Int J Parallel Prog (2013) 41:570–593
DOI 10.1007/s10766-012-0237-2

Graphs for Mining-Based Defect Localization in Multithreaded Programs

Christopher Oßner · Klemens Böhm

Received: 27 June 2012 / Accepted: 29 November 2012 / Published online: 22 December 2012

Abstract Trends in modern multicore architecture design requires software developers to develop and debug multithreaded programs. Consequently, software developers must face new challenges because of bug patterns occurring at runtime and due to the non-deterministic behavior of multi-threaded program executions. This calls for new defect-localization techniques. There has been much work in the field of defect localization for sequential programs on the one side and on the localization of specific multithreading bugs on the other side, but we are not aware of any general technique for multithreaded programs. This paper proposes such an approach. It generalizes data mining-based defect-localization techniques for sequential programs. The techniques work by analyzing call graphs. More specifically, we propose new graph representations of multithreaded program executions as well as two mining-based localization approaches based on these representations. Our evaluation shows that our technique yields good results and is able to find defects that other approaches cannot localize.

Keywords Software-defect localization · Multithreaded programs · Call graphs · Applied data mining · Graph mining

1 Introduction

Problem Statement For some years, chips with several cores have been standard even in desktop computing. Multicore chips facilitate significant speedups of programs by

C. Oßner (✉)· K. Böhm
Institute for Program Structures and Data Organization (IPD), Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76128 Karlsruhe, Germany
e-mail: ossner@kit.edu

K. Böhm
e-mail: klemens.boehm@kit.edu

multithreaded implementations. However, software developers now are confronted with issues not known from sequential programs. The behavior of multithreaded programs is non-deterministic, and thread-specific constructs introduce new kinds of bugs [4,21]. Multithreading defects may result in defective orderings of events. Many new tools have been developed to cope with these issues. Some support the manual search for defects, but do not localize them [2,23]. On the other side, there are localization techniques that pinpoint positions in the source code that are defective. *Race detectors* [8,25,27] detect race conditions by analyzing the locks of a program. However, their localization capability is limited to defects resulting in wrong sequences of memory accesses caused by the wrong use of parallel program-language constructs. We argue that this is too narrow—in particular, defects in multithreading programs can also originate from non-parallel constructs. Example 1 illustrates this.

Example 1 A method without any parallelism decides on the maximal number of threads to be created in the next step. A defect in this method can lead to a wrong number of threads, e.g., one that exceeds the maximum number of threads that is possible. □

The example illustrates that multithreading-specific defects are not necessarily caused by multithreading constructs. This is important, as the problem addressed in this paper is to locate multithreading defects, including this kind of defect. The defect in the example itself is nothing new and is rather unimportant for the point of the example. A distinctive feature of our technique is its generality, e.g., that it also takes sequential parts of the code into account. It makes use of data mining, which is well suited to localize those defects. This is because the reasons for defects are even more versatile in parallel programs. An approach that identifies anomalies in general, without being too specific, is particularly useful. For instance, it should be able to find defects like the one in the example.

Problems Encountered To address a wide variety of defects, a defect-localization technique has to build on a general model. Various data-mining based techniques use *dynamic graphs* as input [5,7,20] that represent the program flow (see Sect. 3). They are recorded during program execution. The techniques require both correct and failed executions. Localization is based on mining for characteristics of the graphs corresponding to correct executions, but not to failed ones, and vice versa. Such techniques have shown to be well suited for defect localization in sequential programs [7,20]. Parallel programs, however, require a more sophisticated representation than conventional ones. The order of events in a multithreaded program i.e., the *interleaving*, changes for repeated executions. Representations must take this into account. It is unclear how this ordering should be modeled in the graphs. We will describe several possibilities to do so. Another open question is if and how threads should be modeled. It then has to be investigated how the model affects localization quality and performance. A problem when analyzing parallel programs is that instrumentation is needed to trace executions. Instrumentation may change the interleaving and make interleavings disappear. Finally, we will propose two localization techniques, one relying on traditional data mining and one that additionally uses frequent subgraph mining (FSGM). FSGM detects all subgraphs in a set of graphs that are embedded in at least a certain number of graphs. This minimal number of occurrences is abbreviated with *minsup* (short for: minimal support), a parameter of the algorithm. With FSGM, more sophisticated relationships might be revealed. But this has not been investigated before quantitatively. Part of the FSGM algorithm is to check the occurrence frequency of candidates against the database. To facilitate this check, a subgraph isomorphism test is needed, which is NP-complete [13].

Contributions This paper is the first in-depth investigation of data-mining based defect localization in multithreaded programs. From a data-mining perspective, it is a description of a significant application, studying how an important real-world problem can be tackled by means of (known) data-mining methods, and how good these solutions are from an application viewpoint. More specifically, our contributions are as follows:

- *New Graph Representations*. We discuss how to model threads and interleavings in call-graph extensions. We propose three different graph representations that extend vanilla call graphs, but have different kinds of *temporal edges* in addition, to model the interleaving.
- *Use of a Noise Maker*. A *noise maker* is a tool that produces interleavings artificially. To our knowledge, this is the first study using a noise maker during the trace of executions to generate the data-mining input. Using a noise maker has two reasons: First, an instrumentation is needed to trace executions. This affects the interleaving. A noise maker can produce interleavings that have disappeared due to a conventional instrumentation. Second, noise makers produce executions with different interleavings systematically.
- *Localization techniques*. We present two localization techniques to derive the *suspiciousness* of methods. This is an estimation how likely a method is to contain a defect. Both techniques use *information gain* to quantify how discriminative a method invocation is with respect to the outcome of the execution (correct, failed). The two techniques are different in that one makes use of frequent subgraph mining (FSGM), to gain additional knowledge about the context of a call, while the other one does without this step.
- *Analysis of Manifestations*. Farchi et al. [10] survey concurrent bug patterns. To understand the value of the graph representations proposed here, we analyze how these bug patterns manifest themselves in the different representations.
- *Evaluation of Precision*. We evaluate how precisely our techniques locates defects, using the multithreading-specific, comprehensive benchmark described in [9]. One result of our experiments is that our approach identifies the defective method in five of eight programs right away and performs well in the other cases. We stress that our approach has identified one defect not mentioned in the benchmark documentation.

Clearly, any FSGM-based approach faces scalability issues. In the context of defect localization however, previous work has covered a hierarchical approach to defect localization that uses FSGM and scales well [7]. There, the localization takes place at different levels of detail (e.g., methods vs. classes vs. packages). That work is orthogonal to this current one and could be combined with it. However, since our approach without such extensions has turned out to work well on the benchmark programs, we have not seen the immediate need to evaluate this combination.

Paper outline: Sect. 2 introduces our localization techniques, Sect. 3 the new graph representations. Section 4 analyzes how different bug patterns manifest themselves in the graphs just introduced. Section 5 evaluates the approach. Section 6 discusses related work, and Section 7 concludes.

