Lightweight State Capturing for Automated Testing of Multithreaded Programs

Kari Kähkönen and Keijo Heljanko

Helsinki Institute for Information Technology HIIT
Department of Computer Science and Engineering
School of Science, Aalto University
{kari.kahkonen, keijo.heljanko}@aalto.fi

Abstract. We present a lightweight approach to capture abstract state information that can be used to avoid testing redundant interleavings of multithreaded programs. Our approach is based on modeling states that are observed during the test executions as a Petri net. This model is then used to determine if some execution paths lead to an already explored state. In such cases exploring execution paths from the same state multiple times can be avoided. Our approach does not capture the complete global states of programs but instead it relies on particular commutativity of transitions to determine if they lead to already known abstract states. We have combined this lightweight state capture technique with a dynamic symbolic execution based approach to systematically test multithreaded programs. Experiments show that even without complete state information, the lightweight state capturing technique can sometimes reduce the number of redundant test executions substantially.

1 Introduction

Testing multithreaded programs is challenging due to the large number of execution paths caused by input values and interleavings of threads. One way to avoid redundant test executions is to capture program states and stop a test execution when an already explored state is encountered. However, capturing and storing states of real world multithreaded programs can add a considerable time and space overhead to a testing algorithm. Furthermore, matching states can be nontrivial if the states are expressed symbolically as in some testing approaches such as dynamic symbolic execution (DSE) [8]. For example, in a symbolic state a variable x could have any value that satisfies a constraint $x > 0$. If another execution path leads to an identical symbolic state except that the constraint for x is $x > 5$, the first symbolic state *subsumes* the second one (i.e., the first state represents all concrete states of the second symbolic state). To determine if a symbolic state has been visited before, a subsumption check by a constraint solver is needed. This can be computationally expensive if the constraints are complex and therefore we will not use such an approach in this paper.

An alternative to capturing states is to use stateless algorithms that explore execution paths through a program without explicitly storing state information.

A naive way to do such exploration is to consider all possible input values and interleavings. Approaches like DSE and partial order reductions [7, 15] can be used to avoid redundant test executions. DSE expresses symbolically the sets of input values that cause the same execution path to be followed. Partial order reduction algorithms avoid redundant tests based on the fact that it is not necessary to explore different interleavings of independent state transitions. However, even with such reduction techniques, stateless algorithms can explore the same subset of the state space multiple times.

In this paper we present a lightweight approach to state capturing that can be combined with DSE without the need for subsumption checks that use constraint solvers or for storing complete global states. Our approach is based on the observation that sometimes it is easy to see that interleavings even with dependent transitions commute and thus lead to the same state. As an example, consider a program that has an array in shared memory and the access to this array is synchronized with a lock. This means that if two threads want to read a value from the array (or to update distinct indexes), both threads need to acquire the same lock. The corresponding transitions are dependent and therefore both ways to interleave the accesses to the array need to be explored even if using partial order reductions. However, after both threads have acquired the lock, read a value and released the lock, the program ends up in the same state regardless of the execution order. In this case it is not necessary to know the exact global states of the program to be able to determine that both interleavings result in the same state. In this paper we detect such cases by constructing a Petri net model of the program under test based on the information collected during test executions. This model is constructed such that any modeled state transitions lead to new abstract states unless it is easy to determine that a transition leads to an already known state.

We also present a systematic testing algorithm that constructs the model on-the-fly and uses it to perform state matching. The new algorithm can be seen as extending our previous unfolding based testing approach [10] with lightweight state matching. Naturally without complete state information, the cases where state matching can be done are limited. Nevertheless, experiments show that in cases where our approach can detect that a given state has been visited before, the savings both in testing time and the number of test executions can be substantial. The main contributions of this paper are: (1) a lightweight approach to match states without capturing complete global states or using symbolic subsumption with a constraint solver, (2) a testing algorithm that combines net unfoldings, dynamic symbolic execution and the lightweight state capturing to reduce unnecessary test executions, and (3) an experimental evaluation of the new approach.

2 Background

To keep the presentation simple we assume that in programs to be tested the number of shared variables is fixed and the only nondeterminism in threads is

caused by concurrent access of shared memory or by input data from the environment. We also assume that operations accessing shared memory are sequentially consistent. The state of a multithreaded program consists of the local states of threads and the shared state consisting of the shared variables. The operations on shared memory that are 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. Local operations, such as if-statements, are evaluated solely on the values in the local state and therefore cannot access shared variables directly. In real programs the statements can be modified automatically to satisfy these assumptions by using local temporary variables. The algorithms discussed in this work are based on analyzing sequences of operations observed during test executions and therefore language constructs such as loops, goto-statements and function calls are also supported.

2.1 Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) [8, 14], which is also known as concolic testing, is a systematic test generation approach in which a program is executed both concretely and symbolically at the same time. The concrete execution corresponds to the execution of the actual program and symbolic execution computes constraints on values of the variables in the program by using symbolic values that are expressed in terms of input values. 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, executing a program `x = x + 1; if (x > 0)...;` generates constraints $input_1 + 1 > 0$ and $input_1 + 1 \leq 0$ at the if-statement assuming that the symbolic value $input_1$ is assigned initially to x. 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 through the program under test to be explored systematically.

