Autom Softw Eng (2015) 22:475–515
DOI 10.1007/s10515-014-0150-6

Unfolding based automated testing of multithreaded programs

Kari Kähkönen · Olli Saarikivi · Keijo Heljanko

Received: 31 August 2013 / Accepted: 20 April 2014 / Published online: 3 May 2014

Abstract In multithreaded programs both environment input data and the nondeterministic interleavings of concurrent events can affect the behavior of the program. One approach to systematically explore the nondeterminism caused by input data is dynamic symbolic execution. For testing multithreaded programs we present a new approach that combines dynamic symbolic execution with unfoldings, a method originally developed for Petri nets but also applied to many other models of concurrency. We provide an experimental comparison of our new approach with existing algorithms combining dynamic symbolic execution and partial order reductions and show that the new algorithm can explore the reachable control states of each thread with a significantly smaller number of test runs. In some cases the reduction to the number of test runs can be even exponential allowing programs with long test executions or hard-to-solve constraints generated by symbolic execution to be tested more efficiently. In addition we show that our algorithm generates a structure describing different interleavings from which deadlocks can be detected efficiently as well.

Keywords Dynamic symbolic execution · Unfoldings · Automated testing · Partial order reduction

K. Kähkönen (✉) · O. Saarikivi · K. Heljanko
Department of Computer Science and Engineering, School of Science,
Helsinki Institute for Information Technology HIIT, Aalto University,
Box 15400, 00076 Aalto, Finland
e-mail: kari.kahkonen@aalto.fi

O. Saarikivi
e-mail: olli.saarikivi@aalto.fi

K. Heljanko
e-mail: keijo.heljanko@aalto.fi

1 Introduction

Designing correct multithreaded programs is challenging due to the large number of ways how different threads can interleave their execution. For example, if there are n independent operations being executed concurrently, there are $n!$ possible interleavings. Different interleavings can lead to different states in the program and therefore a programmer needs to make sure that no interleaving leads to an erroneous state. It is easy for the programmer to miss some of the possible interleavings and therefore an automated way to test multithreaded programs systematically can be of great help in increasing the confidence that the program is functioning as intended.

To test a multithreaded program automatically, it is easy to generate an execution tree that represents all the possible interleavings of the program. This execution tree can be finite or infinite depending whether there is a cycle in the state space of the program. A large number of these interleavings can be irrelevant for checking properties like the reachability of a control state in the program. This is because some of the operations that are executed by one thread can be independent with operations in other threads in such a way that the program ends up in the same state regardless in which order the independent operations are executed. Because of this the execution paths though the program can be partitioned into equivalence classes such that it is enough to test one execution path from each equivalence class and still capture interesting defects, such as all assertion violations and deadlocks.

One approach to achieve such partitioning is to use partial order reduction methods such as persistent sets by Godefroid (1996), stubborn sets by Valmari (1991) and ample sets by Peled (1993). With these methods it is possible to compute which subtrees in the execution tree can be safely ignored. An alternative way to fight state explosion is to use a "compression approach" by constructing a symbolic representation of all the possible interleavings such that the representation is more compact than the full execution tree. Unfoldings, first introduced in the context of verification algorithms by McMillan (1992), is an example of such a representation (see Esparza and Heljanko 2008 for an extensive survey on the topic).

In this paper we present a new testing approach based on unfoldings that enables us to construct such a compact representation of the interleavings of multithreaded programs. This makes it possible for our algorithm to avoid exploring irrelevant interleavings efficiently when the aim is to cover all the local control locations of the threads. This still allows errors such as assertion violations to be detected. Furthermore, the resulting symbolic representation can be used to detect deadlocks as well. By using unfoldings, our testing algorithm can sometimes cover the program with a substantially smaller number of test runs than current related approaches based on partial order reductions.

We will also show how this approach can be combined with dynamic symbolic execution (DSE) to test multithreaded programs that use input values as part of the execution. Symbolic execution of sequential programs can also be seen as a compression approach. While it is easy to consider all possible combinations of input values, this becomes quickly infeasible because just two 32-bit integer input values would generate $(2^{32})^2$ possible combinations. Symbolic execution constructs a symbolic representation that partitions the input values to equivalence classes such that all the possible input values are still represented by the symbolic structure. Given this similarity, we feel that the unfolding approach integrates to DSE in a natural way.

It is also possible to combine partial order reduction methods with DSE. Persistent set based dynamic partial order reduction (Flanagan and Godefroid 2005) and race detection and flipping (Sen and Agha 2006b) are examples of algorithms presented recently that are suitable for this. In this work we compare our approach with these algorithms and discuss the advantages and disadvantages of using a compression approach over reduction methods in testing multithreaded programs. Especially, we show that our algorithm handles better cases where some of the threads in a program are highly independent.

This paper extends our previous work (Kähkönen et al. 2012) and the main contributions of this paper are: (1) giving an extended description of the testing algorithm, (2) further reducing the search space that an algorithm computing new test targets needs to explore, (3) describing how our approach can be extended to detect deadlocks, (4) giving a more detailed comparison of the differences between our algorithm and related approaches, and (5) reporting results of experiments on additional case studies. Based on the experiments, we show that our approach that combines unfoldings and dynamic symbolic execution is a competitive alternative to approaches using partial order reductions together with dynamic symbolic execution. In particular, our algorithm is computationally more expensive per test execution but it can avoid generating tests that current partial order reduction based approaches cannot. In some cases the difference between the number of test runs can even be exponential.

2 Background

In this section we will first state our assumptions about programs that are given as input to our algorithm. We will also give a brief overview of dynamic symbolic execution and net unfoldings, as those are the two main techniques behind our testing algorithm. We also discuss related approaches that use DSE and partial order reductions to test multithreaded programs in order to give a better understanding of the advantages and disadvantages of our new approach.

2.1 The language for programs

To formalize our algorithm, we use a simple language to describe the programs that can be tested. To keep the presentation simple, this language does not not contain dynamic thread creation or dynamically varying number of shared variables. Instead, these are fixed at program start time. Handling these features in the context of Java programs is discussed in Sect. 4. The syntax of the language for describing threads in the programs is shown in Table 1 and it can be seen as a subset of programming languages such as C, C++ or Java.

We assume that the only nondeterminism in threads is caused by concurrent access of shared variables or by input data from the environment. We also assume that operations accessing the shared memory are sequentially consistent. Operations that access shared variables are called visible operations as they can be used to share informa-

Table 1 Simplified language syntax

T	::=	Stmt*	(thread)
Stmt	::=	l: S	(labeled statement)
S	::=	lv := e | sv := lv | **if** b goto l' | **lock**(lc) | **unlock**(lc)	(statement)
e	::=	lv | sv | c | lv op lv | input	(expression)
b	::=	**true** | **false** | lv = lv | lv $\neq$ lv | lv $<$ lv | lv $>$ lv | lv $\geq$ lv | lv $\leq$ lv	(boolean expression)
		op $\in$ {+, -, *, /, %, …},	
		lv is a local variable, sv is a shared variable, lc is a lock and c is a constant	

tion between the threads. The state of a multithreaded program consists of the local state of each of the threads and the shared state consisting of the shared memory. The visible operations considered in this work are read and write of shared variables and acquire and release of locks. We assume that a read operation reads a value from a shared variable and assigns it to a variable in the local state of the thread performing the operation. Write assigns either a constant or a value from a local variable to a shared variable. Non-visible operations, such as if-statements, are evaluated solely on the values in the local state of the executing thread and therefore cannot access shared variables directly. In real programs the statements can be modified automatically to satisfy these assumptions without changing the behavior of the program.

2.2 Dynamic symbolic execution

Dynamic symbolic execution (DSE) (Godefroid et al. 2005), which is also known as concolic testing, is a popular systematic test generation approach used in many tools such as PEX (Tillmann and de Halleux 2008), CUTE (Sen 2006) and KLEE (Cadar et al. 2008). In dynamic symbolic execution a program is executed both concretely and symbolically at the same time. The concrete execution corresponds to the execution of the actual program under test and symbolic execution computes constraints on values of the variables in the program by using symbolic values that are expressed in terms of inputs to the program. At each branch point in the program's execution, the symbolic constraints specify the input values that cause the program to take a specific branch. As an example, let us consider a program `x = x + 1; if (x > 0);` and assume that x has initially the symbolic value $input_1$. Executing this program symbolically would generate constraints $input_1 + 1 > 0$ and $input_1 + 1 \leq 0$ at the if-statement.

A path constraint is a conjunction of the symbolic constraints corresponding to each branch point in a given execution path. All input values that satisfy a path constraint will explore the same execution path for sequential programs. If a test execution goes through multiple branch points that depend on the input values, a path constraint can be constructed for each of the branches that were left unexplored along the execution path. These constraints are typically solved using SMT-solvers in order to obtain concrete values for the input symbols. This allows all the feasible execution paths thought the program under test to be explored systematically. This can be seen as a bridge between testing and model checking. The main challenge with DSE is that the number of feasible execution paths is in many cases too large to explore exhaustively. However, compared to random testing, DSE can often obtain better coverage of the program under test in cases where there exists execution paths that require very specific input values and are therefore unlikely to be generated randomly.

2.3 Testing multithreaded programs with dynamic symbolic execution

In multithreaded programs the order in which the operations from different threads are executed affects the execution path as well. This additional nondeterminism can be handled by taking control of the scheduler and considering the schedule as an input to the system. To be more precise, each thread in the program under test is always executed one thread at a time to a point where the next operation for each thread is a visible operation (e.g., access to a shared memory location). At such scheduling points the test scheduler chooses one enabled thread to proceed until the next visible operation is encountered. A backtracking search can then be made to explore all scheduling possibilities. In this approach the execution from one scheduling point to another contains one visible operation as well as the local operations of the scheduled thread up to the next visible operation. This way the interleavings of local operations are automatically ignored. This approach of avoiding unnecessary interleavings is used in many algorithms such as DPOR by Flanagan and Godefroid (2005).

The execution paths resulting from different interleavings and input values can be represented as a symbolic execution tree as described by Saarikivi et al. (2012). The nodes in such a tree are either scheduling points, with branches for different enabled operations, or local branching points, where the branches correspond to symbolic constraints generated by DSE. As an example, Fig. 1 shows a simple program and its symbolic execution tree. The scheduling nodes in the figure are marked with a darker color. Every possible execution of the program follows one of the paths from the root node to a leaf node. In the figure, $input_1$ and $input_2$ denote the symbolic input values for thread 1 and 2, respectively.

The problem with exploring a full symbolic execution tree is that the number of possible interleavings is generally too large to explore exhaustively even without input value based branching. However, many interesting properties such as assertion violations or deadlocks can be checked without exploring all interleavings. For example, the order in which two concurrent read operations are executed does not affect the behavior of the program as the operations are independent. It is possible to partition the execution paths into equivalence classes such that any two paths in the same equivalence class can be obtained from one another by performing a number of swaps between adjacent independent operations. These equivalence classes are often called

```
Global variables:   Thread 1:                        Thread 2:
int x = 0;          1: local i1 = input() + 1;       6: local i2 = input();
int y = 0;          2: if (i1 == 5)                  7: local b = x;
                    3:     x = 5;                    8: if (i2 > 10)
                    4: else                          9:     b = 0;
                    5:     local a = x;
```

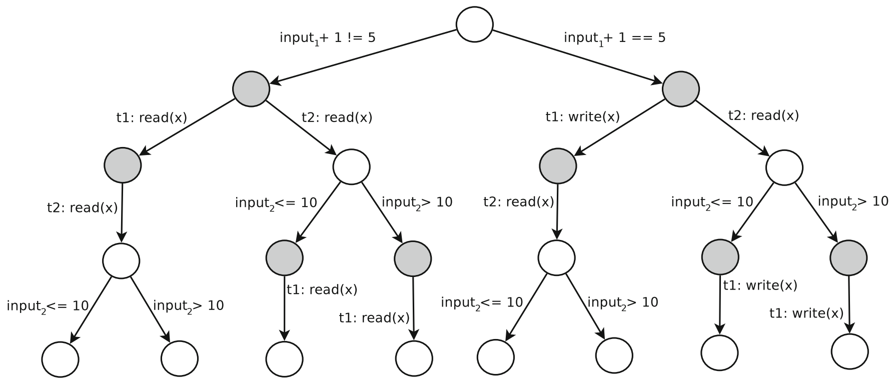
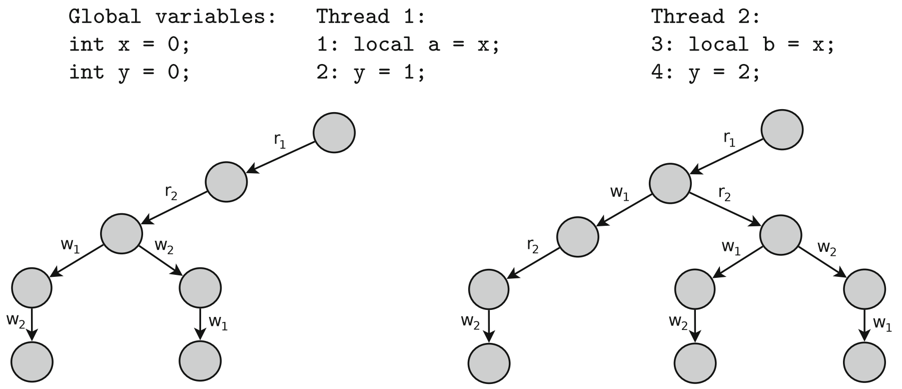


Fig. 1 Symbolic execution tree with all interleavings

Mazurkiewicz traces (Diekert 1995). It is therefore possible to ignore some subtrees of the symbolic execution tree if it can be shown that the same behavior will be explored in another subtree. For example, in Fig. 1 there is a scheduling point where two reads are enabled. Exploring both of these subtrees is not necessary.

