Unfolding based Minimal Test Suites for Testing Multithreaded Programs

Hernán Ponce-de-León, Olli Saarikivi, Kari Kähkönen and Keijo Heljanko
Helsinki Institute for Information Technology HIIT and
Department of Computer Science, Aalto University
Helsinki, Finland
Email: {Hernan.PoncedeLeon, Olli.Saarikivi, Kari.Kahkonen, Keijo.Heljanko}@aalto.fi

Javier Esparza
Fakultät für informatik, Technische Universität München, Germany
Email: esparza@in.tum.de

***Abstract*—This paper focuses on the problem of computing the minimal test suite for a terminating multithreaded program that covers all its executable statements. We have in previous work shown how to use unfoldings to capture the true concurrency semantics of multithreaded programs and to generate test cases for it. In this paper we rely on this earlier work and show how the unfolding can be used to generate the minimal test suite that covers all the executable statements of the program. The problem of generating such a minimal test suite is shown to be NP-complete in the size of the unfolding, and as a side result, covering executable transitions of any terminating safe Petri net is also NP-complete in the size of its unfolding. We propose SMT-encodings to these problems and give initial results on applying this encoding to compute the minimal test suite for several benchmarks.**

I. Introduction

Testing[1] multithreaded programs is a very challenging problem since the number of possible combinations of concrete input values and interleavings of threads is typically so large that exhaustively testing all of them is not practical. In this paper we build on our earlier work on testing terminating multithreaded programs using net unfoldings [1], [2] which allows us to generate small test suites that are often even smaller than those generated by other partial order reduction methods for testing. The approach contains dynamic symbolic execution [3], [4] techniques for symbolic handling of data values, which allows many concrete input values to be covered with a single test case. Thus we have techniques to both minimize the number of interleavings with unfoldings and the number of concrete input values with dynamic symbolic testing techniques. One of the side results of our testing approach is the unfolding of the terminating multi-threaded program in a symbolic form, which can be exploited for further test suite optimization tasks, as shown below.

Sometimes the test suites our earlier approach has generated can be too large. We might be in a testing setup where initializing the test environment can be very expensive, for example the system might need an expensive manual initialization procedure between test cases. In such cases it is natural to ask whether we could optimize the test suite by generating a minimal test suite that still covers the same events of the unfolding. Another relevant question is whether we can generate a minimal test suite that still covers all executable statements of the program, which might be significantly smaller, but still sufficient for many testing purposes.

[1]We use the term *testing*, but *verification* has also been used in the literature.

A. Related Work

One popular approach to systematically test single threaded programs is dynamic symbolic execution [3], [4] that allows all execution paths of a program to be covered without explicitly testing all input combinations. This testing approach can also be extended to multithreaded programs by using a runtime scheduler that controls the execution of threads [5]; the runtime scheduler can be forced to execute the execution steps of threads in an specific order.

If one intends to find errors such as assertion violations, it is not necessary to explore every possible interleaving because some of the operation of the programs are independent and the final state after executing them is the same regardless of the order in which the operations are executed. Execution paths can therefore been partitioned into equivalence classes called Mazurkiewicz traces [6]. Partial order reduction is one of the techniques that exploit independence between operations by reducing the number of explored interleavings, but still analyzing at least one representative for each equivalence class [7]. Recently, an improvement to this method have been proposed to explore exactly one execution for each Mazurkiewicz trace [8]. When using unfoldings, each Mazurkiewicz trace is represented by a maximal configuration and therefore the same optimality can be obtained by exploring only one execution for each maximal configuration.

Even using techniques such as dynamic symbolic execution or partial order reduction to alleviate the space state explosion problem, the number of execution paths typically grows fast. In order to achieve scalability, an alternative approach

is to only cover local states of each thread instead of all the Mazurkiewicz traces. This approach still allows assertion violations to be detected. The testing algorithm presented in [1] is based on Petri net unfoldings and dynamic symbolic execution and explores all the reachable local states of threads. This approach is extended to contextual nets in [2] allowing in general a more succinct representation of the execution paths.

B. Contributions

Given an unfolding representation of the program under test, we study the problem of finding minimal test suites to cover every event in the unfolding. We show that this problem is NP-complete and propose an SMT-encoding for it. Additionally we explain how a similar encoding can be used to cover all the statements of the program under test. We use several benchmarks to compare the results obtained by these encodings with the test suites generated by the algorithms in [1] and [2]. As a side result, we show that the problem of covering all the transitions in a safe and terminating Petri net is also NP-hard in the size of its unfolding.

As a technical note, we have symbolic data in multithreaded programs, which makes it very difficult to use cut-off events in our unfoldings to handle programs with cyclic state space: as each event corresponds to many concrete input values, the reasoning about cut-offs will need reasoning about symbolic data, which currently requires too much overhead. See [9] for more discussion. We will discuss how adding cut-offs will change the setup.

The rest of the article is structured as following: Section II presents the assumptions on the kind of programs we consider, the basic notions of dynamic symbolic execution, regular and contextual nets and their corresponding unfoldings; Section III explains how to model a given program with different kind of nets and assumptions; Sections IV and V state respectively the problems of covering all events of an unfolding and all transitions of a Petri net and give solutions based on SMT-encodings; we conclude in Section VI.

II. Background

In this section we state our assumptions on the programs under test and give a brief overview of the central concepts needed to understand the rest of the paper.