2 Localizing Defects with Call Graphs

Data mining on program executions represented as graphs has been successful to localize defects in sequential programs [3,5,7,20]. In line with [30], we distinguish between *failure, infection* and *defect*. A failure is an observable incorrect output of a program. Infections are invalid program states. A defect is a position in the source code where a change is necessary to make a failure disappear. We now explain the localization techniques used in this paper. Our approach is *dynamic*. This means that the program analyzed has to be executed, and information on the execution is collected. In a nutshell, the idea behind our approach is to execute the program repeatedly, and to identify differences in the runtime behavior between correct and failed executions. Executing the program may reveal particularities of executions which have failed. *Static* approaches in turn do not execute the program. Thus, static approaches are typically not able to analyze runtime behavior. Some kinds of defects, e.g., wrong loop boundaries, are very hard to detect with static approaches, because it would be necessary to understand the program. Turning to race detectors, static approaches suffer from a high number of false positives [25]. Hence, will describe two dynamic techniques in the following we.

2.1 Overview

Algorithm 1 is an abstract description of the two dynamic techniques studied in this paper. The first step (Line 3) is to record the graphs. This is done by instrumenting a program and tracing executions. In contrast to program-execution tracing, as in, say, [16], we use an instrumentation based on AspectJ. The instrumentation overhead will make some interleavings disappear, but we counter this effect by means of a noise maker. See Sect. 3.3 for details. Tracing can be done on different levels of abstraction, e.g., basic blocks or methods. This paper uses the method level. This abstraction level determines the granularity of the output. If we used an instrumentation that recorded the trace on basic block level, we could output basic blocks instead of methods. Of course, this would lead to significantly higher instrumentation overhead.

Each method is represented as a vertex, each call as an edge. Note that certain defects can not be seen directly at this level, e.g., a wrong variable assignment. Such defects may cause infections that affect the graph, but this is not guaranteed.

We assume that an oracle is available that can decide if an execution is failed or correct, Line 4. For most software projects, tests are available that can serve as such an oracle. Localization techniques often need failed and correct executions, because they rely on differences between them. The traced graphs have to be reduced (Line 5), i.e., transformed to graphs that are much smaller. This is because the original graphs easily become too large to be mined. We explain our reduction technique in Sect. 2.2. Finally, we calculate a list of methods sorted by their suspiciousness P. To derive the suspiciousness, we do data mining. A software developer now can inspect the methods of the list top-down, until he has identified and fixed the bug.

Algorithm 1 The localization process

```
1: $G = \{\}$
2: for all $test \in$ test suite do
3:     $g \leftarrow$ trace of execution of $test$
4:     assign test outcome $o(g) \in \{failed, correct\}$ to $g$
5:     $G \leftarrow G \cup reduce(g)$
6: end for
7: calculate $P(m)$ for each method $m$
8: return list of methods sorted by $P(m)$
```

2.2 Weighted Totally Reduced Call Graphs

Program executions may have a huge number of method calls. The graphs recorded initially represent each method call as a distinct edge, referred to as *call edge* ($\rightarrow$), with a new sink vertex. Each vertex is labeled with the fully qualifying name including its parameters. All existing mining-based localization approaches that use graphs do a reduction. We for our part use the weighted total reduction [5], which has yielded good results for sequential programs.

Definition 1 (*Weighted Total Reduction*) The weighted total reduction notated as R_{total_w} maps an unweighted graph g to a weighted graph g'. All vertices $v1, v2$ with the same label in g are mapped to exactly one vertex v' in g'. If there is an edge (u, v) in g then there is an edge (u', v') in g'. An edge in g' has a weight that counts how many times an edge in g has been mapped to it.

The modeling of recursion is straightforward: In the R_{total_w} reduced call graph, recursion manifests itself in the form of a self-edge.

2.3 Deriving Suspiciousness

In order to derive the suspiciousness, we do feature selection based on the well-known information gain (*InfoGain*). The *InfoGain* of an attribute quantifies how well its values allow to discern between the values of a class attribute. In our case the class attribute is the outcome of the oracle (correct or failed). The process of deriving the suspiciousness is similar to [6]. In contrast to [6] however, we do not use gain ratio for the feature selection, but the well-known information gain (*InfoGain*). Gain ratio is able to deal with data sets where the distribution of class-attribute values is biased. However, since we balance this distribution (see Sect. 5), gain ratio does not yield any advantage over *InfoGain*.

The input of the information-gain algorithm is a table, as follows: Each row represents an execution of the program analyzed. There is a column for each edge of any graph. The graph class (i.e., correct or failed) is stored in the last column. The other cells contain the weight of the respective edge (column) in the graph. If the graph does not have this edge, then the value in the cell is 0. Calculating the information gain is an unsupervised process, and it is calculated for the entire data set without a training set. We calculate the *InfoGain* for each column except for the last one. Let D be the set of tuples of the table, and let C be the set of classes, i.e., $C = \{correct, failed\}$ in our case. m is the number of classes in C, C_i, for $i = 1, \ldots, m$ is a class in C. $C_{i,D}$ is the set of tuples that have class C_i. The probability for a tuple in D belonging to class C_i is p_i, which is approximated by $p_i = |C_{i,D}|/|D|$. $Info(D)$ (Eq. 1) quantifies the expected information needed to classify a tuple. $Info(D)$ is known as *entropy*.

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i) \tag{1}$$

In order to classify a tuple, the data set can be split as described in the following: Let $\{a_1, a_2, \ldots, a_v\}$ be the values of attribute A. D can be split in v subsets $\{D_1, D_2, \ldots, D_v\}$ so that D_j contains tuples with value a_j for A. Ideally this split would yield an exact classification. But in general, the class values of the tuples in D_j will not be uniform. This means that additional information is needed to exactly classify a tuple. A measure of this additional information is $Info_A$, Eq. 2.

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} Info(D_j) \tag{2}$$

Finally, the information gain $InfoGain$ is calculated as follows.

$$InfoGain(A, D) = Info(D) - Info_A(D)$$

We interpret the information gain as suspiciousness. Methods can call several other methods, so there usually is more than one edge with the same source vertex. For this reason, we set the suspiciousness of a method as the maximum of all information-gain values of outgoing edges. It is intuitive to assign this value to the source of an edge (the *call site*), as the sink is the method called and is not active in the call. We report the whole edge (not only source vertices) to the software developer to provide background information for the ranking. It can happen that the information gain is the same for several methods. In this case we use a second ranking criterion. It is the number of lines of code (LOC) of Method m in descending order. The number of LOC is a simple criterion to estimate suspiciousness [24].

2.4 Obtaining Contexts with FSGM

Various localization approaches [3,5,20] apply FSGM to derive suspiciousness. The approach in Sect. 2.3 in turn only relies on call frequencies. But this may not be sufficient. Example 2 illustrates why FSGM can be helpful.

Example 2 Think of a call $a \rightarrow b$ that happens in all executions, be they correct or not. Its frequency is unrelated to the outcome. But the call can still be suspicious in a specific context: Imagine a second edge $a \rightarrow c$. Method c sets a global Boolean to False, and this leads to a state where b should not be called. In the context of the call $a \rightarrow c$, the call $a \rightarrow b$ must not be present and should be suspicious. □

The reason to apply FSGM is to discover contexts such as the edge $a \rightarrow c$ in the example. In Example 2, a subgraph including both calls is more frequently contained in graphs representing failed executions. The difference in the frequencies will be helpful when calculating the suspiciousness.

