The published version of the paper is available on ScienceDirect
https://www.sciencedirect.com/science/article/pii/S0164121219302298

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Journal of Systems and Software 00 (2019) 1–42

# Detection of intermittent faults in software programs through identification of suspicious shared variable access patterns

Panagiotis Sotiropoulos, Costas Vassilakis

*Department of Informatics and Telecommunications, University of the Peloponnese, Tripolis, Greece*
*Tel.: +30-2710-372203*
*Fax: +30-2710-372160*
*panossot@uop.gr costas@uop.gr*

## Abstract

Intermittent faults are a very common problem in the software world, while difficult to be debugged. Most of the existing approaches though assume that suitable instrumentation has been provided in the program, typically in the form of assertions that dictate which program states are considered to be erroneous. In this paper we propose a method that can be used to detect probable sources of intermittent faults within a program. Our method proposes certain points in the code, whose data interdependencies combined with their execution interweaving indicate that they could be the cause of intermittent faults. It is the responsibility of the user to accept or reject these proposals. An advantage of this method is that it removes the need for having predefined assertion points in the code, being able to detect potential sources of intermittent faults in the whole bulk of the code, with no instrumentation requirements on the side of the programmer. The proposed approach exploits information from the dynamic behavior of the program. In comparison with parser-based approaches which analyze only the program structure, our approach is immutable to language term changes and in general is not depending on any user-provided assertions or configuration.

*Keywords:* Intermittent faults, Fault detection, Shared variables, Model-based checking

## 1. Introduction

An intermittent fault in computer software is a malfunction of a software program that occurs at intervals, usually irregular, while the software functions normally at other times. Avizienis et al. [1] defines intermittent faults as the union of (a) *elusive permanent faults*, i.e. faults that manifest themselves conditionally, with their activation conditions depending on complex combinations of internal state and external requests, that occur rarely and can be very difficult to reproduce and (b) *transient faults*, which includes *physical faults* (i.e. faults associated with the hardware) as well as *interaction faults*, stemming from reciprocal actions with external systems. The root causes of intermittent faults can be traced to (a) particular hardware conditions, e.g. radiation-induced transient faults are caused by alpha particles found in chip packages and atmospheric neutrons [2], (b) limit conditions (e.g. out of memory or disk storage, lost interrupts, not initialized memory, unexpected data from external sources including interactions with other systems) and (c) concurrency errors, including race conditions and scheduling decisions [3][1].

Software-rooted intermittent faults are referred to as MandelBugs [4][5]. A Mandelbug is a bug residing some location in the code, however applying test cases on the code even under seemingly exact conditions does not always lead to a failure. The reason for this non-deterministic behavior is twofold: firstly, the execution of the buggy code leads to an erroneous internal condition (e.g. a wrong variable value) which does not necessarily manifest itself as a failure immediately, but rather it may necessitate a chain between errors (*error propagation*) until the system uses elements (e.g. variable values) involved in the erroneous internal conditions in a way that influences a perceivable system behavior. And secondly, other elements of the software system, including other applications, the operating system or the hardware, may affect the behavior of a fault in a specific application. For instance, if a multi-threaded application lacks adequate synchronization mechanisms, race conditions may occur, depending on the choices made by the operating system scheduler regarding the exact time points that threads are dispatched on the CPU for execution, or preempted. Gray [3] uses the term *Heisenbugs*, to refer to software bugs that either do not appear or change their behavior when attempts are made to discover them. Typically this is owing to the fact that when programs are debugged the execution environment and conditions change [6]: optimization features are turned off; debugger programs may initialize memory contents to zero or modify the memory layout during execution; stepwise execution alters timings; statements inserted to print out variable values differentiate register values and so forth. Grottke et al. [7] identifies aging-related bugs as an interesting a sub-type of Mandelbugs: aging-related bugs are faults capable of causing degraded performance or increased failure rate because they either accumulate internal error states and/or the activation and/or error propagation of the fault is influenced by the total time the system has been running. MandelBugs and Heisenbugs are contrasted to *Bohrbugs* [3], which refers to the class of bugs that always produce a failure on retying the operation that involves the bug; in this respect, a Bohrbug is a solid and easily detectable bug, that can be isolated by standard debugging techniques.

In the analysis presented in [7] Mandelbugs correspond to the 36.5% of the total number of the faults discovered in the on-board software for 18 JPL/NASA space missions. Carrozza et al. [5] studied an industrial mission-critical software system, in which Mandelbugs accounted for the 14.56% of the total number of faults. Cotroneo et al. [8] examine four major open source projects, and report that the percentage of Mandelbugs ranges from 7.5% (for the AXIS project) to 50.2% (for the Linux project); they also assert that in their sample, the fault densities for Bohrbugs and Mandelbugs are similar for large software projects, while for smaller projects the fault density for Bohrbugs tends to be higher and that Mandelbugs take more time to fix than Bohrbugs. Chillarege [9] concludes that Mandelbugs predominantly affect non-functional aspects, such as reliability, availability and serviceability, while they rarely affect software functionality.

A common cause of software-rooted intermittent faults in applications, is the erroneous order of accessing shared variables in multi-threaded applications, e.g. when a write-after-write (WAW) hazard occurs, a shared variable is written by a thread while it should have first been written by another one, and so forth. The more complex the software program, the greater the likelihood of an intermittent fault to occur and the harder to locate its root cause. Many research efforts have targeted the issue of intermittent faults, and in this context a number of concurrency anti-patterns, (i.e. concurrency control mechanisms that have been proven to be ineffective and error-prone) and possible solutions have been identified, e.g. [10, 11]).

Intermittent faults can be detected both using static and dynamic debugging techniques [12]. For the detection of logical faults, in particular, in the context of dynamic approaches, the programmer typically needs to add appropriate *assert* statements expressing program-specific invariants (i.e. conditions that must always hold), which is evaluated at runtime. When the invariant is found not to hold, then a fault is flagged and the developer may use a dynamic debugger to examine the program state, trying to trace back the root cause of the error.

The method proposed in this paper, intends to help programmers discover locations in the code that could cause intermittent faults that are owing to improper order of accessing shared variables. On top of an existing debugging and verification tool, we add mechanisms that create traces of shared variable access sequences and rules that are able to identify such improper access patterns within these traces; these patterns may be manifestations of intermittent fault presence. Then, the system is able to suggest to the developer code locations that may be the root cause of these intermittent faults. In this paper, we have chosen Java Path Finder (JPF [13]; a brief overview of JPF is given in section 2.3) as the base debugger tool, on top of which the proposed method is built; we exploit the capabilities of JPF to extract runtime information from the executing program. Our approach is immutable to any user configuration (e.g. parser configurations), as it exploits information from the dynamic behavior of the program, which is sourced through the mechanisms provided by JPF. More specifically, JPF functionalities are used to gather all the information about possible interleavings of the accesses of the shared variables from the different threads in a tree structure, and after the tree structure is shaped, it is searched for the presence of shared variable access patterns that indicate the presence of an intermittent fault. Code locations that are involved in the suspectable shared variable accesses are then identified, and these locations are proposed to the user (i.e. the developer) for check (e.g. code review to verify whether synchronization mechanisms are used appropriately). The developer is the one who makes the final decision on whether a suggestion made by the tool should be accepted or not. Contrary to other algorithms in the literature, the proposed approach needs no instrumentation (e.g. insertion of appropriate assertions in selected code locations) to work. In this way, the whole extent of the executed code is always checked, and no additional effort on the side of the developer is required. The proposed technique can be used in conjunction with other intermittent fault detection techniques, both at hardware and software level (e.g. [14][15][16][17]); combined application can be achieved either by the simultaneous use of individual techniques (this is directly applicable for other techniques that are hardware-based, e.g. [14][16]; for software-based techniques, an integration step will be required), or through a more loosely coupled approach where the proposed algorithm is run in parallel with other techniques and their results are combined.

In addition, in this paper, we examine the complexity of the proposed intermittent fault detection algorithm, by experimentally quantifying the effect that partial order reduction techniques [18] have on to the limitation of this number of paths.

The rest of the paper is structured as follows: section 2 overviews related work, including static and dynamic verification tools and elaborating on JPF, which is used in our approach. Section 3 introduces the proposed algorithm, while section 4 discusses the complexity of the algorithm. Section 5 explores methods for speeding up the execution of the proposed algorithm by (a) exploiting parallelism and (b) pruning the possible execution paths tree, with the latter techniques being able to also tackle the state explosion issue, which is inherent in state space-based approaches. Section 6 presents an experimental evaluation of the algorithm, and finally section 7 concludes the paper and outlines future work.

## 2. Related work

Since reliability is a key objective in software development, numerous techniques have been proposed and employed to aid developers to localize, identify and remove faults. Some techniques examine the source code statically to identify *code smells*, i.e. characteristics that may indicate a deeper problem. Towards this direction, code smell detectors have been employed [19][20]. Similarly, software fault prediction aims to identify fault-prone software modules by using some underlying properties of the software project before the actual testing process begins [21].

Considering the dynamic behavior of the software, using test cases for unit-level [22] or integration testing was one of the first tools to verify software correctness [22]. Considering the size and complexity of modern software, methods for automatically generating comprehensive test case suites have been developed [23][24][25]. Since test case-based fault detection may miss certain faults, even under high code coverage, approaches to identifying faults that evade test-case based detection processes have also been proposed [26]. Additionally, taking into account that execution of test cases consumes time and resources, their minimization and management have been explored [27][28]. With security aspects gaining increasing attention in the past few years, specialized methods for analyzing and detecting software vulnerabilities have been developed [29].

When faults do manifest in software, either in the context of testing or while execution in production environments, testers and developers need to pinpoint the actual fault location and root cause: to this end, a number of relevant algorithms and techniques have been proposed. Besides "traditional" fault localization techniques, which include logging, assertions, breakpoints and profiling, a number of advanced fault localization techniques have been proposed, which are classified as (a) slice-based, (b) program spectrum-based, (c) statistics-based, (d) program state-based, (e) machine learning-based, (f) data mining-based and (g) model-based techniques. Wong et al. [30] provides a survey on fault localization techniques.

Intermittent faults however, due to their nature, may evade detection from typical fault discovery tools [3], therefore specialized methods have been developed to assist developers in identifying and removing intermittent faults. In the rest of section we overview related work for intermittent fault detection. We initially survey work in the domain of static debuggers, and subsequently we examine approaches using dynamic debuggers. Finally, we give a brief introduction to JPF, the dynamic debugging tool used for the instrumentation of the proposed intermittent fault detection approach.