A. The program under test

In order to simplify the presentation, we consider programs with an acyclic state space (programs which terminate) where the number of threads and shared variables is fixed and the only nondeterminism is given by the concurrent access to shared variables and by input data from the environment. We also assume that the operations accessing shared memory are sequentially consistent. The states of the program consist of local states of the threads and the program's shared state; those states are modified by the execution of operations. Operations are divided into invisible operations which only modify the local state of a thread and visible operations which modify the global state of the program and are the only operations that can directly affect the execution in other threads. If-statements are evaluated only on the values in the local state of the executing thread and therefore cannot access shared variables directly. Programs can be modified (e.g. by using temporary variables) to satisfy these assumptions without changing the behavior of the program. Visible operations include acquire and release of a lock and reading from or writing to a shared variable. Read operations access the value of a shared variable and assign it to a variable in the local state of the thread performing the operation. Write operations assign either a constant or a value from a local variable to a shared variable. A test execution is a sequence of program operations; a test is a set of input values and a schedule.

B. Dynamic symbolic execution

Dynamic symbolic execution (DSE) or concolic testing [3], [4] is a test generation approach which executes a program 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, 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 and therefore is it not necessary to test them all. 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 allowing to test other branches in another test execution. 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. In the modeling of our program we consider the branching of the program dependent on inputs from the environment.

C. Petri nets and their unfoldings

Our approach consists of modeling the observable behavior of a multithreaded program during testing with different kinds of net unfoldings. Different modelings of the program are presented in the next section and in the following we describe the basic concepts needed to understand them.

Regular nets. A net is a triple (P, T, F) where P and T are disjoint sets of places and transitions and $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation. Places and transitions are called nodes and elements of F arcs. The preset and postset of a node x are respectively defined as ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$. A marking of a net is a mapping

$P \to \mathbb{N}$. A Petri net is a tuple $\mathcal{N} = (P, T, F, M_0)$ where M_0 is the initial marking of the net (P, T, F). Graphically markings are represented by putting tokens on circles that represent the places of a net. We restrict to the so-called safe nets where each marking puts zero or one token at each place. A transition t is enabled in any marking that puts tokens on all the places in the preset of t. The causality relation $<$ in a net is the transitive closure of F while its reflexive and transitive closure is denoted by $\leq$. A set of causes of a node x is defined as $\lfloor x \rfloor = \{t \in T \mid t \leq x\}$. Two nodes x and y are in conflict (denoted by $x \# y$) if there are transitions $t_1 \neq t_2$ such that ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ and $t_1 \leq x$ and $t_2 \leq y$.

In the same way a directed graph can be unrolled into a tree that represents all paths through the graph, a Petri net can be unrolled into an acyclic net called an occurrence net. An occurrence net is an acyclic net (B, E, G) where B and E are called conditions and events and G is the partially ordered flow relation. The occurrence net also satisfies the following conditions: *(i)* for every $b \in B$, $|{}^\bullet b| \leq 1$; *(ii)* for every $x \in B \cup E$ the set $\lfloor x \rfloor$ is finite; and *(iii)* no node is in conflict with itself.

A branching process is a tuple $(\mathcal{O}, l) = (B, E, G, l)$ where $l : B \cup E \to P \cup T$ 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 $Min(\mathcal{O})$ is a bijection between $Min(\mathcal{O})$ and M_0, where $Min(\mathcal{O})$ 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 $e = f$. The labeling l relates each event and condition with its corresponding transition and place in the (folded) net. The branching process represents all the possible interleavings between transitions of the net. Different branching processes can be obtained by stopping the unrolling process at different depths. The maximal (possibly infinite when the state space is acyclic) branching process is called the unfolding of a Petri net. To simplify the discussion in this paper, we use the term unfolding for all branching processes and not just the maximal one.

Given an unfolding $\mathcal{U} = (B, E, G)$, any causally closed and conflict-free set of events forms a configuration: $C \subseteq E$ is a configuration iff *(i)* $e \in C \wedge e' \leq e \Rightarrow e' \in C$, and *(ii)* $e \in C \wedge e \# e' \Rightarrow e' \notin C$. Configurations of the unfolding represent executions paths.

Example 1. *Fig. 1 presents two unfoldings modeling the behavior of a program with two threads reading a shared variable* x*. The first unfolding keeps only one copy of the variable (conditions labeled by* x*) while the second one keeps local copies for each thread (conditions labeled by* x_1 *and* x_2*). Tokens represent permission to access the variable or one of its local copies. The first unfolding only allows serialized access to the shared variable while read operations are considered independent in the second unfolding; this is done by replicating the conditions representing the variable* x *(for each variable there is one condition representing it for each thread). Events* r_1 *and* r_4 *are causally related* $(r_1 < r_4)$ *while events* r_1 *and* r_2 *are in conflict* $(r_1 \# r_2)$*. Events* r_5 *and* r_6 *are neither causally related nor in conflict and are called concurrent* $(r_5 \ \mathbf{co} \ r_6)$*. The configurations* $\{r_1, r_4\}$ *and* $\{r_2, r_3\}$ *of the first unfolding show the two possible ways in which the read operations can be sequentially executed in (a) while the configuration* $\{r_5, r_6\}$ *shows that the operations can be done independently in (b).*