2.5 Deriving Suspiciousness with Contexts

Table 1 shows the input for the information-gain based feature selection with contexts. As before, each row of the table represents an execution of the program analyzed, and the last column contains the class of the graph. The other columns are the set of all edges in all frequent subgraphs, Lines 2–6 in Algorithm 2. The first column is edge $a \rightarrow b$ in Subgraph 1, the third column is the same edge but in Subgraph 2. In case a subgraph is not contained in a graph, all corresponding cells are 0, as for SG_1 and $Graph_2$ in Table 1. The next step of the algorithm is to derive the suspiciousness of methods. Line 9 calculates the information gain. The remainder of the algorithm derives the suspiciousness based on the information gain. Line 10 is needed, because an edge usually appears in different contexts. Analogously to the variant without contexts, we assign the maximum of all information-gain values of outgoing edges to a method (Line 14).

Table 1 Feature table with contexts

	SG_1 $a \rightarrow b$	SG_1 $a \rightarrow c$	SG_2 $a \rightarrow b$	$\cdots$	$Class$
$Graph_1$	2	1	2	$\cdots$	$failed$
$Graph_2$	0	0	4	$\cdots$	$correct$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Algorithm 2 Deriving Suspiciousness

```
Require: frequent subgraphs of execution traces $SG$
 1: $E(SG) = \{\}$
 2: for all frequent subgraph $sg \in SG$ do
 3:     for all edges $e \in sg$ do
 4:         $E(SG) \leftarrow E(SG) \cup \{e\}$
 5:     end for
 6: end for
 7: assemble feature table; rows: executions; columns:
    $E(SG) \cup graph\ class$; cells: edge weights
 8: for all edges $e \in E(SG)$ do
 9:     calculate $InfoGain(e)$
10:     $P(e) = max\{InfoGain(e) | e = (u, v) \in E(SG),$
        $\forall u_i, v_i, i \in [1, n] : u_1 = \cdots = u_n \wedge v_1 = \cdots = v_n\}$
11: end for
12: $S = \{\}$
13: for all methods $m$ do
14:     $P(m) = max\{P(e) | e = (m, m_x)\}$
15:     $S \leftarrow P(m) \cup S$
16: end for
17: return suspiciousness values $S$
```

For the evaluation we used the *CloseGraph* [29] implementation of the ParSeMiS[1] suite. In comparison to other FSGM algorithms, *CloseGraph* does not output all frequent subgraphs, but only *closed* ones. A subgraph g is closed if there is no supergraph of it with the same support. The closed frequent subgraphs still contain the information that is of interest for localization. At the same time, the result set is relatively small. FSGM has been applied for defect localization in sequential programs, but it has not been verified if it improves the localization for multithreaded programs.

Calls that are part of the infection may manifest themselves in a graph. It is likely that methods belonging to these calls have a high rank. Such a ranking may be helpful for a software developer, because the infection leads him to the defect. However, any method ranked higher than the defective one makes the result worse. It will be an interesting outcome of the evaluation to see to which degree infections influence the result.

3 Call Graphs for Multithreaded Defect Localization

In this section we first study how threads can be represented in graphs. We then propose three representations of multithreaded programs. Finally we highlight important technical issues.

3.1 Representation of Threads in Graphs

It is important to represent the interleaving in the graphs, because multithreading bugs may affect it. The interleaving of a multithreaded program is the order of operations. This includes the information which thread an operation is executed in. Because of the

[1] http://www2.informatik.uni-erlangen.de/research/ParSeMiS.

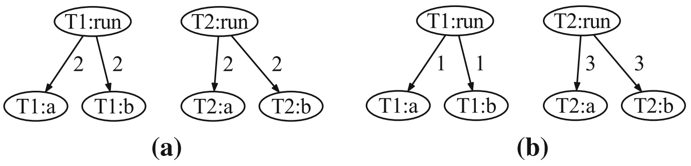

Fig. 1 Work load

dynamic allocation of resources, the executing thread is not known a priori. We say that an operation is *associated with a thread in a specific execution* if it is executed within the thread. The following paragraph features an illustration.

Thread Association Figure 1 shows excerpts of graphs for two different executions (a) and (b). A label contains the name of a method and the ID of the executing thread. The executions have different thread associations, leading to different weights. Suppose that one execution is correct, and the other one has failed. While Fig. 1a, b are different, this does not necessarily have to do with the class of the graphs. Example 3 illustrates a situation leading to this kind of noise.

Example 3 Suppose that the threads `T1` and `T2` are of the same class. Their job is to pull work from a queue. In case of Fig. 1a the load is balanced between the threads. Figure 1b shows an execution where the methods `a` and `b` are executed three times in thread `T2` and only once in thread `T1`. The reason why executions swap to thread `T2` could be that thread `T1` has been suspended, and for this reason `T2` was instructed one more time, say `T2` was faster. □

It seems natural to model thread IDs as vertex labels. But since our localization technique relies on differences in the graphs, differences as in Fig. 1 might wrongly be assumed to be suspicious, while, they are due to noise. The takeaway from the discussion is that it is not obvious how to deal with thread association when it comes to the graph representation.

Thread-IDs The assignment of thread IDs is non-deterministic, i.e., they are not valid over several program runs. So it is not possible to use them to compare different graphs. Another problem when labeling vertices with thread IDs is that threads can lead to repeated substructures [6]. Often similar tasks are assigned to more than one thread, as in Fig. 1. With the representation used there, a pattern appears repeatedly. This increases the graph size, but does not provide additional helpful information. All this means that there is no obvious way how to model thread IDs. The following subsection proposes alternative ways to deal with threads in the modeling.

3.2 Evaluation of Graph Representations

The previous section has shown that modeling multithreaded program executions as graphs is not trivial. We now introduce three new notions to address the specifics of multithreaded software, namely next-call, inter-thread and intra-thread relations.

Definition 2 (*Next-Call Relation*) Two methods u and v have a next-call relation *iff* (i) u and v have the same call site, and (ii) the call to v happens immediately after the call to u without any call from any other call site in between.

Definition 3 (*Inter-Thread Relation*) Given two calls c_1: $x_1 \rightarrow y_1$, c_2: $x_2 \rightarrow y_2$, there is an inter-thread relation between y_1 and y_2 *iff* (i) c_1 and c_2 happen immediately after each other, without any other calls in between, and (ii) x_1, x_2 are not in the same thread.

Definition 4 (*Intra-Thread Relation*) There is an intra-thread relation between two methods y_1 and y_2 if there are calls $c_1 : x_1 \rightarrow y_1$, $c_2 : x_2 \rightarrow y_2$ such that (i) c_1, c_2 happen immediately after each other, and (ii) x_1 and x_2 are in the same thread.

We now propose graph representations for each relation. These graphs are extensions of total reduced call graphs. Some characteristics apply to all three graphs. For these points we use the term *temporal* and use the subscript $_{temp}$.