Partial order reduction algorithms provide one possibility to determine which parts of the symbolic execution tree need to be explored. Typical partial order reduction algorithms often use static analysis to compute a set of transitions that need to be explored from each state of the program. However, in this paper we are considering dynamic analysis techniques that obtain the necessary information during runtime. Therefore we need to use dynamic variants of the partial order reduction algorithms. Two such existing algorithms that have been combined with DSE are dynamic partial order reduction (DPOR) by Flanagan and Godefroid (2005) and race detection and flipping by Sen and Agha (2006b). The main idea behind these algorithms is to look back in the history of the current execution to determine if there are some transitions that can be in race with the current transition. If such transitions are found, firing the transitions in different order can lead to different Mazurkiewicz traces. Therefore a backtrack search is performed to explore the alternative interleaving. These kind of approaches, however, can explore unnecessary interleavings in cases where there are independent transitions between the transitions that are in race. Let us consider a program with two threads that first read a shared variable x and then write to a shared variable y. Let us assume that the program is tested such that in the first test both of the read operations are executed before the writes. In this case a dynamic partial order reduction algorithm notices that the writes are in race and simply swaps them. This results in exploring the execution paths shown on the left side of Fig. 2. In the figure, the reads are denoted with r and writes with w. The subscript indicates the executing thread. However, if the first execution is such that the first thread performs



```
Global variables:   Thread 1:             Thread 2:
int x = 0;          1: local a = x;       3: local b = x;
int y = 0;          2: y = 1;             4: y = 2;
```

Fig. 2 The effect of execution order in avoiding unnecessary interleavings

both of its operations before the second thread, the partial order reduction algorithm again notices that the writes are in race but in order to swap them, the read operation of the second thread needs to be executed before the writes are executed. This leads to testing an execution path where the read operations occur first. In this execution path the write operations are again in race which leads to performing also a third test execution. These execution paths are illustrated on the right side of Fig. 2.

The problem here is that the algorithms do not consider other branches in the tree to determine if some equivalent execution has already been performed. This problem can be alleviated by using sleep sets (Godefroid 1996). For example, in Fig. 2, sleep sets would prevent the execution path (r_1, r_2, w_1, w_2) to be followed in the larger tree as it would use the information that a path starting with (r_1, w_1) has been explored. Therefore in any other branch after r_1 there is no need to execute w_1 before a transition that is dependent with it is encountered. With sleeps sets the number of test executions that do not end up in a state where all enabled threads are in the sleep set is the same as the number of Mazurkiewicz traces. However, DPOR might still perform unnecessary test executions where all threads end up in the sleep set.

Let us consider another program with $2n$ threads and n shared variables such that for all the shared variables there is one thread that reads the variable and one that writes to it. In other words, there are n pairs of threads that are independent to each other. In such program the number of Mazurkiewicz traces grows exponentially as the number of threads increases. The existing partial order reduction algorithms will test all of these Mazurkiewicz traces which results in exploring an execution tree similar as in Fig. 3. However, assuming that we are interested in covering only the local control states of programs, it would be enough to perform just two tests for this program: one in which each thread pair performs the read operation first and another where the writes are performed first.

The examples above motivate the need for an alternative approach to represent possible interleavings when the aim is to cover the local states in the program under test instead of all Mazurkiewicz traces. In this paper we present one such alternative based on unfoldings.

```
Global variables:   Thread 1:          Thread 2:   Thread 3:          Thread 4:
int x = 0;          1: local a = x;    2: x = 1;   3: local b = y;    4: y = 2;
int y = 0;
```

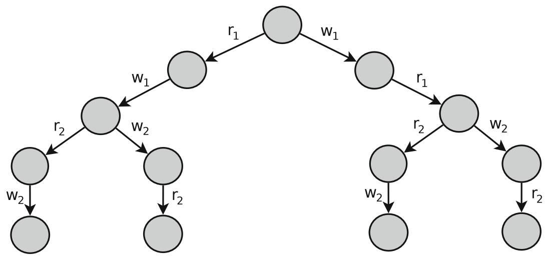


Fig. 3 Exponentially growing example

2.4 Definitions for net unfoldings

In the following we define unfoldings based on Petri nets and will use them to unfold our multithreaded programs.

2.4.1 Petri Nets

A *net* is a triple (P, T, F), where P and T are disjoint sets of places and transitions, respectively, and $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation. Places and transitions are called nodes and elements of F are called arcs. The preset of a node x, denoted by ${}^\bullet x$, is the set $\{y \in P \cup T \mid F(y, x) = 1\}$. The postset of a node x, denoted by $x^\bullet$, is the set $\{y \in P \cup T \mid F(x, y) = 1\}$. A marking of a net is a mapping $P \mapsto \mathbb{N}$. A marking M is identified with the multiset which contains $M(p)$ copies of p. Graphically markings are represented by putting tokens on circles that represent the places of a net. A Petri net is a tuple $\Sigma = (P, T, F, M_0)$, where (P, T, F) is a net and M_0 is an initial marking of (P, T, F).

2.4.2 Causality, conflict and concurrency

Causality, conflict and concurrency between two nodes x and y in a net are defined as follows:

- Nodes x and y are in causal relation, denoted as $x < y$, if there is a non-empty path of arcs from x to y. In this case we say that x causally precedes y.
- Node x is in conflict with y, denoted as $x\#y$, if there is a place z different form x and y such that $z < x$, $z < y$ and the paths from z to x and from z to y take different arcs out of z.
- Node x is concurrent with y, denoted as $x\ co\ y$, if nodes are neither causally related ($x < y$ or $y < x$) nor in conflict.

Any two nodes x and y, such that $x \neq y$ are either causally related, are in conflict or are concurrent. A *co-set* is a set of nodes where all nodes in the set are pairwise concurrent.

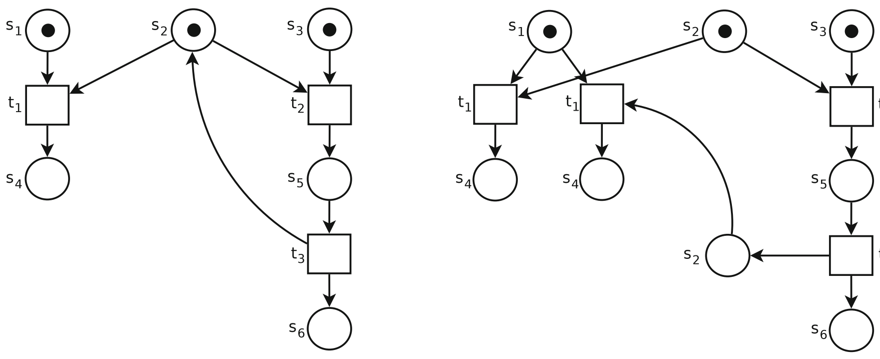


Fig. 4 A Petri net and its unfolding

2.4.3 Unfoldings

A directed graph can be unwound into a (possibly infinite) tree when starting from a root node such that each node in the tree is labeled by a corresponding node in the graph. For nondeterministic sequential programs this unwinding would be the computation tree of the program, where the computations maximally share their prefixes. In a similar way, Petri nets can be unfolded into labeled *occurrence nets*. These nets have a simple DAG-like structure. A multithreaded program can be unwound to a computation tree but it can also be represented as an unfolding that allows the prefixes of computations to be shared even more succinctly. Intuitively these unfoldings represent both the causality of events as well as the conflicts between them.

Formally, an occurrence net O is a net (B, E, G), where B and E are sets of places and transitions and G is the flow relation. Furthermore, O is acyclic, that is, for every b in B, $|{}^\bullet b| \leq 1$; for every $x \in B \cup E$ there is a finite number of nodes $y \in B \cup E$ such that $y < x$; and no node is in conflict with itself. To avoid confusion when talking about Petri nets and their occurrence nets, the nodes B and E are called *conditions* and *events*, respectively.

A *labeled occurrence net* (also called a branching process) is, by adapting the definition from the paper by Khomenko and Koutny (2001), a tuple $(O, l) = (B, E, G, l)$ where $l : B \cup E \mapsto P \cup T$ is a labeling function such that:

- $l(B) \in P$ and $l(E) \in T$;
- for all $e \in E$, the restriction of l to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet l(e)$;
- the restriction of l to $Min(O)$ is a bijection between $Min(O)$ and M_0, where $Min(O)$ denotes the set of minimal elements with respect to the causal relation;
- for all $e, f \in E$, if ${}^\bullet e = {}^\bullet f$ and $l(e) = l(f)$ then $e = f$.

It is possible to obtain different labeled occurrence nets by stopping the unfolding process at different times. The maximal labeled occurrence net (possibly infinite) is called *the unfolding* of a Petri net. To simplify the discussion in this paper, we use the term unfolding for all labeled occurrence nets and not just the maximal one.

Example 1 Figure 4 shows a Petri net on the left side and its unfolding on the right side. Notice that the conditions and events in the unfolding are labeled with the cor-

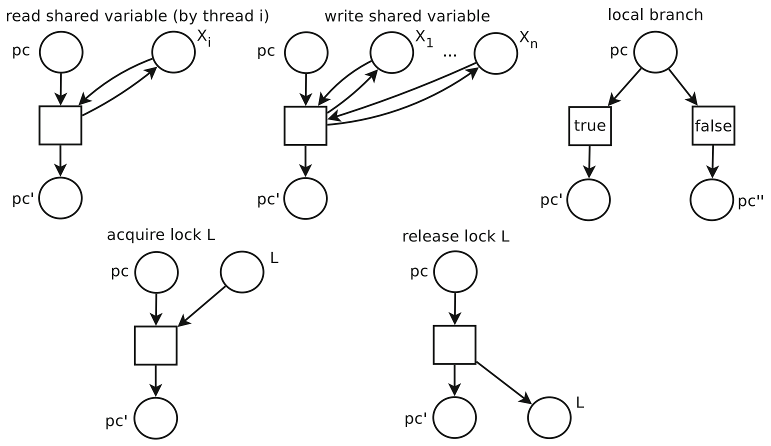


Fig. 5 Modeling constructs

responding places and transitions in the Petri net. For any computation (sequence of transitions) in the Petri net there is a matching computation in the unfolding. If a computation visits the same node multiple times, in the unfolding there is always a new copy for the node. In the figure this can be seen as the two conditions for s_2.

Let us also consider some nodes in this unfolding. The condition in the initial marking that is labeled with s_2 causally precedes both conditions labeled with s_4. The bottommost condition labeled with s_2 is in conflict with leftmost condition labeled with s_4 and causally precedes the other.

3 Modeling execution paths with unfoldings

Dynamic symbolic execution can be seen as a technique that unwinds a control flow graph of a program to a symbolic representation of possible execution paths (e.g., to a symbolic execution tree). Our unfolding based approach works similarly: the control and data flow of a program can be represented as a Petri net which is then unfolded. This results in an acyclic structure where the control flow paths of each thread are unwound like in standard DSE and additional branching structures are used to model the interleaved accesses to shared memory. Essentially our approach constructs an symbolic execution tree for each thread and uses arcs that represent data flow to express the causality between events belonging to different threads. As in traditional DSE, we do not construct an explicit control flow representation that is unfolded but use the information obtained during runtime instead. However, considering such a representation can be helpful in understanding our algorithm. To obtain a Petri net representing the control and data flow of a program, we use the same constructs as described by Farzan and Madhusudan (2006), who use them to model the control flow of programs to do atomicity checking. These constructs are shown in Fig. 5.

In this approach of representing a program, for every thread there is a set of places that correspond to the program locations (i.e., program counter values) the thread can

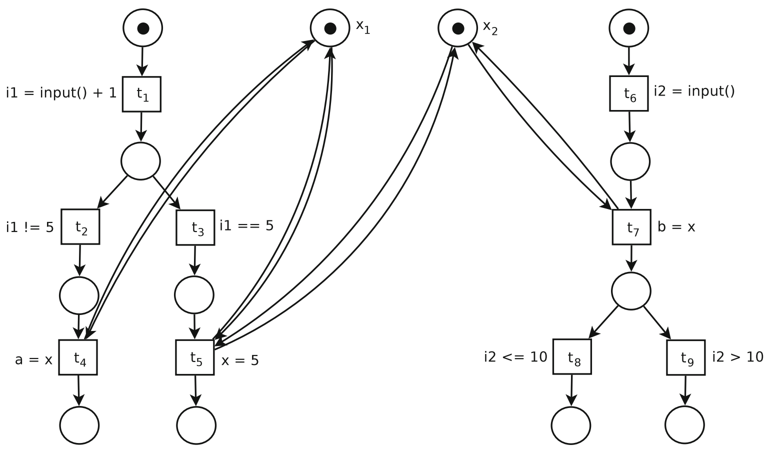


Fig. 6 Petri net representation of the program in Fig. 1

be in. There is also a place for each lock and *n* places for each shared variable, one per thread. The reason why shared variables are modeled with *n* places is that this way each thread has a local copy of the variable and two reads of the same variable become independent by construction. The benefits of this will become more apparent when the unfolding process of such Petri nets is discussed.