### *2.1. Static debuggers*

Static debuggers analyze the software code without running it. Because these debuggers do not rely on tests, they can be extremely thorough. Theoretically, static debuggers can test even code paths which are rarely executed in practice [31]. Because they are based on static analysis and satisfying predefined constraints, they could fail to detect some errors. Moreover, while static debuggers can be used in unsafe languages[1] to reveal potential bugs, they cannot guarantee that the data in memory is coherent according to any high-level criteria [32].

It is very common that a static analyzer tool is used to analyze the software code and then symbolic execution with SMT (Satisfiability Modulo Theory) [33] formulas of defined constraints is used for the verification of the code [34].

Symbiosis is an example of a static debugger [35]. Symbiosis necessitates the existence of a failing scheduling, which is then analyzed to determine the root cause of the fault.

[1]An unsafe language does not ensure that primitive program operations are applied to arguments of the proper form, e.g. does not ensure that array subscripts are within the allowable range.

### *2.2. Dynamic debuggers*

Dynamic debuggers examine the software code while it is running. The code in instrumented and all the possible paths are executed in order to detect candidate errors; the Partial Order Reduction technique ([18]) can be used to reduce the number of paths tested, by avoiding to re-examine some path that has been already examined while exploring some other branch. However, depending on the actual values assigned to input variables of the code, it is possible that some paths are not executed and thus the tools may miss certain code defects. In addition, because dynamic debuggers use information available at run time, which is harder to extract statically from the source code, dynamic debuggers can detect errors that are harder to discover when using static analysis tools [31].

The CHESS tool [36] is an example of a dynamic debugger. CHESS creates multiple versions of the debugged program, each one suitably instrumented to control the scheduling of threads. The instrumentation step generates $O(2^n)$ versions for a function with $n$ components, however [36] reports that the execution of $O(n)$ versions (context switch at one of the components each time) is usually enough to activate a concurrency fault; this however may lead to missing Heisenbugs with complex activation patterns. Furthermore, [37] reports that CHESS necessitates additional scaffolding and test code, on top of the test code that would be normally needed for unit or integration testing.

The SCURF tool [37] also follows the instrumentation approach to create particular combinations of thread interleaving. Then, each of these versions is run against a number of test cases -coupled with test oracles- for checking system functionality that need to be available, and a spectrum-based fault localization [38] is utilized to correlate detected errors with concurrently executing code blocks.

CTrigger [39] focuses on atomicity violation bugs; the fault identification process begins by profiling the software and identifying potential unserializable interleavings, while subsequently infeasible interleavings are pruned and low-probability interleavings are ranked. Afterwards, the unserializable interleaving space is explored. CTrigger also requires testing inputs and oracles.

Java Path Finder (JPF) [13], is another dynamic debugger which is used by the proposed algorithm. In the following subsection, we provide a brief introduction to JPF, to present the core functionalities exploited by the proposed algorithm. JPF can locate a number of concurrency bugs, such as deadlocks and missed signals, as well as Java-related faults e.g. unhandled exceptions and improper heap usage; in order to identify faults related to application semantics (e.g. erroneous variable values), relevant assertions should be given within the application code.

Table 1 summarizes the existing tools and methods, their features and capabilities and compares them to those of the proposed algorithm.

Table 1. Comparison of existing fault identification tools and methods

| Tool-method | Scope | Capabilities | Limitations |
|---|---|---|---|
| Test cases | Unit & integration testing | Mostly detects Bohrbugs. | MandelBugs and Heisenbugs typically evade detection; coverage alone cannot guarantee a comprehensive fault detection. Manual creation of test cases is laborious and tedious, however test case generators are available. |
| Static debuggers | Static checking of code properties; symbolic execution can be also performed | Identification of resource leaks, security issues and code smells. Can be used in conjunction with SMT models for increased detection capabilities. With failing schedules available, faults can be localized. | Cannot capture dynamic behavior and may miss some errors; cannot guarantee data coherence in memory according to any high-level criteria. Use with SMT models necessitates definition of constraints (*program invariants*). |
| Chess [36] | Concurrency faults | Detects faults owing to thread interleaving | May miss Heisenbugs with complex activation patterns; necessitates additional scaffolding and test code. |
| SCURF [37] | Concurrency faults | Detects faults owing to thread interleaving and inadequate atomicity guarantees. | Necessitates pre-crafted test cases and test oracles; errors not foreseen in these cases may be missed. |
| CTrigger [39] | Atomicity violation bugs | Locates faults owing to thread interleaving, catering for efficiency. | Necessitates pre-crafted test cases and test oracles; errors not foreseen in these cases may be missed. |
| JPF [13] | Concurrency faults, including deadlocks and atomicity violations; generic faults. | Powerful detection engine, extendable via the listener mechanism. | Needs programmer-provider assertions to detect errors related to high-level data coherence. |
| Proposed algorithm | Enhances JPF with detection of erroneous/-suspect shared variable access patterns. | Captures all errors detected by JPF and errors related to high-level data coherence, without the need to pre-define assertions, test cases or oracles. | May flag false positives. |

### *2.3. JPF - A brief overview*

Java Path Finder (JPF) is an open-source software verification system, initially developed by NASA, that performs model checking for Java programs. While test case-based software checks only some of the potential program executions and may thus miss errors, model checking automatically combines the behavior of state machines with a specification, which corresponds to the properties that the system should satisfy [13, 40, 41]. In more detail, the model checker accepts as input the state machine (FSM) of the program and the specification, and exhaustively explores all executions in a systematic way, flagging executions where the specification is found not to hold [42, 43]. The JPF code is available at [44].

While the systematic generation of all potential execution paths covers the whole search space of program states, handling millions of combinations which are hard to be modeled by manually crafted test cases [42], and thus expose all errors, this approach entails excessive computation cost, which renders it infeasible [42]. Two techniques can be used here to reduce computation costs, namely backtracking and state matching.

- *Backtracking* is a technique that allows the restoration of previous execution states, to examine if there are unexplored choices left. For instance, if JPF reaches a program end state, it can walk backwards to find different possible scheduling sequences that have not been explored yet. While this theoretically can be achieved by re-executing the program from the beginning, and arranging that a different scheduling sequence is adopted in each execution, backtracking is a much more efficient mechanism if state storage is optimized.

- *State Matching* is another key mechanism to avoid unnecessary work. The *execution state* of a program mainly consists of the heap and thread-stack snapshots. While JPF executes, it checks for every new state, whether an identical one has already been explored (c.f. Fig. 1); in this case, there is no use to explore again from that state onwards. When state matching occurs, JPF backtracks to the nearest non-explored non-deterministic choice.

Since concurrent actions can be executed in any arbitrary order, considering all possible interleavings of concurrent actions can lead to a very large state space. It can be shown that the number of states increases exponentially with the number of threads [45]. JPF uses a technique called

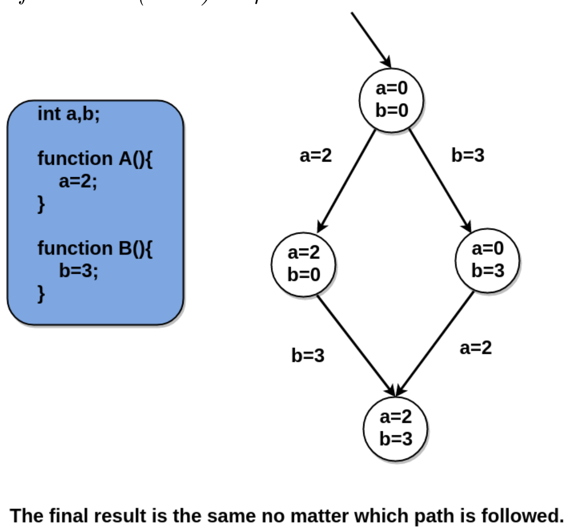


Figure 1. State matching: both execution paths lead to the same state (state 3).

*Partial Order Reduction* (POR [46]), which basically identifies statements whose order of interleaving does not affect in any way the overall program execution, and groups them into a single state transition, reducing thus drastically the number of states that must be maintained. For instance, the instructions of threads *T1* and *T2* in Fig. 2 can be interleaved in any of the six ways listed in the same figure, however to verify program correctness it suffices to explore the two paths highlighted in Fig. 3 [47, 46].

JPF uses a customizable Virtual Machine that supports various features related to model checking, including state storage and state matching. Actually, JPF is a virtual machine (VM) running on top of the Java Virtual Machine (JVM) and controlling its operation. The core JPF model supports checks for generic properties, such as absence of unhandled exceptions, deadlocks, and race conditions.

Listeners are perhaps the most important extension mechanism of JPF. They provide a way to observe, interact with and extend JPF execution through code provided in the form of custom classes. Listeners are dynamically configured at runtime, and therefore they do not require any modification to the JPF core. Listeners are executed at the same authorization level as JPF, so no limitations are imposed to their functionality (c.f. Fig. 4). In our approach, we use one listener that observes shared variable access by threads, and logs these accesses for further analysis.

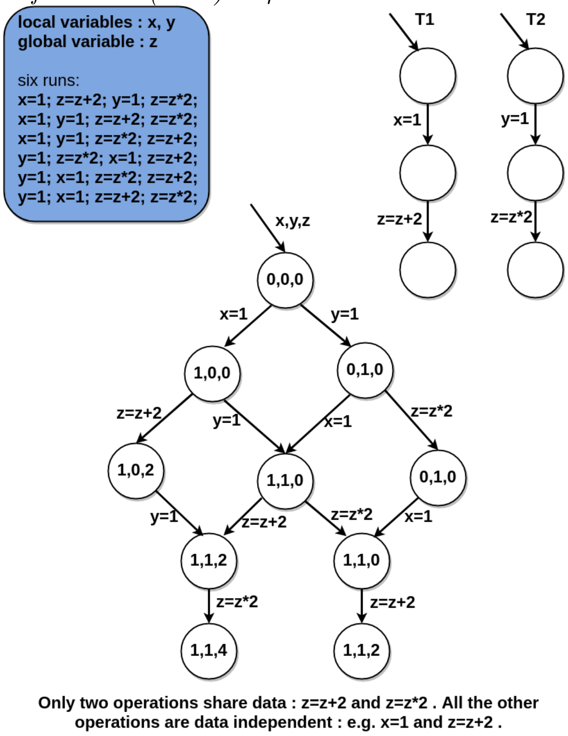


Figure 2. POR - Partial Order Reduction Example

## 3. The Proposed Intermittent Fault Detection Algorithm

The method proposed in this paper comprises three parts: The first one encompasses the development of a rule base for the detection of shared variable access patterns that may indicate sources of intermittent faults. The second one is about the generation of complete execution traces for the target program, to record all possible shared variable access patterns. In this part, JPF, augmented with additional logging listeners, is used to implement the generation of the program traces. The third one comprises the application of the rule base developed in part 1 on the traces generated during step 2, in order to detect possible sources of intermittent faults within the program. These points are proposed to the user for review and, if appropriate, application of the necessary corrections.

