides an overview of our contribution in the fields of automated verification of multithreaded programs and efficient detection of concurrency errors, and summarizes the included publications. All these publications present the results of research projects on which I participated in the role of a principal investigator, but nevertheless the research has always been conducted as a team effort — together with my supervisors, colleagues, and students. In our research, we have built upon many of the concepts and techniques described in Chapter 2 — specifically, we designed new algorithms that improve and extend state-of-the-art in terms of precision, performance, and scalability.

We divided the publications into two groups. The common topic of the first group are algorithms that improve precision and performance of partial order reduction (Chapters 4-8). The second group contains publications about techniques that enable faster search for concurrency errors through heuristics and randomization (Chapters 9-12). Here follows the complete list of publications included in the habilitation thesis:

[Chapter 4] Pavel Parízek and Ondřej Lhoták. **Identifying Future Field Accesses in Exhaustive State Space Traversal**. Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 93–102, IEEE CS.

[Chapter 5] Pavel Parízek and Ondřej Lhoták. **Model Checking of Concurrent Programs with Static Analysis of Field Accesses**. Science of Computer Programming, volume 98, part 4, pp. 735–763, Elsevier, 2015.

[Chapter 6] Pavel Parízek. **Hybrid Analysis for Partial Order Reduction of Programs with Arrays**. Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2016), LNCS, volume 9583, pp. 291-310, Springer.

[Chapter 7] Pavel Parízek and Pavel Jančík. **Approximating Happens-Before Order: Interplay between Static Analysis and State Space Traversal**. Proceedings of the 21st International Symposium on Model Checking of Software (SPIN 2014), pp. 1–10, ACM.

[Chapter 8] Pavel Parízek. **Hybrid Partial Order Reduction with Under-Approximate Dynamic Points-to and Determinacy Information**. Proceedings of the 16th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2016), pp. 141–148, IEEE.

[Chapter 9] Pavel Parízek and Ondřej Lhoták. **Randomized Backtracking in State Space Traversal**. Proceedings of the 18th International Workshop on Model Checking of Software (SPIN 2011), LNCS, volume 6823, pp. 75–89, Springer.

[Chapter 10] Pavel Parízek and Ondřej Lhoták. **Fast Detection of Concurrency Errors by State Space Traversal with Randomization and Early Backtracking**. International Journal on Software Tools for Technology Transfer, accepted, published online, Springer, 2018, https://link.springer.com/article/10.1007/s10009-018-0484-7.

[Chapter 11] Pavel Parízek and Tomáš Kalibera. **Efficient Detection of Errors in Java Components Using Random Environment and Restarts**. Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010), LNCS, volume 6015, pp. 451–465, Springer.

[Chapter 12] Pavel Parízek. **Fast Error Detection with Hybrid Analyses of Future Accesses**. Proceedings of the 31st ACM Symposium on Applied Computing (SAC 2016), pp. 1251–1254, ACM.

The next two sections reflect the aforementioned groups of publications in the list.

## 3.1 Improving Partial Order Reduction

We begin this section by describing limitations of the prevailing approaches to partial order reduction (POR), which motivated our work.

The variants of POR used in tools such as Java Pathfinder [18] determine whether a given action is possibly interfering based on the current state and execution history. The state space traversal procedure with POR does not see what pairs of interfering actions may occur in the rest of program execution. Consequently, in order to guarantee soundness, the respective POR techniques have to perform overly conservative decisions, thus achieving only limited precision and suboptimal performance. A popular strategy is to create a thread scheduling choice at each instruction that accesses a heap object reachable from multiple threads (e.g., writing to some field or acquiring the object's monitor), assuming that some other concurrent thread might perform an interfering action upon the same object later during its execution after the current dynamic state. Some of thread choices made using this strategy are redundant, because they are not really needed to cover all interleavings of the interfering actions.