2.2 Petri nets and Unfoldings

In the following we describe Petri nets and their unfoldings that are used in our testing algorithm to model and explore the states of the program under test.

Definition 1. *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 $\{y \in P \cup T \mid (y,x) \in F\}$. The postset of a node x, denoted by $x^\bullet$, is the set $\{y \in P \cup T \mid (x,y) \in F\}$. 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. 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).

Graphically markings are represented by putting tokens on circles that represent the places of a net. A transition t is enabled in a marking that puts tokens on the places in the preset of t.

Definition 2. *The causality relation $<$ in a net is the transitive closure of F. The reflexive and transitive closure of F is denoted by $\leq$.*

Definition 3. *Two nodes x and y are in conflict if there are distinct transitions t_1 and t_2 such that ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ and $t_1 < x$ and $t_2 < y$.*

Similarly as a directed graph can be unwinded into a tree that represents all paths through the graph, a Petri net can be unfolded into an acyclic net called occurrence net. For acyclic Petri nets the causality relation is a partial order.

Definition 4. *An occurrence net O is an acyclic net (B,E,G), where B and E are sets of conditions (places) and events (transitions) and G is the flow relation. Occurrence net O also satisfies the following conditions: 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. If an occurrence net is obtained by unfolding a Petri net, the events and conditions in it can also be labeled with the corresponding transitions and places.

Definition 5 (Adapted from [11]). *A labeled occurrence net is a tuple $(O,l) = (B,E,G,l)$ where $l : B \cup E \mapsto P \cup T$ is a labeling function such that: (i) $l(B) \in P$ and $l(E) \in T$; (ii) for all $e \in E$, the restriction of l to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet l(e)$; (iii) 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; and (iv) for all $e, f \in E$, if ${}^\bullet e = {}^\bullet f$ and $l(e) = l(f)$ then $e = f$.*

Different labeled occurrence nets can be obtained by stopping the unfolding process at different times. The maximal labeled occurrence net (possibly infinite) is called *the unfolding* of a Petri net [3]. To simplify the discussion in this paper, we use the term unfolding for all labeled occurrence nets and not just the maximal one. To illustrate the concepts above, let us consider the Petri net shown on the left in Fig. 1. Next to the Petri net are its computation tree and unfolding that represent the computations (sequences of transitions) of the Petri net in an acyclic manner. The nodes in the computation tree represent the reachable markings of the Petri net (i.e., global states) and the edges represent transitions that lead from one marking to another. In the unfolding each event

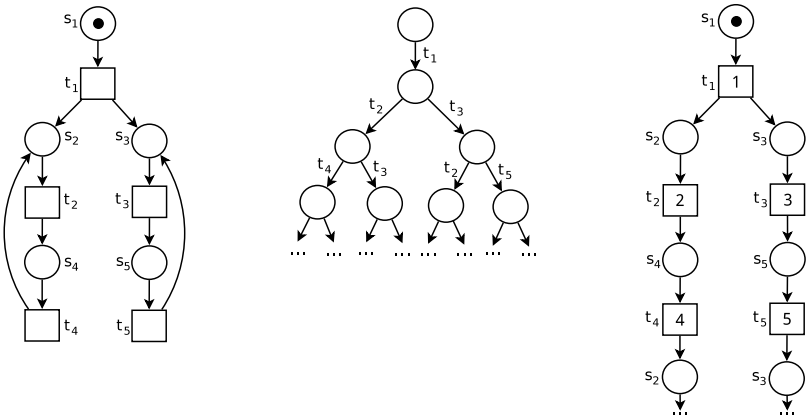

Fig. 1. A Petri net, its computation tree and unfolding

and condition are labeled with the corresponding transition or place of the Petri net. For some Petri nets its unfolding can be exponentially more succinct than the corresponding computation tree.

All the reachable markings of a Petri net can be explored by traversing its computation tree. However, exploring the full computation tree is not always necessary. In our example, the marking reached after firing t_1 is $\{s_2, s_3\}$. Firing transition sequence $t_1t_2t_4$ leads to the same marking and therefore there is no need to expand the computation tree further from the node that corresponds to this transition sequence. In a similar way, it is also possible to compute a finite prefix of the unfolding that captures all the reachable markings of a Petri net. However, as an unfolding is a more succinct representation than the computation tree, computing a finite prefix is not as straightforward. In the following we use notation similar to [3].

Definition 6. *The local configuration of an event e in an unfolding is the set $\{e' \mid e' \leq e\}$.*

Definition 7. *Let e be an event in the unfolding of Petri net N. **St**(e) denotes the marking of N reached after firing the transitions corresponding to the events in the local configuration of e.*

Definition 8. *Let $\prec$ be a partial order on the events of an unfolding. An event e in a prefix of the unfolding is a terminal (also known as a cut-off event) if there exists an event e' such that $e' \prec e$ and* $\mathbf{St}(\text{e'}) = \mathbf{St}(\text{e})$.