The first phase (rule base development) need not be performed for each program. Instead, a generic rule base can be developed once and be subsequently applied to all target programs. It is possible that derivatives of the generic rule base are created to match the requirements of specific program

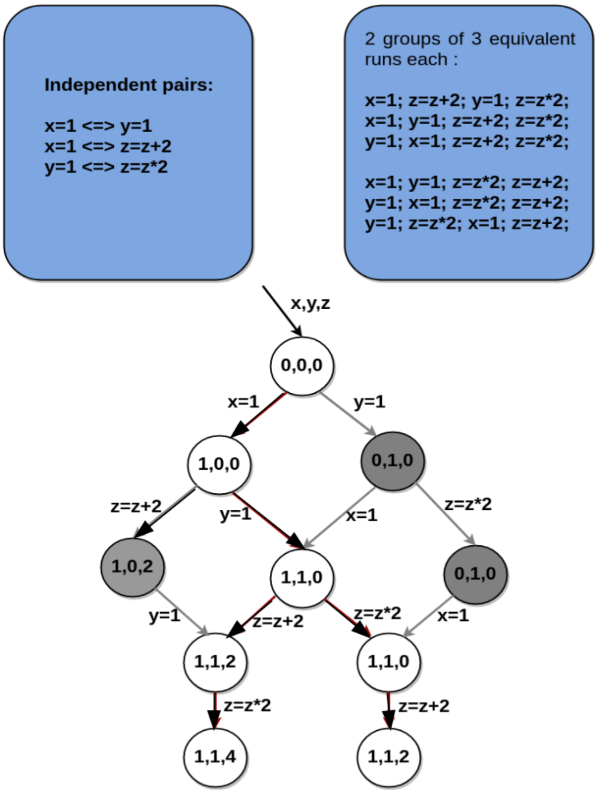


Figure 3. POR - Partial Order Reduction Example

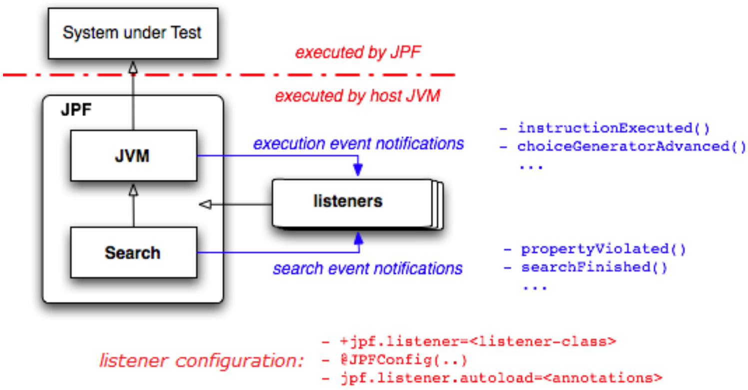


Figure 4. JPF listeners

classes, e.g. programs with different isolation levels. The basic rules that we use in this paper, are :

1. sequences of operations of the form $read_{T1}(X)$, $write_{T2}(X)$, $write_{T1}(X)$, corresponding to *write-after-write* hazards (the notation $read_{T}(X)$ denotes that thread T reads variable X; and similarly for $write_{T}(X)$)
2. sequences of operation of the form $read_{T1}(X)$, $write_{T2}(X)$, $read_{T1}(X)$, corresponding to the *read-after-write* hazard

When either of these patterns is detected in the program traces, then the corresponding code *may* be the cause of intermittent faults. In case of the write-after-write hazard rule, we can observe that this order is not equivalent to any serializable order: if T1 were scheduled before T2, then the value finally stored in X would be the one written by T2 instead of the value written by T1, which is finally stored by the schedule above (this is known as the lost update problem [48]); and if T2 were scheduled before T1, T1 would have read the value of X stored by T2 instead of the value previously stored for X (recall that T1 may have used the value read for X to compute a new value for X, so reading a different value leads to erroneous computation). In case of the *read-after-write* hazard rule, before we read our shared variable X for the second time on thread T1, it is modified on T2 with a write operation, which makes the value of X on thread T1 to have a new value (without having saved the product of the previous X read value or simply using different values of X in different parts of the computation in T1; the latter is also known as the non-repeatable read problem [49]), which may be a potential cause of some intermittent fault.

In the second phase, we create a tree structure, where we store the data of the states of the JPF run. Each state can be a combination of data from the different threads that are accessed concurrently. Typical data that we store during this state are: the variable name, the class name, whether the access is a read or write one, the thread id, the method name, synchronization info, the package name, the value of the variable, if there is a monitor enter or exit operation, lock info, the line of the code that is executed and the source line, etc. When the JPF run is done, the detailed trace about the accesses of the state variables is available for analysis. Since each state may have one or more previous states (recall that the state matching mechanism specifically searches for cases where multiple execution paths have led to the same state) and may lead to multiple subsequent states, the trace effectively forms a directed acyclic graph (DAG).

In the third phase (processing phase), we apply our rule base on this structure in order to detect the points in the code that match our rules. Multiple rule bases or even ad-hoc rules could be applied during this phase. The JPF should be executed only once while the tree structure can be stored and reused for the application of all user rules.

As an example of applying the proposed algorithm, consider the case that the rule base contains the two rules listed above, and that the code illustrated in Listing 1 is checked for the presence of intermittent faults. Before the execution of line (3) of this code, the value of the ***filled*** variable should always be smaller than ***MAXNUM***, a condition that is checked by the condition at line (1) and the code associated with it. However, in the context of concurrent executions of the ***put*** method, it is possible that two distinct threads detect that the value of the ***filled*** variable is equal to ***MAXNUM-1***, and subsequently each one increases the value of the variable by 1, therefore violating the invariant $filled \leq MAXNUM$; this is owing to the premature lock release, occurring at line (2).

Listing 1. A simple code example, which produces faults intermittently

```
    private final static Lock l = new ReentrantLock();
    private static int filled = 0;
    private static ArrayList queue = new ArrayList();
    private static final int MAXNUM = 2;

    public void put(Object elem) {

        l.lock();
   (1)  if (filled < MAXNUM) {
            //other code
   (2)      l.unlock();
        } else {
            l.unlock();
            return;
        }
        l.lock();
        // assert (filled < MAXNUM);
   (3)  filled++;
        queue.add(elem);
        l.unlock();
        return ;
```

```
    }

    public Object get() {
        Object elem = null;
        l.lock();

        if (filled > 0) {
(4)         filled--;
            elem = queue.remove(0);
        }
        l.unlock();

        return elem;
    }
```

---

Fig. 5 demonstrates the different access interleavings that may occur when the code of the ***put*** method is executed concurrently by two threads, *T1* and *T2*, assuming that the condition at line (1) evaluates to ***true*** for both threads. We can notice that six distinct interleavings are possible, out of which four (the 1st, 2nd, 4th and 5th branches of the tree) entail the appearance of the non-repeatable read problem. For instance, in the second branch of the tree, the following shared variable accesses will be performed: $read_{T1}(filled)$, $read_{T2}(filled)$, $read_{T1}(filled)$, $write_{T1}(filled)$, $read_{T2}(filled)$, $write_{T2}(filled)$, with the first two reads corresponding to the checking of condition at line (1), and subsequently each read/write pair corresponding to the variable increment at line (3). In this sequence, we can observe that a *read-after-write* hazard occurs, since a write operation on variable ***filled*** is performed by thread *T1* between the two read operations performed on the same variable by thread *T2*, therefore the read performed by *T2* is non-repeatable.

Fig. 6 shows the respective states of the ***filled*** variable, again assuming that the condition at line (1) evaluates to ***true*** for both threads. Notably, when the ***filled*** variable is less than *MAXNUM-1* all interleavings lead to a correct state, increasing the ***filled*** variable by two. However, if ***filled*** *== MAXNUM-1*, the execution of the branches entailing the *read-after-write* hazard leads to an incorrect state, where the ***filled*** variable is set to ***MAXNUM+1***. The proposed algorithm can thus identify code that is bound to cause intermittent faults, without any knowledge about the correctness of the states.

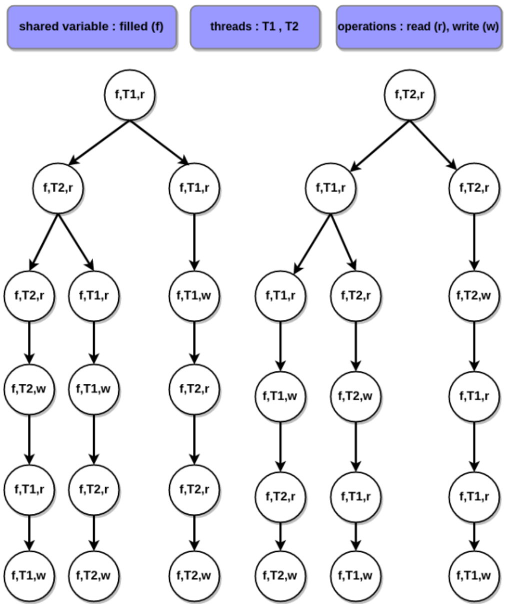


Figure 5. The possible access interleavings of the shared variable ”filled” when two put methods of two different threads are executed concurrently.