Fully dynamic approaches to POR (see, e.g., [22]) are very precise, since they recognize memory locations truly accessed by multiple threads during the program execution. On the other hand, a limitation of dynamic POR is that it performs redundant computation — it has to (i) explore each trace up to the end state, because choices are created retroactively, and (ii) check every pair of globally visible actions to detect interference. This all may have a negative impact on the performance of state space traversal especially in the case of programs with large state spaces and long execution traces.

Our goal was to design more precise POR algorithms and use them within state space traversal to avoid many of the redundant thread scheduling choices, achieving much better performance and scalability of verification in this way. Elimination of redundant thread choices from the program state space also helps to reduce the number of thread interleavings that a given verification tool will systematically explore, while preserving soundness and coverage of all possible distinct thread interleavings over interfering actions. For this purpose, we combined static analysis together with dynamic analysis and knowledge of program state during the traversal into a **hybrid analysis**, which is the basic component of our **hybrid POR**. Such hybrid analysis exploits the strengths of all its parts and mitigates their respective limitations. When given an instruction to be executed next and the current state, a hybrid analysis identifies — soundly and with a good precision — all possibly interfering actions that other threads may perform on any execution trace starting in the given state. In other words, our hybrid analysis can detect possibly interfering pairs of actions by inspecting possible future behavior of program threads. Analysis results are used on-the-fly during the state space traversal to avoid creation of many redundant thread choices, and therefore to avoid redundant exploration of some thread interleavings. For example, assume that the next instruction of the current thread accesses some field of a heap object. If the hybrid analysis provably determines that no other thread may access the same field in the rest of the program execution starting in the current state, the respective thread choice before the field access instruction would be actually redundant and can be safely omitted.

We have gradually designed three variants of the hybrid analysis, where each of them processes actions of a different kind — accesses to fields of heap objects, accesses to individual array elements, or thread synchronization — and therefore enables POR to eliminate a different subset of redundant thread choices. All variants follow the same principles that are described here. The hybrid analysis has two phases: static and dynamic.

The static phase is performed in advance before the state space traversal and computes only partial results. For each program point $p$ (a code location), it collects the set of actions of a corresponding kind that may occur between the point $p$ and return from the method that ccontains $p$ (including nested calls). In this way, the static analysis can provide data for any possible fragment of the execution of any method.

The dynamic phase is performed on-the-fly during the state space traversal, and uses information taken from a dynamic program state, including the current dynamic call stack of each thread and runtime values of program variables. When the POR technique queries the hybrid analysis in a particular state during the systematic traversal, the analysis processes all runnable threads except the current one, and for each thread computes the full results that cover its behavior from the current program counter up to the end state. Specifically, given the thread $T$, the analysis takes partial results of the static phase for the program counter in each call stack frame of $T$, and then merges all these data together.

Full results of the hybrid analysis are very precise, because of the usage of dynamic call stacks that represent precise contexts, but they are always specific to the current dynamic program state. In addition, some other elements of the dynamic program state are used to further improve the precision of statically computed information. We provide more details below in the following paragraphs and in the respective included publications.

Chronologically, the first variant of the hybrid analysis collects accesses to fields of heap objects that individual threads may perform during the rest of the program execution from the current state. Results of this analysis variant are therefore used to detect pairs of interfering field accesses — that means accesses to the same field on the same object that are performed concurrently by different threads. In publications that are included as Chapters 4-5, we introduce and formally define the basic principles of the whole hybrid analysis, including usage of its results by POR during the state space traversal. We also evaluated several pointer analyses for the purpose of distinguishing between heap objects, discussing their benefits and drawbacks.

A special case is formed by fields that we call immutable. Such fields are updated only during the enclosing object's initialization by a thread that allocated the object, and never written afterwards. We assumed that the initialization phase ends when the object escapes to the heap and thus becomes shared. No thread scheduling choice has to be created when an immutable field is accessed, because only read actions may possibly happen concurrently, once the field is reachable from multiple threads. We define the concept of immutable fields more precisely in Chapter 4, and then in Chapter 5 we present a complementary static analysis that detects immutable fields automatically. The analysis extends known approaches to (1) computing method summaries that capture side effects and (2) escape analysis.

