Department of Distributed and Dependable Systems
Technical report no. D3S-TR-2022-01
July 25, 2022

Incremental Verification of Multithreaded Programs by Checking Interleavings for Pairs of Threads

Pavel Parízek, Filip Kliber

Abstract: Many techniques of automated verification target multithreaded programs, because subtle interactions between threads may trigger concurrency errors such as deadlocks and data races. While many techniques and tools involving systematic exploration of the whole space of possible thread interleavings have already been developed, a great majority of them does not scale to large real-world software systems, despite various clever algorithmic optimizations. A viable approach is to use incremental verification techniques that, in each run, focus just on the recently modified code and the relatively small number of affected execution traces, and therefore can provide analysis results (including bug reports) very quickly.

We present a new algorithm for incremental verification of multithreaded programs that is based on the *pairwise approach*. The key idea of the pairwise approach is systematic exploration of possible thread interleavings just for specific pairs of threads, such that behavior of the subject program is analyzed separately for every relevant thread pair.

We implemented the algorithm with Java Pathfinder as the backend verification tool, and evaluated it on several multithreaded Java programs. Results show that our incremental algorithm (1) can find errors very fast, (2) greatly reduces time needed for complete safety verification, and (3) it can find the same errors as full verification of the whole state space. Performance of our prototype implementation of the algorithm is also comparable with state-of-the-art techniques based on static analysis.

This work was partially supported by the Czech Science Foundation project 20-07487S and partially supported by the Charles University institutional funding project SVV 260588.

1 Introduction

An important subject of automated verification and bug detection techniques are multithreaded programs, especially because subtle interactions between threads, unforeseen by the developers, may trigger concurrency errors such as deadlocks and data races (atomicity violations in general) during the program execution. Techniques most suitable for precise detection of possible concurrency errors involve systematic exploration of the whole space of all possible thread interleavings that may occur at runtime. While many techniques and tools in this category have already been developed (cf. [22, 32, 6, 11]), exhaustive systematic verification does not scale to large real-world multithreaded software systems due to state space explosion [4]. The main cause of limited scalability is the huge number of thread interleavings that have to be analyzed even for rather small multithreaded programs. State-of-the-art techniques and tools partially address this challenge by using many clever algorithmic improvements, optimizations and heuristics, including different variants of partial order reduction [1, 9, 10], thread-modular verification [8], iterative context-bounded verification [21], and symbolic predictive analysis based on dynamic event recording [33]. Nevertheless, scalability of verification to large and complex software systems remains to be an issue.

One viable approach is to use incremental verification techniques that, in each run, analyze just the recently modified source code (differences from the previous version) and the corresponding small subset of affected thread interleavings. The main practical benefit of such incremental procedures is that they can be run very often — for example, after each commit to a source code repository, or possibly even on-the-fly as a background process in an IDE while developers are editing the source code — and they provide results in the form of a list of detected bugs very quickly, allowing the developers to fix the reported bugs while they have the respective code in fresh memory. Several research groups already published some work in this direction [2, 13, 19, 35], targeting also efficient incremental search for concurrency errors.

Overview. In this paper, we present an algorithm for incremental verification of multithreaded programs that is based on systematic exploration of possible thread interleavings just for specific pairs of threads. We call it the *pairwise approach*. The key idea behind our pairwise approach is that, when developers edit the code of a specific program thread, it is sufficient to check all possible interleavings of the modified code with other threads just in a pairwise manner — that means checking the interleavings for each relevant pair of threads separately. Every run of our verification algorithm focuses just on the most recent source code modification, that means on detecting possible concurrency errors that involve the modified code, assuming the previous version of the input program was already verified earlier and deemed free of concurrency errors.

For the purpose of explaining our approach here and in the rest of this paper, we consider only those source code modifications that involve program statements that represent possible interaction between threads — reading or writing a field of a shared heap object, reading or writing an element of a shared array, lock acquire statement, lock release, starting of a new thread, and other means of thread synchronization. In this paper, we use the phrase *thread interaction statements* to denote such statements. Thread interaction statements may be identified using static analysis that computes over-approximate sets of shared heap objects (fields, array elements) [29, 24, 25].

A run of the incremental verification procedure based on our pairwise approach is triggered when the developer inserts or deletes a thread interaction statement to/from the program source code. By starting the verification procedure only upon a source code modification that involves some thread interaction statements, we avoid the issue of running the verification procedure too frequently (e.g., after each inserted character or keyword). As the first step, the verification procedure then identifies all static program threads that may execute the modified code fragment. We denote this set of threads by the term *modified threads*. Subsequently, the procedure checks all possible interleavings for every pair of threads, where at least one element of a pair belongs to the set of modified threads. In this way, our verification procedure checks all possible interactions of the modified source code fragment (affected block of program statements) in the respective threads with the current version of the source code in all other threads, and performs the checks in a pairwise manner. Note that, unlike some other existing techniques (cf. [13, 35, 14]), our algorithm does not have to compute the set of possibly affected thread interleavings (schedules) because it explores all interleavings between the relevant pairs of threads in order to search for concurrency errors.

For illustration, consider the program in Figure 1, which involves three static threads (T1, T2, and T3). Two dynamic instances of T2 are created at runtime, while just one instance is created for each of the other threads T1 and T3. When the developer adds a call of print(o.f) into the code of T2, which is

underlined in the source code listing, then our incremental verification algorithm checks the following pairs of threads: (T2 T1), (T2 T2), (T2 T3). The pair (T2 T2) has to be analyzed because the program involves two dynamic instances of thread T2.

```
1 T1: o.f = x ; t2a = start T2 ; t2b = start T2
2 T2: evaluate(o) ; print(o.f)
3 T3: y = p.g ; z = r.h; w = y + z
```

Figure 1: *Example program*

We also want to emphasize that, even though our verification algorithm checks all interleavings just for each pair of threads separately, it reports all concurrency errors that involve the modified code — and, in particular, even all errors in synchronization patterns that involve more than two concurrent threads. Details are provided in Section 3.4.

One important practical aspect related to usage of this incremental verification procedure is that the developer may continue to edit the program source code while the procedure runs in the background. When the analyzed code is edited again, the verification results might be obsolete and therefore the currently running verification process should be terminated or restarted. However, for the purpose of explaining our algorithm in the paper, we ignore this scenario.