Notation 1 (Temporal Edges) *The realization of next-call relations (inter-thread relations, intra-thread relations) is the next-call edge (inter-thread edge, intra-thread edge). We represent these edges as* $u \rightarrow_{nc} v$ *(next-call edge),* $u \rightarrow_{it} v$ *(inter-thread edge), and* $u \rightarrow_{in} v$ *(intra-thread edge) for vertices* u, v*. An edge weight models how often the respective relation has come into existence in the execution that the graph represents.*

Note that temporal edges model the point of time of the invocation of the sink. To distinguish edges of R_{total_w} from temporal edges, the different types of edges have different labels. Each graph in Fig. 2 shows an excerpt of the same execution. (a1), (a2) show next-call graphs, (b1) and (b2) inter-thread graphs and (c1), (c2) intra-thread graphs. The graphs are call graphs extended by the corresponding temporal edge (dotted). For simplicity we have omitted initial steps: Execution of the main method and initialization of two threads are not shown. The graphs (a1), (b1) and (c1) are correct executions, and (a2), (b2) and (c2) are failed executions. Suppose that the defect behind the failed executions is as follows: The sequence of calls to a and b should have been protected, but it is not. Method invocations, such as calls to a and b in the example, have to be protected if, say, a reads the value of a shared variable g, and b writes g. Otherwise, g may be read by method a of two different threads

Table 2 Event order

correct		failed	
Thread	Call	Thread	Call
Thread1	$run \rightarrow a$	Thread1	$run \rightarrow a$
Thread1	$run \rightarrow b$	Thread1	$run \rightarrow c$
Thread2	$run \rightarrow a$	Thread2	$run \rightarrow a$
Thread2	$run \rightarrow b$	Thread1	$run \rightarrow b$
Thread1	$run \rightarrow c$	Thread2	$run \rightarrow b$

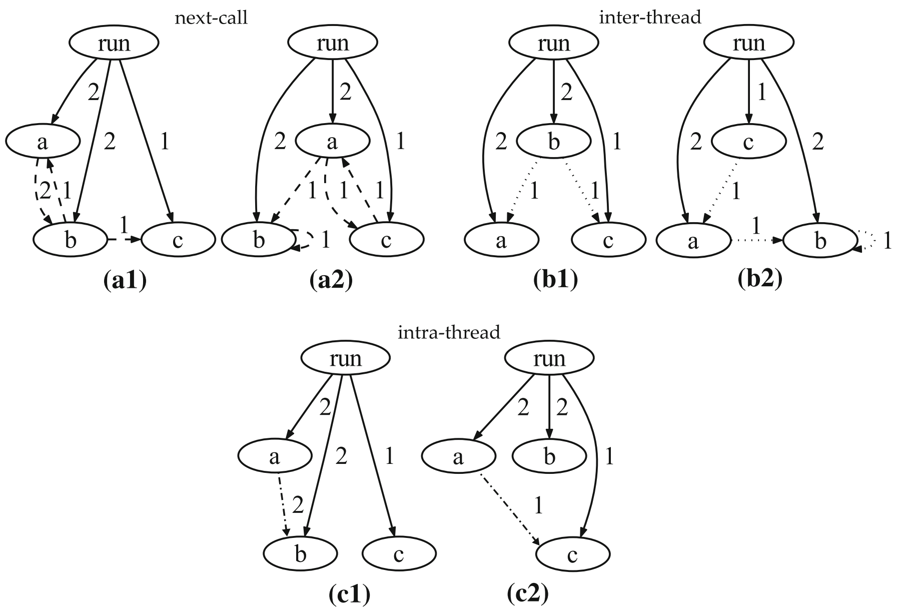

Fig. 2 Variants of reduced graphs; *correct: (a1), (b1), (c1); failed: (a2), (b2), (c2)*

`T1, T2`. Before `T2` processes the value read, `T1` writes a new value to `g` in method `b`, leaving `T2` with an orphaned value.

Table 2 shows the order of events corresponding to the graphs illustrated. For simplicity the table does not take into account that executions can be parallel. The event orders shown can come into existence if the example program is executed on a single processor.[2]

Next-Call Graphs A next-call edge is of interest for multithreaded defect localization, because a change of its weight between different executions might be due to an incorrect interleaving.

Example 4 The bug in Fig. 2 (a2) is that `a` is executed from Thread1 and Thread2 before Thread1 executes `b`. The incorrect execution order manifests itself in the call graph: The edge $a \rightarrow_{nc} b$ has weight 1. In the correct execution, Fig. 2(a1), this edge has weight 2. This tells us that `b` is called two times immediately after `a` is called (and from the same call site). This complies with the order events should be executed in. □

Thus, the interesting edge $a \rightarrow_{nc} b$ in Example 4 has different weights in the correct and the failed execution. The information-gain step within the localization process looks out for this kind of difference.

Inter-Thread Graphs Next-call graphs make the defect from Example 4 explicit. But the example also shows that not all next-call edges are helpful to localize a defect. Edge

[2] In this paper we use the terms processor core and processor interchangeably.

$a \rightarrow_{nc} x$ is such an edge that is not related to the defect. It is of advantage to reduce the number of edges without losing important information. This is the rationale behind inter-thread graphs. One can think of such an edge as representing thread switches. Observe that, for the executions illustrated, the inter-thread graphs have fewer edges than the next-call graph. Example 5 illustrates how inter-thread graphs help to localize the defect shown in Fig. 2.

Example 5 Edge $a \rightarrow_{it} b$ in Fig. 2(b2) is the manifestation of the defect: It indicates that after the call of `a` there is a thread switch. In the correct execution order this edge is not present. □

Intra-Thread Graphs The intuition behind the intra-thread relation is to reduce the number of edges, compared to the previous representations.

Example 6 In the correct execution (Fig. 2(c1)), there is an edge $a \rightarrow_{in} b$ with weight 2. This is the sequence of calls that should have been protected and that has happened twice. In the failed execution, this edge is not present. □

Locating Defects with Temporal Edges The process to derive the suspiciousness of temporal edges is different from the derivation for call edges. The vertices of a temporal edge may be suspicious for two reasons. The first case is that there is a defect at a call site of the methods connected by a temporal edge. For instance, an `if`-statement that decides if a method is called has to be protected together with the call itself. The other case is that two methods influence each other. This happens if no lock or the wrong one has been acquired for a sequence of operations that should be protected. In both cases the interleaving with respect to the involved methods may be defective.

Fixing Defects Related to Temporal Edges The previous paragraph opens a way to fixing a defect pinpointed to by a temporal edge, i.e., a temporal edge with high suspiciousness. Algorithm 3 illustrates this process.