Our second variant of the hybrid analysis focuses on accesses to individual array elements, which represent another kind of possibly shared memory locations besides object fields. Some tools in their default configuration disable thread choices at statements that read or write array elements in order to avoid explosion of the state space. However, they can miss some concurrency errors as we show on a small example program in Chapter 6. We designed the hybrid analysis for array elements with two goals in mind: (1) optimize the existing popular approaches to POR in the context of programs that use arrays and, consequently, (2) allow tools to make choices at selected accesses to array elements while avoiding state explosion at the same time. The analysis considers also possible concrete values of element indexes, using the combination of a symbolic bytecode interpreter with information retrieved from dynamic program states. A concrete value of a program variable can be safely and soundly used to augment the precision of analysis results only when the variable is certainly not updated in the rest of the program execution — the hybrid analysis can determine that too by inspecting allocations of new heap objects, explicit assignments, and method calls. In addition, our analysis supports arbitrary expressions as indexes and multi-dimensional arrays. Chapter 6 provides more details especially about the analysis that determines possible concrete values of array element indexes and whether they could be updated.

The hybrid analyses of future accesses to object fields and individual array elements proved to be quite useful, but their limitation precision-wise is that the collected information always covers the whole lifetime of a given thread. Both analyses do not consider thread synchronization events that may prevent some thread interleavings from occurring at runtime. For example, a pair of field accesses is not interfering when just a single interleaving of the accesses is possible because of thread synchronization. We addressed this limitation with the third variant of the hybrid analysis, which computes approximation of the may-happen-before ordering between accesses to data (object fields, array elements)

and synchronization events (lock acquire, lock release, wait, notify, thread join). Usage of the happens-before ordering helps to identify those pairs of actions that cannot be interleaved arbitrarily. The publication included as Chapter 7 shows examples of the happens-before patterns that our analysis supports, together with a precise description of all components of the hybrid analysis that we designed to recognize these patterns in program control flow. Each of the analysis components gathers information about specific kind of thread synchronization events.

Building upon the hybrid analysis, we also designed a hybrid POR algorithm that addresses the limitations of dynamic POR that we discuss above. It combines dynamic POR with our hybrid analyses of future accesses to object fields and array elements. While the standalone dynamic POR identifies possibly interfering actions just retroactively, i.e. by inspecting the current execution trace up to the current state and using the precise knowledge of dynamic heap objects, hybrid POR augments the whole process by hybrid analyses whose results provide lookahead into the rest of program execution. The key feature of the hybrid POR algorithm is usage of under-approximate dynamic points-to and determinacy information, which is very coarse at the start and gradually refined during a run of the state space traversal procedure. A given variable is determinate at a particular code location, if it has the same value every time program execution reaches the code location [51]. Knowledge of the dynamic points-to sets for local variables improves precision of the hybrid field access analysis. The verification procedure based on state space traversal with hybrid POR soundly covers all interleavings of interfering actions, but termination of the iterative refinement process is not guaranteed, and the whole procedure is optimized towards fast detection of concurrency errors. We present formal definition of the hybrid POR algorithm, together with additional technical details, in Chapter 8.

We evaluated the hybrid analyses and the hybrid POR experimentally on multithreaded Java programs from various benchmark suites and public repositories. All the benchmark programs involve high degree of concurrency. Results show that usage of the hybrid analyses significantly improves the precision, performance, and scalability of POR techniques. These benefits are clearly apparent especially in the case of benchmarks that have larger state spaces, for which many redundant thread choices were eliminated during the systematic traversal. Both data and thorough discussions of results, including the cost of hybrid analyses in terms of the running time, are provided in the respective individual publications (Chapters 4-8). Overall, the work done in this line of research shows that knowledge of dynamic program states can be used to improve precision of various static analyses on-the-fly during the state space traversal at a reasonable cost. Usage of the hybrid analyses in POR allows it to create much less thread choices and in this way enables successful application of program verification to more complex multithreaded programs.