Contribution. The main research contributions presented in this paper include:

- Algorithm for incremental verification of multithreaded programs based on exploring all possible interleavings just for specific pairs of threads.
- Experimental evaluation of the prototype implementation of our algorithm within Java Pathfinder [38] on 9 multithreaded Java programs.

Results of experiments that we performed show that runs of the incremental verification procedure finish very quickly for every subject program and every source code modification that we considered in our evaluation. Its performance is comparable with state-of-the-art techniques based on static analysis. For the most complex program in our benchmark set, the verification procedure finished on average in 16 seconds (when considering just experiments focused on fast search for concurrency errors), respectively in 18 seconds (experiments focused on checking safety). In addition, our incremental verification algorithm can find the same errors as if the full verification of the whole program state space is run every time.

In the next section, we define important terminology and notation used within the rest of this paper. Technical details of our pairwise approach to incremental verification are explained in Section 3. We present the results of our experimental evaluation in Section 4 and discuss related work in Section 5. Finally, in Section 6 we provide the concluding remarks.

2 Notation and Terminology

We define *thread interaction statements* more precisely as follows. It is any program statement that accesses (reads or updates) a memory location reachable from multiple threads. The memory location can be an object field, an array element, or a lock status variable, for example. Execution of a thread interaction statement represents possible exchange of information between multiple threads, or their mutual synchronization.

An atomic *transition* in the program state space is defined as a sequence of statements that (i) begins with a thread interaction statement and (ii) then contains any number of thread-local statements, whose effects are not visible to other threads. All statements within a transition are executed by the same thread. A program code location is marked as a *scheduling-relevant point*, if the next instruction to be executed represents a thread-interaction statement.

We use the term *modified code fragment* to denote a piece of source code that includes at least one added or deleted thread interaction statement, together with the affected nearby code within the same procedure. More precisely, we define the modified code fragment as a list l_{mod} of adjacent program statements within a single procedure. Without loss of generality, here we assume that the developer edited (added or removed) just a single thread interaction statement st in each step of iterative program development. Every modified thread interaction statement (i.e., the corresponding modified fragment)

must be processed separately (one by one). In the case of addition, the modified code fragment l_{mod} covers the statement st just inserted by the developer, together with all the nearby code (other statements) that load operands for st and use its result. For example, when considering addition of a field read statement, l_{mod} has to include (1) the statement that loads a reference to a target heap object and (2) the following code (located after the added statement) that uses the read field value. In the case of deletion, the modified code fragment l_{mod} covers the removed statement st, together with all the nearby code that loads operands for st and uses its result. Note also that the whole sequence l_{mod} always belongs to the code of single program thread T.

3 Exploration of Thread Interleavings: Pairwise Approach

For the purpose of explaining our approach, we assume that the input for a run of the verification procedure consists of a modified code fragment, which is specified by a pair of program code locations that represent boundaries of the respective sequence l_{mod} of program statements. We use the symbol T_{mod} to denote the static modified thread (in the program source code) that contains the modified code fragment l_{mod}.

The static thread T_{mod} is identified according to reachability of procedures in the program call graph. If the modified source code procedure, which contains the list l_{mod} of statements, is reachable from the entry point procedure of a given thread T then such thread is the modified thread T_{mod}.

If the given subject program involves multiple dynamic threads (instances) of the static thread T_{mod}, our verification algorithm processes every dynamic instance of T_{mod} separately. This also means that our algorithm considers every two dynamic instances of T_{mod} as distinct dynamic threads for the purpose of creating pairs of threads subject to verification. All the dynamic instances of T_{mod} will contain the updated code. Note also that the set $\mathcal{T}$ of dynamic threads can change between states of the program subject to verification. In each run, our algorithm considers the upper bound of the set of possible dynamic thread instances.

3.1 Main Algorithm

When given the modified code fragment l_{mod} and the static modified thread T_{mod} as input, our core verification algorithm explores all possible interleavings that involve l_{mod} for all pairs (T_i, T_j) of dynamic threads, where T_i is a dynamic instance of T_{mod} and T_j is a dynamic instance of some static thread from the set $\mathcal{T}$ of all program threads. Note that, in case of a modified thread with several dynamic instances, also these instances will be paired with each other and checked. As we indicated above, the key feature of the verification algorithm is that it processes each pair of threads separately, one pair at a time. Figure 2 captures the most important aspects of our verification algorithm, which we explain below in this section.

```
1  for Ti ∈ getDynamicInstances(Tmod) do
2    for Tj ∈ getDynamicInstances(𝒯) do
3      exploreInterleavingsForThreadPair(𝒯, Ti, Tj)
4    end for
5  end for
6
7  procedure onThreadSchedulingChoice(Tcur, Ti, Tj)
8    pi = getThreadCurrentPC(Ti)
9    if Tcur == Tj then
10     if pi == lmod.entry then return {Ti, Tj}
11   end if
12   if Tcur == Ti then
13     if pi ∈ lmod then return {Ti, Tj}
14   end if
15   return {Tcur}
16 end proc
```

Figure 2: *Main algorithm that explores all interleavings for a pair of threads*

The set $I_{i,j}$ of thread interleavings explored for a given pair of dynamic threads, (T_i, T_j), where T_i is a dynamic instance of T_{mod}, is defined as follows. For every scheduling-relevant point p_j in the

code of T_j, the set $I_{i,j}$ contains an interleaving in which the first statement in the sequence l_{mod} is scheduled to be executed at p_j. Thread T_i executes the code fragment l_{mod} in that particular interleaving. In addition, for each specific point p_j and for the specific thread interleaving, in which execution of statements in l_{mod} immediately follows the point p_j, the set $I_{i,j}$ also contains a thread interleaving for every scheduling-relevant point p_i in l_{mod} where thread T_j is scheduled to run at p_i within that particular interleaving. This definition of the set $I_{i,j}$ covers also the case when program execution begins with the statements in the sequence l_{mod}, because all thread start events are scheduling-relevant points too. All the possible interleavings of the modified code fragment l_{mod} with the existing code of thread T_j are captured in this way. The body of the event handler onThreadSchedulingChoice in Figure 2, invoked within the scope of the procedure exploreInterleavingsForThreadPair, captures the definition of $I_{i,j}$ in an imperative style. Here, the symbol T_{cur} represents the current (running) thread in the program state when a scheduling choice was reached on the currently explored state space path. This event handler procedure is called for each program state where a thread scheduling choice needs to be created; it returns the set of all threads to be explored from that state. The backend verification tool, which is actually used to systematically explore possible interleavings for the given pair of threads, has to create all the required thread scheduling choices in the program state space, in order to ensure that all thread interleavings in the set $I_{i,j}$ are truly explored. Figure 3 shows just a fragment of the tree of all thread interleavings and scheduling choices.