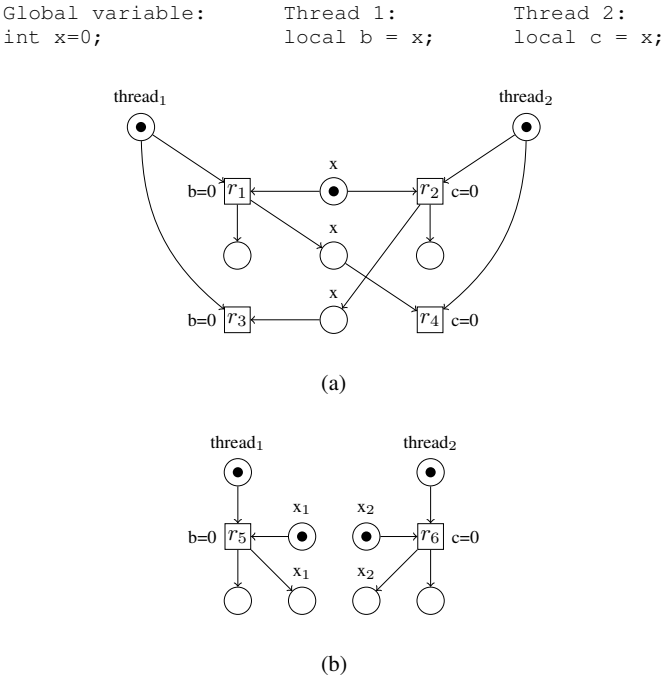


Fig. 1: An example program with its unfolding representation with (a) serialized access to shared variables and (b) place replication.

Contextual nets. Even if unfoldings allow to represent the possible interleavings between transitions of a Petri net in a compact way, this representation can be done more succinctly by extending regular nets with read arcs [10]. A contextual net (c-net) is a tuple (P, T, F, C), where (P, T, F) is a regular net and $C \subseteq P \times T$ is a context relation which elements are called read arcs. The context of a transition t is defined as $\underline{t} = \{p \mid (p, t) \in C\}$. The causality relation $<$ in a c-net is the transitive closure of $F \cup \{(t, t') \in T \times T \mid t^\bullet \cap \underline{t'} \neq \emptyset\}$. Two transitions t and t' in a c-net are in asymmetric conflict, denoted by $t \nearrow t'$, iff *(i)* $t < t'$, or *(ii)* $\underline{t} \cap {}^\bullet t' \neq \emptyset$, or *(iii)* $t \neq t' \wedge {}^\bullet t \cap {}^\bullet t' \neq \emptyset$. The asymmetric conflict $t \nearrow t'$ represents the fact that in any execution where both t and t' happen, t should precede t'.

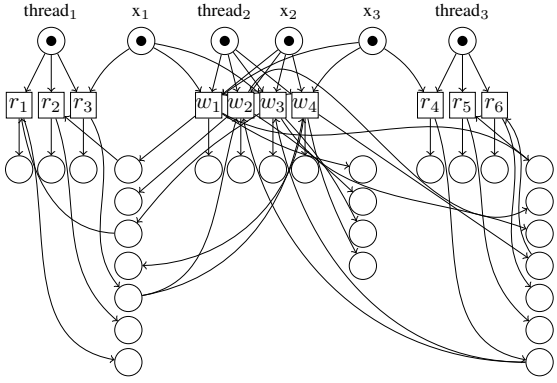
As in the case of regular nets, c-nets can be unfolded into an acyclic c-net describing all the possible paths from its initial marking. A contextual occurrence net is an acyclic c-net (B, E, G, C) such that: *(i)* for every condition b we have $|{}^\bullet b| \leq 1$, *(ii)* the causal relation is irreflexive and its reflexive closure $\leq$ is a partial order such that $\lfloor x \rfloor$ is finite for any node $x \in B \cup E$, and *(iii)* and $\nearrow_{\lfloor e \rfloor}$ is acyclic for every $e \in E$.

The configurations of a contextual unfolding are formed by causally-closed (considering both the flow relation and the context) and $\nearrow$-cyclic-free set of events.

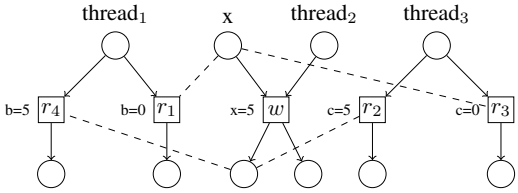
Example 2. *Fig. 2 shows a program with three threads, two of them reading a shared variable and a third one writing it. The behavior of this program is represented by a regular unfoldings in (a). The same program can be modeled by the c-net in (b) using read arcs. In the unfolding (a) the execution of a read operation is modeled by an event which generates a new condition representing the variable. All these conditions enable new write operations and four events (w_1 - w_4) are added to the unfolding. In the case of c-nets, the read operations can be modeled with read arcs and since new variable conditions are not generated, only one event is necessary to model the write operation.*

```
Global variable:
int x;

Thread 1:          Thread 2:       Thread 3:
local b = x;       x = 5           local c = x;
```



(a)



(b)

Fig. 2: Regular and contextual unfolding of a program.

III. Modeling multithreads programs

It is possible to represent the execution paths of a program with its computation tree where every interleaving is explicitly represented. However, to exploit the independence between some operations, net unfoldings can be used to obtain a more succinct representation in many cases. This can be done by representing shared variables, locks and local states of threads with conditions and operations with events. Each event represents the execution of the statements of a visible operation and any subsequent invisible operations from the same threads. Note how this definition groups the execution of any invisible operations together with the previous visible one, thus omitting the interleavings of invisible operations. As typical with approaches that used DSE, we do not model the local operations of threads unless their result depends on input values.

We present three different ways to model a program: the first approach (which we call naive) does not take into account that concurrent reads to the same shared variable can be done independently; the second approach uses a technique called place replication to avoid unnecessary dependencies between reads; the final approach uses contextual nets which may reduce the size of the unfolding by introducing read arcs.