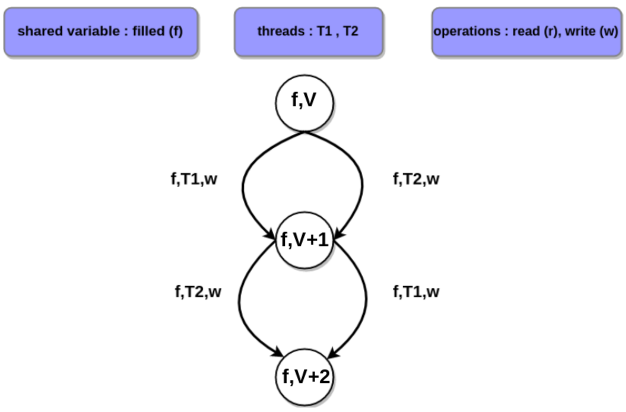


Figure 6. The states of the shared variable ”filled” when two put methods of two different threads are executed concurrently.

Using JPF to generate our state tree structure, has the advantage that the user does not need to give any a-priori information about the code (shared variables, atomic blocks, etc.), while JPF is not sensitive in possible code structure changes as a static analyzer could potentially be.

An implementation of the algorithm is available in open source at `https:`

//github.com/pansot2/JPF.

## 4. Complexity analysis

When a multithreaded program with $k$ threads executes, at each time point the scheduler may pick any of the threads that are not in a suspended state to execute, thus having a maximum of $k$ alternative choices. Since we focus on operations that access shared variables (because inappropriate shared variable access patterns are a major cause of intermittent faults), if we consider that each thread $t_i$ performs $n_i$ shared variable accesses, then the corresponding states of a multithreaded program can be arranged in a tree whose rank is equal to the number of threads $k$ and its depth is equal to:

$$depth = \sum_{i=1}^{k} n_i \tag{1}$$

The root of the tree corresponds to the initial state of the program, while an edge denotes a transition from a state to a subsequent one, through the execution of an instruction that accesses a shared variable, with the instruction belonging to some thread $t_i$ (c.f. Fig. 5). Sibling tree states correspond to different scheduling decisions. The total number of paths in the tree is equal to the number of ways to interleave k ordered sequences.

In order to compute the number of paths, we consider that the instructions in each thread $t_i$ ($1 \leq i \leq k$) are essentially ordered lists, and we want to interleave the elements of these lists while preserving the order of the elements in each ordered list.

According to equation (1), there will be $n_1 + n_2 + \cdots + n_k$ places that we must fill (one place for each level of the tree). We can proceed by first assigning the elements of the first list, corresponding to the instructions of the first thread, to places. Therefore, we select $n_1$ out of the available $n_1 + n_2 + \cdots + n_k$ places, and we assign to the selected $n_1$ places the instructions of the first thread, preserving their order. The number of possible alternatives is $\binom{n_1+n_2+\cdots+n_k}{n_1}$.

Next, we choose $n_2$ of the remaining places that will accommodate members of the second list, which correspond to the instructions of the second thread. Out of the total number of $n_1 + n_2 + \cdots + n_k$ places in the list, $n_1$ are now occupied by elements of the first list, therefore the number of available places in the list is $n_2 + \cdots + n_k$. Consequently, the number of possible alternatives is $\binom{n_2+\cdots+n_k}{n_2}$. Working in the same way with the remaining ordered lists, when placing the elements of the $k^{th}$ list there are $n_k$ elements to be placed in $n_k$ positions, therefore there exist $\binom{n_k}{n_k}$ alternatives.

Combining all the above, the mathematical formula that calculates the number of ways to interleave $k$ ordered sequences is:

$$\prod_{i=1}^{k} \binom{\sum_{j=i}^{k} n_j}{n_i} \tag{2}$$

In each state of the execution tree, we store information about the current accesses of the shared variables, synchronization info, etc. for all active threads. If a thread progresses, by accessing a shared variable, recording changes in the synchronization info, etc., then a new tree node is created as a child of the previous state (c.f. Fig. 7).

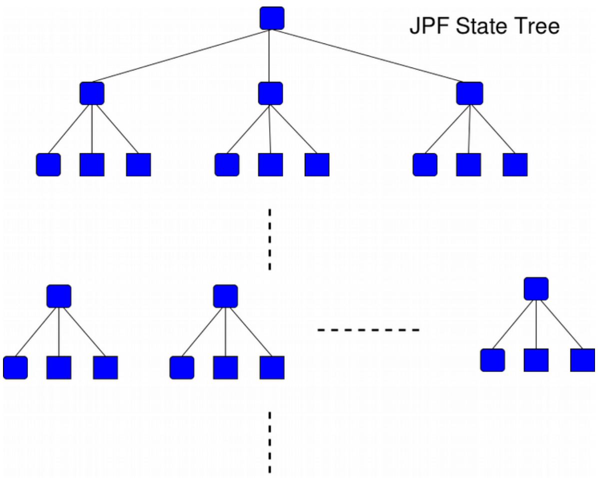


Figure 7. JPF State Tree - Rank of the tree: The maximum number of threads that can run in parallel. Depth of the tree: The sum of the accesses of all shared memory variables for all the threads.

In our work, execution path traversal is instrumented via the JPF model checker, which is a so-called explicit-state model checker, since it enumerates all visited states, and therefore suffers from the state-explosion problem inherent in analyzing large programs [50], while the number of paths to be examined also increases rapidly, with the number of threads and instructions per thread (c.f. equation 2. However, JPF employs a number of techniques including POR (Partial Order Reduction), state matching, and branch coverage [50] to reduce the number of states and the number of paths that will be examined. Using these techniques, JPF can scale up to analyzing programs up to 100,000 lines of code [51].

Insofar, there has not been any theoretical analysis of the effect that POR and state matching have on the complexity of the algorithms that explore the search space of possible execution paths. This is due to the fact that the final effect is highly dependent on the specific instruction placement for each program (which affects the number of cases that POR can be applied), existence and location of lock/unlock instructions (which may limits the actual choices available to the scheduler at each step), as well as volatility of external inputs, which is a determinant factor for the number of cases that states will be actually matched. Further theoretical analysis of this aspect is part of our future work.

The proposed algorithm dictates that shared variable accesses that are performed along an execution path are recorded, and access sequences are scanned for occurrences of the two concurrency hazards, i.e.:

1. access patterns of the form $read_{T1}(X)$, $write_{T2}(X)$, $write_{T1}(X)$, corresponding to *write-after-write* hazards
2. access patterns of the form $read_{T1}(X)$, $write_{T2}(X)$, $read_{T1}(X)$, corresponding to the *read-after-write* hazard

The complexity of recording shared variable access sequences within an execution path is $O(n)$, where $n$ is the number of shared variable access sequences occurring within the execution path. Once the access sequence is recorded, the next task is to determine whether this sequence contains any of the access patterns (1) and (2) above. In the following, we discuss the matching procedure, considering initially the first form of access pattern for a specific thread, and subsequently we generalize for the second form of access patterns and all threads of a program.

When searching for access patterns of the form (1) above, it is not necessary that the instructions are found in strict sequence. The following types of instructions may intervene between the first instruction of the pattern ($read_{T1}(X)$) and the last one ($write_{T3}(X)$):

- accesses to other shared variables, either by the same or by other threads, e.g. $read_{T1}(Y)$ and $write_{T2}(Y)$;
- accesses to the same shared variable by threads other than T1, e.g. $read_{T3}(X)$ and $write_{T2}(X)$;

If such instructions occur, then still the hazard can be flagged, since they have no effect on the semantics of the pattern. Additionally, we can note the following:

- if a *read*$_{T1}$*(X)* instruction occurs between the first and the second instruction of the pattern (i.e. we have an access sequence *read*$_{T1}$*(X),* ***read***$_{T1}$***(X)****, write*$_{T2}$*(X), write*$_{T1}$*(X)*), then the hazard still exists, and it actually maps to the instructions 2-4 of the extended access pattern (i.e. the first *read*$_{T1}$*(X)* is not a part of the hazard; the hazard occurs later on).

- similarly, if a *write*$_{T1}$*(X)* instruction occurs between the second and the third instruction of the pattern (i.e. we have an access sequence *read*$_{T1}$*(X), write*$_{T2}$*(X),* ***write***$_{T1}$***(X)****, write*$_{T1}$*(X)*), then the hazard still exists, and it actually maps to the instructions 1-3 of the extended access patterns (i.e. the last *write*$_{T1}$*(X)* is not a part of the hazard; the hazard has already occurred upon the execution of the third instruction of the extended access pattern).

- if a *read*$_{T1}$*(X)* occurs between the second and the third instruction of the pattern (i.e. we have an access sequence *read*$_{T1}$*(X), write*$_{T2}$*(X),* ***read***$_{T1}$***(X)****, write*$_{T1}$*(X)*), then the hazard *does not occur*, since the computation of the value written by the fourth instruction has been performed based on a "fresh" copy of variable *X* (i.e. a copy obtained after thread *T2* has written a new value [48]).

- finally, if a *write*$_{T1}$*(X)* instruction occurs between the first and the second instruction of the pattern (i.e. we have an access sequence *read*$_{T1}$*(X),* ***write***$_{T1}$***(X)****, write*$_{T2}$*(X), write*$_{T1}$*(X)*) then the hazard *may occur*, since the value stored by the fourth instruction *may* be dependent on the value read by thread *T1* during the execution of the first instruction of the extended access sequence.

Considering all the above, the target access pattern may be formulated as a regular expression of the form:

read$_{T1}$(X) (all except read$_{T1}$(X))* write$_{T2}$(X) (all except read$_{T1}$(X))* write$_{T1}$(X)

where the notation *all except read*$_{T1}$*(X)* means any either a read or write on any shared variable, by any thread except for a read access by thread $T1$; note here that since both the number of shared variables and the number of threads are finite, the notation *all except read*$_{T1}$*(X)* corresponds to a finite set of elements, whose cardinality is *(#threads * #shared variables - 1)*. Furthermore, the star operator denotes "zero or more occurrences of the preceding element".

In order to match a regular expression against an element sequence, a deterministic finite state automaton can be used [52]; Fig. 8 depicts the deterministic finite state automaton which matches the regular expression described above. The finite state automaton performs the match in linear time, performing one state transition for each input symbol. Therefore, matching a single instance of a rule, pertaining to a specific thread and a specific shared variable, can be performed in linear time.