## 3.2 Fast Search for Concurrency Errors

Our primary goal in this research project was to investigate new ways of using randomization and heuristics in the state space traversal for the purpose of fast detection of concurrency errors — improving upon concepts and specific approaches that were published by other

people. We developed three complementary techniques that are described in the following paragraphs. All of them target explicit depth-first traversal of a program state space.

The main idea of the first technique, which we call **randomized early backtracking**, is that the search procedure is allowed to backtrack early from states that still have unexplored outgoing transitions. Some parts of the state space are pruned in this way, resulting in an incomplete traversal that may reach error states faster but without any coverage guarantees. Random number choice and values of three parameters are used to control the decisions whether to backtrack or continue forward. One parameter specifies the minimal search depth (threshold) at which early backtracking is allowed, the second parameter defines the range of values produced by the random number generator that trigger early backtracking, and the last parameter determines the length of backward jumps. Actual values of all three parameters are computed on the basis of a user configuration. We presented the basic idea of randomized backtracking and preliminary experiments in the publication included as Chapter 9. Later we extended the whole approach in a journal publication (Chapter 10). More specifically, we augmented the core procedure and the set of supported templates for expressing parameter values such that they also dynamically reflect the current state space path and the current state, we designed and validated an automated ranking procedure with the goal to identify configurations that yield overall consistently good performance with small variation, and we also presented results of a broad experimental evaluation. Our experiments show that state space traversal with specific configurations of randomized early backtracking achieves better error-detection performance than many state-of-the-art approaches on many benchmark programs.

Besides the actual state space traversal, randomization is often used also in generated test drivers for open programs. Verification tools like Java Pathfinder cannot be directly applied to open programs, e.g. isolated components and libraries, and therefore some kind of a test driver (environment) has to be created. Note that terms **abstract environment** and **test driver** are used interchangeably in this context. A test driver performs various sequences and concurrent interleavings of method calls with various combinations of parameter values in order to exercise different control-flow paths in the given program and trigger as many errors in the code as possible. Consequently, a verification tool takes as input a full executable program that is composed of the open program and its environment. Our approach is based on automated generation of test drivers that run multiple threads and execute a random sequence of method calls in each thread. Another challenge that we addressed in this work are the long running times of verification and bug detection tools, even if a test driver executes only few threads and short sequences of method calls. Some randomized test drivers (environments) may not trigger any error, in which case the whole program state space is traversed without any benefit, or the traversal simply runs for too long when a particular environment is used. The key idea of our solution is to avoid the long running times by frequently restarting the whole error detection process according to a specific strategy. If the running time of a verification tool exceeds a predefined small limit, the process is stopped, a new environment that involves random choice is generated, and the verification is started again. Our evaluation showed that, with the help of restarts, at least some concurrency errors are found in a reasonable time. This approach was greatly inspired by the usage of restarts in SAT solvers [27, 31] and other long-running software processes. Further details are presented in Chapter 11.

We also designed two heuristics that are based on the hybrid analysis of possible accesses to shared objects. The first heuristic changes the order in which transitions outgoing from a state are explored. It prioritizes threads that may execute actions possibly interfering with some past accesses. The second heuristic prunes in each state all those outgoing transitions that are not associated with threads that may, according to the hybrid analysis, perform some interfering actions in the future. In both cases, results of the hybrid analysis are used to control the process of state space traversal and search for concurrency errors. Our experiments indicated great improvements of the error detection speed over the baseline for some configurations and benchmarks. Chapter 12 provides detailed explanation of the proposed heuristics and complete results of experiments together with their discussion.