The transitions in the model correspond to the operations that threads can perform. In Fig. 5 the places for program locations are labeled with program counter values (e.g., a thread is at program location *pc* and after firing a transition it will be in a location pc'). The shared variable places are labeled with the name of the variable and the subscript indicates to which thread that local copy of the variable belongs to. Lock places are labeled with the name of the lock.

Initially all places for locks, shared variables and initial program locations are marked. As with control flow graphs, the data is abstracted away. For this reason all the valuations of shared variables are represented by the same set of places and any transition taking a token from these places puts it immediately back. As an example, Fig. 6 shows a Petri net representation of the program from Fig. 1. Note that like control flow graphs, the Petri net representations are overapproximations of possible execution paths. The runtime information obtained during testing then determines which of the paths are feasible when data values are also taken into consideration.

Constructing an unfolding of the program under test can be done by modeling operations encountered during test executions with the unfolded counterparts of the constructs in Fig. 5. As with the symbolic execution tree example in the previous section, we only add events for visible operations and branching operations depending on input values to the unfolding. The resulting net can then be seen as an unfolding of the Petri net representation without events for local non-symbolic transitions. We also store the symbolic constraints obtained from DSE to the events that model the branching decisions. Note that in standard DSE that is based on depth first search of the control flow graph, there is no need to store the symbolic constraints or to construct the symbolic execution tree explicitly. This is because in depth first search

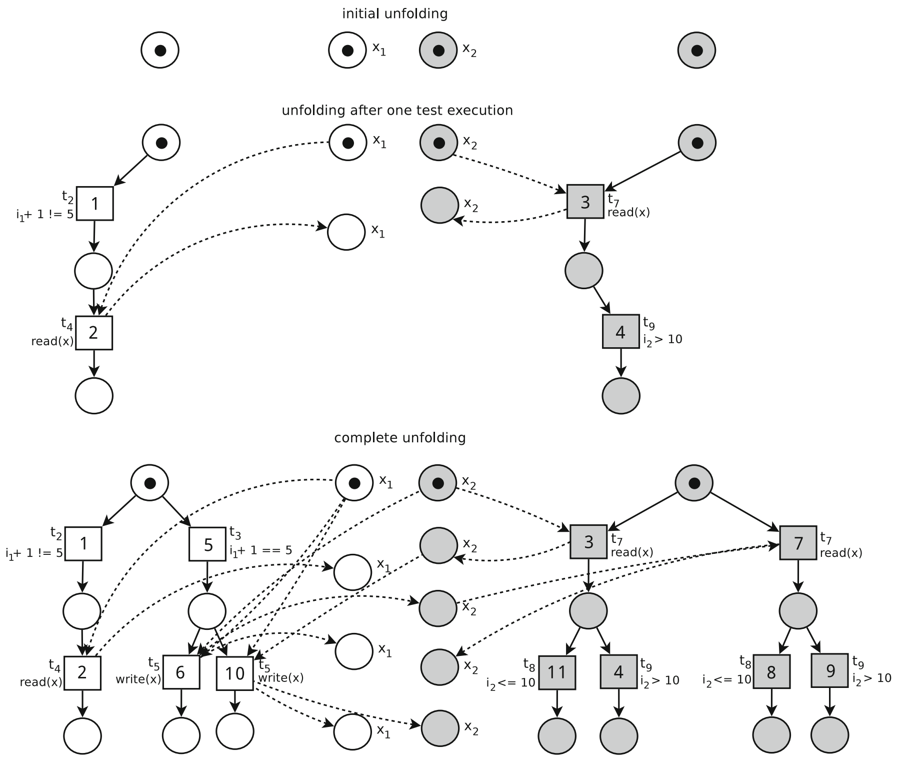


Fig. 7 Representing execution paths as an unfolding

the path constraint for a test execution can be obtained directly from the previous one. Compared to such an approach, our algorithm has additional space requirements as the unfolding and the constraints need to be stored. This, however, makes it possible to explore the execution paths in arbitrary order. It is also possible to use other than depth first search with standard DSE by storing additional information like in our approach. When talking about unfoldings, we refer to conditions that correspond to places for program location, locks and shared variables as *thread conditions*, *lock conditions* and *shared variable conditions*, respectively.

To illustrate how unfoldings can be constructed based on test executions alone, let us consider the program in Fig. 1. Let us assume that we want to first model a test execution corresponding to the sequence of program locations (1, 2, 6, 5, 7, 8, 9). The modeling process starts with an initial unfolding that corresponds to the initial marking of the Petri net representation of the program. This is shown as the topmost net in Fig. 7. We also maintain a marking M that at the beginning of each test execution matches the initial marking of the unfolding. Executing line 1 corresponds to a local operation that does not depend on input values and is not explicitly modeled in the unfolding. The result of executing line 2, however, depends on input values. A new event with a label 1 is added to the unfolding to represent that the false branch can be executed at this point. The symbolic constraint $i_1 + 1 \neq 5$ captured by DSE is also stored to this event, where i_1 denotes the input value read by thread 1. The marking M is then updated by firing the event. This results in a new marking that indicates in which symbolic state the execution is at the moment. The reachable markings in the unfolding correspond to the nodes in the symbolic execution tree as they also represent reachable symbolic states. The operation executed at line 6 is again a local operation that is not modeled. After this, thread 1 performs a read operation and this is represented by the event 2. Note that this event is simply the unfolded version of the read construct in Fig. 5. To make the graphical representations of unfoldings more readable, the arcs between shared variable conditions and events are shown as dashed lines. Such arcs represent the data flow of shared variables but do not differ semantically from normal arcs. The event 2 is then fired to update M. The rest of the operations observed by the test execution are processed in a similar manner. The resulting unfolding that models the first test execution is shown in Fig. 7.

The unfolding process can then be continued by performing additional test executions. In these executions, an event for an observed operation is added to the unfolding only if a matching event does not already exist. The full unfolding obtained by exploring all execution paths through the example program is shown at the bottom of Fig. 7. Note that the write and read operations in the example program can be in race as illustrated by events 3 and 6. In the unfolding both ways to interleave such operations are explicitly represented with different sets of events. This is similar as having different branches for the interleavings in an execution tree. If two co-enabled operations are not in race, like the read operations in the example, both ways to interleave the operations are represented by the same events. Note also that if the shared variables were not represented by n places, the unfolding process would consider the reads to be dependent. This would lead to exploring unnecessary interleavings.

It is easy to model the explored execution paths as nets with the approach shown above. However, assuming that we start with a random test execution, the question remains how the net corresponding to the executions observed so far can be used to compute new test inputs and schedules such that all local states of the threads in the program get eventually covered. In other words, a set of test executions that cover all the transitions in the Petri net representation of the program is needed. In the next section we will show how this can be computed systematically.

4 Systematic testing with unfoldings

In this section we describe our algorithm that systematically tests a given program by constructing and exploring an unfolding representation of it. We start by describing the overall idea of our algorithm and then give further details of the subroutines used by the algorithm in the following subsections.

4.1 The testing algorithm

The main structure of the testing algorithm is presented in Fig. 8. The algorithm starts with an initial unfolding that is constructed as described in Sect. 3 and it also maintains a set of possible extensions that are events that can be added to the unfolding to extend

Fig. 8 Unfolding algorithm

```
Input: A program $P$
 1: $unf$ := initial unfolding
 2: $extensions$ := events enabled in the initial state
 3: while $extensions \neq \emptyset$ do
 4:     choose $target \in extensions$
 5:     if $target \notin unf$ then
 6:         $operation\_sequence$ := EXECUTE($P, target, k$)
 7:         $marking$ = initial marking
 8:         for all $op \in operation\_sequence$ do
 9:             $e$ = CORRESPONDINGEVENT($op, marking$)
10:             if $e \notin unf$ then
11:                 add $e$ and its output conditions to $unf$
12:                 $extensions := extensions \setminus \{e\}$
13:                 $pe$ := POSSIBLEEXTENSIONS($e, unf$)
14:                 $extensions := extensions \cup pe$
15:             $marking$ = FIREEVENT($e, marking$)
```

it. The algorithm then performs the following loop until it cannot find new events to be added to the unfolding. The loop starts by selecting one event from the possible extensions that is not already in the unfolding (lines 4–5). The EXECUTE subroutine then performs a test execution using dynamic symbolic execution to cover the selected event by a concrete test execution (line 6). It is also possible that the target event is not reachable with any input values. This results in cutting the event from the unfolding. A bound k is used to limit the length of the test execution. Using such bound is standard in DSE based approaches to guarantee the termination of the algorithm. The execution collects a sequence of visible operations and symbolic branch operations that characterize the observed execution. The algorithm then proceeds to model this sequence as events in the unfolding by first creating a marking that corresponds to the initial marking. The operation sequence is then processed one operation at a time. First it is determined if there already exists an event in the unfolding that corresponds to the operation. This is done by the subroutine CORRESPONDINGEVENT, which checks from the unfolding if there is in the current marking an event enabled which belongs to the same thread as the operation and which is of the same type (line 9). If no such event exists, the algorithm adds a new event to the unfolding as described in Sect. 3. If the event was in the set of possible extensions, it is also removed from there so that it is not selected as a target node in further iterations. The algorithm then computes a set of new events that can become enabled in some marking that is reachable after the event that was added has been fired. These events are added to the set of possible extensions. The possible extension computation is performed by the subroutine POSSIBLEEXTENSIONS. Finally the current marking is updated by firing the event corresponding to the operation that was processed (line 15) and the algorithm can start processing the next operation.

Example 2 To better understand the testing algorithm, let us consider the simple program shown in Fig. 9. Let the first test execution be such that thread 1 is first executed fully followed by thread 2. The operation sequence for this test consist of a read operation `a = x` by thread 1 and the operations `b = x` and `x = input()` by thread 2. Notice that the if-statements do not generate operations for the sequence as they do

```
Global variables:   Thread 1:           Thread 2:
int x = 0;          local int a = x;    local int b = x;
                    if (a > 0)          if (b == 0)
                       error();           x = input();
```

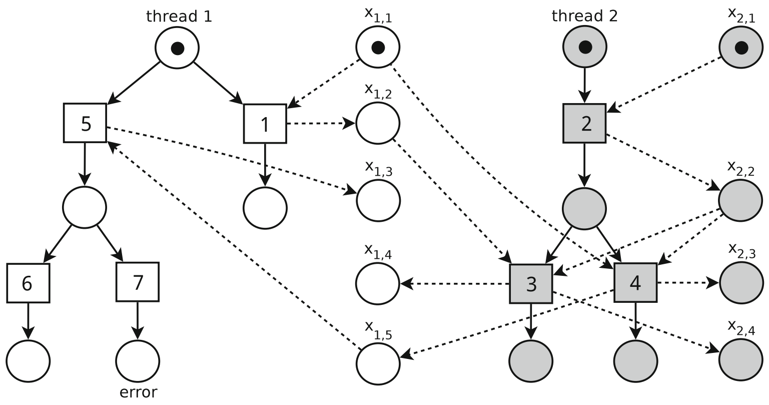


Fig. 9 Complete unfolding of the example

not depend on input values in this particular execution. The algorithm then determines if there exists an event corresponding to the first operation in the generated sequence. As no such event exists, a new one is added to the unfolding. This event is the read event labeled with 1. The algorithm then checks if it is possible to add any new events to the unfolding that can be executed after event 1. There are no such possible extension events and therefore the event 1 is fired in order to obtain a new marking that corresponds to the current state of the execution. The algorithm processes the second operation from the sequence and adds the event 2 to the unfolding. For this event the possible extensions algorithm notices that a write event can be performed in the current marking (event 3) or the write event can be performed in a marking where event 2 has been fired but event 1 has not (event 4). These two events are added to the set of possible extensions and event 2 is fired. For the last operation, the algorithm adds the event 3 to the unfolding and removes it from the set of possible extensions. No possible extensions are found for this event.

For the second test run, an event from the set of possible extensions is selected as a new test target. In this example the only possibility is the event 4. The algorithm determines from the unfolding that this event can be reached by a schedule that executes two visible operations of thread 2 from the initial state (and after that is free to follow an arbitrary schedule). In the resulting test execution thread 2 assigns a random input value to x and thread 1 reads this value and branches at the if-statement based on this value. Let us assume that the false branch is followed in this test run. By processing the resulting operation sequence, the events 4 and 5 get added to the unfolding (for the first operation the CORRESPONDINGEVENT subroutine returns the event 2). After the event 5 has been added, the possible extensions algorithm notices that thread 1 will perform next a branch operation that depends on input values. This leads to adding events for the true and false branches to the set of possible extensions (i.e., events 6 and

7). Assuming that event 6 corresponds to the false branch, it is added to the unfolding and removed from the set of possible extensions. This leaves the event 7 unexplored. For the final test run this event is selected as the test target. The schedule to reach this event is computed as before and in addition the path constraint corresponding to the scheduled execution path is solved. In this case the path constraint is simply $input > 0$. This constraint is solved to obtain a concrete value to be returned by the input statement and this value together with the computed schedule causes the program to reach the error location.