We assume that there is for each thread a set of conditions for each program location (i.e. program counter values) the thread can be in. We also assume that there is a set of conditions for each lock in the program. The constructs of Fig. 3 can be used to model a program by initially constructing conditions for each thread, shared variable and lock that exists in the initial part of the program. A marking containing these conditions represents the initial state of the program. The constructs (a),(b) and (c) represent symbolic branching of the program depending on inputs and acquiring or releasing locks for any of the three modeling approaches. Sections III-A and III-B explain how reading from and writing to shared variables can be modeled with different kind of nets and assumptions.

A. Modeling programs with regular unfoldings.

A naive way to model access to a shared variable using regular unfoldings is to associate each variable with a condition and then every read or write event consumes it and produces a new condition representing the variable (see Fig. 3 (d)). The executions of the simple program of Fig. 1 with a shared variable and two threads reading it can be modeled by the unfolding (a). This unfolding shows that the naive approach only allows serialized access to the shared variable, i.e. it contains two possible executions $r_1 \cdot r_4$ and $r_2 \cdot r_3$ represented by its configurations.

To avoid the serialized access of reading operations, shared variable conditions can be duplicated for each thread: each shared variable is modeled by n conditions, 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 (see Fig. 3 (e) and (f)). This approach is known as place replication [11] and it has the effect that two concurrent reads of the same shared variable become independent. Fig. 1 (b) shows the unfolding modeling the program with place replication; the events representing the read actions become independent and the two test executions of the program $r_5 \cdot r_6$ and $r_6 \cdot r_5$ can be obtained as linearizations of its configuration $\{r_5, r_6\}$. This unfolding contains only two events instead of four as in the naive case.

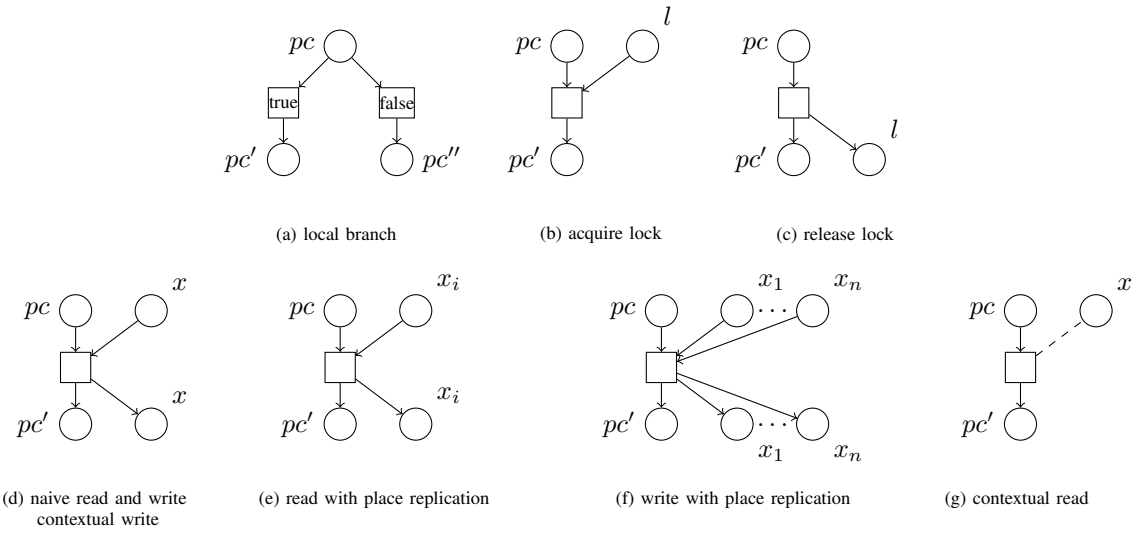


Fig. 3: Modeling programs with unfoldings.

One of the disadvantages of the place replication technique is that it forces to fix the number of threads in the program and thus dynamic creation of threads is not supported. Programs with thread creation can be modeled by using contextual nets (see [2]).

B. Modeling programs with contextual occurrence nets.

Even if the place replication technique allows to represent the independence between concurrent reads, it may generate unnecessary instances of a write operation. Consider the program of Fig. 2 where two threads read a shared variable and a third one writes it. The unfolding (a) is the one obtained using place replication and the constructs (e) and (f) of Fig. 3. There are four different instances w_1-w_4 of the write operation which correspond to the four ways to interleave the access to the shared variable.

In order to obtain a smaller unfolding, contextual nets can be used. The shared variable places are no longer replicated for each thread (recall that the reason for the place replication is to make two concurrently enabled read operations independent in the unfolding). With contextual nets read operations can be modeled using read arcs. The construct for a write transition is the same as in the naive approach with regular nets while read transitions have shared variable conditions in their context (see Fig. 3 (d) and (g)). The program of Fig. 2 can be modeled by the c-net (b). Notice that the four instances of the write operation are replace by a single write event, but the four ways to interleave the executions of the program are still represented by the four configurations of the c-net.

C. Program unfolding

Sections III-A and III-B explain how to model a program using regular and contextual unfoldings representing all the possible ways in which operations of the program can be interleaved. Each unfolding represents symbolically all the possible executions of the program, i.e. any linearization of a configuration represents an execution of the program. Even if a program has different representations (for example the nets obtained using place replication or the contextual approach) any test execution of the program can be obtained as the linearization of some configuration in any unfolding. For each of the unfolding representations, every maximal (w.r.t set inclusion) configuration corresponds to a Mazurkiewicz trace of the program. Since read operations are considered independent, the program of Fig. 2 has four Mazurkiewicz traces representing the final states of the program

$$b = 0, c = 0 \quad b = 0, c = 5 \quad b = 5, c = 0 \quad b = 5, c = 5$$

This traces correspond to the four maximal configurations of both unfolding in Fig. 2:

$$\{r_3, r_4, w_2\}, \{r_3, w_3, r_1\}, \{r_4, w_4, r_6\}, \{w_1, r_2, r_5\}$$

for the regular unfolding (a), and

$$\{r_1, r_3, w\}, \{r_1, w, r_2\}, \{r_3, w, r_4\}, \{w, r_4, r_2\}$$

for the contextual one (b).