A finite prefix of an unfolding can be constructed by leaving out any events that are causally preceded by a terminal event. That is, if an event e' has been added to the unfolding before event e and $\mathbf{St}(\text{e'}) = \mathbf{St}(\text{e})$, the unfolding process can be stopped at event e. To illustrate this, let us consider the unfolding in Fig. 1

again. The numbers on the events in the unfolding denote the order in which they have been added to the unfolding and therefore follow the $\prec$ partial order. Let us assume that e_n denotes the event labeled with n. The local configuration of e_4 consists of events e_1, e_2 and e_4. The marking reached by firing these events corresponds to the marking $\mathbf{St}(e_4) = \{s_2, s_3\}$. Note that event e_1 has been added to the unfolding before e_4 and $\mathbf{St}(e_1) = \mathbf{St}(e_4)$. This means that e_4 is a terminal and no events that are causally preceded by it need to be added to the unfolding. Similarly the event e_5 is also a terminal.

It is important to note that not all partial orders $\prec$ lead to complete prefixes [3]. In other words, if the prefix is not complete, some reachable marking of the Petri net is not represented in the prefix. It has been shown that if events are added to the unfolding in so called *adequate order*, the prefixes are always complete. In the implementation of our algorithm that is used in the experiments, we use the ERV adequate order as described in [4]. For further details about unfoldings, see [3].

3 Modeling Test Executions

Our lightweight state capturing technique is based on modeling behavior observed during test executions as a Petri net. This model can then be used for state matching in a testing algorithm. The initial state of the program is modeled by having a place for each thread, shared variable and lock in the program under test. Places for threads are abstract representations of their local states and places for shared variables represent valuations of that shared variable. To model execution paths through the program, transitions are added to the model such that they correspond to operations that have been observed during test executions. To model these operations we use the constructs shown in Fig. 2. For clarity, the places corresponding to shared variables and locks have a darker color than places for abstract local states.

The intuition behind the modeling constructs is as follows. When a thread is in a local state such that the next operation to be executed is a write, the operation always results in the same subsequent local state and the same value to be written to the shared memory regardless of the current valuation of the shared variable. This is represented in Fig. 2 such that if the write transition marked with dashed lines is added to the model after the transition with solid lines from the same local state, the transitions result in the same places for the local state and the shared variable. For read operations the resulting local states are always different if the shared variable places are different (i.e., in cases where the values being read might be different). This is again illustrated by a second read transition marked with dashed lines in Fig. 2. However, reading a value does not change it and therefore a read operation can be modeled with a transition that returns the token back to the original shared variable place. For each lock there is always only one lock place and acquiring a lock takes a token from this place and releasing the lock puts the token back to the same place. Local operations of threads are not modeled explicitly as a thread executes them always

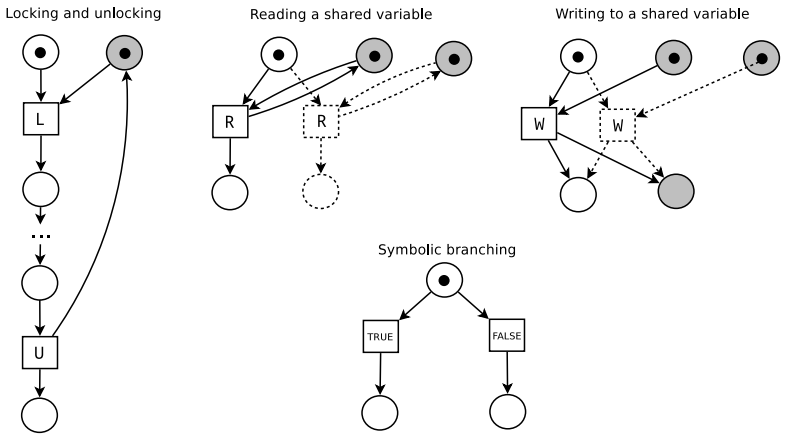

Fig. 2. Modeling constructs

in the same way until the next global operation is encountered. An exception to this are branching operations that depend on input values. In our approach we use symbolic execution to collect constraints that describe which input values cause either the true or false branch to be followed at a given local state. These constraints are added to the transitions for the true and false branches. As such constraints restrict the possible values in a local state, branching transitions lead to new abstract local states.

To model a test execution, a marking that corresponds to the initial state is first created. This marking is used to denote the current state of the test execution. The operations enabled in the initial state are then modeled. The test execution then executes one of these operations and the corresponding transition is fired to update the current marking. In the resulting state all the enabled operations are again modeled unless a corresponding transition already exists in the model. This process is then continued until the whole test execution has been processed. Note that as we model sequences of observed operations, loops in the program are unrolled. Also as we do not track the full local states of threads, we cannot determine if a thread can loop its execution back to an earlier abstract local state. This means that any transition in the model can be fired at most once in any given test execution. The only cycles in the Petri net model occur with places for shared variables and locks. As a test execution can be infinite for nonterminating programs, we limit the length of each test execution by a given bound in order to guarantee termination.