## 3.3 Tools and Experiments

Very important aspects of our research work that I want to emphasize here are tool building and solid experimental evaluation. Tools are needed (1) to evaluate and validate the proposed methods, (2) to highlight the benefits and limitations of each method with respect to performance and precision, and (3) to enable experiments on realistic software systems that point to interesting open research problems (challenges). We implemented most of the techniques described above in this chapter using Java Pathfinder, which is a highly extensible platform for verification and analysis of multithreaded Java programs, and the WALA libraries for static analysis of Java programs, and released everything in the form of open source software. Although we evaluate our techniques typically on Java programs, we strive to design them in a general way, so that they can be applied also to multithreaded programs written in other languages (e.g., C++ and C#).

# Chapter 4

# Identifying Future Field Accesses in Exhaustive State Space Traversal

**Pavel Parízek and Ondřej Lhoták**

**Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 93–102, IEEE CS**
**DOI: 10.1109/ASE.2011.6100154**

# Chapter 5

# Model Checking of Concurrent Programs with Static Analysis of Field Accesses

**Pavel Parízek and Ondřej Lhoták**

**Science of Computer Programming, volume 98, part 4, pp. 735–763, Elsevier, 2015**
**DOI: 10.1016/j.scico.2014.10.008**

# Chapter 6

# Hybrid Analysis for Partial Order Reduction of Programs with Arrays

**Pavel Parízek**

**Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2016), LNCS, volume 9583, pp. 291-310, Springer**
**DOI: 10.1007/978-3-662-49122-5_14**

# Chapter 7

# Approximating Happens-Before Order: Interplay between Static Analysis and State Space Traversal

**Pavel Parízek and Pavel Jančík**

**Proceedings of the 21st International Symposium on Model Checking of Software (SPIN 2014), pp. 1–10, ACM**
**DOI: 10.1145/2632362.2632365**

# Chapter 8

# Hybrid Partial Order Reduction with Under-Approximate Dynamic Points-to and Determinacy Information

**Pavel Parízek**

**Proceedings of the 16th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2016), pp. 141–148, IEEE**
**DOI: 10.1109/FMCAD.2016.7886672**

# Chapter 9

# Randomized Backtracking in State Space Traversal

**Pavel Parízek and Ondřej Lhoták**

**Proceedings of the 18th International Workshop on Model Checking of Software (SPIN 2011), LNCS, volume 6823, pp. 75–89, Springer**
**DOI: 10.1007/978-3-642-22306-8_6**

# Chapter 10

# Fast Detection of Concurrency Errors by State Space Traversal with Randomization and Early Backtracking

**Pavel Parízek and Ondřej Lhoták**

**International Journal on Software Tools for Technology Transfer, accepted, published online, Springer, 2018,** https://link.springer.com/article/10.1007/s10009-018-0484-7
**DOI: 10.1007/s10009-018-0484-7**

# Chapter 11

# Efficient Detection of Errors in Java Components Using Random Environment and Restarts

**Pavel Parízek and Tomáš Kalibera**

**Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010), LNCS, volume 6015, pp. 451–465, Springer**
**DOI: 10.1007/978-3-642-12002-2_37**

# Chapter 12

# Fast Error Detection with Hybrid Analyses of Future Accesses

**Pavel Parízek**

**Proceedings of the 31st ACM Symposium on Applied Computing (SAC 2016), pp. 1251–1254, ACM**
**DOI: 10.1145/2851613.2851935**

# Chapter 13

# Conclusion and Future Work

New techniques presented in the publications that we included in this habilitation thesis have two main benefits: (1) improved performance and scalability of automated verification of multithreaded programs through more precise partial order reduction, and (2) faster discovery of concurrency errors by state space traversal augmented with randomization and heuristics. Using tools that implement our techniques, developers could apply checks to larger software systems and perform them more often, all in order to create software that contains less bugs and with less effort. However, many open problems and challenges still must be addressed to make automated verification and debugging of large industrial multithreaded software systems really practical. From the perspective of theory, almost every interesting verification and program analysis question is undecidable. Therefore, much like the whole community, we strive to develop algorithms and tools that perform reasonably well in as many cases as possible and provide useful answers. In the rest of this chapter, we briefly introduce our research plans in this area for the next several years, including work that is already in progress.