Note that if the part of an unfolding that belongs to a single thread is considered in isolation (e.g., the events with white color in Fig. 9), it can be seen as a symbolic execution tree constructed by DSE together with additional branching for scheduling related operations. In other words, instead of constructing a single big symbolic execution tree, the unfolding approach constructs a symbolic execution tree for each of the threads such that the shared variable and lock conditions describe the causality between events belonging to different threads. This means that the control flow paths of each thread are unwound as in standard DSE. Note also that if some operation in the program can be reached by multiple execution paths or non-equivalent schedules, the respective events in the unfolding are distinct as the set of events that causally precede a given event exactly describe an execution path through the control flow graph of the program and a set of schedules belonging to the same Mazurkiewicz trace.

4.2 Computing possible extensions

The subroutine PossibleExtensions is responsible for finding new events that can extend the unfolding. An event can be added to the unfolding if there exists a reachable marking that corresponds to a program state where an operation that has not yet been modeled is enabled. To find such events, the algorithm needs to be able to determine which operations are enabled in a given marking. When unfolding Petri nets, this information can be obtained directly from the Petri net itself. In our algorithm we do not have the explicit Petri net available. However, the thread conditions can be used to obtain the same information. Note that in each reachable marking there is one thread condition for each thread. These conditions correspond to local program locations and for each location there is a single operation type that can be executed next. Also, if we assume that all possible extensions are known when a new event is added to the unfolding, it is enough to consider only those reachable markings that contain at least one of the conditions in the postset of the newly added event. This is because for other markings the possible extensions have already been computed.

If we had a Petri net representation of the program, we could use standard possible extensions algorithms. These algorithms, however, have been designed for arbitrary Petri nets and are computationally the most expensive part of unfolding Petri nets. Fortunately, the modeling approach used in our algorithm generates unfoldings that have a very restricted structure. This makes it possible to create a specialized possible extensions algorithm that is typically much more efficient than general algorithms. In this section we will first describe an easy to understand but very inefficient possible extensions algorithm to get a basic understanding of possible extensions computation.

```
Input: An event $e$
1: $extensions = \{\}$
2: for all $c \in e^\bullet$ do
3:     if $c$ is a thread condition then
4:         $extensions = extensions \cup$ EXTENSIONSFROMTHREADCONDITION($c$)
5:     else if $c$ is a shared variable condition then
6:         $extensions = extensions \cup$ EXTENSIONSFROMSHAREDCONDITION($c$)
7:     else if $c$ is a lock condition then
8:         $extensions = extensions \cup$ EXTENSIONSFROMLOCKCONDITION($c$)
9: return $extensions$
```

Fig. 10 Possible extensions algorithm

In Sect. 5 we then describe how this naive approach can be improved for the types of unfoldings generated by out testing algorithm.

4.2.1 Naive possible extensions algorithm

As explained above, to find possible extensions, it is enough to consider reachable markings that contain at least one of the conditions in the postset of event e for which possible extensions are being computed. Therefore possible extensions can be found by considering each of the conditions in $e^\bullet$ separately. Figure 10 shows an algorithm that does this to find possible extensions. The algorithm iterates through the conditions in $e^\bullet$ and searches for possible extensions that are enabled in a marking that contains the condition c being considered. As there are three possible types of conditions, these are considered separately.

Extensions from thread conditions If c is a thread condition, the type of the possible extension events depends on the operation op that is to be executed from the program location determined by c. If this operation is a lock release or a symbolic branch, computing possible extensions is trivial. Events for lock releases have only one condition in their presets, that in this case is the thread condition c. Therefore a new release event with c in the preset can be added as a possible extension. Events for symbolic branching also have only a single thread condition in their presets. For symbolic branching, however, we need to add two events to the unfolding: one for the true branch and one for the false branch. The symbolic constraints generated by DSE are also stored to these events.

For other types of event, we need to find a set of shared variable conditions or lock conditions to form the preset of an extension event together with the thread condition c. Note that each event from c corresponds to the same transition in the Petri net representation of the program. This means that the presets of events from c will have the same labels. A naive way to find all possible extensions is then to consider every possible set of conditions that have these same labels. If the conditions in such set are concurrent, there exits a reachable marking where an event with this set as its preset can be fired. Every such event is added to the set of possible extensions. As an example, let us assume that op is a read operation. The naive algorithm would then go through the unfolding and collect all shared variable conditions labeled with the shared variable being read and belonging to the same thread as c. For each of these conditions that are concurrent with c, there exists a read type possible extension.

Extensions from shared variable and lock conditions If c is a shared variable condition, there can exist read or write type possible extensions having c in their presets. The naive way to find all read type possible extensions is to collect all thread conditions such that the next operation from it reads the variable represented by c. Any such thread condition that is concurrent with c forms a preset for a possible extension. Finding lock type possible extensions can be done in the same way.

To find write type possible extensions, a set of thread conditions can be collected as above. If the conditions are concurrent, we still need to find the rest of the shared variable conditions that from a preset of a write event. A naive way to do this is to again consider all possible sets of shared variable conditions that are labeled such that the set together with c and the thread condition forms a valid preset.

Checking for concurrencys A part of the possible extensions computation is to perform checks to determine if a set of conditions are concurrent. This *co-check* can be done by collecting the events and conditions that causally precede the conditions for which the co-check is performed. If two events are found that have the same condition in their presets, the conditions are in conflict. If one of the conditions is in the set of causally preceding conditions of another condition, the conditions are causally related. Otherwise the set of conditions are concurrent. Such check is in the worst case linear to the size of the unfolding but often significantly faster.

This naive approach can be seen as a brute-force search to find possible extensions. As the unfolding grows, the number of sets that needs to be checked grows rapidly. Furthermore, in typical programs most of these sets are not co-sets. Therefore an efficient way to compute possible extensions is necessary to make the testing algorithm practical. Such an approach is described in Sect. 5.

4.3 Obtaining test inputs

The EXECUTE subroutine performs a single test execution to generate a sequence of operations by using dynamic symbolic execution. The aim of this execution is to cover a previously unvisited event in the unfolding. To obtain input values and a schedule for the test execution, all the events that causally precede the target event are first collected. The target event can then be reached by executing the operations corresponding to these events. To obtain a valid order in which these operations can be executed, the collected events need to be sorted. As the event labeling in our algorithm corresponds to the order in which the events are added to the unfolding (*i.e.,* the labeling with numbers in Fig. 7), an event with a larger label number cannot causally precede an event with a smaller number. Therefore a valid order to fire these events (and execute the operations) is obtained by sorting the collected events to an ascending order according to the labeling. This allows the target event to be reached by requiring the runtime scheduler to execute the visible operations corresponding to the collected events in the obtained order.

As described before, the testing algorithm stores the symbolic constraints obtained by symbolic execution to events. The path constraint for an execution path leading to the target event can be computed by taking a conjunction of all the symbolic constraints stored in the collected events. If the path constraint is unsatisfiable, the target event is unreachable and it can be removed from the set of possible extensions. A satisfiable path constraint gives input values that together with the computed schedule forces a test run to reach the target event.

Let us consider why it is safe to remove an event from the set of possible extensions when the obtained path constraint is unsatisfiable. The sequence of events computed as above corresponds to an execution path where only those operations that are necessary to reach the target are executed. There can also be other possible paths to the target event. However, such execution paths must always execute the operations corresponding to the obtained event sequence because the target event cannot be reached without firing all events that causally precede it. Therefore alternative execution paths can only differ from the computed path by interleaving additional operations to it. By construction any such additional operation cannot be data-dependent with the target event. This is further illustrated by the Example 3 below. This means that the path constraints for alternative execution paths include the constraints from the computed path. If the conjunction of these constraints is unsatisfiable, so is the path constraint for the alternative path as adding constraints to an unsatisfiable path constraint cannot make it satisfiable. Similarly if the path constraint for an alternative execution path is satisfiable, so is the path constraint computed by EXECUTE.

There can also be execution paths that lead to an event that represents the same transition as the target event. However, as the execution paths are unwound in the unfolding representation, the event reached by such an alternative execution cannot be the same as the target event unless it correspond to the path computed by EXECUTE. Note also that the obtained execution path up to the target event is not necessarily a prefix of some previously explored path as is the case with algorithms like DPOR. In other words, the unfolding algorithm uses the information from all previous test execution to compute new test targets (possible extensions). This is one reason why the unfolding based approach can avoid redundant test execution when covering reachable local states of threads.

Example 3 Let us consider the complete unfolding in Fig. 7 and let us assume that we want to perform a test execution that covers the event labeled with 9. To reach this event, a test execution must execute the operations corresponding to events 9, 7, 6 and 5 (i.e., the target event itself and the events that causally precede it). A valid order to fire these events is obtained by sorting the collected events to an ascending order (5, 6, 7, 9). This corresponds to a schedule where at the first schedule point thread 1 performs a write operation (event 6) and in the following schedule point thread 2 performs a read operation. The path constraint for the execution path is a conjunction of the constraints stored to events 5 and 9. This constraint is then solved to obtain concrete input values that force a test execution to take the correct branches at branching statements. Let us also consider the case where we want to perform a test execution to cover event 4. The EXECUTE subroutine covers this event by following a path corresponding to an event sequence (3, 4). It is also possible to reach the event with an alternative execution path that fires an event sequence (3, 5, 4). Note that this sequence contains the originally computed sequence. However, it is not possible to reach event 4 with an execution that performs the write operation of thread 1 before the read operation of thread 2. This is because after the write, the event 3 can never be enabled. Instead, in this execution path, the read operation of thread 2 is represented by a different event (i.e., event 7) that is in conflict with the event 3. This illustrates that if the path constraint for event 4 was unsatisfiable, it would be safe to remove it from the set of possible extensions as the different execution paths containing data-dependencies are unwound to separate paths in the unfolding.

Note also that in our algorithm the possible extensions computation is done without taking the path constraints into account. This can lead to a situation where a possible extension corresponds to an execution path that requires conflicting symbolic branches to be followed. As an example, consider a program where thread 1 executes statements `if (a == 2) then x = 1;` and thread 2 executes statements `if (b != 2) then x = 2;` where a and b are local variables having the same symbolic value and x is a shared variable. This means that there is no execution path where both $x = 1$ and $x = 2$ are executed. However, in the unfolding of this program the events for the writes to x are concurrent if the symbolic constraints are ignored (i.e., both true and false events for symbolic branches are concurrently enabled). This means that if the event for $x = 1$ is added during the first test execution and the event for $x = 2$ is added during the second test execution, the possible extensions algorithm after the second test execution finds that $x = 1$ could be performed right after $x = 2$ and vice versa. The path constraints for these extensions are, however, unsatisfiable. This makes it necessary to check that the path constraint to an target event is satisfiable even if the event corresponds to a visible operation. As this can lead to checking the same path constraints multiple times, caching the solutions for path constraints helps improving the efficiency of the algorithm. This also illustrates that our algorithm searches alternative schedules based on the information collected from all previous executions and not just the most recent execution as with approaches like DPOR.

4.4 Extending the algorithm to detect deadlocks

The testing algorithm is just going to cover all events in the unfolding of the program and thus also all local states. However, it does not guarantee that all global states containing a deadlock get explicitly explored. However, the unfolding is a symbolic representation that contains also the thread interleavings that have not been explicitly tested. This means that it can be used to find the deadlocks that are missed by the concrete test executions. Approaches, like DPOR, that explore every Mazurkiewicz trace of the program are guaranteed to find all deadlocks during testing. If we want to make the unfolding approach comparable to DPOR, an additional search for deadlocks is needed.

A deadlock in the program under test corresponds to a reachable marking in the unfolding generated by the testing algorithm such that there are no events enabled in the marking and at least one of the threads has not reached its final state. The problem now is to find such reachable markings. As the only operations that can be blocked

Fig. 11 Algorithm for deadlock detection

```
Input: An unfolding $unf$, a thread condition $c$
Output: A set of found deadlocks containing $c$
 1: $wantedResource = wants(c)$
 2: $foundDeadlocks = \emptyset$
 3: SEARCH($c$, $\{c\}$)
 4: $foundDeadlocks$ = FILTERUNSAT($foundDeadlocks$)
 5: return $foundDeadlocks$

 6: function SEARCH($c$: condition, $coSet$: set of conditions)
 7:     $candidates$ = FINDNEXT($c$)
 8:     for all $candidate$ in $candidates$ do
 9:         if $coSet \cup \{candidate\}$ is a co-set then
10:             $S = coSet \cup \{candidate\}$
11:             if $holds(candidate) \in wantedResource$ then
12:                 $foundDeadlocks = foundDeadlocks \cup S$
13:             else
14:                 SEARCH($candidate$, $S$)

15: function FINDNEXT($c$: condition)
16:     $S = \{a \mid a \ co\ c, wants(a) \in holds(c) \wedge holds(a) \neq \emptyset\}$
17:     return $S$
```

in the programs we are considering in this paper are lock operations[1], we can restrict the deadlock detection to two different cases: either some thread acquires a lock and never releases it or there is a circular dependency on the locks such that a number of threads wait on each other.