Example 1. Let us consider the program shown in Fig. 3 and a test execution that executes the statements on lines 1,2,3,4,5,6 in that order. Modeling this execution starts with an initial marking $\{s_1, s_2, x_1, y_1, l_1\}$. In the initial state the lock acquire operations of both threads are enabled. These are modeled as the transitions t_1 and t_2. The lock transition belonging to thread 1 is then fired

to obtain a marking $\{s_3, s_2, x_1, y_1\}$. In this new state the operation `x = 1` is the only one that is enabled. As the model does not contain a transition that is enabled in the current marking, the transition t_3 is added to the model and fired. The rest of the test execution is processed in a similar manner to obtain the net in Fig. 3. Note that if a second test execution is made such that thread 2 performs its operations first, no new transitions need to be added to the model. Furthermore, both of these executions end up in the same marking indicating that the resulting states are the same.

```
Global variables:
int x = 0;
int y = 0;

Thread 1:
1: acquire(lock);
2: x = 1;
3: release(lock);

Thread 2:
4: acquire(lock);
5: y = 1;
6: release(lock);
```

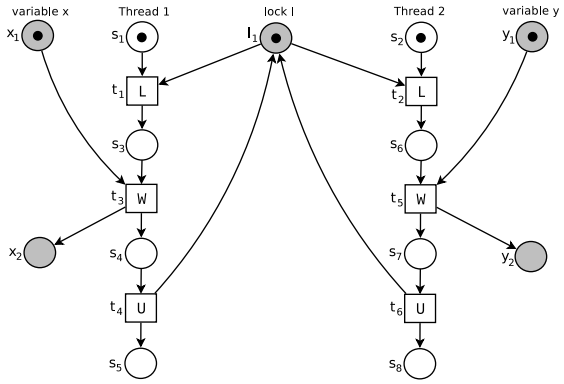

Fig. 3. Locking example

Example 2. Fig. 4 shows another example program where three threads write concurrently to the same shared variable. The partial model on the top of the figure is obtained by performing a test execution where thread 1 is executed first, thread 2 second and thread 3 last. In the initial state the writes for all threads are enabled and this is modeled by the transitions t_1, t_2 and t_3. After executing the write of thread 1, the enabled writes of thread 2 and thread 3 are modeled as transitions t_4 and t_5. The final write of the test execution is modeled as transition t_6. The model on the bottom shows the complete model for the program. In this case there are six possible ways to interleave the write operations. However, there are only three possible end states (markings) for these interleavings and therefore if the program continues after the writes, it is possible to cut the exploration of some of these interleavings.

3.1 Advantages and Limitations for State Matching

As discussed in the examples, test executions following different interleavings can lead to the same marking. This can be used to avoid unnecessary tests

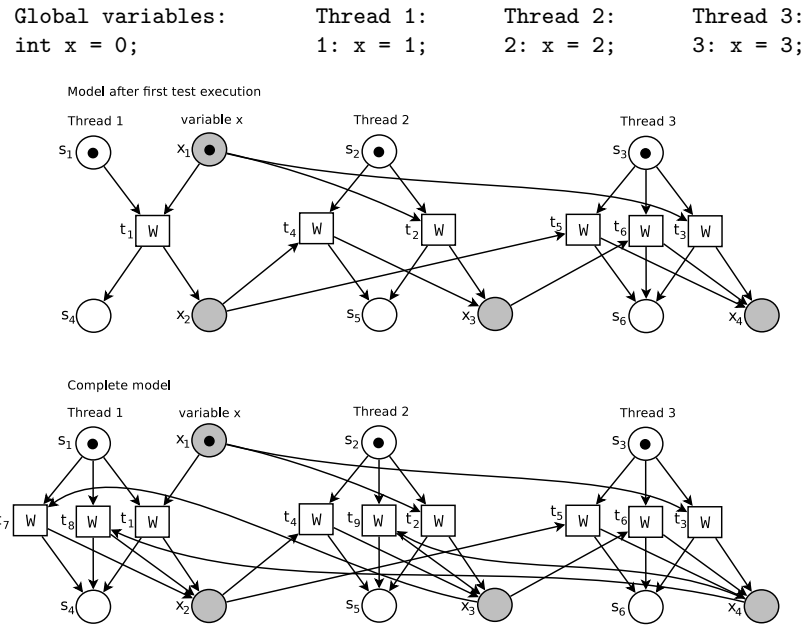

Fig. 4. Concurrent writes example

in automated testing by storing the visited markings, for example, to a hash table. Naturally the cases where our modeling approach can determine that a test execution leads to an already visited state are limited as it does not do any complex reasoning on the symbolic data values. Furthermore, as in the model the loops in the program are unrolled, it cannot be used to detect cases where a test execution loops back to a state that was already visited during the same execution. However, the model can be in some cases used to detect when different interleavings of the same operations lead to the same state. Cases such as accessing different variables in a shared data structure that is protected by a lock do occur and in such cases our approach has potential to scale much better than stateless testing approaches, even if they reduce interleavings of independent operations. Furthermore, the space requirement of storing markings can be considerably smaller than storing full state information.