Algorithm 3 Locating a Defect with Temporal Edges

```
Require: temporal edge $e$
  if call to source or sink of $e$ is conditioned but not protected then
    protect call and if-statement together
  else
    protect operations of source and sink that use a shared variable with common lock
  end if
```

3.3 Obtaining Graphs of Multithreaded Programs

With sequential programs, call graph based localization techniques trace executions with different inputs. The reason to use different inputs is to obtain failed and correct executions. In case of a multithreaded bug, due to non-determinism, the output for a given input may change with different interleavings. It may be sufficient to execute the program many times with the same input to obtain correct and failed executions. Some interleavings are rather unlikely to happen. It may even be the case that a defect never happens, or it occurs all the time on a specific system; the interleaving may vary heavily on different systems. But it is possible to systematically explore the space of interleavings with a *noise maker*. We use the noise maker of the ConTest tool [4]. This tool adds instrumentation points before and after each concurrent event. It inserts "noise" at the instrumentation points, i.e., the Java methods `yield`, `priority` and `sleep`. The number of interleavings grows exponentially with the number of threads. This makes it infeasible to produce all possible interleavings. However, ConTest tries to produce a different interleaving in each run.

Instrumentation may change the program behavior. Any instrumentation introduces overhead, and this overhead might make some interleavings disappear. Bugs that do not appear any more if the program is instrumented are called *Heisenbugs* [14]. The reason for this disappearance is the following: To implement the trace graph, we need a global data structure, and this is problematic. The instrumentation for each thread has to access the global structure, and this of course has to be synchronized. Because of this, the possible interleavings of the instrumented program may be different from the original ones. But through the use of a noise maker, those interleavings are likely to occur again.

3.4 API-Calls

We now explain how we deal with calls of methods which are not part of the code currently under observation, e.g., the JRE library. In principle, when analyzing a program, one can take libraries into account. To keep our approach focused, we do not consider defects in libraries—they should be eliminated on their own. Since we localize defects by identifying suspicious edges, we cannot leave aside API calls: They might reveal a defect of the call site. In our graph representations, there is a vertex for each method of a library called because we record any call to an API. However, the size of the graphs is of matter when applying FSGM. Therefore, the API calls can become a bottleneck. If the computation time of *CloseGraph* exceeds a limit (1 h), we propose the following: Stop the computation, aggregate all vertices that represent an API method to one *dummy vertex* and restart the computation. This is analogous to the work of Eichinger et al. [7].

4 Manifestation of Defects

A design pattern is a guideline to solve recurring problems. Bug patterns in contrast describe recurring programming mistakes. The manifestation of a defect differs in the proposed graph representations, and it is important to look at how different patterns manifest themselves. On the one hand, the defect can have an immediate effect on the graph, be it on the structure, be it on a weight. On the other hand, the defect might not manifest itself in the graph, but an infection it has caused might. Farchi et al. [10] propose a taxonomy of eight multithreading bug patterns, and we rely on in it this paper. In the following, we review these patterns and say how they manifest themselves. We do this to provide a theoretical evaluation of our graph representations and how well they cope with the various multithreaded bugs. The benchmark from [9] we will use in Sect. 5 has been created to cover the bug patterns described here. From our data-mining based perspective, it is important to understand the manifestations. In the field of defect localization, one usually generates the representations himself, like we do. This means that we have full control over the representation. Understanding the data can help to improve the representation and the mining process.

Pattern: "Wrong Lock or no Lock"/"Two-Stage-Access" These two patters describe failures caused by obtaining the wrong lock or no lock at all, so that a code segment is not protected although it should be. This can cause defective interleavings, see the examples in Fig. 2. As the manifestations differ for the different graph representations, we look at them separately.

Call Graphs without any temporal edge: Neither the structure nor the weights of a call graph change when the execution order changes. So we cannot see this pattern directly. We might however, see a manifestation of an infection.

Next-Call Graphs: The reason for protecting events is to have them as atomic units. Suppose that calls to Methods a and b shall be protected, in this order. If the order is maintained, a next-call edge will go from a to b, otherwise not.

Inter-Thread Graphs: If the sequence of calls that should have been protected is interrupted after the call to method a, there is an inter-thread edge from a to the method that has interrupted the sequence.

Intra-Thread Graphs: Analogous to next-call graphs.

The so-called *two-stage access* bug pattern is a special case of the *wrong-or-no-lock* pattern. If variables are shared between events, these events often must be protected as a whole. The pattern describes situations where the software developer has wrongly assumed that protecting each operation separately is sufficient. The manifestation of this bug pattern is analogous to the *wrong-or-no-lock* pattern.

Pattern: "Not Atomic" Programmers tend to assume that certain operations, like the ++-operator, are atomic, but in fact they are not. We look at program executions at the method level. The pattern instead describes bugs at the level of operations. The graphs do not show any direct manifestation. But if the variable is used later, we might see effects of the wrong value, e.g., calls not executed or the wrong number of times.

Pattern: "Double-checked Locking" Code used for lazy object initialization, which has been explicitly proposed [28], tends to lead to the double-checked locking bug, as follows: To achieve fast execution, only the initialization itself is synchronized. The most common code path, a check if the initialization has taken place, is unsynchronized. –The proposed code may lead to situations where an object reference is not null, but the object is only partially initialized. This happens due to a reordering by the compiler. The reordering is not transparent on the source-code level, and thus one can see the effect only through infections.

Pattern: "Sleep Bug Pattern"/"Loosing a Notify" In parallel programming, it is often useful to do calculations in dedicated child threads. At a specific point, the result of a child thread must be available to the parent. To ensure that the child has finished its work, the `join` method should be used. Otherwise the parent thread may hold an initial or old result value. Any interleaving where the calculation of the child is finished when the result is read is correct with respect to the *sleep* bug pattern. This leaves us with a possibly huge number of interleavings that differ without affecting the correctness. For this reason, differences in the graphs for correct and failed executions usually are not helpful, but infections may be. If a result is not available in time, the variable storing it will hold an initial or old value that might infect the program execution.

The *sleep* bug pattern does not induce a data race. A data race happens if a variable is overridden, but with this pattern the value is not written in time. Hence, a race detector cannot find this pattern.

The so-called *losing a notify* bug pattern is very similar to the one of the *sleep-bug* pattern. If a call to `notify` is executed before the corresponding `wait`, the notification is lost. The waiting code remains sleeping. As with the *sleep-bug* pattern, the software developer implicitly assumes an ordering of events without ensuring it. The effects on the graph are similar.

Pattern: "Blocking Critical Section"/"Orphaned Thread" This bug pattern describes situations where a thread is assumed to return control, but does not do so. This can cause the system to freeze if all other threads are waiting. A behavior that easily can be seen in inter-thread graphs: There is no outgoing inter-thread relation from the blocking thread, hence the weight of the edge is lower than the number of calls to the corresponding `start` method. For the other graph representations, the manifestation depends on the specifics of the bug. The hang of the system can come along with a different program flow. For instance, a missing invocation of a method might have caused the hang. The so-called *orphaned thread* bug pattern manifests itself in the same fashion. In this pattern, there is a master thread that coordinates the actions of some worker threads, a common design for multithreaded programs. A way to induce the pattern is to place messages on a queue, which the worker threads then work of. In case the master thread terminates abnormally, the worker will be stuck waiting for input from the queue. Neither the *blocking critical section* pattern nor the *orphaned thread* pattern result from wrong memory accesses. This is why a race detector cannot find them.