The deadlocks caused by the first case are easily detected. We need to find all thread conditions that try to acquire one of the locks that are held but not released by a successfully terminated thread such that the corresponding thread becomes enabled if the terminated thread releases its locks. Such thread conditions can be found by temporarily adding release events for each acquired lock to the final thread state of the terminated thread and computing their possible extension events. Each possible extension then corresponds to a case where a thread is waiting for a release event that never happens. Furthermore, if the execution model guarantees that a thread releases all the locks it has acquired when it terminates, considering the deadlocks caused by the second case is enough.

Detecting circular deadlocks is more challenging. For a circular deadlock to occur, there must exist a reachable state where a set of threads hold at least one lock each and each thread wants to acquire a lock held by some other thread in the set. If the locking dependencies are circular, no thread can proceed. One approach to find such circular dependencies is to systematically search for them by starting a search from each thread condition that holds at least one lock and is followed by an acquire lock operation. One possible algorithm for this is the recursive backtrack search shown in Fig. 11.

[1] Thread join can be implemented similarly as locks by assuming that a thread holds a shared resource initially and releases it upon termination. If we assume that wait and notify can be called only if a corresponding lock is held by a thread, then the algorithm explores all possible orders of wait and notify calls because the algorithm executes all interleavings of acquires of related locks.

To simplify the discussion, we use the following definitions related to the algorithm in Fig. 11. Given a thread condition c:

– *operation(c)* refers to the type of operation that a thread wants to execute after reaching c,
– *wants(c)* refers to the shared resource (e.g., a lock) that a thread wants to acquire after reaching c, and
– *holds(c)* refers to the shared resources held by a thread after reaching c.

The search algorithm is assumed to be called every time when a thread condition c is added to the unfolding such that $holds(c) \neq \emptyset$ and $wants(c) \neq \emptyset$. This corresponds to a state in the execution of a thread where a circular deadlock is possible. The algorithm performs a backtracking search to find a set of thread conditions such that if they are reached concurrently, the execution will be in a deadlock. The algorithm maintains a set of thread conditions that are concurrent and are candidates to be in a set forming a circular deadlock. At each search step the algorithm searches for all such thread conditions t that are concurrent with the current candidate set, wants to acquire some lock held by the previous thread condition that was added to the set and holds some lock itself (function FINDNEXT in the algorithm). In other words, the algorithm searches for a new candidate in an attempt to build a circular chain leading to a deadlock. If such a thread condition t is found, t is added to the candidate set. If t holds the lock that the first condition c in the candidate set wants to acquire, a potential circular deadlock has been found, otherwise the algorithm proceeds with a new search step to find the next condition in a possible circular dependency chain. If the explored candidate set does not lead to a deadlock, the algorithm backtracks and searches other possible ways to extend the candidate set. The deadlocks found by the algorithm are only potential deadlocks as the search is done in the unfolding without taking the path constraints collected by dynamic symbolic execution into account. Therefore it is possible that there are no concrete input values that lead to the deadlock. For this reason the path constraint corresponding to the reachable marking for the potential deadlock must be checked (function FILTERUNSAT).

The FINDNEXT function that searches for the next candidate thread condition can be implemented naively by iterating though all thread conditions but this is naturally highly inefficient. Fortunately in typical cases it is possible limit the search space. To see this, we first consider the following lemma.

Lemma 1 *Let c_1 be a thread condition such that it is causally preceded by a lock event e_1 but not the corresponding unlock event. Let c_2 be a thread condition that is concurrent with c_1 and the next operation from c_2 is an acquire on the same lock as event e_1. In any unfolding it holds that either $c_2^\bullet = \emptyset$ or there exists an event or a possible extension $e_2 \in c_2^\bullet$ such that ${}^\bullet e_1 \cap {}^\bullet e_2 \neq \emptyset$ or $e_3^\bullet \cap {}^\bullet e_2 \neq \emptyset$, where e_3 is some unlock event that releases the lock acquired by e_1.*

Proof Let l_1 be the thread condition in ${}^\bullet e_1$. As $l_1 \neq c_2$, there are four possible cases: (1) $c_2 < l_1$, (2) $l_1 \# c_2$, (3) l_1 *co* c_2 or (4) $l_1 < c_2$. From our assumptions we know that $l_1 < c_1$. Now cases 1 and 2 directly imply that c_1 and c_2 are not concurrent which is against our assumptions and therefore both of these cases are impossible. In case 3 we know that l_1 *co* c_2 and *operation*(c_2) is a lock, therefore an event or possible extension

e_2 must exist as required by the lemma. Let us finally consider the case 4. Since $l_1 < c_2$ and $c_1\ co\ c_2$, it must hold that $e_1 < c_2$. Otherwise c_1 and c_2 would not be concurrent which is against our assumptions. Now as *operation*(c_2) is a lock that wants to acquire the same lock as e_1, we know that if there is a condition $e \in c_2^\bullet$, it must hold for the lock condition $l \in {}^\bullet e$ that $e_1 < l$. This implies that e must be causally preceded by some event e_u that releases the lock acquired by e_1. Let l_3 be the lock condition in $e_u^\bullet$. We know that l_3 and c_2 cannot be causally related as otherwise c_1 and c_2 would not be concurrent. This leaves the possibilities that either $c_2\ co\ l_3$ or $c_2 \# l_3$. The first possibility implies that there exist an event or possible extension e_2 as required by the lemma. The second possibility implies that no lock condition that is causally preceded by e_u cannot be concurrent with c_2. Therefore either there exist an event e_2 or $c_2^\bullet = \emptyset$. $\square$

The condition c_1 in Lemma 1 corresponds to the last thread condition that has been added to the candidate set in the algorithm of Fig. 11 and all the suitable candidates returned by FINDNEXT satisfy the same assumptions as c_2. Now based on Lemma 1, we can restrict the search for new candidates by considering only those thread conditions that are in the presets of lock events that are similar to e_2 or e_3. Finding such events can be done efficiently by traversing the arcs in the unfolding. We also need to check thread conditions with empty postsets. A separate list of such conditions can be maintained to do this efficiently. The restricted search is therefore similar to the one done when computing possible extensions.

The algorithm described above can be used find all circular deadlocks in the program under test if it is executed for each added thread condition such that the executing thread both holds a lock and wants to acquire a lock. If there are no such thread conditions (e.g., in the case of the program having only one lock or no locks at all), there is no need to perform a search and in these cases the deadlock checking does not cause any additional overhead.

4.5 Limitations of the algorithm

The unfolding algorithm described in this paper assumes that all shared variables are explicitly defined in the beginning and the number of threads is fixed. It is possible to extend the algorithm to support dynamically changing sets of shared variables. In order to do this each such variable needs an unique identifier across all possible executions. In the context of Java programs it is possible to obtain such identifiers by adding an event to the unfolding when a new object is created and then identifying a shared variable (a field of this object) by combination of the object creation event and the field name of the variable. Static variables can be identified by their class and field names. Notice also that the concrete execution in dynamic symbolic execution gives precise aliasing information for the variables. Therefore, it is always known if two references point to the same shared variable.

Handling dynamic number of threads is more challenging. The main problem is that when a thread performs a write operation, the corresponding write event accesses shared variable conditions from all threads in the program. It is not enough to consider only those threads that are running at the time the operation is executed as it is possible

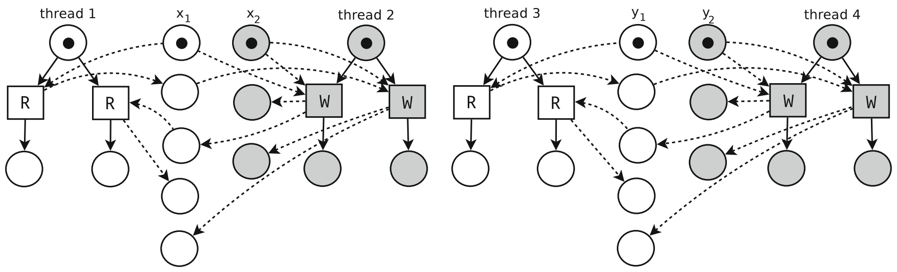


Fig. 12 Unfolding of independent pairs of threads

that there exists an execution of the program where there are additional threads running and the write affects the behavior of those threads as well. One simple way to address this problem is to update each write operation in the unfolding to access the conditions of a new thread when the thread is added to the unfolding. This, however, is an expensive operation. Another way is to model the thread creation with an event that reads all the shared variables of the parent thread and writes the shared variables to the child thread. The problem with this approach is that thread creations can now be considered to be in race with some write operations and this can lead to unnecessary test runs. Perhaps the most promising alternative is use contextual unfoldings that use read arcs (Vogler et al. 1998). With this approach there is no need to have local copies of shared variable conditions for each thread, instead one copy shared by all threads is enough. This can be done by modeling read events such that they use read arcs that require that a token is in some condition but firing the read event does not remove the token. This approach would potentially even simplify the possible extension computation described in this paper but, on the other hand, make it more expensive to compute when given conditions are concurrent. Currently our algorithm does not support dynamic thread creation and finding the best solution to the problem is part of future work.

4.6 Further observations

In the set of possible extensions from which the target event for the next test execution is selected, there can be multiple events that are concurrent. It is therefore possible that one test run covers more than one test target. This property can provide further reduction to the number of needed test runs especially in situations where there are independent components in the program under test and the random scheduler and random inputs have a good chance of exploring different execution paths. It is also possible to compute a schedule and input values for the next test execution such that it will cover the maximum number of possible extension events. This computation, however, is potentially computationally expensive (we conjecture that it is NP-hard) and it is left for future work to study different heuristics and approaches to take advantage of this property and improve the runtime of the algorithm.

To illustrate this property further, consider the example program from Sect. 2 which consists of n pairs of threads that are independent. Figure 12 shows an unfolding resulting from such program. With unfoldings the events in different pairs are concurrent and therefore it is possible to explore the whole program with only two test runs regardless of the number of threads. This can lead to an exponential reduction compared to DPOR-like approaches.

Another interesting property of the generated unfoldings is that if the testing process is repeated multiple times, the resulting unfoldings are isomorphic. This is not necessarily true when using trees and partial order reduction based approaches. It is possible to take advantage of this property to partially check the correctness of the algorithm's implementation. For example, it is possible to compare the resulting unfoldings from two implementations and check if they are isomorphic. An even quicker check is to compare the number of events from two different algorithms. If the resulting unfoldings are different, with high probability they have different number of events especially for non-trivial programs. It is also possible to exhaustively test all possible interleavings of a program, which is easy to implement correctly, build an unfolding based on these tests in a similar way as in Sect. 3 and then compare the result with the unfolding based testing algorithm. Although this cannot show that the implementation is correct, it can be an useful testing and debugging aid while developing unfolding based algorithms.

The unfolding approach can also in some cases detect errors that DPOR might miss when using the same bound on the execution length. This is because the execution path that DPOR follows to a test target (a backtrack point) is a prefix of some earlier test execution. Such path, however, might not be the shortest path to the target. The unfolding approach, however, does not execute any unnecessary operations to reach a target event. This means that the unfolding approach can detect some errors with shorter executions and therefore with a smaller bound on execution lengths. Naturally, DPOR can detect the same error if it follows the same execution path as the unfolding approach. However, DPOR might ignore the part of the execution tree where the shortest path to the error is represented. Therefore this observation relates to the fact that the parts of the execution trees explored by DPOR can differ based on the order in which the execution paths are explored.

The unfolding based testing approach brings other benefits as well. For example, it generates a structure that contains all the interleavings and input dependent execution paths of the program under test. In a way, this structure can be seen as a model of the tested program. If additional information is collected during the testing, such as storing the concrete or symbolic values of shared variables to the shared variable conditions in the unfolding, it would be possible to search the unfolding to answer reachability questions such as is there a global state in the program where the sum of two shared variables is over some given limit. For large programs, the unfolding would typically be incomplete, but even in this case it would be possible to find violations of some interesting properties on global states that were not visited directly by any of the performed tests.

Finally, it is possible to generate an initial unfolding based on tests that are generated randomly or by some heuristic approach, model these test execution with unfoldings and then compute possible extensions to increase the coverage of the initial testing effort. With trees this would not be as easy as two executions following the same Mazurkiewicz trace might follow different branches in the symbolic execution tree making it difficult to use partial order reduction efficiently. With unfoldings the events corresponding to the executions following the same trace are the same. Exploring such additional uses of the generated unfolding is part of further research.

5 Efficient possible extensions computation

In this section we show how possible extensions can be computed efficiently. To do this, we take the structural properties of the unfoldings generated by our algorithm into consideration. We do a case analysis of all the different types of possible extension events that the algorithm in Fig. 10 can find. As every event has one thread condition in its preset, the subroutine EXTENSIONSFROMTHREADCONDITION needs to consider five possible cases: a possible extension is (1) a symbolic branch event, (2) a read event, (3) a write event, (4) a lock event or (5) an unlock event. The subroutine EXTENSIONS-FROMSHAREDCONDITION needs to consider two cases: the extension is (6) a read event or (7) a write event. With the subroutine EXTENSIONSFROMLOCKCONDITION there is only one possible case: (8) the extension event is a lock event.

To find possible extensions, there are therefore eight possible cases to consider. In the following we show how each of these cases can be handled efficiently. Before that, however, we will describe a structural property that connects possible extensions. This property allows the possible extensions to be computed efficiently.

5.1 Characterizing the connection between possible extensions

To make the discussion precise, we use the following definitions.