4 Systematic Model Construction

Modeling test executions as a Petri net is easy. However, to systematically test a given program, we want to perform test executions that cover the complete model. This can be done by starting with a random initial test execution, modeling it and using the obtained information to compute inputs for subsequent test executions. In this section we present two algorithms to do this.

```
Require: A program P
 1: model := empty Petri net
 2: visited := ∅
 3: extend model with a random test execution
 4: EXPLORE(M_0, ∅)
 5: procedure EXPLORE(M, S)
 6:     if M ∉ visited then
 7:         visited := visited ∪ {M}
 8:         PREDICTTRANSITIONSFROMMODEL(M)
 9:         if model is incomplete at M then
10:             EXTENDMODEL(P, S, k)
11:         for all transitions t enabled in M do
12:             M' := FIRE(t, M)
13:             S' := S appended with t
14:             EXPLORE(M', S')
```

Fig. 5. Naive testing algorithm

4.1 Naive Stateful Approach

A simple way to construct a complete model of a program under test is to initially model a random test execution and start traversing the *computation tree* of the model. If the model does not have enough information to determine how the computation tree should be expanded at some state, a new test execution is performed to update the model. An algorithm based on this idea is shown in Fig. 5. It performs a depth-first search on the reachable markings of the model by calling recursively the EXPLORE subroutine that takes the state (marking M of the model) and a sequence S of transitions that lead to this state as input. At each state the algorithm determines that a model is incomplete if it is not known what operations the threads want to perform next or if there are no transitions in the model for these operations. This requires keeping track of the end states of threads observed during test executions (i.e., a thread does not perform any operations after reaching such a state). If the model does not have the necessary information, a test execution to explore the current state is performed. After this the transitions enabled in the current state are known. The algorithm also stores the visited states and backtracks if an already explored state is encountered.

In some cases it is easy to determine from the model which operation a thread wants to perform next even if the corresponding transition in the model is missing. As an example, let us consider the program and the partial model in Fig. 4. Let us assume that we are exploring a marking $m = \{s_1, s_5, s_3, x_3\}$. The model is incomplete at this marking because no transition for thread 1 is enabled in this state. However, each transition from a place representing a local state of a thread has the same type (i.e., from a given local state, the operation the thread wants to perform is always the same). As there is a write transition in the postset of s_1, we know that thread 1 wants to perform a write operation. In cases like this, the missing transition (t_7 in our example) can be added

to the model without performing a test execution. The subroutine PREDICTTRANSITIONSFROMMODEL performs such analysis for each visited marking. To be more precise, PREDICTTRANSITIONSFROMMODEL checks the postsets of the places for local thread states to determine the operations the threads want to perform and adds any missing transitions to the model. For reads and writes this is trivial. For lock operations it needs to be checked that the lock is free in the current state (i.e., the marking contains the respective lock place). Using PREDICTTRANSITIONSFROMMODEL can sometimes significantly reduce the need for test executions. For example, the final model for program in Fig. 4 can be constructed with information obtained from a single test execution.

The EXTENDMODEL subroutine performs a test execution both concretely and symbolically. The subroutine takes as input a sequence S of transitions that leads to the state that is being explored. Bound k for the execution length is used to guarantee termination. To get concrete input values for the test execution, all the symbolic constraints associated with the branching transitions in S are collected and their conjunction is solved using a constraint solver. The sequence S is also given to a runtime scheduler that schedules the execution such that the operations are performed in the same order as the corresponding transitions in S. After reaching the target state, the scheduler is free to follow any schedule.

We call the algorithm in Fig. 5 naive because it explores interleavings of global operations even if they are independent. This is unnecessary to find errors such as assertion violations or reachability of control states. Naturally different interleavings of independent operations lead to the same state and the algorithm backtracks in such cases. To avoid exploring unnecessary interleavings, we present next an algorithm based on unfolding the model. This approach is only guaranteed to cover the reachable local states of threads and to detect all assertion violations. Detecting all deadlocks is not guaranteed. Another approach would be to use partial order reduction algorithms such as DPOR [6]. As using state matching with DPOR requires special care to guarantee the completeness of the algorithm [16], investigating such possibilities is left for future work.