If one is interested only in the local states of the threads, it can be observed that `Thread 1` and `Thread 3` only have two local states: $b = 0$ or $b = 5$ and $c = 0$ or $c = 5$. Partial order reduction techniques preserving Mazurkiewicz traces do not take into account local states of threads and therefore any such algorithms would explore at least four executions paths for the this program. In the next section we show how to reduce the number of test executions in the program while covering every local state of the threads.

IV. Minimal Test Suites Based on Unfoldings

The goal of this section is, given an unfolding representation of a multithreaded program, compute the minimal test suite covering every event. We show that this problem is NP-complete in the size of the unfolding and solve it using an

SMT-encoding. If in addition the information about which statements are executed by each event is given, the encoding can be modified to minimize the test suite covering every statement of the program.

A. Events covering

Given the unfolding representation of a multithreaded program, we define the following decision problem to cover the unfolding with a fixed number of test executions.

Definition 1 (EVENTS-COVER). *Given an unfolding $\mathcal{U} = (B, E, G)$ and an integer k, decide whether there exists a set $\{C_1, \ldots, C_k\}$ of configurations of $\mathcal{U}$ covering E.*

Deciding if there exists a set of k configurations that covers every event in the unfolding is an NP-complete problem.

Theorem 1. *EVENTS-COVER is in NP.*

Proof: We create a nondeterministic algorithm with a polynomial running time. The requirements for the set $\{C_1, \ldots, C_k\}$ are:

- each C_i is a configuration,
- $\bigcup_{i \leq k} C_i = E$

Algorithm 1 first guesses a set of k subsets of E, thus the amount of nondeterminism needed is polynomial in the size of the inputs. Then the algorithms checks that the conditions mentioned above are fulfilled. All the loops of the program have a polynomial upper bound on the number of iterations in the size of the input (both k and n_i are smaller than $|E|$), and thus the program has a polynomial running time after the nondeterministic initial guess has been made. To simplify the presentation we use two subroutines $CAUSALLY\text{-}CLOSED(e, C)$ and $CONFLICT(e, e')$. The first one returns true iff all the events in the past of e are in C; the other returns true iff events e and e' are in conflict. For regular unfolding, the latter can be checked by traversing the past of both events and checking if there exist $e_1 \leq e$ and $e_2 \leq e'$ such that ${}^\bullet e_1 \cap {}^\bullet e_2 \neq \emptyset$. For a contextual unfolding, we need to check that there exist no cycles of asymmetric conflict containing e and e': the direct graph $G = (V, A)$ were $V = E$ and $(e, e') \in A$ iff $e \nearrow e'$ is polynomial w.r.t the size of the unfolding and we can detect cycles with complexity $\mathcal{O}(|V| + |A|)$.

■

Theorem 2. *EVENTS-COVER is NP-hard.*

Proof: We show a reduction from the graph coloring problem to EVENTS-COVER; since a regular unfolding is also a contextual one with an empty context, we construct a regular unfolding in the reduction. Let G be a graph with set of vertices V and edges A, we construct an unfolding $\mathcal{U} = (B, E, G)$ in the following way:

- for each each vertex $v \in V$ there is an event e_v in E and conditions c_v, c'_v in B such that $c_v \in {}^\bullet e_v$ and $c'_v \in e_v{}^\bullet$
- for each edge $a = (v_1, v_2) \in A$ there is a condition c_a in B with $c_a \in {}^\bullet e_{v_1} \cap {}^\bullet e_{v_2}$.

Algorithm 1

Input: An unfolding net $\mathcal{U} = (B, E, G)$ and and integer k
Output: Accept/reject.

```
Guess a set of sets {C_1, ..., C_k} where C_i = {e^i_1, ... e^i_{n_i}} ⊆ E
for i := 1 ... k do
    for j := 1 ... n_i do
        if ¬CAUSALLY-CLOSED(e^i_j, C_i) then reject
        for l := 1 ... n_i do
            if CONFLICT(e^i_j, e^i_l) then reject
for e ∈ E do
    found := False
    for i := 1 ... k do
        if e ∈ C_i then found := True
    if ¬found then reject
accept
```

The resulting unfolding has not causal dependencies, i.e. $\leq = \emptyset$, and $e_{v_1} \# e_{v_2} \Leftrightarrow (v_1, v_2) \in A$ (which is equivalent to e_{v_1} **co** $e_{v_2} \Leftrightarrow (v_1, v_2) \notin A$). It is easy to see that the created net is linear in the size of the input graph and it is also straightforward to generate it in polynomial time.

We claim that G is k-colorable iff E is covered with k configurations.

$\Rightarrow$) Given a coloring of G, let V_i be the set of vertices colored by i. For every pair of vertices $v_1, v_2 \in V_i$ we know that $(v_1, v_2) \notin A$ (if they have the same color, they cannot be adjacent) and therefore V_i represents a conflict-free set of events. Since $\leq = \emptyset$, every V_i represents a causally-closed set and it follows it represents a configuration. Since every vertex is colored using k colors, every event is covered with just k configurations.