Definition 1 A *memory access set* of an event e with a thread condition c in its preset is the set $^\bullet e \setminus \{c\}$ (i.e., the set of shared variable or lock conditions in the preset of e).

Definition 2 Let P be a set of places. A set S of conditions *represents* P if $|P| = |S|$ and for each $x \in P$ there exists $y \in S$ such that $l(y) = x$. For single conditions we write that a condition c represents a place p if $l(c) = p$.

The intuition here is that if we consider the Petri net representation of the program that is being unfolded, a shared variable in it is modeled with a fixed set of places P. If we have two sets of shared variable conditions, the sets represent each other if they contain conditions that correspond to the same set of shared variable places.

Definition 3 Let S_1 and S_2 be memory access sets that represent the same set of places. S_1 and S_2 are *adjacent* if there exists a sequence of events $e_0 \ldots e_n$ and a reachable marking M containing all the conditions in S_1 such that firing the event sequence from M leads to a marking where all conditions in S_2 are marked and there is no prefix of the event sequence $e_0 \ldots e_{i<n}$ leading to a marking that contains a memory access set that represents the same places as S_1 and S_2.

Example 4 Let us consider the sets $a = \{x_{1,2}, x_{2,2}\}$, $b = \{x_{1,1}, x_{2,1}\}$ and $c = \{x_{1,2}, x_{2,1}\}$ in Fig. 9. The sets a and b are not adjacent as from any marking containing b, a marking containing a can be reached only with event sequences that fire both event 1 and event 2. One of these events must be fired first which leads to a marking containing a memory access set representing the same places as a and b. However, b is adjacent with c and c is adjacent with a.

Before proving that possible extension events are connected in a way that helps finding them quickly, let us first consider a property that helps in proving the main results we need.

Lemma 2 *Let $L = S_1, S_2, \ldots, S_n$ be a sequence of memory access sets representing the same places and let for all $1 \leq i < n$ hold that S_i and S_{i+1} are adjacent such that at least one condition in S_i causally precedes the corresponding condition in S_{i+1}. If the conditions in S_1 and S_n are concurrent with a condition t, then all the conditions in the sequence L are as well.*

Proof Let c_1 be a condition from S_1 and let c_n and c' be conditions representing the same place as c_1. Furthermore, assume that $c_n \in S_n$ and $c' \in S_i$ where $1 < i < n$. It must now hold that $c_1 \leq c' \leq c_n$ as the conditions representing the same place in adjacent memory access sets are either the same or causally related.

If $c' = c_1$ or $c' = c_n$, then c' is trivially concurrent with t. Let us now assume that $c_1 < c' < c_n$ and c' is not concurrent with t. There are three possible cases: (i) $c' < t$, (ii) $t < c'$ and (iii) $t \# c'$. Case (i) implies that $c_1 < t$, case (ii) implies that $t < c_n$ and case (iii) implies $t \# c_n$. All these cases contradict the assumptions that c_1 and c_n are concurrent with t. Therefore it must hold that $c' \; co \; t$. As the same analysis can be done for all conditions in the sequence L, the conditions in this sequence must be concurrent with t. □

Theorem 1 *Let t be a thread condition in an unfolding such that the next operation from t is either a read or write, and let S be a set of all memory access sets of events that have t in their presets. Let G be a graph such that there is a vertex in G for every memory access set in S and an edge between two vertices if the respective memory access sets are adjacent. The graph G is connected.*

Proof Let G' be a graph that has a vertex for every memory access set in the unfolding that represent the same places as the sets in S and has an edge between two vertices if their respective memory access sets are adjacent. Clearly G is a subgraph of G'. Furthermore, G' is connected as there exits a vertex S_0 in G' such that S_0 contains conditions that are initially marked in the unfolding and all other memory access sets representing the same places must be reachable from this initial marking.

Let us now assume that G is not connected. This means that there exits two vertices S_1 and S_2 such that there is no path between S_1 and S_2 in G. Let us consider the vertex S_0. If $S_0 = S_1$ or $S_0 = S_2$, then according to Lemma 2, there must exist a path from S_1 to S_2 such that all conditions in the intermediate vertices are concurrent with t. If all the conditions in S_0 are concurrent with t, then there exists a path from S_1 to S_2 via S_0. In these cases the graph G is therefore connected.

Let us now consider the case where S_0 is not equal to S_1 or S_2 and not all conditions in S_0 are concurrent with t. This means that there exists a condition in S_0 that causally precedes t. Let M be the marking that is reached by firing from the initial marking all

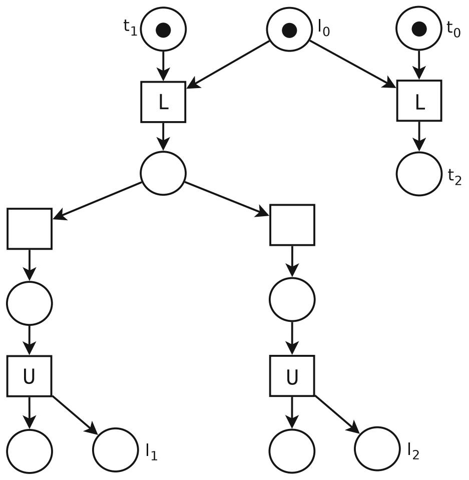


Fig. 13 Example of alternative lock conditions

events that causally precede t. No matter in which valid order the events are fired, the resulting marking M is always the same. Let S_0' be the memory access set presenting the same places as the sets in S that such that $S_0' \in M$. Such a memory access set must exist as read and write events are modeled such that in every marking one condition corresponding to each thread's shared variable copy is marked. The conditions in this set are trivially concurrent with t. Furthermore, all memory access sets that are concurrent with t and represent the same places as S_0' must be reachable by firing sequences of events from a marking containing S_0'. Therefore there exists a path in G' from the vertex corresponding to S_0' to the vertex corresponding to S_1 and also similarly to S_2. According to Lemma 2, the intermediate vertices in these paths must be concurrent with t and therefore there exist a path between S_1 and S_2 in G. $\square$

Based on Theorem 1, when we want to compute possible extensions starting from a thread condition t and we already known one event from t, the rest of the possible extensions can be found by using the known event as a starting point and restricting the search to the graph G. This limits the search space in comparison of searching the whole unfolding as with more general unfolding approaches. In typical cases this can speed up the possible extensions computation considerably but in worst case the whole unfolding still needs to be searched.

When considering possible extensions relating to lock events, we cannot use the result above. This is because with lock conditions the assumption that in each reachable marking there is one condition representing the lock does not hold. However, a similar result can be obtained for locks as well. For this, we need the following definition.

Definition 4 Let S_0, S_1 and S_2 be memory access sets that represent the same set of places and contain lock conditions l_0, l_1 and l_2, respectively. Let S_0 be adjacent to both S_1 and S_2 such that $l_0 < l_1$ and $l_0 < l_2$. If there exists a lock event e such that $l_0 \in {}^\bullet e$, $e < l_1$ and $e < l_2$, the sets S_1 and S_2 are said to be *alternatives*.

Example 5 Let us consider the unfolding in Fig. 13 in which the lock events are labeled with L and unlock events are labeled with U. In this unfolding the memory access sets corresponding to lock conditions l_1 and l_2 are alternatives. However, if there were an unlock event that is causally preceded by the thread condition t_2, the lock condition in the postset of this event would not be an alternative to l_1 and l_2 even though all three conditions would be preceded by the same lock condition l_0. This is because the closest preceding lock events for these conditions are not the same.

Theorem 2 *Let t be a thread condition in an unfolding such that the next operation from t is lock acquire, and let S be a set of all memory access sets of events that have t in their presets. Let G be a graph such that there is a vertex in G for every memory access set in S and an edge between two vertices if the respective memory access sets are adjacent or alternatives. The graph G is connected.*

Proof Let us assume that there are two lock conditions c_1 and c_2 that are concurrent with t but in graph G there is no path between their respective vertices. It must be the case that either $c_1 < c_2$, $c_2 < c_1$ or $c_1 \# c_2$ as the conditions are assumed to be separate and in any marking at most one lock condition corresponding to the same lock can be marked.

Let us consider the case $c_1 < c_2$. From Lemma 2 we know that in an execution path that starts from a marking containing c_1 and reaches a marking containing c_2, all the conditions representing the same lock as c_1 and c_2 are also concurrent with t. As these conditions correspond to a sequence of adjacent vertices in G, there exists a path in G between c_1 and c_2. The case $c_2 < c_1$ is symmetrical.

Let us now consider the case $c_1 \# c_2$. Let G' be a graph such that there is a vertex for every memory access set in the unfolding that represent the same lock as the conditions in S and has an edge between two vertices if their respective memory access sets are adjacent. Note that no edges between alternative memory access sets are added. The graph G' is a tree as each memory access set contains a single condition that represents the same lock and has the vertex corresponding to the initial lock condition as the root of the tree. In G' there must therefore exist a minimal subtree that contains both vertices corresponding to c_1 and c_2. Let c_0 be the condition corresponding to the root of this subtree. If c_0 is concurrent with t, based on Lemma 2 there exists a path in G that connects c_1 to c_2 via c_0. Let us now consider the case where c_0 is not concurrent with t. There are three possibilities: (i) $c_0 \# t$, (ii) $t < c_0$ or (iii) $c_0 < t$. The first two cases imply $t \# c_1$ and $t < c_1$, respectively, which contradicts the assumption that t is concurrent with c_1. The case (iii) implies that there must exist an event e that has c_0 in its preset such that $e < t$. If the paths from c_0 to c_1 and c_2 require firing different events at a marking containing c_0, then either c_1 or c_2 has to be in conflict with t which is again a contradiction. Therefore the only possibility is that $e < c_1$ and $e < c_2$. As c_1 and c_2 are in different branches in G', the next unlock events after firing e must be different in paths from c_0 to c_1 and to c_2. This means that the next vertices with conditions c'_1 and c'_2 in the paths from c_0 to c_1 and c_2, respectively, are alternative. It is easy to see that $c'_1 \; co \; t$ has to hold because if c'_1 causally precedes t, c_2 cannot be concurrent with t as it is in conflict with c'_1. If t causally precedes c'_1 or is in conflict with it, the same has to hold for c_1 as well because either $c_1 = c'_1$ or $c'_1 < c_1$. This contradicts the assumption that $c_1 \; co \; t$. The same holds symmetrically for c'_2 as well. Based on Lemma 2, there is a path from c'_1 to c_1 and c'_2 to c_2 in G. As c'_1 is alternative to c'_2, there is a path in G from c_1 to c_2. As such a path can be found in all cases G is connected. □

Compared to our previous work (Kähkönen et al. 2012), the formalization above gives a stronger requirement for alternative lock conditions, potentially leading to a smaller search graph. Also in the previous formalization, a search graph was described such that it could be used for memory access sets containing either lock conditions or shared variable conditions. Here the results are split into two cases as the shared variable case does not need to take alternative nodes into account. This again results in a smaller search graph for the possible extensions algorithm to consider.

We will next use these obtained results to show how the search space for possible extensions can be reduced when compared to the naive approach described earlier. This can be seen as improving the subroutines of the algorithm in Fig. 10. We will consider optimizing each of these subroutines separately.

5.2 Optimized possible extensions computation from thread conditions

We will first consider the different types of possible extensions that have a thread condition t in their presets. The types of such possible extensions depend on the operation that can be executed after reaching t.

5.2.1 Case 1: the operation branches on a symbolic value

If the operation to be executed after reaching t is a symbolic branch, there is no need to perform any search as the only possible extensions are the events for the true and false branch. These can be added directly to the set of possible extensions like in the naive approach.

5.2.2 Case 2: the operation reads a shared variable

If the next operation from a thread condition is a read, one possible extension can be found trivially. This is because our algorithm maintains a marking that represents the current state of the test execution. If the program is in a state where a thread wants to perform a read (or a write) operation, such operation is always enabled. Therefore one possible extension can be computed directly from the marking maintained by the algorithm. The rest of the possible extensions can be found by locating those shared variable conditions that are adjacent to the shared variable condition in the preset of the initially found extension. If such shared variable condition s is concurrent with t, a preset for a read type possible extension has been found. In these cases a similar search needs to be done recursively to the shared variable conditions adjacent to s as well. Based on Theorem 1 this search finds all sets of conditions that together with t form the presets of possible extensions.

5.2.3 Case 3: the operation writes a shared variable

The case where the next operation is a write can be handled in the same way as the read case. The only difference is that the trivially found possible extension has a larger set of shared variable conditions in its preset. A recursive search for adjacent sets of shared variable conditions then needs to be performed.

5.2.4 Case 4: the operation acquires a lock

For a thread condition t that has an acquire of lock l as the next operation, it is not necessarily the case that the marking maintained by the algorithm contains a lock condition labeled with l. In the case where the marking contains such condition, it can be used as the starting point for a similar search as for read and write events. This time, however, the algorithm needs to explore alternative lock conditions in addition to adjacent ones as required by Theorem 2.

In the case where the marking does not contain a lock condition labeled with l, there are two possibilities: (1) no lock conditions that are concurrent with t and labeled with l exists, or (2) one such condition is either in the preset of the last lock acquire event e_l executed in the operation sequence currently being processed by the testing algorithm or in one of the unlock events that release the lock acquired by e_l. If the case (1) holds, no possible extensions from t currently exists. Therefore a starting point for the recursive search of possible extensions, if any exists, can be found by checking lock conditions satisfying (2).