**Incremental techniques.** The prevailing verification methodology is to analyze the program or its state space each time completely from the start, ignoring the results of previous runs of a verification tool. An alternative methodology is based on the usage of incremental techniques, some of which (i) exploit differences between consecutive versions of a given software system and (ii) perform checks that focus on recent source code modifications. It is motivated by the need to avoid the high cost of program verification and enable its application to large systems in practice. The process works in the following way. Full verification is performed only for the initial version of a given system (e.g., the first release). Then, for each subsequent version, analysis focuses just on program behaviors that are influenced by source code changes made since the previous run of a verification tool. Unaffected fragments of the program code and its state space do not have to be re-analyzed. Results of every run are saved in order to enable their reuse during analysis of the next version, thus making it less time consuming. Since two consecutive versions of a software system are typically quite similar, i.e. the differences are usually small, usage of incremental techniques can significantly reduce the overall cost of verification during the whole process of software development.

Many algorithms and other results, either directly on the topic of incremental verification or closely related, were published recently, but mostly just for sequential systems (e.g., [11, 37, 47]). We plan to work on incremental techniques designated specifically for verification and analysis of multithreaded programs. The focus of the research community have started moving towards this particular area quite recently [29]. Related challenges include, for example, (1) very fast detection of concurrency errors in large systems, (2) fully automated analysis of incomplete programs where some components and threads are not yet implemented, and (3) efficient representation of intermediate results and fragments of the state space for subsequent verification of a more complete program. Therefore, in order to fulfill the overall goal we have to do also the following:

- design compact representations of program behavior and partial verification results,
- propose incremental methods for capturing the effects of source code changes on program behavior (e.g., interference among threads),
- develop efficient algorithms for incremental refinement of previous verification results (based on recent source code edits),
- find and use suitable compact over-approximations of the missing pieces of subject incomplete programs.

We would also like to explore the possibility of incremental algorithms of partial order reduction that will on-the-fly reflect program code edits. Another direction is to create fast incremental techniques for detecting concurrency errors through adaptation of selected approaches that check the whole program at once (using either static or dynamic analysis) — many of such approaches have been published by other researchers.

**Debugging and code repair.** Besides scalability and performance of verification, another hard problem is fixing the detected concurrency bugs properly. More specifically, once a verification tool reports an error together with a rather complex trace, then developers have to perform these steps: (1) locate the root cause of the observed error, (2) modify the program source code in order to repair it and eliminate the bug, and (3) check that the applied fix has actually been correct and did not introduce any new bugs to the program code. One of my long-term research plans is to work on the development of automated methods for easier debugging of concurrent programs. Like in the case of verification, we are especially interested in automated incremental techniques for debugging. A specific goal is to automatize common debugging actions that developers perform manually (1) in order to identify root causes of concurrency errors and (2) to create valid repairs of program code that eliminate the respective bugs.

Our overall grand vision is fast continuous detection and repair of errors in multithreaded programs, such that tools would be able to guide developers by providing timely reports about the results of verification and suggestions of possible bug fixes, and to do that on-the-fly while developers are editing program source code. The solution will involve analysis of changes to the program code made by the developer, which should be performed continuously at the time of editing, and immediate search for concurrency errors.

**Automatically generating abstractions.** A widely used approach for improving scalability of program verification is abstraction combined with iterative refinement. We have already started working on a procedure that, for a given complex software system in Java, automatically generates its abstraction that is more amenable to verification. The whole procedure consists of three phases: (1) dynamic analysis that records information about native methods and library methods that perform I/O, (2) static analysis that identifies possible side-effects of library methods and creates their summaries, and (3) program code transformations. Generated abstract programs can be also used as benchmarks in future experimental studies.