4.2 Unfolding Based Approach

To explore an unfolding of the model instead of the computation tree, we make a small modification to the modeling approach presented in Sect. 3: instead of modeling a shared variable with a single place in each reachable marking, we duplicate the place for each thread. In other words, each shared variable place in the model is replaced with n places, where n is the number of threads in the program. A write transition is made to access each of the n copies while a read transition accesses only the local copy belonging to the thread performing the read. This approach is known as place replication [5] and it has the effect that two concurrent reads of the same shared variable become independent. If place replication is not used, the unfolding process would explicitly explore different interleavings of read transitions. The use of place replication is demonstrated in the example at the end of this section.

```
Require: A program P
 1: model := empty Petri net, unf := initial unfolding
 2: visited := ∅
 3: extend model with a random test execution
 4: extensions := events enabled in the initial state
 5: while extensions ≠ ∅ do
 6:     choose ≺-minimal event e from extensions
 7:     M := St(e)
 8:     predictTransitionsFromModel(M)
 9:     if model is incomplete at M then
10:         extendModel(P, e, k)
11:     else
12:         add e to unf
13:         extensions := extensions \ {e}
14:         if M ∉ visited then    // e is not a terminal
15:             visited := visited ∪ {M}
16:             extensions := extensions ∪ PossibleExtensions(e, unf)
```

Fig. 6. Unfolding algorithm

The unfolding based testing algorithm is shown in Fig. 6 and the idea behind it is similar as with the naive algorithm. Initially a random test execution is performed to start the model construction. The algorithm maintains a set of events that can be used to extend the unfolding. Initially such events are those that are enabled in the initial state. The algorithm then starts adding these extensions to the unfolding in the order specified by the partial order $\prec$ (lines 5-6). As discussed in Sect. 2, we use the ERV adequate order to guarantee completeness.

To be able to avoid exploring states multiple times, the algorithm computes $\mathbf{St}(e)$ for each event e added to the unfolding. This can be seen as the state that is reached by following the shortest execution path to the event e. For the obtained marking (state), the algorithm performs the same analysis for missing transitions as the naive algorithm does (line 8). If after adding the predicted transitions the model is incomplete at the obtained marking, a new test execution is performed to update the model. Otherwise the algorithm adds the selected event to the unfolding and determines if it is a terminal. This is done by checking if an event with the same marking $\mathbf{St}(e)$ has already been added to the unfolding (line 14). If the event is not a terminal, the algorithm computes a set of new events that can be added to the unfolding and adds these events to the set of possible extensions. To be more precise, a possible extension is an event that has not yet been added to the unfolding but could be fired in some reachable marking.

Computing Possible Extensions. There exists several algorithms for computing possible extensions. Most of these algorithms, however, have been designed for arbitrary Petri nets and are computationally the most expensive part of building unfoldings. Such algorithms can be used in our testing approach but this could adversely affect the performance. Fortunately, the Petri net models constructed

in our approach have a restricted structure that makes computing possible extensions more efficient than in the general case. We have recently described an efficient possible extensions algorithm in [10] that works with unfoldings that are constructed in a similar way as in our new testing algorithm. We use this efficient algorithm in the implementation of the algorithm in Fig 6.

Computing Inputs For Test Executions. Extending a model with a test execution can be done similarly as in the naive algorithm. The difference is that with unfoldings we do not have directly a sequence of transitions that leads to the state (i.e., marking M) we want to explore. However, obtaining such a sequence is easy. The state is reached by firing the transitions corresponding to the local configuration of e. If the events in the unfolding have labels that describe the order in which they were added to the unfolding (e.g., the numbers on events in Fig. 1), an event with a larger label cannot causally precede an event with a smaller label. This means that the transitions can be fired in the order given by the labeling of their corresponding events to reach the marking M.

Example 3. To illustrate the unfolding based algorithm, let us consider the Petri net model and its unfolding in Fig. 7. The model represents a program with two threads that acquire a lock and read a shared variable x. The first thread also branches its execution based on input values at the end. Note that the places for x have been replicated for each thread (i.e., x_1 has been replicated to x_1^1 and x_1^2). This makes the read transitions independent as explained earlier. To construct the net in Fig. 7, the algorithm first performs a random test execution. In this example, any execution provides enough information to model all the transitions shown in the Petri net model. However, depending which branch the first thread follows at the end, the model remains incomplete at place s_9 or s_{10} as the corresponding local state is not explored. From the initial state it is possible to fire transitions t_1 and t_2. The events 1 and 2 correspond to these transitions and are added to the set of possible extensions. The algorithm selects event 1 to be added to the unfolding and this results in a new reachable marking where it is possible to fire event 3. The found event is added to the set of possible extensions and the same process is continued until the algorithm selects the event 12 to be added to the unfolding. The marking computed at line 7 is the same for this event as well as for event 11. Therefore event 12 is a terminal (marked with a cross) and possible extensions for it are not computed. Let us assume that the initial test execution did not explore the state corresponding to place s_9. To add event 13 to the unfolding, the algorithm needs first to perform a test execution to explore s_9 so that it has enough information to compute possible extensions for event 13. This is achieved by a test execution that follows the transitions corresponding to the events 1, 3, 5 and 13. Let us assume that the symbolic constraint associated with t_7 is $input_1 > 5$. Solving this constraint gives the test execution a concrete input value (e.g., the value 6). After performing the test execution, the algorithm knows that s_9 corresponds to an end state and can continue the unfolding process by adding event 13 and finally event 14.