Finally, let us prove why the case (2) must hold if (1) does not. The lock condition in the preset of e_l must either be concurrent with t, which already satisfies (2), or it must causally precede t as e_l has been fired in the current test execution that leads to the condition t. In the case where the lock condition causally precedes t, the lock conditions L in the postsets of the unlock events that release the lock acquired by e_l cannot be causally related. If this were not the case, e_l would not be the most resent acquire of a lock labeled with l. This means that the lock conditions in L are either concurrent or in conflict with t. If one of them is concurrent with t, the case (2) holds and if all of them are in conflict with t, then (1) holds.

5.2.5 Case 5: the operations releases a lock

Lock release has only a single thread condition in its preset. In this case it has to be t and such event can be added directly to the set of possible extensions.

Example 6 Let us consider the unfolding in Fig. 14 and let us assume that the possible extension algorithm wants to compute extensions starting from the thread condition t_1. If the program performs a read operation after reaching t_1, a possible extension that can be fired from the current marking corresponds to the event R_1. This gives the algorithm the initial possible extension. The algorithm then computes conditions adjacent to s_2. In this example there are three such conditions s_1, s_3 and s_4. After finding these conditions, the algorithm checks if they are concurrent with t_1. Let us assume that s_1 is concurrent with t_1 but the rest of the found conditions are not. This means that there is a read type possible extension that has t_1 and s_1 in its preset. The algorithm then continues by performing recursively a similar search for the adjacent conditions of s_1.

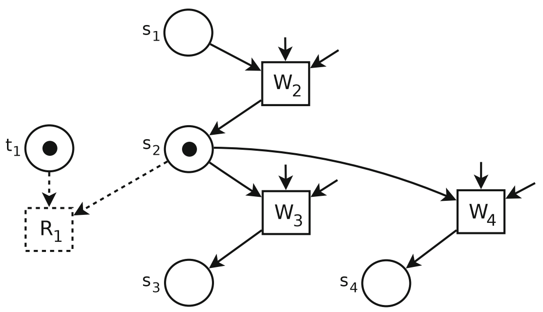


Fig. 14 Example of computing possible extensions from thread conditions

5.3 Optimized possible extensions computation from shared variable conditions

Let us assume that $s \in e^{\bullet}$ is the shared variable condition for which the possible extensions are being computed. Also assume that $s' \in {}^{\bullet}e$ is a shared variable condition with the same label as s. To find possible extensions from s efficiently, we take advantage of the following observation. In a Petri net representation of the program, firing a transition t_1 that corresponds to e (i.e, either a read or write event) does not disable or enable any transitions belonging to other threads. This means that for any possible extension event from s, the transition t_2 corresponding to this event is concurrently executable with the transition t_1. In other words, there exists a data race. This also means that an event e' labeled with t_2 is enabled in a marking that contains the condition s' and the same thread condition as the possible extension (i.e., t_2 and t_1 are both enabled). If all possible extension have been computed so far, the event e' is either already added to the unfolding or is in the set of known possible extensions. As there must exist an event like e' for each possible extension from s, the possible extensions can be computed through these events.

5.3.1 Case 6: an operation can read the accessed shared variable

To find read type possible extensions, we need to find all read events that have s' in their presets. This can be done simply by iterating though the events in the postset of s'. If a thread condition in the preset of such reach event is concurrent with s, the conditions form a preset for a possible extension event.

5.3.2 Case 7: an operation can write to the accessed shared variable

To find write type possible extensions, it is not necessary to consider each shared variable condition in the postset of e separately. This is because e is the most recent event added to the unfolding and therefore any marking where e has been fired contains all the conditions in $e^{\bullet}$. This means that any write type possible extension needs to have all of the shared variable conditions in $e^{\bullet}$ in its preset. The possible extensions can then be found by searching write events that have the shared variable conditions in ${}^{\bullet}e$ in their presets. Given such a write event e', a possible preset for an extension

Fig. 15 Possible extensions from shared variables

```
Input: An event $e$
 1: $extensions = \{\}$
 2: $cadidates = \{\}$
 3: for all $c \in {}^\bullet e$ do
 4:     for all $e' \in c^\bullet$ do
 5:         $cadidates = cadidates \cup e'$
 6: for all $candidate \in candidates$ do
 7:     $preset$ = ComputePreset($e, candidate$)
 8:     if $preset$ is a co-set then
 9:         $ext$ = event with the $preset$
10:         $extensions = extensions \cup ext$
11: return $extensions$
```

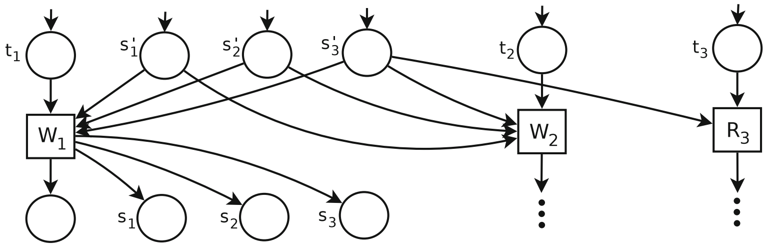


Fig. 16 Example of computing possible extensions from shared variable conditions

event consists of the conditions in ${}^\bullet e'$ except that any condition having a same label as a condition $c \in e^\bullet$ is replaced by c.

Both of these cases can be summarized by the algorithm in Fig. 15 where computePreset constructs presets for possible extensions as explained above. For simplicity the algorithm considers each shared variable condition in $e^\bullet$ separately even for write type extensions. This is not optimal as explained above but does not affect correctness. In the above we have assumed that all possible extensions are known and already added to the unfolding. However, typically possible extensions are being computed before all the previously known extensions have been added to the unfolding. To make sure that the algorithm in Fig. 15 can be used, one possibility is to do the search in a version of the unfolding that contains all known possible extensions. This, however, is not strictly necessary. When an extension that acts as the event e' is added to the unfolding, the possible extensions computation performed at this point will find the possible extension that was originally to be computed.

Example 7 Let us consider the unfolding in Fig. 16 and let us assume that the event W_1 has been just added and we are computing the possible extensions from the shared variable conditions in the postset of this event. As described above, the algorithm needs to iterate though the events that are found in the postset of conditions s'_1, s'_2 and s'_3. In the example such events are W_2 and R_3. The algorithm then checks if the set $\{t_2, s_1, s_2, s_3\}$ is a co-set. If it is, a write type possible extension has been found. Similarly the algorithm adds a read type possible extension if $\{t_3, s_3\}$ is a co-set.

5.4 Optimized possible extensions computation from lock conditions

If there exists possible extensions from a lock condition l, the only possibility is that some thread can acquire the lock released by the event e for which possible extensions are being computed. This leads to our last case.

5.4.1 Case 8: an operation can acquire the released lock

A thread condition that is concurrent with l and corresponds to a program location where a thread wants to acquire a lock represented by l forms a preset of a possible extension event together with l. Therefore all such thread conditions need to be found. Let us first assume that a lock event e' exists in the unfolding such that e' and a possible extension from l have the same thread condition t in their presets. In this case, based on Theorem 2, there has to exist an event (or a previously found possible extension) that has t and a lock condition l' in its preset such that l is adjacent or alternative to l'. Therefore, all thread conditions that are possibly concurrent with l can be computed through events that have a lock condition adjacent or alternative to l in their presets. If a thread condition in the preset of such event is concurrent with l, a possible extension is found.

There is, however, still the possibility that e' does not exist. This corresponds to the case where a lock release transition enables a new transition that was not enabled before. In other words, in the postset of a thread condition t that is concurrent with l there are no currently known events. This can happen if any event from t has not yet been enabled in the current execution or any previous test execution that reached t resulted in a deadlock. One possibility to guarantee that no possible extensions are missed due to this is to maintain a list of such thread conditions. Every time when extensions from lock conditions are computed, the thread conditions in this list need to be checked to see if they can form a valid preset together with l. In typical programs the size of such list remains small as a test execution will always find an event that follows a thread condition unless the execution ends up in a deadlock.

Example 8 Let us consider the unfolding in Fig. 13 and let us assume that we are computing possible extension events from l_2. The conditions adjacent or alternative to l_2 are l_0 and l_1. The thread conditions in the presets of lock events that take a token from these conditions need to be checked for concurrency with l_2. In this example the only such thread conditions are t_0 and t_1. The thread condition t_1 causally precedes l_2 and therefore cannot be concurrent with it. If t_0 is concurrent with l_2, there exists a possible extension with these conditions in its preset.

6 Experimental evaluation

In this section we evaluate our algorithm by using it to test several Java programs and comparing it with dynamic partial order reduction based testing approaches.

6.1 Experimental setup

We have implemented our algorithm and a DPOR algorithm variant that takes advantage of commutativity of read operations as described by Saarikivi et al. (2012) in a prototype tool that supports a subset of Java programs. We have also combined the DPOR algorithm with sleep sets. The DPOR algorithms used in these experiments use a heuristic to determine which transition must be fired at a backtrack point computed by the algorithm. For example, let us consider the program discussed in Sect. 2 and the test cases illustrated in Fig. 2. Let us assume that the execution path (r_1, w_1, r_2, w_2) has been explored and there is a third transition t enabled after firing r_1. DPOR detects that there is a race between the write transitions and sets a backtrack point before the first write event. From that backtrack point the algorithm can then either fire the transition t or the transition r_2 and needs to make a choice between them. The DPOR algorithms in these experiments use a heuristic that tries to fire a transition belonging to a thread that the transition causing the race belongs to. In the example the race is found when processing the transition w_2 which belongs to thread 2 and therefore the algorithm chooses r_2 to be fired. If no such transition is found, the algorithm plays it safe and explores all enabled transitions from the backtrack point. Sleep sets complicate this situation as it is possible that the transition which the heuristic chooses to be explored is in a sleep set and therefore set to be ignored. This leads to two possible alternatives to handle the situation. Either the sleep set is ignored in cases like this or all the other transitions that are enabled but not in a sleep set are chosen to be explored. In these experiments we have implemented both approaches and call the first alternative as DPOR-1 and the second as DPOR-2.

To enable symbolic execution and to control the scheduling of threads, our tool instruments the program under test using the Soot framework (Vallée-Rai et al. 1999). As a constraint solver the tool uses Z3 (de Moura and Bjørner 2008). The number of test execution required by the algorithms can vary depending on the order in which the execution paths are explored. For this reason the experiments were conducted such that the algorithms follow a random schedule after the current test target in an unfolding or a tree is reached. Also, the experiments were repeated 20 times and the average number of performed tests and the total average time required are reported. A timeout of two hours is used for each experiment.

A similar tool to our prototype is jCUTE by Sen and Agha (2006a) that uses a race detection and flipping algorithm. Contrary to our earlier work, we decided not to include jCUTE in these experiments as the tool is no longer maintained and it fails to explore all equivalence classes for programs where there exist a write event and a number of read events that are all concurrent. The simplest example of such program has three threads where thread 1 executes the operation `x = 1;` on a shared variable x and the other two execute the operation `local a = x;`.

6.2 Evaluation subjects

The file system benchmarks are from a paper by Flanagan and Godefroid (2005) where they are used for experimental evaluation of the DPOR algorithm. They are examples of programs where static partial order reduction algorithms cannot accurately predict the use of the shared memory and therefore conservatively explore more states than dynamic approaches. Parallel pi-algorithm is an example of a typical program where a task is divided to multiple threads and the results of each computation are merged in a block protected by locks. The Dining benchmarks implement a variation of the dining philosophers problem where each philosopher tries to eat twice. The number of philosophers in the benchmarks are 4, 5 and 6. The test selector benchmarks represent an abstracted and faulty implementation of a modification to a small part of our tool that allows multiple test executions to be run concurrently. The pairs program is an artificial example in which the unfolding approach can achieve exponential reduction when compared to partial order reduction approach based on using trees. This can be considered as a close to optimal case for the unfolding approach. The single lock benchmark is a program where multiple threads access various shared variables in a way that all accesses are protected by the same lock. In this case all the approaches explore the same execution paths (the interleavings of the lock operations) and there are no operations that can cause the DPOR algorithm to explore unnecessary paths due to scheduling choices. Fib and Szymanski programs are from the 1st International Competition on Software Verification (SV-COMP). These programs have been translated to Java and simplified by limiting how many times some loops are executed in order to limit the number of execution paths. Updater is a program with four threads where two threads update shared variables independently based on input values and the remaining two read the values to do computation with them. The second variant has been modified in order to introduce a deadlock to the program. Synthetic examples are handmade programs reading input values and performing arbitrarily generated sequences of operations.

6.3 Results and discussion

Table 2 shows the results of the experimental evaluation. For each algorithm the number of tests needed to fully cover the benchmarks are shown together with the execution time of the algorithm. The best results are made bold in the table. Some of the programs contain deadlocks and those are marked by using italics in the name of the program. All three algorithms detect these deadlocks successfully. For the unfolding algorithm the table includes two additional columns. The additional time to perform the deadlock check as described in Sect. 4.4 is given in the column DL and the number of events in the final unfolding is given in the column Events. The number of events gives an idea of the memory requirements of the algorithm as the size of the pre- and postsets of any event is at most the number of threads in the program under test plus one, and each event and condition requires constant amount of memory.