$\Leftarrow$) Suppose we have a set of configurations $\{C_1, \ldots, C_k\}$ such that every event $e \in E$ belongs to at least one configuration and let e_{v_1}, e_{v_2} be two events of C_i. Events in the same configuration are not in conflict, then $(v_1, v_2) \notin A$ and v_1, v_2 can be colored with the same color. It follows that for every event e_v in C_i the vertex v can be colored by i. We need one color per configuration (only k) and since every event belongs to at least one configuration, every vertex is colored; i.e. G is k-colorable.

■

Fig. 4 shows an example of the reduction from graph coloring to EVENTS-COVER. The graph has a clique (complete maximal subgraph) of size 3 composed by vertices $\{v_1, v_2, v_3\}$ and therefore at least 3 colors are needed. The Figure shows a way to color the vertices using 3 colors and thus it is minimal. We have $C_1 = \{v_1, v_4\}, C_2 = \{v_2, v_5\}, C_3 = \{v_3\}$ and therefore the sets $\{e_1, e_4\}, \{e_2, e_5\}, \{e_3\}$ can be used as the configurations to cover the net.

Example 3. *Consider the program of Fig. 2. Clearly if we represent the program with a regular unfoldings and the naive*

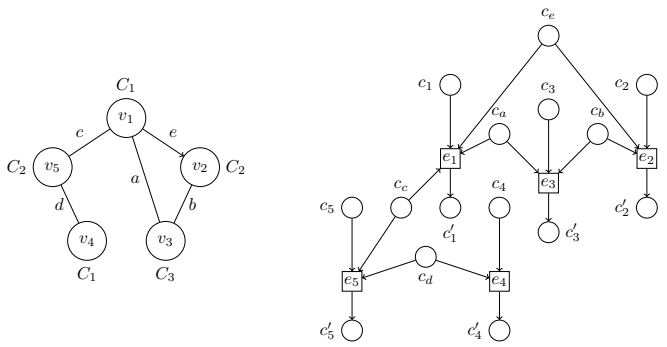


Fig. 4: Reduction of graph coloring to EVENTS-COVER.

approach, this representation would explicitly enumerate all the six possible interleavings accessing the place representing the shared variable; events representing the write operation would be pairwise in conflict and therefore at least 6 executions are needed to cover every event using this representation. A similar analysis can be done in the unfolding of Fig. 2 (a) and at least 4 executions are needed to cover events w_1-w_4. If one considers the contextual representation of the program in Fig. 2 (b), every event can be covered executing, for example, all the reads first ($r_1 \cdot r_3 \cdot w$) and executing the write before any read ($w \cdot r_2 \cdot r_4$). We will see that these are not only lower and upper bounds respectively for the number of executions, but actually the minimal number of test cases needed to cover the unfoldings.

B. SMT-encoding of EVENTS-COVER

This section shows how to encode the EVENTS-COVER problem for regular and contextual unfoldings with SMT based on the encodings of their configurations [12], [13], [14] and additional formulas that capture the path constraints.

In order to cover every event of the unfolding with k configurations, we need to

- find k configurations (this can be done by coping k times the configuration encoding) such that
- every event belongs to at least one configuration

Given an unfolding $\mathcal{U} = (B, E, G)$ and an integer k, we encode the EVENTS-COVER problems using variables $\varphi_{e,i}$ for each event $e \in E$ and $i \leq k$.

The following formula represent causal dependence; for each event e and each $i \leq k$:

$$\varphi_{e,i} \Rightarrow \bigwedge_{e' \in {}^\bullet({}^\bullet e)} \varphi_{e',i} \tag{C1}$$

Since the constraints generated at each branching point are mutually exclusive (one event represents the constraint being true and the other the constraint being false as shown in Fig. 3 (a)), the variables representing inputs of the program need to be renamed. Let g_i be the constraint g where each variable $input$ have been renamed as $input_i$. For each branching event e with a symbolic constraint g and each $i \leq k$ we have:

$$\varphi_{e,i} \Rightarrow g_i \tag{C2}$$

The following constraint encodes conflict-freeness for regular unfoldings; for each condition c, each event $e \in c^\bullet$ and each $i \leq k$:

$$\varphi_{e,i} \Rightarrow \bigwedge_{e' \in c^\bullet \setminus \{e\}} \neg \varphi_{e',i} \tag{C3}$$

Finally each event should be part of at least one configuration; for each event e:

$$\bigvee_{1 \leq i \leq k} \varphi_{e,i} \tag{EC}$$

For regular unfoldings the EVENTS-COVER problem can be encoded as the conjunction of formulas (C1)-(C3),(EC). To extend the encoding for contextual nets, we need to consider read arcs: if a read event is fired, the write event that has most recently updated the value being read must have been fired.

For each read event e and each $i \leq k$ we have:

$$\varphi_{e,i} \Rightarrow \bigwedge_{e' \in {}^\bullet \underline{e}} \varphi_{e',i} \tag{R1}$$

An encoding consisting of formulas (C1)-(C3) and (R1) is an over-approximation of the configurations because the encoding does not take into account possible $\nearrow$-cycles. To accurately capture configurations of a contextual unfolding, additional constraints are needed that make the translation unsatisfiable if a configuration contains a $\nearrow$-cycle. To complete the translation, let $n_{e,i}$ be a natural number associated with event e in configuration $i \leq k$. Intuitively the numbers associated with events describe the order in which they must be fired in they corresponding configurations. The firing order that eliminates $\nearrow$-cycles can then be expressed with the following formulas.

For each event e and each $i \leq k$:

$$\varphi_{e,i} \Rightarrow \bigwedge_{e' \in {}^\bullet({}^\bullet e) \cup {}^\bullet \underline{e}} n_{e',i} < n_{e,i} \tag{R2}$$

For each read event e, write event e' and each $i \leq k$:

$$\varphi_{e,i} \Rightarrow \bigwedge_{{}^\bullet e' \cap \underline{e} \neq \emptyset} n_{e,i} < n_{e',i} \tag{R3}$$

The formulas above have the following meanings: for any event e, all the events that put tokens in its preset and context should be fired before e; and whenever a condition in the preset of a write event is part of the context of a read event, the read should be fired before the write.