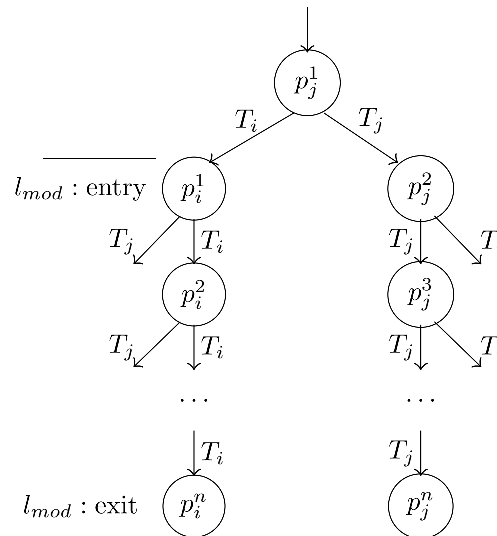

Figure 3: *A fragment of the tree of all thread interleavings and thread scheduling choices*

Note that the set $I_{i,j}$, defined above, has to include just those interleavings where thread preemption is enabled from the state in which the next instruction to be executed by thread T_i is the first instruction of the sequence l_{mod}. Therefore, all preemptive non-deterministic thread scheduling choices may be disabled on the particular analyzed execution trace in the program state space, until T_i reaches the beginning of l_{mod} on the execution trace. When exploring the respective prefix of the execution trace, the backend verification tool just needs to pick a single runnable thread at all scheduling-relevant points — a heuristic optimization that we implemented is to always pick thread T_i, when it is runnable. In addition, since the purpose of our algorithm is to explore just all the possible interleavings of the modified code fragment l_{mod} with the existing code of other threads, this also means that the set $I_{i,j}$ does not have to include any thread interleaving, in which the execution of any thread is preempted at a non-blocking statement while the next instruction of T_i does not belong to the sequence l_{mod}.

Other specific requirements on the backend verification procedure that actually explores thread interleavings in the set $I_{i,j}$ are discussed below in Section 3.3.

Exploring relevant interleavings of multiple threads. In order to enable sound detection of all error states (e.g., deadlocks) that involve N threads for $N > 2$, we extended our core verification algorithm (described above) such that it uses a dynamic happens-before ordering relation [16] computed over the execution trace prefix up to the first instruction (exclusively) of the modified code fragment l_{mod}. Note that if l_{mod} is reached multiple times on a given execution trace, then each of these cases is processed separately with respect to the whole respective trace prefix. Given an execution trace prefix w, our algorithm determines the list E of relevant events (such as lock acquire, lock release, and thread join) in

the prefix w, where each event in the list satisfies these two conditions: (1) it is associated with a thread other than T_i or T_j, and (2) it is not guaranteed to happen strictly before the first instruction of l_{mod} is reached by T_i. For every event $e \in E$, the algorithm has to consider both cases with respect to thread scheduling; it explores (i) the interleaving in which the instruction corresponding to e is executed before the first instruction of l_{mod} and (ii) also the complementary interleaving where the event e occurs after the end of l_{mod}, all that while preserving the program code order for all events associated with the same dynamic thread instance. The set $I_{i,j}$ is expanded to contain the additional interleavings that involve events in the list E.

For illustration, consider an example program that involves the global initialization statement $\texttt{i} = 0$, four dynamic threads $\texttt{T1} - \texttt{T4}$ with the same code $\texttt{i}++;\ \texttt{i}--$, and one dynamic thread T5 with the code $\texttt{assert}(\texttt{i} < 4)$. When the modified thread is T5 and the verification procedure analyzes interleavings for the pair of threads T4 and T5, the computed dynamic happens-before ordering relation specifies that actions of other threads (T1, T2, T3) do not happen strictly before or after T5 and therefore belong to the set E. Our algorithm will then explore also the interleaving (trace) beginning as $\texttt{T1}:\texttt{i}++$, $\texttt{T2}:\texttt{i}++$, $\texttt{T3}:\texttt{i}++$, $\texttt{T4}:\texttt{i}++$, $\texttt{T5}:\texttt{assert}(\texttt{i} < 4)$, ..., which reaches the error state (assertion violation).

3.2 Affected Thread Interleavings

The key ideas behind our algorithm, described in the previous subsection, express also our general approach to identification of the set of affected thread interleavings, which have to be explored for a given program code modification to guarantee sound verification and search for possible concurrency errors.

Unlike some existing techniques, our algorithm does not have to precisely identify the sets of thread interleavings and execution paths that are possibly affected by a specific change of the program source code. In particular, our algorithm does not have to compare the previous and new version of the program code by performing some kind of change-impact analysis [13], for example, and it also does not need to compute differences between the respective program versions in terms of possible behaviors using techniques such as directed incremental symbolic execution [26].

When the set $I_{i,j}$ of all interleavings between a given pair of threads, which captures all their possible interactions that involve l_{mod}, is relatively small, then performing a dedicated comparatively expensive analysis to precisely identify the set of affected thread interleaving (i.e., a subset of $I_{i,j}$) would incur an unnecessary overhead. The set $I_{i,j}$ effectively represents an over-approximation of the set of really affected thread interleavings, which is generated implicitly during the process of exploration. Usage of such over-approximation has the consequence of some parts of the whole program state space possibly being explored repeatedly, over the whole process of incremental program development and over all runs of the verification procedure — but that is not a problem with respect to performance and scalability, as long as each run of the verification procedure explores just a small part of the state space and finishes quickly.

Our verification procedure also does not have to use change-aware prioritization of thread interleavings [14], because all thread interleavings that are explored in a given step of incremental verification involve the modified code (l_{mod}).