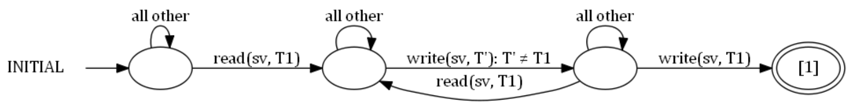


Figure 8. Deterministic finite state automaton for matching the access pattern

In the detection phase multiple instances of rules must be matched against the shared variable access traces of each execution path; effectively, each of the two rules corresponding to the *write-after-write* and the *read-after-write* hazard must be specialized for each thread and each shared variable. Therefore a maximum of (2 * #threads * #shared variables) rules must be matched; the number may be lower if some thread $T$ does not read or write some shared variable $V$, in which case the respective rule instances specialized for thread $T$ and shared variable $V$ need not be considered.

Matching of all rule instances can be performed by a single reading of the shared variable access trace, by combining the deterministic finite state automata into a single deterministic finite state automaton capable of recognizing all suspect shared variable access patterns. The procedure for building the automaton is described in [52] and summarized in the following. Let $M_i = (K_i, \Sigma, s_i, F_i, \delta_i)$ be the automaton that realizes the match of rule instance $R_i$, where $K_i$ is the set of states of the automaton, $\Sigma$ is the alphabet (read and write operations on shared variables by threads), $s_i$ is the start state of the automaton, $F_i$ is the set of final states and $\delta_i$ the transition function for $M_i$. A new non-deterministic automaton $M_{nd} =$

$(K_{nd}, \Sigma, s_{nd}, F_{nd}, \delta_{nd})$ is constructed where:

$$K_{nd} = S_{nd} \bigcup (\bigcup_i K_i) \tag{3}$$

$$s_{nd} = S_{nd} \tag{4}$$

$$F_{nd} = \bigcup_i F_i \tag{5}$$

$$\delta_{nd} = (\bigcup_i \delta_i) \bigcup (\bigcup_i \{S_{nd} \xrightarrow{\epsilon}\}) \tag{6}$$

Effectively, a new start state $S_{nd}$ is introduced which is non-deterministically linked to all start states of the individual automata under an $\epsilon$-transition (i.e. a transition that occurs with no input), and all final states of the individual automata are considered as final states in the merged automaton. The non-deterministic automaton is depicted in Fig. 9. Finally, the non-deterministic automaton is converted to a deterministic one, using the algorithm described in [52].

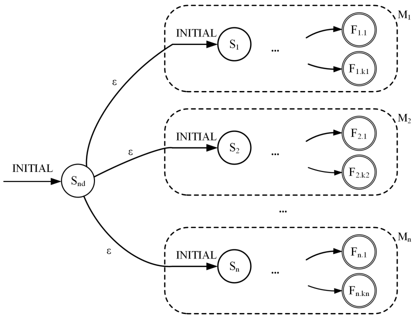


Figure 9. The non-deterministic automaton.

Since an execution path typically includes other instructions besides accesses to shared variables (e.g. accesses to local variables, computations, etc.), the overall complexity of shared variable access recording and matching is inferior to that of the execution of the path. Additionally, note here that in the context of the execution performed by JPF as part of the explicit state-model checking, some operations such as state matching are expensive ones, needing to examine a number of data elements (e.g. values of state variables), as contrasted to the shared variable access recording and rule matching operations introduced by the algorithm, where each shared variable access is recorded or processed in the merged deterministic finite state automaton in a $O(1)$ operation. Therefore, the overall complexity of the suspicious shared variable access pattern detection procedure is dominated by the complexity of the execution of the different execution paths, which is instrumented by JPF which -as noted above- can satisfactorily handle programs of the magnitude of 100,000 lines of code.

## 5. Optimizing Intermittent Fault Identification

While the presented algorithm leverages the intermittent error detection potential, it introduces additional overheads. In this section, we examine methods for limiting overheads and increasing the efficiency of intermittent fault detection.

### *5.1. Separate analysis of independent thread partitions*

Our method targets the identification of access patterns on shared variables which may lead to errors; to test whether such patterns may appear, all possible execution paths are examined. However, when two threads do not access any variable in common, it is not necessary to test all possible interleavings of these threads' execution, since obviously no "suspect" variable access patterns may be identified among these threads.

Generalizing, we can partition the threads in the program in subsets $TS_1$, $TS_2$, ..., $TS_n$ where:

- $TS_i \cap TS_j = \emptyset$, $\forall$ i, j: i $\neq$ j
- $\bigcup_i TS_i = T$, where T is the set of all program threads
- $SVA(TS_i) \cap SVA(TS_j) = \emptyset$, $\forall$ i,j: i $\neq$ j, where $SVA(TS_j)$ is the set of all shared variables accessed by any thread in $TS_j$

In order to exploit this aspect towards the optimization of the intermittent fault identification process, we override the default choice generation and backtracking behavior of JPF, to allow the user to specify the threads whose execution will be monitored in a particular execution. At implementation level, this is realized by overriding the `stateAdvanced` method of the listener, which controls what happens when a new state is generated. In more detail, the user defines in the configuration file the threads that contain related shared variables, which form a thread subset, and the rest of the threads being partitioned into trivial, single-thread subsets which are ignored in order not to produce any alternative choices; only a single choice is considered for these threads. Effectively we have a model involving some "interacting threads" and some "independent threads". This is accomplished using the following configuration parameter:

*vm.watched.threads = the threads that should trigger alternative choice generations in JPF*

The code in the listener that handles this functionality is illustrated in the following Algorithm (Listing 2):

Listing 2. Process followed by the listener used for choice generation only for a subset of the threads of the software program

```
1. Get information about the watched threads defined
   with the configuration parameter vm.watched.threads
2. Get information about the threads that the watched
   threads depend on; this is defined via the
   configuration parameter vm.watched.threads.seqdeps
3. In the stateAdvanced overriden method, ignore the
   states that are not caused by executing
   instructions of the watched threads or the threads
   they depend on. This is accomplished by invoking
   the search.getVM().ignoreState() method.
```

As it can be noticed in the process above, the ***ignoreState()*** method that JPF provides is used in order to ignore the states related to a thread change that are not included at the ***vm.watched.threads*** list in the configuration. There is no need to make alternative choices for the scheduling of threads that are not included in the ***vm.watched.threads*** variable and have no watched thread depending on them, as these, in general, do not influence the subset of threads for which the intermittent fault analysis is conducted.

In order to comprehensively analyze the program for existence of intermittent faults, the intermittent fault detection procedure should be run for each thread subset $TS_i$. If the number of states examined when analyzing thread subset $TS_i$ is equal to $numStates(TS_i)$, then the analysis of all subsets entails the examination of $\sum_i numStates(TS_i)$ states. Contrary, if a combined analysis of all threads is employed (i.e. the thread "independence property" is not exploited), then the analysis will entail the examination of $\prod_i numStates(TS_i)$ states, under the assumption that no dependent threads exist for each thread subset. It is clear that the thread partitioning scheme introduces significant performance gains.

The formulation of independent thread partitions that are separately examined for suspect shared variable access patterns contributes to the reduction of the state space that need to be examined, alleviating thus the issue of state explosion. The gains regarding the aspect of state space size reduction are quantified through the experiments presented in subsection 6.3.

At this stage of development, we have delegated the responsibility of partitioning threads to thread subsets to the user; in our future work, we will consider automatic or semi-automated ways to determine independent thread subsets.

### *5.2. Pruning state subtrees of specific nodes*

In this section we explore the potential to optimize the intermittent fault analysis time, by reducing the nodes of the JPF tree for different ranks and depths. This method can be used for software programs employing a Boss-workers model [53] (or the dispatcher-worker model, as listed in [54]), where the different tasks are distributed to workers which use code/libraries that are independent (in terms of shared variables) from the rest of the program. A typical case of this example is the web server process service loop, where requests are accepted by the main thread and then their execution is delegated to worker threads, with worker threads being totally independent and not accessing any shared variables. In this case, we could avoid the expansion of alternatives for worker thread states, since -by virtue of their independence- are not bound to be the source of intermittent faults (cf. Fig. 10).

Pruning state subtrees can be configured by specifying the depth of the state tree at which pruning will occur and the order of the child nodes at this level to be allowed: at the present state of development, pruning is regulated via the properties listed in table 2. At runtime, when the listener detects that a subtree should be pruned it executes the statement ***`ti.breakTransition(true);`***, which breaks the current transition and forces an end state.

Pruning the state subtrees of specific nodes contributes to the alleviation of the state explosion problem, since the state space that is explored within the program execution is reduced. The gains regarding the aspect of state space size reduction are quantified through the experiments presented in subsection 6.3.

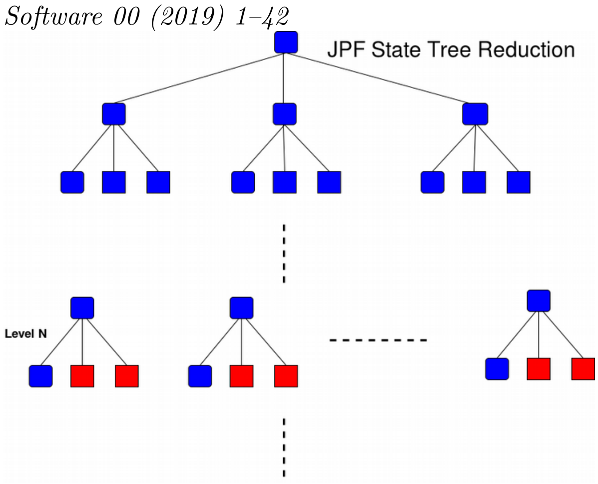


Figure 10. JPF State Tree Reduction : Pruning subtrees of nodes at Level N by allowing execution of the first child of each node only.

| Property name | Description |
|---|---|
| vm.parallel.allow.depth | the level of the nodes where the reduction will be applied (single value or a range) |
| vm.parallel.allow.child | the order of the child node(s) at the specified level that will be allowed to continue (e.g. the value "2" designates that the 2nd child will be allowed to continue, while execution of other children will be inhibited) |

Table 2. Parameters regulating the pruning of state subtrees

### *5.3. Exploiting processing power in share-nothing architectures*

The proposed method is orchestrated on top of JPF, and can thus benefit from JPF's potential to run efficiently on shared memory architectures, exploiting multiple execution cores for accelerating the state space search procedure [55]. To further scale parallelism potential and take advantage of share-nothing architectures, the user could designate specific JPF paths whose exploration would be assigned to a different machine. More specifically:

- the listener examines all transitions
- when a path that is designated to be transferred to another machine is

reached, the state space is serialized and transferred over the network to the destination machine

- at the local machine further exploration is inhibited by means of the `ti.breakTransition(true);` statement, which breaks the current transition and forces an end state.