**Remark.** We also see possible applications of our techniques and tools in teaching. For example, students of introductory courses on programming and concurrency might apply our tools to get immediate feedback when they make some change of the program code.

# Bibliography

This section contains bibliography entries just for Chapters 1-3 and 13. Each of the included papers (Chapters 4-12) has its own bibliography section at the end.


[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal Dynamic Partial Order Reduction. In Proceedings of POPL 2014, ACM.

[2] T. Ball, S. Burckhardt, K. Coons, M. Musuvathi, and S. Qadeer. Preemption Sealing for Efficient Concurrency Testing. In Proceedings of TACAS 2010, LNCS 6015.

[3] T. Ball, S. Burckhardt, P. de Halleux, M. Musuvathi, and S. Qadeer. Predictable and Progressive Testing of Multithreaded Code. IEEE Software, 28(3), IEEE, 2011.

[4] T. Bergan, D. Grossman, and L. Ceze. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. In Proceedings of OOPSLA 2014, ACM.

[5] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. Advances in Computers, 58, Elsevier, 2003.

[6] S. Biswas, J. Huang, A. Sengupta, and M.D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In Proceedings of PLDI 2014, ACM.

[7] G. Brat and W. Visser. Combining Static Analysis and Model Checking for Software Analysis. In Proceedings of ASE 2001, IEEE CS.

[8] Y. Cai, S. Wu, and W.K. Chan. ConLock: A Constraint-Based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs. In Proceedings of ICSE 2014, ACM.

[9] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In Proceedings of PLDI 2002, ACM.

[10] E.M. Clarke, O. Grumberg, and D.A. Peled. Model Checking. MIT Press, 2001

[11] C.L. Conway, K. Namjoshi, D. Dams, and S.A. Edwards. Incremental Algorithms for Inter-procedural Analysis of Safety Properties. In Proceedings of CAV 2005, LNCS 3576.

[12] B. Cook, D. Kroening, and N. Sharygina. Symbolic Model Checking for Asynchronous Boolean Programs. In Proceedings of SPIN 2005, LNCS 3639.

[13] K.E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: Effective Unit Testing for Concurrency Libraries. In Proceedings of PPoPP 2010, ACM.

[14] L. Cordeiro and B. Fischer. Verifying Multi-threaded Software Using SMT-based Context-Bounded Model Checking. In Proceedings of ICSE 2011, ACM.

[15] T. Dinsdale-Young, L. Birkedal, P. Gardner, M.J. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. In Proceedings of POPL 2013, ACM.

[16] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-Guarantee Reasoning. In Proceedings of ESOP 2009, LNCS 5502.

[17] M.B. Dwyer, S.G. Elbaum, S. Person, and R. Purandare. Parallel Randomized State-Space Search. In Proceedings of ICSE 2007, IEEE CS.

[18] M. Dwyer, J. Hatcliff, Robby, and V. Ranganath. Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs. Formal Methods in System Design, 25, Springer, 2004.

[19] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In Proceedings of SOSP 2003, ACM.

[20] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic Testing. In Proceedings of ESEC/FSE 2013, ACM.

[21] C. Flanagan and S.N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In Proceedings of PLDI 2009, ACM.

[22] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In Proceedings of POPL 2005, ACM.

[23] C. Flanagan and S. Qadeer. Thread-Modular Model Checking. In Proceedings of SPIN 2003, LNCS 2648.

[24] M. Ganai. Scalable and Precise Symbolic Analysis for Atomicity Violations. In Proceedings of ASE 2011, IEEE.

[25] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. LNCS 1032, Springer, 1996.

[26] P. Godefroid, A. Nori, S. Rajamani, and S. Tetali. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In Proceedings of POPL 2010, ACM.

[27] C.P. Gomes, B. Selman, and H.A. Kautz. Boosting Combinatorial Search Through Randomization. In Proceedings of AAAI 1998, AAAI Press.