To design a graph representation that suits the needs of localization well, it is important to understand how the chosen representation reacts to defects. We apply mining algorithms on graphs, not on the source code. However, we want to identify a location in the code. One interesting point is that various bug patterns cannot be seen directly, but possibly through infections. An infection may change the program flow. This is why it is important to include the sequential behavior of a program: Changes in the program flow can be seen in the call graph.

5 Evaluation

In this section we first describe the programs used for evaluation, namely the benchmark from [9]. It is written in *Java*, but contains many bugs that apply to other languages as well. The benchmark programs cover the different bug patterns known from the literature. They are comparable in size to the widely used Siemens tools [15], which, however, do not feature parallelism.

5.1 Programs used for Evaluation

We now explain the benchmark programs, but only briefly due to space limitations. Observe that these programs are difficult to analyze: For instance, experiments of Eichinger et al. [6] show that the race detector implemented in the MulticoreSDK [22] does not find any of the defects in *AllocationVector, DeadlockException, Liveness* and *MergeSort*. (The other programs we have used in our evaluation and explained next are not covered by the evaluation of Eichinger et al. [6].)

Account (Pattern: *wrong/no lock*) The program is a bank simulation that executes deposit, withdrawing and transfer actions on several accounts. Each account is held by a dedicated thread. The transfer action involves two accounts. It accesses the amount variable of a second account. As the access is not locked, other actions are executed during (parallel to) the access to the recipient account. This causes dirty reads.

AirlineTickets (Pattern: *not atomic*) The program simulates the selling of tickets, and there are 10 % more tickets available than there are seats. To model the sale, a variable that reflects the number of seats sold is incremented. An `if`-statement immediately follows. This sequence of statements should have been protected to avoid data races.

AllocationVector (Pattern: *two-stage access*) The program simulates allocation and deallocation of memory blocks, a typical OS activity. Allocating a block consists of two actions: finding a free block and the allocation itself. Each of the two methods/actions is protected separately, but they should have been protected together. The effect is that two threads can allocate the same memory.

BubbleSort (Pattern: *not atomic*) This program is a bubble-sort implementation. Threads iterate over the input array and swap two elements if they are not sorted; a separate method does the swapping. Although the method itself is protected, the comparison that may lead to its call and the call itself are not protected together. Two threads might do the comparison, the outcome is the same in both cases, and the swapping method is called from both threads.

BubbleSort 2 (Pattern: *not atomic, sleep bug pattern*) This program again is a bubble-sort implementation. Critical interleavings may occur. For example, in case of an array sorted descendingly there is only one interleaving that ensures a proper sorting. It occurs if all `run`-methods are executed in the order their corresponding `start`-methods are called. This actually is a sequential execution, with no parallelism. The `run`-method is the place where the defect occurs: A new thread instance *t* is generated here that might run too early. So, at first glance, this method is the appropriate place to call `join` and wait until all the work is done before *t* runs. But this would cause a thread to wait for its end within its `run`-method, a deadlock situation. There is no way to change the program so that it makes use of parallelism and always has a correct interleaving. Further, there is another defect, which is not mentioned in the benchmark documentation, an atomicity violation. As with the other BubbleSort program, the method that swaps numbers is protected, but the `if`-statement just before the call to that method is not protected together with the call. (We point out that our approach has localized this previously unknown defect.)

DeadlockException (Pattern: *blocking critical section*) This program creates several worker threads, and the number of threads is limited. A thread starts by incrementing a global counter, followed by a mathematical operation (the actual work) and finally decrementing the counter. As the operation might throw an exception, causing the method to be left abnormally, the counter might not get decremented.

Liveness (Pattern: *liveness bug/orphaned thread*) This program simulates a server-client application, where each client is processed in a thread, and the maximum number of threads is limited. If there are more requests than client threads available, incoming requests are queued. These suspended clients are resumed when other client threads finish their work. As the queuing can be delayed, it can happen that all threads finish their work, and there still are suspended threads. The bug pattern is a special case of the *orphaned thread* pattern, except that the master thread does not terminate. However, when the failure appears, we are left with threads that are orphaned.

MergeSort (Pattern: *not atomic*) This program tries to realize a parallelized merge-sort. The maximal number of threads is fixed. If there are available threads left, threads can generate subthreads. If two or more threads read the number of available threads concurrently, they will generate subthreads, and more than the maximum number of threads might be generated. The documentation of the benchmark suggests that protecting a sequence of statements that read the number of available threads and generate subthreads depending on the outcome would fix the bug. But within the code to protect, there is a call to `join` from the subthreads. With this, the protected area could only be left if the subthreads finished their work. But to do this, the subthreads would have to execute the same protected code, leading to a deadlock.

All programs of the benchmark provide an oracle and a class that triggers the execution. We use the same parameters for all executions and execute each program 500 times. Then we balance the execution traces, so that the same number of failed and correct traces is left. If <100 traces are left, we execute the program and balance the traces again until more than 100 traces are left. Executing the program a second time has been sufficient in most cases. We use the standard parameters for ConTest and a noise frequency of 1,000. This value means that noise is added to each concurrent event. It is the maximum value possible, and lower values do not affect each event. A possible reduction of the overhead could be to use a lower noise frequency for ConTest.

Our approach has a runtime overhead that is produced by the call-graph trace on the one hand and the overhead from ConTest on the other hand. E.g., executing *Account* 500 times took 5:18 min when we used ConTest and the instrumentation for inner-thread graphs. Executing the instrumented version without ConTest took 2:17 min; 500 runs with the original, not instrumented version of *Account* took 1:24 min The other instrumentations have a similar overhead compared to the inner-thread graph instrumentation: 500 runs with ConTest took 5:11 min for inter-thread graphs and 4:57 min for next-call graphs. So the runtime overhead is not a bottleneck of our approach, and we did not see the need to further evaluate it.

5.2 Results

In the following, we analyze how well our approach performs. We examine how many methods a software developer has to investigate and compare this for the three graph representations. Table 3 shows the results of our experiments for the different graphs and localization techniques. For instance, an entry of *2* means that the corresponding approach has ranked the defective method second. We explain for each program used how the defect manifests itself, and we investigate the effect of the number of executions used for the analysis. It can happen that it is not possible to derive a result (†). This is the case if the suspiciousness of all methods is zero, the defective method is not in the result set, or the result set is empty. Table 4 shows if the ranking in Table 3 is caused by a temporal edge or a call edge.

It is possible that the graphs become too large for FSGM. For the programs *Liveness* and *MergeSort* we were not able to analyze the temporal relations. With these relations, we faced scalability issues with FSGM and stopped any experiment after 1 h. All experiment were run on a standard Windows 7 desktop machine: 2.4 GHz dual core CPU (*Intel Core i5 M520*), with 4 GB RAM.