- On the remote machine, the state is deserialized, and execution resumes from the point that it was suspended; in this case, the listener does not issue the `ti.breakTransition(true);` statement, allowing the exploration of the path.

The state serialization, transfer and execution resumption mechanisms are currently under implementation.

## 6. Experimental evaluation

In this section, we present the experiments conducted to:

1. validate the proposed algorithm in terms of its fault detection potential,
2. experimentally assess the complexity of the algorithm and quantify the overhead introduced over the "plain JPF" software validation and
3. assess and quantify the gains reaped from applying the optimization methods presented in section 5.

### *6.1. Algorithm validation*

#### *6.1.1. Small-scale validation*

Initially, when we ran the non-extended version of JPF against the code illustrated in listing 1 using three parallel threads. The validation succeeds without identifying any potential error sources, exhibiting thus a false negative.

Then we run the proposed algorithm to generate data access traces, applying the rules *read(s,T1)-write(s,T2)-write(s,T1)* and *read(s,T1)-write(s,T2)-read(s,T1)*, which can identify data access patterns that are potentially erroneous. After the processing of the tree, which has been generated via our listener, the following instruction interleavings are identified as possible intermittent fault causes:

1. $t_1(1) - t_2(3) - t_1(3)$

2. $t_1(1) - t_2(4) - t_1(3)$
3. $t_1(1) - t_2(4) - t_1(4)$

where $t_i(j)$ denotes the execution of instruction $j$ (cf. code example in Section 3) by thread $i$.

The user is invited to review the relevant code and accept or reject those proposals. In the above case, the first case flagged by the algorithm is a source of intermittent faults, since it may lead to the violation of the $filled < MAXNUM$ program invariant, as we would like in every case before executing line (3) to have a shared variable which is smaller than $MAXNUM$ ($filled < MAXNUM$). The two last sequences do not generate an intermittent fault in our case, however it is worth noting that when the instructions `l.unlock();` (line (2) in listing 1) as well as the corresponding $l.lock()$ instruction immediately preceding line (3) in the same listing, effectively thus removing the atomicity violation which is the root cause of the error flagged in the first case, the second error flagging are removed and only the third one is reported.

Our approach is able to identify previously missed intermittent faults. On the other hand, it introduces some false positives. An annotation-based approach could be used to inhibit the reporting of specific patterns that have been validated not to cause intermittent faults.

#### *6.1.2. Validation in real-world scale*

To validate our approach in a real-world scale, we conducted experiments using the multithreaded Java websever available in [56], which extensively uses multithreading (using a thread pool of configurable size) and shared variables. We initially ran the non-extended version of JPF against the simulation code given in GitHub [57], and no errors were flagged. The code was also checked by the proposed algorithm, and no errors were flagged either.

Subsequently, we followed a fault injection approach [8], to inject faults within the code and test whether these faults are detected by (a) the non-extended versions of the JPF and (b) the proposed algorithm (i.e. JPF extended with our listener and the potentially erroneous access pattern detection). The results of the tests are summarized in table 3.

Effectively, the proposed algorithm was able to detect all faults detected by JPF (which underpins the proposed algorithm), plus errors related to erroneous access patterns, which were missed by JPF. Therefore, the proposed algorithm offers more comprehensive error detection, at the expense

| Fault type | JPF | Proposed algorithm |
|---|---|---|
| Deadlock | Yes | Yes |
| Unhandled exception | Yes | Yes |
| Race conditions | Yes | Yes |
| Application-specific assertions | Yes | Yes |
| Erroneous shared variable access patterns as in listing 1 | No | Yes; for the injected erroneous access pattern faults, one related false positive was also raised |

Table 3. Detection of injected faults by JPF and the proposed algorithm

of flagging a limited number of false positives and a performance overhead, which has been quantified to be up to 10.7%, as discussed in subsection 6.2. Recall here that the potential of the proposed algorithm to detect erroneous shared variable access patterns is advantageous over the detection of errors based on application-specific assertions, in that (a) the former does not necessitate any instrumentation by the programmer (i.e. insertion of *assert* statements), while the latter does and (b) assertion-based error detection is limited to detecting errors at the locations that assertions have been inserted and related to the conditions within the assertions, whereas erroneous shared variable access pattern detection can identify errors at any location and under any condition.

An instance of the code of [56] with injected faults is available at [58] [2]

### 6.2. Complexity Assessment Experiments

In this subsection, we report on the experiments conducted to gain insight regarding the size of the state space and the execution time needed under different thread mixtures, and present the obtained metrics. To promote example clarity, we initially present a complexity analysis on the code depicted in listing 3, while subsequently we present our complexity analysis findings on our real world-scale example of the multithreaded Java websever [56]. All the experiments reported in this subsection, as well as in the following one, have been performed on a PowerEdge M910 blade server, with

[2]At different phases of the test, different faults were injected; [58] contains a specific set of faults.

256 GBytes of physical memory and four 8-core E7-4830 Intel Xeon processors. The Java environment had been configured to use up to 40 GBytes of memory.

The example in listing 3 entails instructions belonging to three threads, namely *T1*, *T2* and *T3*. Threads *T1* and *T2* access two shared variables *A* and *B*, therefore the interleaving of their instructions can be the root cause of intermittent fault occurrence. On the contrary, thread T3 accesses only local variables, and consequently no intermittent faults can occur due to the instructions of this thread.

Listing 3. Thread code

```
shared int A, B;

T1:
  B = 2;
  A = B + 1;

T2:
  B = 0;
  B = B + 2;
  A = B + 1;

T3:
  local int c, d;
  d = 0;
  d = d + 2;
  c = d + 1;
```

In terms of shared variable read and write operations, threads *T1* and *T2* can be written as:

Listing 4. Thread read and write operations

```
T1: w(B), r(B), w(A)
T2: w(B), r(B), w(B), r(B), w(A)
T3: -
```

Some possible execution schedules of threads *T1* and *T2* are presented in the following list. For conciseness, we have only included those execution schedules which end with the last instruction of thread *T1*; inclusion of cases that end with the last instruction of *T2* is done in an identical fashion.

The instructions of *T1* (i.e. instructions of thread *T1* for which at least one instruction of *T2* intervenes between them and the last instruction of *T1*)are denoted using boldface, to promote readability.

For the creation of the execution schedules presented in the following list, we assume a simple processor addressing mode, where each instruction can fetch access only one memory location (corresponding to a variable), fetching its contents to a register or storing register contents to it. This is in-line with the Java model, where instructions operate on operands individually stored on the operand stack, and results are then copied to variables[3]. In this sense, read and write operations executed in the context of the same instruction (e.g. *r(B)T1*, *w(A)T1* corresponding to the instruction *A = B + 1;* are separable, in the sense that thread switching can occur between these two accesses. However, in some processors it is possible to execute multiple variable accesses in a single instruction: for instance in the Pentium it is possible to map the program instruction *B = B + 2;* to a single machine-language instruction *ADD WORD PTR [ESI],0x2* (assuming that the ESI register points to the memory location of variable *B*) [59]. Notably, this can happen when Java bytecode is compiled into optimized machine instructions e.g. through the Java HotSpot VM[4]. In these cases, execution schedules in which the involved read and write operations appear separated cannot occur and therefore should not be considered; henceforth, the following list contains only execution schedules where the operations *r(B)T2* and *w(B)T2* realizing the *B = B + 2;* instruction of thread *T2* are adjacent.

1. w(B)T2, r(B)T2, w(B)T2, r(B)T2, w(A)T2, w(B)T1, r(B)T1, w(A)T1
2. **w(B)T1**, w(B)T2, r(B)T2, w(B)T2, r(B)T2, w(A)T2, r(B)T1, w(A)T1
3. w(B)T2, **w(B)T1**, r(B)T2, w(B)T2, r(B)T2, w(A)T2, r(B)T1, w(A)T1
4. w(B)T2, r(B)T2, w(B)T2, **w(B)T1**, r(B)T2, w(A)T2, r(B)T1, w(A)T1
5. w(B)T2, r(B)T2, w(B)T2, r(B)T2, **w(B)T1**, w(A)T2, r(B)T1, w(A)T1
6. **w(B)T1**, **r(B)T1**, w(B)T2, r(B)T2, w(B)T2, r(B)T2, w(A)T2, w(A)T1
7. **w(B)T1**, w(B)T2, **r(B)T1**, r(B)T2, w(B)T2, r(B)T2, w(A)T2, w(A)T1
8. **w(B)T1**, w(B)T2, r(B)T2, w(B)T2, **r(B)T1**, r(B)T2, w(A)T2, w(A)T1
9. **w(B)T1**, w(B)T2, r(B)T2, w(B)T2, r(B)T2, **r(B)T1**, w(A)T2, w(A)T1
10. w(B)T2, **w(B)T1**, **r(B)T1**, r(B)T2, w(B)T2, r(B)T2, w(A)T2, w(A)T1

---

[3] `https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html`

[4] `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html`

11. w(B)T2, **w(B)T1**, r(B)T2, w(B)T2, **r(B)T1**, r(B)T2, w(A)T2, w(A)T1
12. w(B)T2, **w(B)T1**, r(B)T2, w(B)T2, r(B)T2, **r(B)T1**, w(A)T2, w(A)T1
13. w(B)T2, r(B)T2, w(B)T2, **w(B)T1**, **r(B)T1**, r(B)T2, w(A)T2, w(A)T1
14. w(B)T2, r(B)T2, w(B)T2, **w(B)T1**, r(B)T2, **r(B)T1**, w(A)T2, w(A)T1
15. w(B)T2, r(B)T2, w(B)T2, r(B)T2, **w(B)T1**, **r(B)T1**, w(A)T2, w(A)T1

Table 4 depicts the experimental results obtained from running the code in listing 3 with a varying number of instances of threads *T1*, *T2* and *T3*. The experimental results are contrasted with the theoretical maximum of possible paths, which is equal to the number of possible distinct execution schedules (c.f. Section 4).

| Complexity Comparison Table | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 2 Threads (1xT1, 1xT2) | 3 Threads (1xT1, 1xT2, 1xT3) | 4 Threads (1xT1, 1xT2, 2xT3) | 4 Threads (2xT1, 2xT2) | 5 Threads (1xT1, 1xT2, 3xT3) | 5 Threads (2xT1, 2xT2, 1xT3) | 6 Threads (1xT1, 1xT2, 4xT3) | 4 Threads (2xT1, 2xT2, 2xT3) |
| Theoretical number of possible paths | 56 | 56 | 56 | 40360320 | 56 | 40360320 | 56 | 40360320 |
| Experimentally determined number of paths (JPF) | 13 | 18 | 63 | 49766 | 63 | 90627 | 63 | 154081 |
| Practical number of final states (JPF) | 2 | 2 | 2 | 3 | 2 | 3 | 2 | 3 |
| Experimentally determined number of final states (JPF) | 4 | 4 | 4 | 11 | 4 | 11 | 4 | 11 |
| JPF end states | 29 | 33 | 37 | 331 | 41 | 342 | 45 | 353 |

Table 4. Complexity comparison for varying number of threads

At this point, we note the following :

1. For the number of possible paths, we assume all possible execution schedules regarding instructions accessing shared variables, as calculated by equation 2.
2. The metric *Practical number of final states* corresponds to the number of the different program results, in terms of shared variable values.