When the acyclicity constraints are encoded in SAT [13], [15], the size of the encoding is $\mathcal{O}(n \log n)$ in the best case. However the formulas (R2) and (R3) are linear in the size of the unfolding using the expressivity of SMT.

Example 4. *Example 3 gives lower bounds to the EVENTS-COVER problem using the regular unfoldings of the program in Fig. 2. Using the encoding (C1)-(C3),(EC) for $k = 6$ and*

$k = 4$ *respectively, we obtain that the formulas are satisfiable and then the minimal numbers of executions to cover every event using the naive and place replication approach are respectively 6 and 4. Example 3 also gives a possible solution to cover every event in the contextual representation with two executions. If the encoding (C1)-(C3),(R1)-(R3),(EC) is used with* $k = 1$*, the formula is unsatisfiable and the unfolding cannot be covered with only one execution. We can conclude that the given solutions are optimal.*

C. Statement Coverage

The encoding above shows how to cover the unfolding representation of a multithreaded program. However, different representations give minimal test suites of different size for the same program. If in addition of the unfolding we have information about which statements are covered by each event, we can minimize the test executions not to cover each event, but rather each statement of the program at least once. Suppose we have a mapping $stat$ from the statements of the program to the set of events in its unfolding that execute those statements. A statement j is covered if any of the events in $stat(j)$ is covered by some configuration of the unfolding. Condition (EC) can be replaced by the following formula; for every statement j:

$$\bigvee_{\substack{e \in stat(j) \\ i \leq k}} \varphi_{e,i} \qquad \text{(SC)}$$

The formula above does not require that every event is covered as in the case of EVENTS-COVER, but at least one event should be covered for each statement. In general fewer test executions are needed to cover every statements than to cover every event.

Example 5. *Suppose the following labeling relates every event in Fig. 2 (a) with the statements of the program:*

statement	**events**
$local\ b = x$	r_1, r_2, r_3
$x = 5$	w_1, w_2, w_3, w_4
$local\ c = x$	r_4, r_5, r_6

Using formulas (C1)-(C3),(SC) for $k = 1$ *we obtain the encoding of Fig. 5 which is satisfiable for example for* $\varphi_{w_1}, \varphi_{r_2}, \varphi_{r_5}$ *and thus every statement can be covered with only one test execution. The same result can be obtained using the naive representation of the program and the contextual one.*

The EVENTS-COVER problem is stated for a particular unfolding of the program and allows different minimal test suites depending on the given representation of the program. However, to achieve statement coverage we reason about different ways to interleave the statements of the program. Since every interleaving is represented symbolically in every unfolding, the size of a minimal test suite covering every statement of the program is the same despite its unfolding representation.

Causal clauses:

$$\begin{array}{lcl} \varphi_{w_2} & \Rightarrow & \varphi_{r_3} \wedge \varphi_{r_4} \\ \varphi_{w_3} & \Rightarrow & \varphi_{r_3} \\ \varphi_{w_4} & \Rightarrow & \varphi_{r_4} \\ \varphi_{r_1} & \Rightarrow & \varphi_{w_3} \\ \varphi_{r_2} & \Rightarrow & \varphi_{w_1} \\ \varphi_{r_5} & \Rightarrow & \varphi_{w_1} \\ \varphi_{r_6} & \Rightarrow & \varphi_{w_4} \end{array}$$

Conflict-freeness:

$$\begin{array}{lcl} \varphi_{w_1} & \Rightarrow & \neg\varphi_{r_3} \wedge \neg\varphi_{r_4} \\ \varphi_{w_3} & \Rightarrow & \neg\varphi_{r_4} \\ \varphi_{w_4} & \Rightarrow & \neg\varphi_{r_3} \\ \varphi_{r_3} & \Rightarrow & \neg\varphi_{w_1} \wedge \neg\varphi_{w_4} \\ \varphi_{r_4} & \Rightarrow & \neg\varphi_{w_1} \wedge \neg\varphi_{w_3} \end{array}$$

Statement covering:

$$\varphi_{r_2} \vee \varphi_{r_3} \vee \varphi_{r_6}$$
$$\varphi_{r_1} \vee \varphi_{r_4} \vee \varphi_{r_5}$$
$$\varphi_{w_1} \vee \varphi_{w_2} \vee \varphi_{w_3} \vee \varphi_{w_4}$$

Fig. 5: SMT-encoding for statement covering using the place replication representation.

D. Experiments

We compare the test suites obtained by the testing algorithms of [1] and [2] with the ones obtained by the encodings to execute every event of the unfolding (using place replication and contextual nets) and every statement of the program. The encodings were run with the Z3 SMT-solver [16] using an incremental approach to reuse the information computed by the solver for smaller instances of the problem.

We have conducted the experiments using several benchmarks. Filesystem is used for evaluation of the DPOR algorithm in [17]. Parallel Pi is a program that uses the divide and conquer technique; it divides a task to multiple threads and then merges the results of each computation. The synthetic benchmark performs arbitrarily generated sequences of operations. Dining implements the dining philosophers problem. The Fib benchmark is from the 1st International Competition of Software Verification (SV-COMP); it has been modified to bound the times some loops are executed. For benchmarks that have multiple versions, the versions are similar but involve more threads or increase in complexity.