3.3 Backend Verification Procedure

The backend verification procedure, actually used to analyze the set $I_{i,j}$ of possible interleavings for given pair of threads, may involve any of the commonly used approaches — including static program analysis, model checking, dynamic analysis, or systematic testing — and their combinations. Even rather expensive techniques, such as precise dynamic analyses with the happens-before ordering (cf. [7]), can be used in this setting, because the set $I_{i,j}$ should be quite small for most pairs of threads. We provide evidence for this claim in our experimental evaluation (Section 4).

Nevertheless, selection of the specific backend verification procedure (tool) is orthogonal to the main algorithm. The only important requirement imposed on the backend procedure is that it has to systematically analyze the visible behavior of the input program under every thread interleaving from the set $I_{i,j}$, as defined in Section 3.1. In particular, the backend procedure must be able to find all possible concurrency errors in the given set of thread interleavings.

3.4 Properties of the Algorithm

In this section, we discuss important properties of our verification procedure based on the pairwise approach, including soundness and coverage, all that especially with respect to the ability of detecting concurrency errors.

Even though our verification algorithm explores all possible interleavings just for pairs of threads, it detects also concurrency errors that involve the modified code fragment and N threads for $N > 2$. We discuss what thread interleavings (execution traces) have to be explored in order to detect all concurrency errors for N threads, and we also show that all these (required) interleavings are really covered by our verification algorithm. In particular, we show that, for each error in a given program, our algorithm explores at least one trace leading to the corresponding error state. Our discussion focuses on three main kinds of concurrency errors — deadlocks, atomicity violations and ordering violations. We begin with deadlocks.

3.4.1 Deadlocks over N threads.

Here we leverage the assumption, mentioned already in Section 1, that the whole program state space did not contain any deadlock before the modifications represented by l_{mod} in the code of T_i were made by developers. Execution of the modified code fragment l_{mod} by T_i is a necessary precondition for reaching a deadlock introduced through statements in l_{mod}. Therefore, we consider an arbitrary thread interleaving that reaches a deadlock state involving the set $\mathcal{T}_D$ of N threads, where $\mathcal{T}_D$ contains at least the pair (T_i, T_j) of threads for some T_j. We use the symbol π_D to denote this execution trace (thread interleaving). Next, we explain how it is guaranteed that our verification algorithm explores the trace π_D, discovering the respective deadlock state along the way.

The trace π_D has the following three parts: (1) actions guaranteed to happen strictly before execution of the first action in the sequence l_{mod} according to the dynamic happens before ordering described at the end of Section 3.1, then (2) some interleaving of the sequence of all actions in l_{mod} executed by T_i with actions of T_j and threads in the set $\mathcal{T}_D \setminus \{T_i, T_j\}$, which can happen concurrently with l_{mod}, and finally (3) actions guaranteed to happen strictly after execution of the last action in the sequence l_{mod}. Note that we use the partitioning of π_D only here in this discussion. It is not computed during the run of our algorithm. In order to detect the deadlock state reached by π_D, our verification algorithm needs to explore a thread interleaving π'_D that may differ from π_D only in the sub-sequence of independent actions within the middle part. We exploit the fact that l_{mod} contains just a single thread interaction statement st. Therefore, all other statements in l_{mod} can be moved next to st in π'_D, using the concept of left movers and right movers [18]. Given the set E of actions by threads in the set $\mathcal{T}_D \setminus \{T_i\}$, such that actions in E may happen concurrently with l_{mod}, the interleaving π'_D satisfies the property that each action in the set E is located either before the first statement of l_{mod} or after the last statement of l_{mod}. Such trace π'_D is explored by our verification algorithm for some pair (T_i, T_j) of threads, using the extension described at the end of Section 3.1, and it reaches the same deadlock error state as the execution trace π_D.

We illustrate how the verification algorithm works on the example of a deadlock involving three threads. Consider the scenario where thread 1 attempts to acquire the lock for resources A and B, thread 2 attempts to lock the resources B and C, and finally thread 3 attempts to lock C and A. There is an interleaving in which every thread acquires the lock for the first resource but then waits for the second resource — more specifically, thread 1 owns the resource A and waits for B, thread 2 owns B and waits for C, and thread 3 has access to the resource C but waits for A. For the purpose of our illustration, we assume that source code of thread 1 was edited most recently, meaning that the attempt to acquire the lock for resource B was added into the code of thread 1 in the last source code edit. Then our algorithm checks all possible interleavings for the pair of thread 1 with thread 2 over the modified code segment in these two cases:

- Thread 3 acquires the lock for the resource A before thread 1 or 2 reach the modified code segment on a particular execution trace, and no deadlock occurs.
- Here, thread 3 attempts to acquire the lock for A only after both threads 1 and 2 leave the modified code segment, when the resource A is already locked by thread 1 (that now waits for B), and therefore deadlock is observed. This state corresponds to the thread interleaving described above in the previous paragraph.

Note that, in both cases, thread 3 acquires the lock for C before the execution trace reaches the modified code segment.

3.4.2 Atomicity violations.

In the case of atomicity violations, the problem is much simpler than for deadlocks. Given an atomicity violation, for example a data race, that involves a set $\mathcal{T}_{AV}$ of N threads, it can be observed when any two threads from the set $\mathcal{T}_{AV}$ access a shared memory location without proper synchronization (mutual exclusion). Our verification algorithm detects such data race by exploring all possible interleavings of every pair of threads in the set $\mathcal{T}_{AV}$.

3.4.3 Ordering violations.

Finally, if some statements from the sequence l_{mod} representing the input modified code fragment are involved in an ordering violation error, then the wrong order of actions must be observed for some interleaving of the modified fragment l_{mod} executed by thread T_{mod} with some other thread T_j, i.e. for the pair (T_{mod}, T_j) of threads, and with concurrent actions by other program threads. Note that we assume there was no ordering violation error before changes represented by l_{mod} were made in the scope of the whole iterative software development process.

3.4.4 Coverage of all thread interleavings.

Another desired property of our incremental verification algorithm is the ability to cover, for the final (most recent) version of an input program, the set of all possible thread interleavings that reach a concurrency error state and would be explored by full verification of the whole program state space at once. In particular, we show that our algorithm truly explores all thread interleavings (traces) needed to detect all concurrency errors for an arbitrary number N of threads.