3. The difference between the *Practical number of final states* and the *JPF end states* exists because of the additional information regarding shared data used in JPF (e.g thread shared information).

Regarding the complexity analysis experiments conducted using our real world-scale example of the multithreaded Java websever [56], table 5 depicts the execution statistics of the multithreaded Java websever [56] regarding the number of states, and table 6 depicts the runtimes measured, while varying the parameter of the execution tree depth search limit (c.f. subsection 5.2). In all experiments, all injected faults were flagged, while in the setting where the JPF search depth was set to 350 it was observed that the limit was not reached within the fault detection process execution, indicating that further increments to that parameter would not affect the time and resources needed. We can observe that the overhead introduced by the proposed algorithm over the non-extended version of JPF is up to 10.7%, which is deemed acceptable, considering the increased fault detection potential of the proposed algorithm.

| | |
|---|---|
| new states | 35,185,856 |
| visited states | 24,779,490 |
| backtracked states | 59,965,346 |
| end states | 0 |
| instructions | 499,329,9768 |
| max memory | 30,7 GB |

Table 5. Execution statistics for the fault detection process of the multithreaded Java websever

where:

- *new states* is the number of unique states visited during the run;
- *visited states* is the number of states that are examined and have been revisited during the same execution;
- *backtracked states* refers to the states from which the search backtracked, so as to examine different paths;
- *end states* refers to the concluding states of the program execution, from which there are no forward transitions to try.

| JPF execution time | | | |
|---|---|---|---|
| | JPF search depth=120 | JPF search depth=240 | JPF search depth=350 |
| JPF (not extended) | 01:27:19 | 04:01:59 | 04:05:09 |
| Proposed algotithm | 01:34:25 | 04:26:15 | 04:31:26 |

Table 6. Fault detection process execution time for Java web server Simulation

### *6.3. Optimization Experiments*

In this subsection we report on the experiments conducted to assess the gains introduced by the optimization methods presented in section 5, and present our findings. As noted above, all the experiments reported in this subsection have been performed on a PowerEdge M910 blade server, with 256 GBytes of physical memory and four 8-core E7-4830 Intel Xeon processors. The Java environment had been configured to use up to 40 GBytes of memory.

Table 7 illustrates the performance gains obtained by isolating threads that are not bound to be involved in the occurrence of intermittent faults, regarding the code depicted in listing 3. In more detail, the first data column in Table 7 corresponds to the measurements obtained from the execution of a program whose main thread creates one instance of threads *T1* and *T2*, as well as four instances of *T3* (1xT1, 1xT2, 4xT3); in this execution, JPF monitors all threads. The second data column in Table 7 corresponds to the execution of the same program, with JPF being however instructed to monitor only the main thread and the instances of threads *T1* and *T2*, since no instance of thread *T3* is bound to be involved in the generation of intermittent faults. As described in Section 5.1, this is realized through the setting *vm.watched.threads=main,1,2*.

Table 8 depicts how performance benefits can be obtained from applying the Children Node Reduction technique described in Section 5.2, regarding the code depicted in listing 3. The figures in this table refer to the execution of a java program with 6 threads (1xT1, 1xT2, 4xT3), varying the order of the child that is allowed to continue. Since in our example the first and second children correspond to executions of instructions by threads *T1* and *T2*, which include accesses to shared variables, these choices entail more states to be examined. Given that only these two choices may actually lead to intermittent faults, it suffices to examine only these two cases to fully uncover all intermittent fault root causes.

| watched threads | All Threads | main,T1,T2 |
|---|---|---|
| elapsed time | 00:00:10 | 00:00:03 |
| new states | 5147 | 2192 |
| visited states | 14365 | 1264 |
| backtracked states | 19512 | 3456 |
| end states | 45 | - |
| instructions | 418886 | 88756 |
| max memory | 303MB | 169MB |

Table 7. Examining all possible threads vs. limiting the set of threads examined by JVM.

| Children Node Reduction | | | | |
|---|---|---|---|---|
| | 1st child node for depths between 10-30 | 2nd child node for depths between 10-30 | 3rd child node for depths between 10-30 | No cut off |
| Time | 2 sec | 1 sec | 1 sec | 10 sec |
| New states | 1637 | 523 | 302 | 5147 |
| Visited states | 1118 | 344 | 253 | 14365 |
| Backtracked states | 2755 | 867 | 555 | 19512 |

Table 8. Children Node Reduction effect applied at different thread orders

Table 9 focuses on the scalability of the proposed algorithm under the optimization techniques presented in section 5, depicting the time needed to execute the proposed algorithm to detect faults injected to the open-source multithreaded Java web sever [56] when the thread partitioning and the state subtree pruning of specific nodes techniques (cf. subsections 5.1 and 5.2, respectively) are applied. The configuration used in this experiment is:

```
vm.parallel.allowed.depth=40-350
vm.watched.threads=main,Thread-1,Thread-2,Thread-3,Thread-4
vm.parallel.allowed.child=[1] search.depth_limit = 350
```

This configuration effectively scans the full state tree up to the depth of 40, and beyond that point limits the detection to the first child only,

| Number of threads in the web server request executor thread pool | | | | |
|---|---|---|---|---|
| | 5 | 10 | 50 | 100 |
| Elapsed time | 00:32:41 | 00:47:04 | 00:47:41 | 00:47:54 |

Table 9. JPF execution time for the Java web server simulation

since the Java web server [56] employs the worker thread model discussed in section 5.2 [54], and moreover worker threads are totally independent and are thus not bound to generate any intermittent errors.

We can observe that the time needed to run the fault detection algorithm increases very slowly with the overall number of threads, while additionally significant time savings against the non-optimized version (c.f. table 6) are introduced; these savings are quantified to 82%.

## 7. Conclusions and Future work

In this paper we presented a methodology for intermittent fault detection that is based on the identification of suspicious shared variable access patterns in the code execution traces. Execution traces are generated using the JPF tool, which has been enhanced by a customized listener, while the suspicious access patterns that are searched for correspond to well-known parallel programming hazards. Out method has been shown to be capable of detecting intermittent faults that evade detection when other methods are used, while on the other hand introducing some false positives. In this sense, the programmer is asked to review the potential intermittent fault root causes and accept or reject them. In order to leverage the efficiency of the proposed method, we have introduced optimization methods which can exploit structural properties of the code, such as thread independence and thread subtree isolation, as well as parallel hardware capabilities. Our experiments on optimizations exploiting structural properties of the code have demonstrated that significant performance gains can be reaped.

In the context of our future work we plan to examine the following dimensions:

1. fully implement and evaluate the optimization method for exploiting parallel hardware capabilities.
2. take into account dependencies between global and local variables, which are established via assignment statements.
3. Identify and evaluate additional dependency rules.

4. study the computational complexity of the proposed algorithm, computing a theoretical upper bound for the number of possible execution paths that need to be explored.

## References


[1] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Securure Computing 1 (1) (2004) 11–33. doi:10.1109/TDSC.2004.2.
URL http://dx.doi.org/10.1109/TDSC.2004.2

[2] S. Mukherjee, Architecture Design for Soft Errors, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[3] J. Gray, Why do computers stop and what can be done about it?, Technical Report TR 85.7, Tandem Computers, [Online; accessed 13-July-2019] (1985).

[4] M. Grottke, K. S. Trivedi, Fighting bugs: remove, retry, replicate, and rejuvenate, Computer 40 (2) (2007) 107–109. doi:10.1109/MC.2007.55.

[5] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, S. Russo, Analysis and prediction of mandelbugs in an industrial software system, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 262–271. doi:10.1109/ICST.2013.21.

[6] M. Winslett, Bruce lindsay speaks out: On system r, benchmarking, life as an ibm fellow, the power of dbas in the old days, why performance still matters, heisenbugs, why he still writes code, singing pigs, and more, SIGMOD Rec. 34 (2) (2005) 71–79. doi:10.1145/1083784.1083803.
URL http://doi.acm.org/10.1145/1083784.1083803

[7] M. Grottke, A. P. Nikora, K. S. Trivedi, An empirical investigation of fault types in space mission system software, in: 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN), 2010, pp. 447–456. doi:10.1109/DSN.2010.5544284.

[8] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, K. S. Trivedi, Fault triggers in open-source software: An experience report, in: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 2013, pp. 178–187. doi:10.1109/ISSRE.2013.6698917.

[9] R. Chillarege, Comparing four case studies on bohr-mandel characteristics using odc, in: 2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2013, pp. 285–289. doi:10.1109/ISSREW.2013.6688908.

[10] J. S. Bradbury, K. Jalbert, Defining a Catalog of Programming Anti-Patterns for Concurrent Java, in: Proceedings of the 3rd International Workshop on Software Patterns and Quality, SPAQu'09, 2009, pp. 6–11.

[11] J. Duffy, Solving 11 Likely Problems In Your Multithreaded Code, MSDN Magazine (October 2008).