Account With the inter-thread graph, the edge $transfer \rightarrow_{it} transfer$ is ranked at the second position. This locates the defect, as the failure happens when two instances

Table 3 Evaluation results

Program	*InfoGain*-based			Additional FSGM			# Methods
	Next	Inter	Inner	Next	Inter	Inner	
Account	†	2	†	†	2	†	8
AirlineTickets	1	1	1	1	1	1	3
AllocationVector	2	2	2	2	2	2	8
BubbleSort	2	2	2	1	1	1	11
BubbleSort 2	2	2	2	2	2	2	4
DeadlockException	1	1	1	1	1	1	14
Liveness	1	1	1	1	1	1	11
MergeSort	1	1	1	1	1	1	15

Table 4 Edge types

Program	*InfoGain*-based			Additional FSGM		
	Next	Inter	Inner	Next	Inter	Inner
Account	†	temp	†	†	temp	†
AirlineTickets	call	call	call	call	call	call
AllocationVector	call	call	call	call	call	call
BubbleSort	call	call	call	temp	temp	temp
BubbleSort 2	call	call	call	call	call	call
DeadlockException	call	call	call	call	call	call
Liveness	call	call	call	call	call	call
MergeSort	call	call	call	call	call	call

execute the method in parallel. Next-call and intra-thread graphs do not reveal the defect, because they do not make thread changes explicit. For next-call and intra-thread graphs, we have not obtained any result. This happens as our localization techniques have not been able to find significant differences between failed and correct executions. The information gain for all edges is zero, which is caused by the nature of the defect. The method that facilitates the transfer does not call any method. This is why the defective code does not show up in the call graph.

AirlineTickets The manifestation of the defect is straightforward: In case of a failure, the method facilitating the sale is called too often. More precisely, in case of a failure the method is called more often than in executions the defective interleaving did not occur in. This is revealed by our approach.

AllocationVector The allocation process stops when a block is allocated twice for the first time. For this reason, the call frequency changes when the failure appears. This can be seen in the call graph. The change of the call frequencies is actually not the defect pattern itself, but an infection. The edge ranked higher than the defective call is a call to the JRE that only introduces noise. Besides the infection that is ranked at the second place, we also discover the defect itself, but at a low rank.

BubbleSort With FSGM, the edge ranked first is $swap \rightarrow_{temp} swap$ for all graph variants. This edge gives a good understanding of the defect. Without FSGM, one edge e_x is ranked higher than the defective call. This edge does not exist in the graphs we used with FSGM. It is an API call, and all calls to the JRE library have been reduced to a dummy vertex (see Sect. 3.4). The aggregation with other calls to the JRE library masks the call e_x. Although e_x does not pinpoint the defect, it is not noise. The call is part of the infection, and thus it is not unexpected that the edge is ranked at a high position.

BubbleSort 2 Our approach reveals the atomicity violation. It brings out the defect, because the number of calls to the method swapping elements of the data structure is too high. The method ranked higher than the defective one is the same for all variants. It is part of the infection and has the same suspiciousness as the defective one, which is at the second rank, as it has fewer LOC.

DeadlockException The edge that our approach reports at the first rank is the operation containing a defect. The manifestation in next-call graphs in combination with FSGM is straightforward: In case of an exception, the method is left, and the operation causing the exception is executed less often than in runs without the exception. In case of inter-thread and intra-thread graphs, the output of our algorithm is the edge whose sink is the method called within the catch clause. This is the first step of the infection. The actual fix is to modify the catch clause.

Liveness For the variants using FSGM, the sink of the edge reported is the code segment that resumes suspended clients. Although this segment is not the position of the most obvious fix of the defect, it still gives a good intuition that the access to the data structure holding suspended threads is problematic. The variants without FSGM output different sinks that are not directly related to the defect, but nevertheless the defective method is ranked first.

MergeSort The whole defective method is problematic. It is not possible to identify a specific call in the method that would help to fix the defect. Rather, the whole method needs to be rewritten. Nevertheless, the defective method is ranked first, as it does several suspicious calls.

We have carried out another experiment to determine how stable the results are with respect to the number of executions. We have haphazardly chosen *AirlineTickets* (AT) and *AllocationVector* (AV) from the set of benchmark programs and have lowered the number of executions step by step. Table 5 shows the rank for both programs as well as for both techniques (with and without FSGM). It can be seen that changing the number of executions does not have any effect on the result. This in turn indicates that the number of 100 or more executions is sufficient to get a good covering of the space of interleavings. We are satisfied with this insight and have not evaluated the effect of the interleavings that ConTest produces any further.

Besides the ranking approaches based on information gain that we have evaluated here, a structural ranking is possible as well. The sequential localization technique described investigated by Eichinger et al. [5] uses such a ranking. In further experiments (whose details we omit here), we have applied the structural measure described

Table 5 Effects of number of executions

# Executions	No FSGM		With FSGM	
	AT	AV	AT	AV
150	1	2	1	2
100	1	2	1	2
50	1	2	1	2

there, but it has not improved the localization for any of the graph representations. This is because the structure alone has not been discriminating regarding failing and correct executions.

In Sect. 2.5 we have raised the question how harmful infections are. For *BubbleSort* and *BubbleSort 2*, an infection has been the reason why our approach has ranked the defective method second and not first. In case of *AllocationVector* the infection has even turned out to be advantageous: The defective method is ranked second and not lower because of a call that is part of the infection. This call happens within the defective method.

The evaluation shows that our approach performs well for all experiments. Though the different graph representations perform equally well, they yield quite different result sets. Temporal edges have shown to be useful for the localization in case of *Account* and *BubbleSort*. Inter-thread graphs have been most suitable for the defect localization. In case of *Account*, this representation is advantageous. We have been able to find defects of the following bug patters: *wrong/no lock, not atomic, two-stage access, blocking critical section* and *orphaned thread*. In Sect. 3.2, we have hypothesized that intra-thread graphs can reduce the number of edges, compared to the other representations. However, the opposite applies. The average number of edges of intra-thread graphs is 36.39 compared to 25.64 in case of inter-thread graphs. Next-call graphs have 32.56 edges on average. Next-call graphs are less compact than inter-thread graph, have not performed better in any of the experiments, and the results are almost the same. The average number of vertices is 22.43. Note that an execution has the same number of vertices, no matter if represented as next-call, inter-thread or intra-thread graph.

6 Related Work

Researchers have proposed different approaches to localize defects and support the debugging process in multithreaded programs. Static race detectors like Flanagan et al. [11] try to identify data races by means of a data-flow analysis. The advantage of static tools is that they do not induce Heisenbugs, and they give the same attention to rarely executed parts of the source code. But static tools tend to have the problem that the exact program behavior is not known a priori, and they tend to produce many false positives. Flanagan et al. [11] require the software developer to insert annotations in the source code. This makes the tool aware of design decisions but requires additional effort.