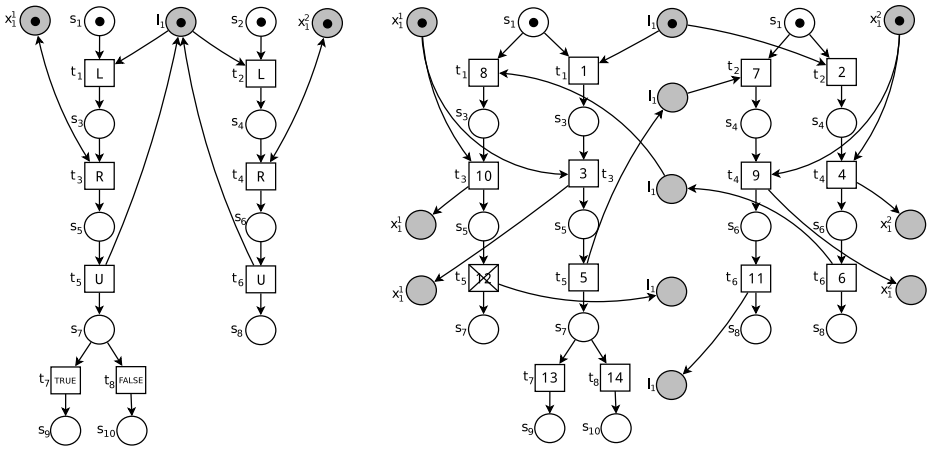

Fig. 7. A model and its unfolding

5 Experiments

We have performed a set of experiments with the new algorithms and compared them with a combination of DSE and DPOR as described in [13] and the stateless unfolding based testing algorithm described in [10]. The naive algorithm described in this paper and DPOR have also been augmented with sleep sets [7] to further reduce the number of test executions needed. The stateless unfolding algorithm constructs similar unfoldings (without constructing the Petri net model) as the new algorithm except that no terminal events are used. In this sense our new approach can be seen as extending the testing approach in [10] by taking some state information into account.

The following benchmarks are used in the experiments. *Fib* and *Szymanski* are from the 1st International Competition on Software Verification except that they have been simplified by limiting how many times some loops are executed. *Filesystem* benchmark is from [6] where it was used to evaluate DPOR. *Dining* implements a dining philosophers problem where each philosopher eats twice. In *Locking* all accesses to shared memory are protected by a single lock. *Updater* contains a set of threads where some threads update values in shared memory and other threads perform work based on these values. *Writes* is similar to the program in Fig. 4 except with more threads and more writes per thread. Finally, synthetic benchmarks perform randomly generated sequences of operations on input values and on global variables. Benchmarks with multiple variants are similar with each other except that the number of threads increases or the program otherwise increases in complexity.

The results of the experiments are shown in Table 1. For each algorithm the table shows the number of test executions needed to fully cover the program under test and the time required to do this. As the algorithms are partially ran-

	Stateless unfolding		Stateless DPOR		Stateful naive		Stateful unfolding	
Benchmark	tests	time	tests	time	tests	time	tests	time
Fib 1	19605	0m 17s	21102	0m 21s	5746	**0m 11s**	4946	0m 15s
Fib 2	218243	4m 18s	232531	4m 2s	53478	3m 45s	46829	**3m 15s**
Filesystem 1	3	**0m 0s**	142	0m 4s	-	(> 30m)	3	**0m 0s**
Filesystem 2	3	**0m 0s**	2227	0m 46s	-	(> 30m)	3	**0m 0s**
Dining 1	798	0m 3s	1161	0m 3s	3	**0m 0s**	4	**0m 0s**
Dining 2	5746	0m 14s	10065	0m 22s	3	**0m 1s**	3	**0m 1s**
Dining 3	36095	1m 29s	81527	3m 29s	2	0m 7s	4	**0m 1s**
Dining 4	205161	12m 55s	-	(> 30m)	-	(> 30m)	2	**0m 3s**
Szymanski	65138	2m 3s	65138	**0m 30s**	50264	0m 43s	46679	2m 35s
Locking 1	2520	0m 8s	2520	0m 6s	20	**0m 1s**	18	0m 3s
Locking 2	22680	0m 56s	22680	0m 47s	29	**0m 2s**	26	0m 9s
Locking 3	-	(> 30m)	-	(> 30m)	115	**0m 21s**	89	3m 32s
Updater	33269	2m 22s	33463	2m 6s	13586	**1m 23s**	12259	1m 52s
Writes	-	(> 30m)	-	(> 30m)	1	**0m 0s**	1	**0m 0s**
Synthetic 1	926	0m 3s	1661	0m 4s	68	**0m 1s**	62	**0m 1s**
Synthetic 2	8205	0m 41s	22462	1m 20s	123	**0m 7s**	97	0m 11s
Synthetic 3	11458	1m 12s	37915	2m 18s	326	1m 8s	298	**0m 30s**