[12] G. Gopalakrishnan, J. Sawaya, Achieving formal parallel program debugging by incentivizing cs/hpc collaborative tool development, in: Proceedings of the 1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models, SCREAM '15, ACM, New York, NY, USA, 2015, pp. 11–18.

doi:10.1145/2753524.2753531.
URL http://doi.acm.org/10.1145/2753524.2753531

[13] P. C. Mehlitz, W. Visser, J. Penix, The JPF Runtime Verification System, http://www.doc.gold.ac.uk/%7Emas01sd/classes/jpf_release/doc/JPF.pdf (2005).

[14] A. Mahmood, E. J. McCluskey, Concurrent error detection using watchdog processors-a survey, IEEE Transactions on Computers 37 (2) (1988) 160–174. doi:10.1109/12.2145.

[15] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante, Soft-error detection using control flow assertions, in: Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems, 2003, pp. 581–588. doi:10.1109/DFTVS.2003.1250158.

[16] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, A watchdog processor to detect data and control flow errors, in: 9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003., 2003, pp. 144–148. doi:10.1109/OLT.2003.1214381.

[17] A. Li, B. Hong, Software implemented transient fault detection in space computer, Aerospace Science and Technology 11 (2) (2007) 245 – 252. doi:https://doi.org/10.1016/j.ast.2006.06.006.
URL http://www.sciencedirect.com/science/article/pii/S1270963806000800

[18] C. Flanagan, P. Godefroid, Dynamic Partial-Order Reduction for Model Checking Software, ACM SIGPLAN Notices - Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages 1 (40) (2005) 110 – 121.

[19] T. Sharma, D. Spinellis, A survey on software smells, Journal of Systems and Software 138 (2018) 158 – 173. doi:https://doi.org/10.1016/j.jss.2017.12.034.
URL http://www.sciencedirect.com/science/article/pii/S0164121217303114

[20] M. S. Haque, J. Carver, T. Atkison, Causes, impacts, and detection approaches of code smell: A survey, in: Proceedings of the ACMSE 2018 Conference, ACMSE '18, ACM, New York, NY, USA, 2018, pp. 25:1–25:8. doi:10.1145/3190645.3190697.
URL http://doi.acm.org/10.1145/3190645.3190697

[21] S. S. Rathore, S. Kumar, A study on software fault prediction techniques, Artificial Intelligence Review 51 (2) (2019) 255–327. doi:10.1007/s10462-017-9563-5.
URL https://doi.org/10.1007/s10462-017-9563-5

[22] P. Runeson, A survey of unit testing practices, IEEE Software 23 (4) (2006) 22–29. doi:10.1109/MS.2006.91.

[23] V. V. Kuliamin, A. A. Petukhov, A survey of methods for constructing covering arrays, Programming and Computer Software 37 (3) (2011) 121. doi:10.1134/S0361768811030029.
URL https://doi.org/10.1134/S0361768811030029

[24] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, P. Pettersson, Automated test generation using model checking: an industrial evaluation, International Journal on Software Tools for Technology Transfer 18 (3) (2016) 335–353. doi:10.1007/s10009-014-0355-9.
URL https://doi.org/10.1007/s10009-014-0355-9

[25] A. Salahirad, H. Almulla, G. Gay, Choosing the fitness function for the job: Au-

tomated generation of test suites that detect real faults, Software Testing, Verification and Reliability 29 (4-5) (2019) e1701, e1701 stvr.1701. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1701, doi:10.1002/stvr.1701.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1701

[26] A. Schwartz, D. Puckett, Y. Meng, G. Gay, Investigating faults missed by test suites achieving high code coverage, Journal of Systems and Software 144 (2018) 106 – 120. doi:https://doi.org/10.1016/j.jss.2018.06.024.
URL http://www.sciencedirect.com/science/article/pii/S0164121218301201

[27] B. S. Ahmed, Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing, Engineering Science and Technology, an International Journal 19 (2) (2016) 737 – 753. doi:https://doi.org/10.1016/j.jestch.2015.11.006.
URL http://www.sciencedirect.com/science/article/pii/S2215098615001706

[28] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, R. Tumeng, Test case prioritization approaches in regression testing: A systematic literature review, Information and Software Technology 93 (2018) 74 – 93. doi:https://doi.org/10.1016/j.infsof.2017.08.014.
URL http://www.sciencedirect.com/science/article/pii/S0950584916304888

[29] S. M. Ghaffarian, H. R. Shahriari, Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey, ACM Comput. Surv. 50 (4) (2017) 56:1–56:36. doi:10.1145/3092566.
URL http://doi.acm.org/10.1145/3092566

[30] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering 42 (8) (2016) 707–740. doi:10.1109/TSE.2016.2521368.

[31] N. Bazan, Static and Dynamic Verification Tools, https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-and-dynamic-verification-tools/, [Online; accessed 24-March-2019] (2017).

[32] M. Felleisen, R. Cartwright, Safety as a metric, in: Proceedings 12th Conference on Software Engineering Education and Training (Cat. No.PR00131), 1999, pp. 129–131. doi:10.1109/CSEE.1999.755192.
URL https://doi.org/10.1109/CSEE.1999.755192

[33] Wikipedia, Satisfiability modulo theories, https://en.wikipedia.org/wiki/Satisfiability_modulo_theories, [Online; accessed 24-March-2019] (2019).

[34] N. Machado, B. Lucia, L. Rodrigues, Production-guided Concurrency Debugging, ACM SIGPLAN Notices - PPoPP '16 51 (8) (August 2016).

[35] N. Machado, B. Lucia, L. Rodrigues, Concurrency Debugging with Differential Schedule Projections, ACM SIGPLAN Notices - PLDI 15 50 (6) (2015) 586–595.

[36] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, I. Neamtiu, Finding and reproducing heisenbugs in concurrent programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 267–280.

URL http://dl.acm.org/citation.cfm?id=1855741.1855760

[37] F. Koca, H. Sözer, R. Abreu, Spectrum-based fault localization for diagnosing concurrency faults, in: H. Yenigün, C. Yilmaz, A. Ulrich (Eds.), Testing Software and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 239–254.

[38] R. Abreu, P. Zoeteweij, A. J. C. v. Gemund, Spectrum-based multiple fault localization, in: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE 09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 88–99. doi:10.1109/ASE.2009.25.
URL https://doi.org/10.1109/ASE.2009.25

[39] S. Park, S. Lu, Y. Zhou, Ctrigger: Exposing atomicity violation bugs from their hiding places, SIGPLAN Not. 44 (3) (2009) 25–36. doi:10.1145/1508284.1508249.
URL http://doi.acm.org/10.1145/1508284.1508249

[40] C. S. Păsăreanu, W. Visser, Symbolic execution and model checking for testing, in: K. Yorav (Ed.), Hardware and Software: Verification and Testing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 17–18.

[41] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, J. ACM 50 (5) (2003) 752–794. doi:10.1145/876638.876643.
URL http://doi.acm.org/10.1145/876638.876643

[42] C. B. Bergersen, Java Path Finder, https://www.uio.no/studier/emner/matnat/ifi/INF5140/v15/slides/jpf.pdf, jPF (2015).

[43] K. Y. Rozier, Survey: Linear temporal logic symbolic model checking, Comput. Sci. Rev. 5 (2) (2011) 163–203. doi:10.1016/j.cosrev.2010.06.002.
URL http://dx.doi.org/10.1016/j.cosrev.2010.06.002

[44] NASA, Java pathfinder, https://github.com/javapathfinder/jpf-core, [Online; accessed 13-July-2019] (2017).

[45] A. Malkis, A. Podelski, A. Rybalchenko, Precise thread-modular verification, SAS'07 Proceedings of the 14th international conference on Static Analysis (2007) 218–232.

[46] NASA, On-the-fly Partial Order Reduction, http://javapathfinder.sourceforge.net/On-the-fly_Partial_Order_Reduction.html, [Online; accessed 24-March-2019] (2009).

[47] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, S. K. Rajamani, Partial-order reduction in symbolic state space exploration, in: O. Grumberg (Ed.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 340–351.

[48] R. Büschkes, M. Borning, D. Kesdogan, Transaction-based anomaly detection, ID'99 Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring 1 (April 1999).

[49] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, B. Saha, Enforcing isolation and ordering in STM, ACM SIGPLAN Notices - Proceedings of the 2007 PLDI conference 42 (6) (2007) 78–88.

[50] M. Mansouri-Samani, P. Mehlitz, C. Pasareanu, J. Penix, G. Brat, L. Markosian, O. O'Malley, T. Pressburger, W. Visser, Program Model Checking: A Practitioner's Guide, https://ti.arc.nasa.gov/m/pub-archive/1439h/1439%20(Mansouri-Samani).pdf, [Online; accessed 11-July-2019] (2012).

[51] NASA, Java pathfinder: A model checker for java programs, `https://ti.arc.nasa.gov/tech/rse/vandv/jpf/`, [Online; accessed 11-July-2019] (2012).

[52] H. R. Lewis, C. H. Papadimitriou, Elements of the Theory of Computation, 2nd Edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[53] I. U. Plale, Thread Design Patterns, `https://www.cs.indiana.edu/classes/b534-plal/ClassNotes/thread-design-patterns4.pdf`, [Online; accessed 26-March-2019] (2001).

[54] A. S. Tanenbaum, M. van Steen, Distributed systems - principles and paradigms, 2nd Edition, Pearson Education, 2007.

[55] P. Parizek, T. Kalibera, Verifying nested lock priority inheritance in rtems with java pathfinder, in: International Conference on Formal Engineering Methods, ICFEM 2016, Springer, Cham, 2016, pp. 417—432. `doi:https://doi.org/10.1007/978-3-319-47846-3_26`.
URL `https://link.springer.com/chapter/10.1007/978-3-319-47846-3_26`

[56] "djessup" GitHub user, Java web server, `https://github.com/djessup/java-webserver`, [Online; accessed 29-July-2019] (2016).

[57] P. Sotiropoulos, Java web server simulation without injected faults, `https://github.com/pansot2/java-webserver/tree/simulation`, [Online; accessed 29-August-2019] (2019).

[58] P. Sotiropoulos, Java web server with injected faults, `https://github.com/pansot2/java-webserver/tree/jpf-simulation`, [Online; accessed 29-August-2019] (2019).

[59] S. Dandamudi, Addressing modes, in: T. editor (Ed.), Introduction to Assembly Language Programming, Undergraduate Texts in Computer Science, Springer, New York, NY, 1998, Ch. 5, pp. 173–206.