We use an inductive approach over incremental code modifications to show that, in order to achieve the desired coverage, it is sufficient to perform systematic exploration of possible interleavings just for pairs of threads and incremental modifications of program code during the software development process. More specifically, we show that every possibly relevant thread interleaving (i.e., that needs to be explored in order to find all errors) in the whole state space of the final (most recent) program version is covered by the pairwise approach to incremental verification for some modified code fragment at a particular step of the incremental program development.

As the base case, we assume that every relevant thread interleaving for the already existing code, that means before the program source code modification represented by l_{mod} is performed, is covered by our verification algorithm in some previous iteration.

Then we need to show that, for an input modified code fragment l_{mod}, our verification algorithm explores all thread interleavings that (1) contain statements from the sequence l_{mod} and (2) may reach an error state. Without loss of generality, we pick an arbitrary interleaving π that involves the modified code l_{mod}. The interleaving π can be decomposed into three segments: the prefix π_{pf}, the middle segment π_{mod} containing all of the modified code (l_{mod}) together with the set E of relevant actions by other threads (not guaranteed to be executed before the first instruction of l_{mod} or after the last instruction of l_{mod}), and the suffix π_{sf}. Note that any of the prefix π_{pf}, the suffix π_{sf}, or the set E can be empty.

From the base case assumption of inductive reasoning, we know that an execution trace created by concatenation of π_{pf} with π_E and π_{sf}, where π_E corresponds to a subsequence of π_{mod} that contains only elements of the set E, is covered by a previous run of our verification algorithm. The current run of the algorithm for the input fragment l_{mod} explores various interleavings of actions in l_{mod} executed by T_{mod} with actions in E (see details at the end of Section 3.1) Note that, in each of those interleavings, some actions from the set E are executed by a thread T_j that makes a pair with T_{mod} associated with the particular interleaving. The middle segment π_{mod} of π corresponds to an interleaving π'_{mod} that is defined as a concatenation of three parts, specifically (i) a sequence of actions in the set E_1, (ii) actions in l_{mod}, and (iii) actions in the set E_2, where $E_1 \cup E_2 = E$. Since l_{mod} contains just a single thread interaction statement st, the interleaving π'_{mod} can be soundly transformed by changing the order of all other statements from l_{mod} in a way that matches π_{mod}. Consequently, also the interleaving π is covered by our algorithm.

To summarize, the discussion above implies that our pairwise approach to incremental verification has the same coverage of thread interleavings as if systematic exploration of the whole program state space was performed after each source code modification.

When the run of the core verification procedure for a specific input modified fragment l_{mod} is completed, it is guaranteed that our algorithm explored — over all its runs for different code fragments

during the whole incremental program development process — all the relevant interleavings of threads as defined within the most recent version of the program code that includes l_{mod} as the last change.

3.4.5 Termination and Decidability.

An important limitation of our approach is that it may not terminate because general pairwise reachability is undecidable [28], and reachability for three or more threads is still an open problem. However, when the run of our algorithm finishes, it covers all relevant thread interleavings and reaches all error states even for $N > 2$ concurrent threads. We are aware of specific restrictions on locking strategies in multithreaded programs for which the reachability problem is decidable (see, e.g., [3] and [15]), but their detailed discussion is out of scope of this paper.

4 Experimental Evaluation

We have implemented our verification algorithm based on the pairwise approach in the Java Pathfinder (JPF) framework [38]. Java Pathfinder satisfies all the requirements on a backend verification tool, defined in Section 3.3 — it performs systematic traversal of concrete state space of a subject program, creates non-deterministic thread scheduling choices at possible thread interaction statements, and faithfully simulates effects of concrete program statements.

In our prototype implementation, we replaced the default JPF module for creating thread scheduling choices with our own component that precisely follows the algorithm described in Section 3.1 (exploring all interleavings just for pairs of threads). We also created a listener module that observes the process of state space traversal to determine whether the current program location on the currently analyzed execution trace is within the sequence of statements that represents the modified code fragment.

Source code of the implementation, together with benchmark programs, scripts and configuration files needed to run all the experiments, is available at https://github.com/d3sformal/incverif-pairwise.

Our main goal for the experimental evaluation was to show (1) that incremental verification based on the pairwise approach finds bugs within the modified code very quickly, and (2) that it detects exactly the same set of errors in subject programs as full verification of the whole state space from scratch, the latter meaning especially to show that our incremental verification algorithm is not less precise than full verification (with respect to possible spurious errors). We also wanted to show that running the incremental verification procedure after each step of the incremental software development process (i.e., after each source code modification) is more efficient than running full verification each time — despite that possible interleavings are explored for many distinct pairs of threads in each run of the incremental procedure. Here we focus mainly on the error detection performance, because fast search for errors is the main usage scenario of incremental verification procedures to be deployed within software development environments (IDEs), but we also evaluate performance improvements achieved for full verification of safety.

4.1 Benchmarks

The set of subject programs that we used for evaluation includes 9 multithreaded Java programs from widely known benchmark collections. More specifically, the set contains five programs from the CTC repository [37] (Alarm Clock, Prod-Cons, RAX Extended, Rep Workers and SOR), two programs from the pjbench suite [39] (Cache4j and Elevator), QSort MT from the Inspect suite [34], and plain Java version of the PapaBench real-time benchmark [23]. The smallest program is Prod-Cons with 130 lines of source code and 2 threads, and the most complex program is jPapaBench with 4500 lines of code and 7 threads. Regarding the level of concurrency, Alarm Clock involves contention of 3 threads over a single common lock, and RAX Extended starts 3 concurrent threads that access a single shared heap object. All other benchmarks involve synchronization just between two threads at a time.

4.2 Methodology and Setup of Experiments

In order to evaluate the performance of our incremental verification algorithm in the setting where it is run for each modified source code fragment, we needed to simulate incremental development of subject programs. The methodology that we actually used was inspired by Conway et al. [5]. We describe the main steps of the procedure that we applied to each subject program.

As the first step, all the possible thread interaction statements in the program code are identified by static analysis. Then, for every thread interaction statement, the respective modified code fragment (that includes the affected nearby code in the same Java method, as defined in Section 2) is determined using traversal of the list of program statements. A set $\mathcal{CF}$ of target code fragments for the program is created in this way.