Table 1. Comparison of different approaches

domized (e.g., the random initial execution), the experiments were repeated ten times and the average results are reported. As a sanity check, it was checked that both the naive and the stateful unfolding algorithms generated models of the same size. From the results it can be seen that the stateful algorithms can sometimes greatly outperform the stateless testing approaches. The naive algorithm is more lightweight than the unfolding approach and therefore typically faster for small programs. However, the naive algorithm scales poorly on some benchmarks. The unfolding based algorithm is only guaranteed to cover the reachable local states of threads and therefore it typically scales better. In some cases, such as with *Szymanski*, our new approach does not provide a significant reduction to the number of test executions. In such cases the stateful algorithms can be slower than the stateless counterparts. With stateless unfolding the order in which events are added does not matter and therefore unfolding with terminals has the additional overhead of sorting the possible extensions. One disadvantage of the stateful approaches is that they require more memory as the model and the visited markings need to be stored.

6 Related Work

Stateful approaches have been successful in model checking. However, when systematically testing real-world programs, storing explored states can require a considerable amount of memory. Even though methods to alleviate this problem have been developed (e.g, compression and hashing [9] and selective caching [2]),

many testing tools rely on stateless exploration. The problem with stateless testing, even when combined with partial order reductions, is that part of the state space may be explored multiple times. Our work can be seen as balancing between complete state capturing and stateless search.

Yang et al. [16] propose a related lightweight approach to capture states at runtime that is based on tracking changes between successive local states without storing full state information. In their approach the captured local states are abstract but they capture the shared portion of the state concretely. Therefore, unlike our approach, their approach cannot directly be combined with DSE. They also describe a stateful DPOR algorithm based on their state capture approach. To guarantee the soundness of their algorithm, additional computation needs to be performed to make sure that any subset of the state space is not missed. With unfoldings this is handled by adding events in an adequate order.

It is possible to take the valuations of variables into account in state matching. With symbolic execution this leads to subsumption checking of symbolic states. Anand et al. [1] propose a combination of subsumption checking and abstractions for data structures such as lists and arrays. Such approaches are considerably more heavyweight compared to our approach but can match states that our approach cannot. An alternative way to reduce the number of states that need to be explored when using symbolic execution is to use state merging [12], where multiple execution paths are expressed symbolically instead of exploring them separately. This, however, makes path constraints more demanding to solve.

7 Conclusions

We have presented a lightweight approach to capture abstract state information of multithreaded programs. This approach is based on modeling programs under test with Petri nets and using this model to avoid exploring reachable states multiple times. We have presented a testing algorithm that combines the modeling approach with DSE. Based on our experiments, lightweight state matching can greatly improve the scalability of DSE based testing algorithms that target multithreaded programs. Potential directions for future work are combining the state capture approach with other partial order reduction algorithms and implementing modeling constructs for common cases such as waits in while loops that do not change the local state of the waiting thread. Another possibility is to take variable valuations in different states into consideration. As discussed in Sect. 1, capturing full state information can be expensive. However, as the Petri net model contains places for shared variables, it is possible to use the same shared variable place whenever the shared variable has the same concrete value (i.e., the value does not depend on inputs). This could make the model more compact.

Acknowledgments

This work was financially supported by Academy of Finland (project 139402).

References

1. Anand, S., Pasareanu, C.S., Visser, W.: Symbolic execution with abstract subsumption checking. In: Valmari, A. (ed.) SPIN. Lecture Notes in Computer Science, vol. 3925, pp. 163–181. Springer (2006)
2. Behrmann, G., Larsen, K.G., Pelánek, R.: To store or not to store. In: Jr., W.A.H., Somenzi, F. (eds.) CAV. Lecture Notes in Computer Science, vol. 2725, pp. 433–445. Springer (2003)
3. Esparza, J., Heljanko, K.: Unfoldings – A Partial-Order Approach to Model Checking. EATCS Monographs in Theoretical Computer Science, Springer-Verlag (2008)
4. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. Formal Methods in System Design 20(3), 285–310 (2002)
5. Farzan, A., Madhusudan, P.: Causal atomicity. In: Ball, T., Jones, R.B. (eds.) CAV. Lecture Notes in Computer Science, vol. 4144, pp. 315–328. Springer (2006)
6. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) POPL. pp. 110–121. ACM (2005)
7. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1996)
8. 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 (2005)
9. Holzmann, G.J.: The model checker Spin. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
10. 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)
11. 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, vol. 2154, pp. 366–380. Springer (2001)
12. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: Vitek, J., Lin, H., Tip, F. (eds.) PLDI. pp. 193–204. ACM (2012)
13. 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)
14. Sen, K.: Scalable automated methods for dynamic program analysis. Doctoral thesis, University of Illinois (2006)
15. 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-Verlag, London, UK (1991)
16. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient stateful dynamic partial order reduction. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN. Lecture Notes in Computer Science, vol. 5156, pp. 288–305. Springer (2008)