[28] A. Groce and W. Visser. Heuristics for Model Checking Java Programs. International Journal on Software Tools for Technology Transfer, 6(4), Springer, 2004.

[29] S. Guo, M. Kusano, and C. Wang. Conc-iSE: Incremental Symbolic Execution of Concurrent Software. In Proceedings of ASE 2016, ACM.

[30] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate Abstraction and Refinement for Verifying Multi-Threaded Programs. In Proceedings of POPL 2011, ACM.

[31] J. Huang. The Effect of Restarts on the Efficiency of Clause Learning. In Proceedings of IJCAI 2007.

[32] J. Huang and C. Zhang. Persuasive Prediction of Concurrency Access Anomalies. In Proceedings of ISSTA 2011, ACM.

[33] J.C. King. Symbolic Execution and Program Testing. Communications of the ACM, 19(7), ACM, 1976.

[34] S. Khoshnood, M. Kusano, and C. Wang. ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrency Bugs. In Proceedings of ISSTA 2015, ACM.

[35] S. Lahiri, S. Qadeer, and Z. Rakamaric. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In Proceedings of CAV 2009, LNCS 5643.

[36] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7), ACM, 1978.

[37] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental State-Space Exploration for Programs with Dynamically Allocated Data. In Proceedings of ICSE 2008, ACM.

[38] Y. Lei and R. Carver. A New Algorithm for Reachability Testing of Concurrent Programs. In Proceedings of ISSRE 2005, IEEE.

[39] K.R.M. Leino and Peter Mueller. A Basis for Verifying Multi-threaded Programs. In Proceedings of ESOP 2009, LNCS 5502.

[40] L. Li and C. Verbrugge. A Practical MHP Information Analysis for Concurrent Java Programs. In Proceedings of LCPC 2004, LNCS 3602.

[41] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In Proceedings of PLDI 2007, ACM.

[42] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In Proceedings of OSDI 2008, USENIX.

[43] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In Proceedings of PLDI 2006, ACM.

[44] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. In Proceedings of ICSE 2009, IEEE CS.

[45] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from Tests. In Proceedings of POPL 2012, ACM.

[46] G. Naumovich, G.S. Avrunin, and L.A. Clarke. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. In Proceedings of ESEC/FSE 1999, LNCS 1687.

[47] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In Proceedings of PLDI 2011, ACM.

[48] S. Qadeer and D. Wu. KISS: Keep It Simple and Sequential. In Proceedings of PLDI 2004, ACM.

[49] E. Ruf. Effective Synchronization Removal for Java. In Proceedings of PLDI 2000, ACM.

[50] A. Salcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In Proceedings of PPoPP 2001, ACM.

[51] M. Schaefer, M. Sridharan, J. Dolby, and F. Tip. Dynamic Determinacy Analysis. In Proceedings of PLDI 2013, ACM.

[52] N. Sinha and C. Wang. On Interference Abstractions. In Proceedings of POPL 2011, ACM.

[53] W. Visser, K. Havelund, G.P. Brat, S. Park, and F. Lerda. Model Checking Programs. Automated Software Engineering, 10(2), Springer, 2003.

[54] C. Wang, R. Limaye, M.K. Ganai, and A. Gupta. Trace-Based Symbolic Analysis for Atomicity Violations. In Proceedings of TACAS 2010, LNCS 6015.

[55] C. Wang, M. Said, and A. Gupta. Coverage Guided Systematic Concurrency Testing. In Proceedings of ICSE 2011, ACM.

[56] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole Partial Order Reduction. In Proceedings of TACAS 2008, LNCS 4963.

[57] M. Wehrle and S. Kupferschmid. Context-Enhanced Directed Model Checking. In Proceedings of SPIN 2010, LNCS 6349.

[58] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A Coverage-Driven Testing Tool for Multithreaded Programs. In Proceedings of OOPSLA 2012, ACM.