For each code fragment F in the set $\mathcal{CF}$ collected for the program, we performed multiple experiments with different configurations. One group of experiments was configured to simulate the scenario in which the fragment F was added into the program source code by the developer (i.e., using the original version of the subject program), and the other group of experiments simulates the scenario where the fragment F was deleted from the program code. Then, for each of those groups, we performed experiments focused on fast search for concurrency errors (bug finding) and experiments focused on verifying safety. Our rationale behind this setup of experiments is the following. When some piece of code (the fragment F) is added or deleted by the developer, typically several iterations of fixing bugs take place, followed by the check for safety performed at the end, before the developer moves on to the next incremental change. Therefore, we cover both scenarios in our experimental evaluation. In order to compare incremental verification with full verification of the whole program state space, we also ran full verification with the default configuration of JPF for all scenarios. That means we ran full verification with the original code and with the modified code (from which the respective thread interaction statement was removed). For the purpose of evaluating the ability to find concurrency errors quickly, we injected synthetic errors into Prod-Cons, Cache4j, Elevator and jPapaBench. But within the scope of experiments focused just on the verification of safety, we used the original benchmarks without any synthetic errors.

In the case of incremental verification, JPF was configured to use our module for creating nondeterministic thread scheduling choices that implements the pairwise approach to state space exploration driven by the respective modified code fragment F — this is the main difference in the setup of JPF from the full verification where all threads are considered at each thread scheduling choice. However, while the actual input code fragment F can be directly used to control the state space exploration in experiments that simulate addition, for the other experiments that simulate deletion it is necessary to use the smallest different code fragment (from the set $\mathcal{CF}$ collected in the first step of the evaluation procedure) that fully wraps the input fragment F. The range of program statements that covers the whole respective Java method, i.e. the method that contains the input target code fragment F, is used as a fallback if the set of all target code fragments does not contain any fragment that wraps the input one.

When preparing all the experiments, for each modified code fragment we generated a variant of the input program where the respective code fragment is removed. For that purpose, we used our custom version generator tool, which is built upon the ASM bytecode manipulation framework [36].

This approach is robust because the results of experiments do not depend on the order in which the individual modified code fragments are processed.

Regarding the backend verification tool, Java Pathfinder (JPF), we configured it to search for deadlocks, race conditions and uncaught exceptions (including violated assertions). In the case of incremental verification, we used the time limit of 60 seconds for each run of JPF within the scope of experiments focused on bug-finding, and the limit of 10 minutes for experiments focused on safety verification. In the case of full verification, we used the time limit of 1 hour as a practical upper bound. We configured these limits in order to give each run of JPF enough time to explore the whole state space at least for smaller benchmarks. The memory limit was set to 16 GB.

4.3 Results and Discussion

Table 1 contains the aggregate results of all our experiments focused on quick bug-finding and especially on fast search for concurrency errors. The second column, labeled as $|\mathcal{CF}|$, shows the number of code fragments considered for each program. For both pairwise incremental verification and full verification, we report the average running time and standard deviation over all code fragments. The running time for a particular code fragment is computed as the sum of the running times of JPF for all pairs of threads. Note that, when computing the overall running times for incremental verification, we ignored the runs of JPF that did not finish within the time limit — the percentage of timed-out JPF runs is reported separately. Similarly, we report the percentage of JPF runs in the full verification mode that failed (i.e., run out of the memory or time limit).

Table 2 contains the aggregate results of experiments focused on checking safety (i.e., when the

Table 1: *Results of experiments focused on fast search for concurrency errors (bug-finding)*

program	$\|\mathcal{CF}\|$	pairwise incremental verification			full verification	
		time: avg ± dev	timed out runs	found bugs	time: avg ± dev	failed runs
Alarm Clock	56	0.25 ± 0.22 s	0 %	yes	1.20 ± 2.60 s	0 %
Prod-Cons	38	0.14 ± 0.08 s	0 %	yes	0.16 ± 0.04 s	0 %
RAX Extended	44	0.17 ± 0.23 s	0 %	yes	0.15 ± 0.05 s	1.1 %
Rep Workers	126	4.67 ± 12.14 s	3.4 %	yes	5.31 ± 29.30 s	2.8 %
SOR	56	0.09 ± 0.17 s	0 %	yes	0.83 ± 0.92 s	0 %
Cache4j	110	8.14 ± 15.06 s	0.84 %	yes	32.46 ± 283.76 s	0 %
Elevator	106	7.92 ± 16.14 s	1.67 %	yes*	12.19 ± 71.40 s	1.0 %
QSort MT	71	1.01 ± 2.27 s	1.25 %	yes*	0.61 ± 0.41 s	0.7 %
jPapaBench	374	15.96 ± 25.93 s	0 %	yes	28.52 ± 178.98 s	0 %

Table 2: *Results of experiments with focused on checking safety (full state space traversal)*

program	$\|\mathcal{CF}\|$	pairwise incremental verification		full verification	
		time: avg ± dev	timed out runs	time: avg ± dev	failed runs
Alarm Clock	56	0.25 ± 0.24 s	0 %	249.54 ± 143.48 s	0 %
Prod-Cons	38	0.16 ± 0.11 s	0 %	6.33 ± 1.82 s	0 %
RAX Extended	44	0.17 ± 0.24 s	0 %	12.71 ± 35.15 s	1.1 %
Rep Workers	126	41.32 ± 128.76 s	4.6 %	4494.43 ± 2859.57 s	14.7 %
SOR	56	0.09 ± 0.16 s	0 %	352.02 ± 74.26 s	0.9 %
Cache4j	110	46.18 ± 128.60 s	2.1 %	4936.50 ± 1987.80 s	2.7 %
Elevator	106	27.38 ± 54.99 s	0.6 %	102.96 ± 394.04 s	88.2 %
QSort MT	71	23.72 ± 92.62 s	3.8 %	735.84 ± 360.06 s	5.6 %
jPapaBench	374	18.41 ± 32.01 s	0 %	8.19 ± 168.26 s	89.3 %

benchmarks do not contain any injected synthetic errors). The meaning of all columns and reported metrics is the same as for the other table.

Results of experiments focused on evaluating the ability to find errors fast, which are provided in Table 1, show that each run of our incremental verification procedure finishes very quickly for all benchmarks and for every modified code fragment. The improvement in running time is apparent especially in the case of large and more complicated benchmarks (Cache4j, jPapabench). Specifically, for the most complex benchmark that we used (jPapaBench), incremental verification of thread interleavings affected by code updates finishes in 16 seconds on average.