Dynamic race detectors either use a *lockset-based* approach, a *happens-before* based approach or a combination of both, as [25]. If an event happens before another one, the first one can be the cause for the second one. Data races fulfill the happens-before relation, and an analysis of the partial ordering of events allows to detect them. Lockset-based detection analyzes the locks held when a shared variable is accessed. The idea is that if access to a variable is sometimes protected, but not always, then the software developer has intended a protection, and a warning is produced. Race detectors face scalability issues. Luo et al. [22] overcome this with a two-stage analysis. As a preprocessing step, uninteresting locks are filtered out. Race detectors are limited to data races and can not localize any other defects. For instance, the approach of Luo et al. [22] has not been able to locate the defects in *AllocationVector, DeadlockException, Liveness* and *MergeSort*. A problem with race detectors is that precise race detection slows down the program execution significantly. Pacer [1] introduces a sampling technique that reduces the overhead of the detector. FastTrack [12] eases the overhead of race detection by an adaptive representation that provides constant time paths for the most common kinds of races. Further, it is possible to extend the definition of data races, in order to address defects caused by several variables. Jannesari et al. [17] is such an approach that features the concepts of *computational units* and *correlated sets*. A computational unit may consist of several reads or writes. A *correlated set* is a set of variables that share a semantic consistency property. The approach is able to detect races that can only be seen if a set of variables is considered.

ConTest [4] and Chess [23] are tools designed to explore the space of interleavings and have a so-called *replay* mechanism. It records the interleaving during an execution and is able to reproduce it. This is interesting, as bugs might appear only for very few interleavings and cannot be easily reproduced. The tools use different strategies to explore the interleaving space: ConTest heuristically inserts `yield`, `sleep` and `priority` statements, Chess produces interleavings systematically. Both tools do not localize defects on their own. Rungta and Mercer [26] gives a comprehensive comparison of different defect localization tools for multithreaded programs. The evaluation there includes ConTest and Chess.

Tarantula [18] is a dynamic approach for single-threaded programs. It counts how often a statement is executed in a correct execution and how often in a failed one. Although the approach is simple, its localization precision is good. But it has not been transferred to and evaluated on multithreaded programs.

[19] is an approach that is able to fix defects automatically. The authors use genetic programming and are able to fix programs without requiring a priori knowledge or specifications. The approach is highly scalable, and 55 out of 105 defects could be fixed in the evaluation. However, even though the approach scales well, it is requires high computational effort. The approach was evaluated on C programs and is not tailored towards multithreading defects. According to the paper, it is unlikely that the approach is able to fix non-deterministic defects.

Several localization approaches [3,5,20] use dynamic graphs to represent program executions. To our knowledge, the only approach for localizing multithreaded defects that also takes sequential parts of the program into account is the one of [6]. They use graphs to model program executions as well. But their representation does not model temporal relations, and the work of Eichinger et al. [6] does not investigate the usefulness of FSGM.

7 Conclusions

Defect localization is a fundamental issue. The time software developer spend on it is huge, in the case of multithreaded software in particular. State-of-the-art approaches either deal with sequential programs only or are limited to specific bug patterns. In this paper, we have proposed an approach without these limitations, based on data mining.

It performs well for defects that race detectors cannot find. The evaluation has shown that modeling the ordering of events in the graphs is important. In the evaluation, we also observed that—for the programs evaluated—FSGM has not been advantageous. Without FSGM it is possible to analyze even larger programs, because one does not have to deal with the performance downsides of the respective algorithms.

A possibility to further enhance the graph representations is to annotate the edges with parameter and return values. Further, another annotation could say which locks are held at the time of a method invocation. This is part of ongoing studies.

References

1. Bond, M.D., Coons, K.E., McKinley, K.S.: Pacer: proportional detection of data races. In: PLDI (2010)
2. Copty, S., Ur, S.: Multi-threaded testing with AOP is easy, and it finds bugs!. In: 11th Int. Euro-Par Conf. (2005)
3. Di Fatta, G., Leue, S., Stegantova, E.: Discriminative pattern mining in software fault detection. In: 3rd Int. Workshop on Software Quality Assurance (SOQUA) (2006)
4. Edelstein, O., et al.: Framework for testing multi-threaded java programs. Concurr. Comput. Pract. Exp. **15**:485–499 (2003)
5. Eichinger, F., Böhm, K., Huber, M.: Mining edge-weighted Call graphs to localise software bugs. In: ECML PKDD (2008)
6. Eichinger, F., et al.: Localizing defects in multithreaded programs by mining dynamic call graphs. In: TAIC PART (2010)
7. Eichinger, F., Oßner, C., Böhm, K.: Scalable software-defect localisation by hierarchical mining of dynamic call graphs. In: SDM (2011)
8. Engler, D., Ashcraft, K.: Racerx: effective, static detection of race conditions and deadlocks. SIGOPS **37**:237–252 (2003)
9. Eytani, Y., Ur, S.: Compiling a benchmark of documented multi-threaded bugs. In: Proc. of the Parallel and Distributed Processing Symposium (2004)
10. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: Proc. Int. Parallel and Distributed Processing Symposium (IPDPS) (2003)
11. Flanagan, C., et al.: Extended static checking for java. In: PLDI (2002)
12. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: Hind, M., Diwan, A. (eds.) PLDI, pp. 121–133. ACM (2009)
13. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, San Francisco, CA (1979)
14. Gray, J.: Why do computers stop and what can be done about it?. In: Symposium on Reliability in Distributed Software and Database Systems (1986)
15. Hutchins, M., et al.: Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Int. Conf. on Software Engineering (ICSE) (1994)
16. Jalan, R., Kejariwal, A.: Trin-trin: Who's calling? a pin-based dynamic call graph extraction framework. International Journal of Parallel Programming **40**:410–442 (2012)
17. Jannesari, A., et al.: Dynamic data race detection for correlated variables. In: Proc. of the Int. Conf. on Algorithms and Architectures for Parallel Processing (2011)
18. Jones, J., Harrold, M., Stasko, J.: Visualization of test information to assist fault localization. In: 24th Int. Conf. on Software Engineering (ICSE) (2002)
19. Le Goues, C., et al.: A systematic study of automated program repair: fixing 55 out of 105 bugs for 8 each. In: ICSE (2012)
20. Liu, C., et al.: Mining behavior graphs for "Backtrace" of Noncrashing Bugs. In: SDM (2005)
21. Lu, S., et al.: Learning from mistakes—a comprehensive study on real world concurrency bug characteristics. SIGARCH **36**:329–339 (2008)
22. Luo, Z., Das, R., Qi, Y.: Multicore sdk: a practical and efficient deadlock detector for real-world applications. In: IEEE Fourth Int. Conf. on Software Testing, Verification and Validation (ICST), pp. 309–318, March (2011)

23. Musuvathi, M., Qadeer, S., Ball, T.: CHESS: A Systematic Testing Tool for Concurrent Software. Technical report, Microsoft Research (2007)
24. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proc. of the Int. Conf. on Software Engineering (ICSE) (2006)
25. O'Callahan, R., Choi, J.: Hybrid dynamic data race detection. SIGPLAN Notices **38**:167–178 (2003)
26. Rungta, N., Mercer, E.G.: Clash of the titans: tools and techniques for hunting bugs in concurrent programs. In: PADTAD (2009)
27. Savage, S., et al.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**:391–411 (1997)
28. Vermeulen, A.L., et al.: In: Vermeulen, A.L. (ed.) The Elements of Java Style. Cambridge University Press (2000)
29. Yan, X., Han, J.: CloseGraph: mining closed frequent graph patterns. In: KDD (2003)
30. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann (2009)