Based on these results, the unfolding approach typically requires less test executions but is slower per test execution when compared to DPOR. However, the reduction obtained can make the unfolding approach faster than the variations of DPOR used in these experiments. This is especially true for programs where the threads are not tightly coupled (i.e., the threads perform long sequences of operations that do not affect each other). Notable examples for this kind of programs are the Filesystem and Pairs programs. In the Filesystem program a number of threads are started and only if there are more than 13 threads running, the same parts of shared memory get accessed by multiple threads. When more threads are running, they start to interact in pairwise manner (i.e., thread 14 accesses the same shared variables as thread 1 and so forth).

Table 2 Comparison of different approaches

	Unfolding				DPOR-1		DPOR-2	
Benchmark	Tests	Time	DL	Events	Tests	Time	Tests	Time
Filesystem 1	**3**	**0m 0s**	+0.0s	178	135	0m 3s	4050	1m 19s
Filesystem 2	**3**	**0m 0s**	+0.0s	226	992	0m 30s	>44661	>2h
Parallel Pi 1	**120**	**0m 0s**	+0.0s	1668	623	0m 1s	165	**0m 0s**
Parallel Pi 2	**720**	0m 7s	+0.1s	9831	4558	0m 16	993	**0m 3s**
Dining 1	**799**	**0m 1s**	+0.2s	8049	26410	0m 48s	1161	**0m 1s**
Dining 2	**5747**	**0m 9s**	+0.8s	52886	>760714	>2h	9890	0m 22s
Dining 3	**36079**	**1m 20s**	+3.7s	306157	>611022	>2h	79776	3m 31s
Test selector	**9504**	0m 35s	+0.5s	144489	122446	4m 24s	9561	0m 20s
Pairs	**7**	**0m 0s**	+0.0s	47	1024	0m 10s	1024	0m 11s
Single lock 1	**2520**	0m 4s	+0.2s	36825	2520	**0m 3s**	2520	**0m 3s**
Single lock 2	**22680**	0m 54s	+0.8s	333376	22680	0m 44s	22680	**0m 43s**
Fib 1	**19605**	**0m 13s**	+0.1s	87457	>1303725	>2h	21058	0m 15s
Fib 2	**218243**	3m 15s	+0.6s	1016562	>903216	>2h	232471	**1m 50s**
Szymanski	**65138**	1m 47s	+0.2	322491	>1074022	>2h	65138	**0m 30s**
Updater 1	**33269**	5m 53s	+1.5s	285571	76424	7m 30s	33449	**5m 51s**
Updater 2	**33506**	**5m 57s**	+1.7s	285571	113408	8m 48s	34025	6m 1s
Synthetic 1	**916**	**0m 3s**	+0.5s	6233	23244	0m 26s	1659	0m 4s
Synthetic 2	**8199**	**0m 39s**	+3.8s	32148	874520	22m 59s	22496	1m 33s

The pairs program is similar to the exponential example discussed in Sect. 2 and can be considered as a close to optimal program for the unfolding approach. For these types of programs the unfolding approach scales better than approaches based on unwinding the program as a tree, as is expected.

In cases where the threads of the programs are more tightly coupled and most of the Mazurkiewicz traces need to be explored even with unfoldings, the DPOR based approaches perform and scale better. This is because DPOR is much more lightweight approach. Examples where DPOR is clearly faster are the Fib2 and Szymanski programs. In these experiments the additional deadlock checking that is needed to make the unfolding approach comparable with DPOR increased the runtime in worst case by 10%. However, in cases where there are no deadlocks or the use of locks is such that circular deadlocks are not possible, the overhead caused by calling the deadlock search algorithm and maintaining the information needed by it is minimal.

The difference between the two DPOR variants shows that approaches based on reducing the execution trees are highly sensitive to small changes in the algorithm. Additional experiments with a version of the DPOR algorithm that does not approximate with heuristics the sets of transitions that need to be fired would be beneficial. Recently an optimal DPOR algorithm has been presented by Abdulla et al. (2014) that never explores more than one interleaving per Mazurkiewicz trace. To guarantee optimality, the algorithm needs to maintain an additional data structure. Implementing such optimal algorithm, however, would bring the runtime overhead of DPOR closer to the unfolding algorithm. Additionally, the unfolding approach does not always need to cover all Mazurkiewicz traces. Therefore even against an optimal version of DPOR, using unfoldings to test multithreaded programs remains a viable alternative.

The benchmarks in these experiments are quite small so that they can be fully tested. This allows the results to be fairly compared. With large programs approaches based on DSE suffer from rapidly growing number of possible execution paths. This means that DSE based approaches cannot typically test a given program exhaustively. However, these approaches can still generate a large number of test cases that are guaranteed to explore different execution paths. This can lead to detecting errors that may be difficult to detect with approaches such as random testing.

7 Related work

There are several approaches to automatically test multithreaded programs. The most closely related approaches to our work perform systematic testing based on dynamic symbolic execution and partial order reductions. The main idea behind these approaches has already been discussed in Sect. 2. Two partial order reduction algorithms that have been combined with DSE are dynamic partial order reduction (Flanagan and Godefroid 2005) and race detection and flipping (Sen and Agha 2006b). The main difference between these two algorithms is that DPOR computes persistent sets and race detection and flipping uses postponed sets (*i.e.*, when a race is detected, in another test run the first transition participating in the race is postponed as much as possible, causing the transitions to swap places in the trace).

As illustrated by the experiments in the previous section, one of the main differences between these approaches to our algorithm is that dynamically computed partial order reductions are typically more lightweight per test execution but in return may require more test executions to cover the program under test as they are required to explore at least one execution from each Mazurkiewicz trace. Having one test per trace also allows these algorithms to detect deadlocks during basic test generation. In our approach we cover the local states of threads and need an additional search for deadlocks. Because of the aim for local state coverage, unfolding based algorithm can sometimes result in exponential reduction in the number of test cases. There is, to our knowledge, no dynamic partial order reduction like approach that can further restrict the search space when only local reachability properties of each thread are to be checked. Testing approaches based on trees can be strongly affected by the order in which transitions are explored in the tests. Lauterburg et al. (2010) experimented with multiple different heuristics for dynamic variants of partial order reduction algorithms for choosing the order in which the transitions are explored and showed that the number of execution paths that need to be tested can vary greatly between the heuristics. With unfoldings this is not a problem and regardless of the order in which transitions are explored, the resulting unfoldings are isomorphic.

Another related approach that can be used to test multithreaded programs is the symbolic execution extension of Java PathFinder (Pasareanu et al. 2008). This approach takes advantage of the on-the-fly partial order reduction implemented in

Java PathFinder that collapses sequences of transitions that can affect other threads into a single transition. To our understanding Symbolic Java PathFinder, however, does not reduce the interleavings cause by executing independent operations (e.g., writes to different variables) like the unfolding and DPOR approaches do. This allows testing for a rich variety of global properties directly as the more global states are explored explicitly. Naturally this means that more test executions need to be performed so that this coverage can be achieved.

There are also different ways to approach automated testing of multithreaded programs that do not systematically try to cover all possible interleavings and execution paths. Perhaps the simplest of these is random testing. The problem with techniques based on random testing is that the probability of exploring a rare interleaving and input value combination can be very low. Our approach can systematically explore these cases but naturally requires more time to compute new test inputs.

The main problem with systemically trying to cover all execution paths of a program is that it requires a huge number of test executions. One approach to reduce the number of tests or direct the testing effort is to use heuristics to select the interleavings to be tested. For example, CHESS (Musuvathi et al. 2008) uses preemption bounding and prioritizes schedules with fewer preemptions. Heuristic approaches can be complimentary to our unfolding based testing technique by prioritizing which possible extension events are selected to be the test targets.

Another class of related work to detect bugs is to monitor executions of a program for potential errors. Runtime monitoring can, for example, be used to detect data races (Flanagan and Freund 2010), deadlocks (Agarwal and Stoller 2006) and atomicity violations (Wang et al. 2010). Approaches such as these can be combined with our testing algorithm by using it to generate test executions that are then monitored.

8 Conclusions

We have presented a new approach that combines dynamic symbolic execution and unfoldings for testing multithreaded programs. With unfoldings the causality and conflicts of events in multithreaded programs can be captured in a natural way. We have shown that this approach can result in some cases even in an exponential reduction to the needed test runs in checking reachability of local states of threads when compared to dynamic partial order reduction based approaches. In particular, we can sometimes cover all of the local states using less test runs than there are Mazurkiewicz traces in the system. The performance of our approach also does not depend on any execution ordering heuristics to improve the partial order reduction effectiveness, unlike approaches that work on execution trees. We have also shown that the unfolding resulting from the testing can be used to detect deadlocks and discussed the advantages and disadvantages of the new approach over existing algorithms. Basically our approach is more efficient in the number of test cases but relies on more expensive algorithms (possible extensions calculation) to do so.

Unfolding based testing is not only useful for finding errors such as uncaught exceptions, assertion violations and deadlocks, as the unfolding generated as part of the testing process can be used in interesting ways. For example, by collecting additional information, the resulting unfolding can be used to determine reachability of global states that were not directly covered by the generated tests. Because the use of unfoldings opens up such interesting directions for further research and as the new testing algorithm provides competitive performance to related partial order reduction approaches working with trees, we we feel that the idea of using unfoldings in testing is a valuable addition to the family of automated testing approaches.

Acknowledgments This work has been financially supported by Tekes - Finnish Agency for Technology and Innovation, ARTEMIS-JU and Academy of Finland (Projects 128050 and 139402).

References

Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.F.: Optimal dynamic partial order reduction. In: Jagannathan, S., Sewell, P. (eds.) POPL, pp. 373–384. ACM, New York (2014)

Agarwal, R., Stoller, S.D.: Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In: Ur, S., Farchi, E. (eds.) PADTAD, pp. 51–60. ACM, New York (2006)

Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008). USENIX Association, Berkely, CA (2008)

de Moura L.M., Bjørner N.: Z3: An efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, Berlin (2008)

Diekert, V.: The Book of Traces. World Scientific Publishing Co. Inc., River Edge, NJ (1995)

Esparza, J., Heljanko, K.: Unfoldings—A Partial-Order Approach to Model Checking. EATCS Monographs in Theoretical Computer Science. Springer, Berlin (2008)

Farzan, A., Madhusudan, P.: Causal atomicity. In: Ball, T., Jones, R.B. (eds.) CAV. Lecture Notes in Computer Science, pp. 315–328. Springer, Berlin (2006)

Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. Commun. ACM **53**(11), 93–101 (2010)

Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) POPL. ACM, New York (2005)

Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer, Secaucus, NJ (1996)

Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005), pp. 213–223. ACM, New York (2005)

Kähkönen, K., Saarikivi, O., Heljanko, K.: Using unfoldings in automated testing of multithreaded programs. In: Proceedings of the 27th IEEE/ACM International Conference Automated Software Engineering (ASE 2012), pp. 150–159 (2012)

Khomenko, V., Koutny, M.: Towards an efficient algorithm for unfolding Petri nets. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR. Lecture Notes in Computer Science, pp. 366–380. Springer, Berlin (2001)

Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Evaluating ordering heuristics for dynamic partial-order reduction techniques. In: 13th International Conference of Fundamental Approaches to Software Engineering, pp. 308–322 (2010)

McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: von Bochmann, G., Probst, D.K. (eds.) CAV. Lecture Notes in Computer Science, pp. 164–177. Springer, Berlin (1992)

Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Draves, R., van Renesse, R. (eds.) OSDI, pp. 267–280. USENIX Association, Berkeley, CA (2008)

Pasareanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M.R., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In: Ryder, B.G., Zeller, A. (eds.) ISSTA, pp. 15–26. ACM, New York (2008)

Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV. Lecture Notes in Computer Science, pp. 409–423. Springer, Berlin (1993)

Saarikivi, O., Kähkönen, K., Heljanko, K.: Improving dynamic partial order reductions for concolic testing. In: Proceedings of the 12th International Conference on Application of Concurrency to System Design (ACSD 2012), pp. 132–141 (2012)

Sen, K.: Scalable automated methods for dynamic program analysis. Doctoral Thesis, University of Illinois (2006)

Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006). Lecture Notes in Computer Science, vol. 4144, pp. 419–423, (Tool Paper). Springer, Berlin (2006a)

Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: Haifa Verification Conference. Lecture Notes in Computer Science, vol. 4383, pp. 166–182. Springer, New York (2006b)

Tillmann, N., de Halleux, J.: Pex—white box test generation for .NET. In: Proceedings of the Second International Conference on Tests and Proofs (TAP 2008). Lecture Notes in Computer Science, vol. 4966, pp. 134–153. Springer, New York (2008)

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot—a Java bytecode optimization framework. In: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999), p. 13. IBM, New York (1999)

Valmari, A.: Stubborn sets for reduced state space generation. In: Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990, pp. 491–515. Springer, London (1991)

Vogler, W., Semenov, A.L., Yakovlev, A.: Unfolding and finite prefix for nets with read arcs. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR. Lecture Notes in Computer Science, pp. 501–516. Springer, Berlin (1998)

Wang, C., Limaye, R., Ganai, M.K., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: Esparza, J., Majumdar, R. (eds.) TACAS. Lecture Notes in Computer Science, pp. 328–342. Springer, Berlin (2010)