The time limit of 60 seconds was reached for a very low percentage of JPF runs over specific pairs of threads. One reason is that no error was discovered by JPF and therefore all the relevant thread interleavings had to be analyzed systematically. Another reason observed in few cases is that the respective experiment simulated the scenario where the removed code fragment was essential for termination of the program run on all thread interleavings (execution traces). For example, in some experiments with Rep Workers, the removed code fragment was responsible for adding smaller parts of input data to the worklist of a recursive divide-and-conquer algorithm — clearly, when such functionality is missing, the program cannot terminate in a standard way.

On the other hand, data in Table 1 also show that the performance benefits of incremental verification in the case of focus on finding bugs quickly are slim for small programs, since also full verification finds errors in such programs very quickly, especially when the errors are shallow. The overhead associated with incremental verification (exploring interleavings for multiple pairs of threads after each program code change) makes the approach useful especially for large programs. As a side remark, here we also want to emphasize that the performance of our pairwise approach to incremental verification is comparable with state-of-the-art techniques based on static analysis, such as D4 [19]. Details can be found in Section 5.

Results of experiments focused on safety verification, presented in Table 2, show that usage of our pairwise approach to incremental verification greatly improves the speed compared to full verification. In particular, full verification of safety run out of the time limit for a great majority of modified

code fragments in the case of some benchmarks (such as Elevator and jPapaBench), while the pairwise incremental verification succeeded in most cases within the limited time and memory resources.

Based on thorough inspection of log files and reports by JPF, we have validated that, for every subject program in our evaluation, our incremental verification algorithm can find the same errors as if the full verification of the whole program state space is run every time. More specifically, for each actual error reported by JPF running in the full verification mode over a given subject program, there was a run of JPF in the incremental verification mode that reported the same error (pointing to the same program code locations with racy statements or actions causing a deadlock state). However, we want to emphasize two specific cases related to the Elevator and QSortMT programs, which are marked by * in the fifth column of Table 1. In the case of Elevator, full verification reported one spurious error — race condition involving an array element — caused by imprecise over-approximate handling of arrays by the race detector plugin distributed with JPF. In case of QSortMT, we found out that, in the scope of experiments over several certain target code fragments, the first error state reached by JPF running in the incremental verification mode is different from the first error reached by JPF running in the full verification mode — this happens because, in each mode, possible thread interleavings are explored in a different order during the state space search.

Some errors reported by JPF are caused by removal of the respective program code fragment. This corresponds to the scenario of an incomplete program still under development, where the run of incremental verification is started at a time when the source code of the subject program is not yet complete — for example, when some arguments to a specific procedure are not yet defined in some caller (so the `null` value is used).

5 Related Work

We compare our work presented in this paper to other techniques and tools that have similar goals (motivation), address the same problems, or use closely related approaches.

Incremental state space exploration. We begin with an overview of techniques based on incremental state space exploration. The work of Lauterburg et al. [17] focuses mainly on efficient incremental model checking of data-intensive properties for programs written in object-oriented languages. Its distinctive features include, for example, saving the full state space after a completed exploration for later reuse (to determine affected execution traces after a program code change) and restoring states through re-execution instead of loading them from disk. On the other hand, Lauterburg et al. do not discuss how the transitions affected by a program code modification are precisely identified. An important difference from our work is that we focus on multithreaded programs and interaction between threads affected by recent source code edits.

Guo et al. [13] developed Conc-iSE, a tool and method for incremental symbolic execution of multi-threaded software. The most important component of this method is an inter-thread and inter-procedural change impact analysis that can precisely identify program statements, execution traces and thread schedules not affected by a specific change in the program source code, eliminating such traces and schedules from re-exploration. Their change impact analysis considers also data values through symbolic execution in order to identify more redundant execution traces that do not have to be explored again. Unlike the Conc-iSE method, in our approach we do not use symbolic input values, since we focus mainly on the search for concurrency errors where we do not need to know symbolic values of program variables.

We are aware of several other prior works that belong into this category. For example, Conway et al. [5] proposed incremental variants of model checking algorithms that detect violations of safety properties. The main limitation of this work from our perspective is the missing support for concurrency. Sery et al. [30] designed an incremental variant of bounded model checking, which uses function summaries derived from Craig interpolants to avoid checking the whole input program from scratch every time. Again, there is no support for programs with multiple threads in this approach.

Fast detection of concurrency errors. The second group of related techniques and tools consists of those focused on very fast detection of concurrency errors and the corresponding bugs in the program source code.

Blaser [2] has developed a tool for efficient detection of concurrency bugs on-the-fly while the developer edits the source code in some IDE. This work has very similar goals to our approach, including the aim for a very fast response, but it performs just bounded concrete interpretation of the input program code and simulates random thread scheduling. On the contrary, our approach is designed to be sound,

i.e. to find all the concurrency errors — in particular, we do not use artificial bounds on the number of explored program steps and thread schedules.

Zhan and Huang [35] developed ECHO, an incremental data race detection technique that can report possible races on shared developers almost immediately after they are introduced to the source code through edits performed by developers within an IDE. ECHO leverages the fact that small code updates can be typically checked very quickly, avoiding re-analysis of the whole program from scratch. The key components of ECHO include (1) incremental change-aware static points-to and happens-before analyses (the latter using a novel graph representation of the happens-before relation), and (2) change-aware race detection algorithm based also on lockset information for heap locations and checking of possibly conflicting accesses to shared heap locations. Evaluation of the ECHO tool on real-world applications, presented in [35], show that is has a very good performance. However, like many other static race detectors, ECHO is incomplete and can report false positives due to the limitations of static analysis, despite optimizations designed by its authors in order to improve precision.

Following upon ECHO, Liu and Huang [19] developed D4, a framework that detects concurrency bugs very fast (interactively) and provides immediate feedback to developers editing the program source code. The main technical contribution includes two novel parallel differential algorithms for static pointer and happens-before analysis of multithreaded programs. After each change of the program sources, both algorithms recompute the analysis results only for affected code. Parallelization of static analyses is achieved by decomposition into multiple graph traversal tasks. Results of experimental evaluation show that D4 can find bugs in large real-word applications really very quickly, under one second in most cases. The main limitations of D4 include restriction to simple concurrency bugs (deadlocks, races) and possible false warnings (due to approximation) — nevertheless, authors of D4 claim that their incremental static analyses achieve the same precision as static whole-program analysis. In comparison, our approach can verify complex high-level properties over programs with multiple threads.

The last technique in this group that we want to highlight is the work of McPeak et al. [20]. It has similar goals as our project, specifically fast incremental analysis of program code for the purpose of detecting bugs and timely reporting while the developer has fresh knowledge of the code. However, authors target running times of several hours (requiring that the analysis must finish in one night build cycle). The key features are modularity, parallelization and usage of summaries — each procedure is analyzed separately and multiple tasks are executed in parallel to achieve better scalability.

Precise identification of affected execution traces. Another group of closely related work includes techniques for automated precise identification of execution traces and thread schedules affected by a source code modification. In the context of regression testing of concurrent programs, Jagannath et al. [14] designed a technique that determines the set of all affected program code locations and then prioritizes exploration of thread schedules involving the affected locations. Person et al. [26] developed an incremental program verification technique that identifies (1) affected statements using static analysis and (2) other effects of program code changes through symbolic execution.

Model checking algorithms for concurrent programs. Finally, we have to mention that numerous (advanced) algorithms for efficient model checking of multithreaded programs have been developed, including techniques based on modular reasoning with abstraction of threads (see, e.g., [8]) and bounding the number of thread preemptions on each state space path (e.g., [27]). Such algorithms can be applied within the backend verification procedure that checks the program behavior under all possible interleavings for a given pair of threads, and in that respect they are complementary to the incremental verification algorithm presented in this paper. However, in this context, an obvious drawback of bounded model checking is that it may not find error states at greater depths in the reachable state space, while our pairwise algorithm explores the whole state space, even though it is done gradually over multiple runs of the incremental verification procedure.

6 Concluding Remarks

Although our prototype implementation of the pairwise approach to incremental verification targets only Java and experimental evaluation considers only Java programs, it should be rather straightforward to create alternative implementations for other programming languages, such as C, provided there exist tools for state space traversal and static analysis that satisfy all the requirements defined in Section 3.3.

Our approach based on checking all interleavings just for specific pairs of threads is complementary

with techniques that prioritize certain thread interleavings during state space exploration, and can be easily combined with them. This includes heuristics for efficient discovery of concurrency bugs [12] and various techniques based on bounded search of some kind [21, 27, 31].

References

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

[2] L. Blaser. 2018. Practical Detection of Concurrency Issues at Coding Time. Proceedings of ISSTA 2018, ACM.

[3] R. Chadha, P. Madhusudan, and R. Viswanathan. Reachability under Contextual Locking. Proceedings of TACAS 2012, LNCS 7214.

[4] E. Clarke, O. Grumberg, and D. Peled. 2000. Model Checking. MIT Press, 2000.

[5] C.L. Conway, K.S. Namjoshi, D. Dams, and S.A. Edwards. 2005. Incremental Algorithms for Interprocedural Analysis of Safety Properties. Proceedings of CAV 2005, LNCS 3576.

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

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

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

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

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

[11] P. Godefroid. 2005. Software Model Checking: The VeriSoft Approach. Formal Methods in System Design, 26(2), Springer, 2005.

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

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

[14] V. Jagannath, Q. Luo, and D. Marinov. 2011. Change-Aware Preemption Prioritization. Proceedings of ISSTA 2011, ACM.

[15] V. Kahlon. Boundedness vs. Unboundedness of Lock Chains: Characterizing Decidability of Pairwise CFL-Reachability for Threads Communicating via Locks. Proceedings of LICS 2009, IEEE CS.

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

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

[18] R.J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. Communications of the ACM, 18(12), 1975, ACM.

[19] B. Liu and J. Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. Proceedings of PLDI 2018, ACM.

[20] S. McPeak, C.-H. Gros, and M.K. Ramanathan. 2013. Scalable and Incremental Software Bug Detection. Proceedings of ESEC/SIGSOFT FSE 2013, ACM.

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

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

[23] F. Nemer, H. Casse, P. Sainrat, J.P. Bahsoun, and M. De Michiel. 2006. PapaBench: A Free Real-Time Benchmark. Proceedings of WCET 2006, OASIcs, volume 4.

[24] P. Parizek and O. Lhotak. 2015. Model Checking of Concurrent Programs with Static Analysis of Field Accesses. Science of Computer Programming, 98, Elsevier, 2015.

[25] P. Parizek. 2016. Hybrid Analysis for Partial Order Reduction of Programs with Arrays. Proceedings of VMCAI 2016, LNCS 9583.

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

[27] S. Qadeer and J. Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. Proceedings of TACAS 2005, LNCS 3440.

[28] G. Ramalingam. Context-Sensitive Synchronization-Sensitive Analysis is Undecidable. ACM Transactions on Programming Languages and Systems, 22(2), ACM, 2000.

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

[30] O. Sery, G. Fedyukovich, and N. Sharygina. 2012. Incremental Upgrade Checking by Means of Interpolation-based Function Summaries. Proceedings of FMCAD 2012, IEEE.

[31] A. Udupa, A. Desai, and S.K. Rajamani. 2011. Depth Bounded Explicit-State Model Checking. Proceedings of SPIN 2011, LNCS 6823.

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

[33] C. Wang, S. Kundu, R. Limaye, M.K. Ganai, and A. Gupta. 2011. Symbolic Predictive Analysis for Concurrent Programs. Formal Aspects of Computing, 23(6), Springer, 2011.

[34] Y. Yang, X. Chen, and G. Gopalakrishnan. 2008. Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah, 2008.

[35] S. Zhan and J. Huang. 2016. ECHO: Instantaneous In Situ Race Detection in the IDE. Proceedings of FSE 2016, ACM.

[36] ASM bytecode manipulation framework, https://asm.ow2.io/

[37] Concurrency Tool Comparison repository,
https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency_Tool_Comparison

[38] Java Pathfinder verification framework (JPF), https://github.com/javapathfinder/jpf-core/wiki

[39] Parallel Java Benchmarks, https://bitbucket.org/psl-lab/pjbench