The result of our experiments are summarized in Table I. As the number of test executions performed by the algorithms in [1] and [2] can vary depending on the order in which the execution paths are explored, the experiments were repeated 10 times and the average results are reported.

The statements of the programs can be covered with less than two executions; since the number of executions is small, the solver does not consume much computational time to find a solution. The number of obtained executions is the same using the regular and contextual representation for the program, however the time results given in the table are those obtained with the contextual unfolding.

For the Filesystem benchmarks, every event in both unfoldings can be covered with just two executions showing that the result of the algorithms in [1] and [2] are close to the optimal. For the rest of the benchmarks, the number of executions grows and the encodings does not scale; the table shows the biggest instance of k for which the solver found a

TABLE I: Experimental results.

Benchmark	Statement coverage		Contextual unfolding		Event coverage		PR unfolding		Event coverage	
	Tests	Time	Tests	Time	Tests	Time	Tests	Time	Tests	Time
Filesystem 1	2	0m 0s	3	0m 0s	2	0m 0s	3	0m 0s	2	0m 0s
Filesystem 2	2	0m 0s	3	0m 0s	2	0m 0s	3	0m 0s	2	0m 0s
Parallel Pi 1	1	0m 0s	24	0m 0s	>11	29m 31s	24	0m 0s	>11	19m 17s
Parallel Pi 2	1	0m 0s	120	0m 0s	>9	2m 37s	120	0m 0s	>10	9m 21s
Parallel Pi 3	1	0m 4s	720	0m 2s	>7	2m 14s	720	0m 2s	>7	0m 53s
Synthetic	2	0m 2s	762	0m 1s	>8	2m 23s	921	0m 2s	>9	29m 12s
Dining	1	0m 1s	798	0m 3s	>8	4m 38s	798	0m 3s	>9	29m 43s
Fib 1	1	0m 25s	4950	0m 17s	>8	15m 44s	19605	0m 3s	>6	5m 35s
Fib 2	1	2m 1s	14546	0m 54s	>5	28m 53s	59908	0m 10s	>3	0m 39s

solution in less than 30 minutes and the corresponding time for obtaining the answer for that instance.

The table shows that the encoding can find solutions with not much computational time for most of the problems when less than 8 test executions are needed. The computational time for $k > 8$ usually grow too fast (see for example Parallel Pi 1); in the case of the Fib benchmark, the encoding approach is slow even for small instances of k since the encoding of the configurations is too big.

V. Minimal Test Suites for Petri Nets

Last sections shows how to cover a multithread program by covering the events of its unfolding representation. The proposed encoding can be used for any finite unfolding and not only those constructed from a multithreaded program. As a corollary of the results of last section, we prove that covering every transition of a safe Petri net with a given number of test executions is NP-hard in the size of a prefix of its unfolding. We study first the problem for acyclic Petri nets and explain the implications of relaxing the acyclicity assumption by adding cut-off events to truncate the unfolding construction.

A. Covering the transitions of an acyclic Petri net

Whenever a safe and acyclic Petri net is unrolled, the obtained unfolding is finite and the EVENTS-COVER problem can be modified to find configurations which may not cover every event, but at least one instance or representative of each transition in the original net.

Definition 2 (TRANSITIONS-COVER). *Given a net $\mathcal{N} = (P, T, F, M_0)$, one unfolding $\mathcal{U}$ and an integer k, decide whether there exists a set $\{C_1, \ldots, C_k\}$ of configurations of $\mathcal{U}$ such that $\bigcup_{i \leq k} l(C_i) = T$.*

Covering all the transitions of a net with a given number of executions is an NP-complete problem, the in NP result is trivial by the same configuration guessing argument as before.

Theorem 3. *TRANSITIONS-COVER is NP-hard.*

Proof: Since the occurrence net for the graph coloring reduction is an acyclic net and coincides with its unfolding, covering all the events of an unfolding $\mathcal{U}$ can be reduced to solve the TRANSITIONS-COVER problem when $\mathcal{U}$ is interpreted both as the net and its unfolding. The result is trivial using this remark. ■

When the Petri net is acyclic, its unfolding is finite and the SMT-encoding proposed in last section can be modified to solve the TRANSITIONS-COVER problem. Since we have information about the original net and every event can be related with its original transition by the labeling l, condition (EC) can be replaced by the following formula; for each transition t in the original net:

$$\bigvee_{\substack{l(e)=t \\ i \leq k}} \varphi_{e,i} \qquad \text{(TC)}$$

The TRANSITIONS-COVER problem can be encoded by the conjunction of (C1)-(C3),(R1)-(R3),(TC) for contextual nets (constraints (R1)-(R3) can be removed for regular Petri nets). The constraint (TC) is weaker than (EC) since it does not state that every event should be part of a configuration, but one representing every transition. This implies that the number of a test executions to cover every transition is usually smaller than the number of executions to cover every event.

B. Adding cut-off events

If we remove the acyclicity restriction from the Petri net, we need to truncate the unfolding to obtain a finite prefix. Cut-off events can be used to obtain finite prefix of the unfolding; cut-off events stop the unfolding algorithm since no new events are added after them. Different notions of cut-off have been proposed in the literature [18], [19], [20] and each of them generate a different prefix of the unfolding. Fig. 6 presents a cyclic Petri net together with two finite prefixes of its unfolding where cut-off events are displayed in grey. If one considers the unfolding (b) to solve the TRANSITIONS-COVER problem, two test executions ($e_1 \cdot e_3$ and e_2) are needed to cover every transition in the original net. However if one consider the unfolding (c), the transitions of the original net can be covered with the single execution $e_1 \cdot e_3 \cdot e_5$. This example shows that the minimality of the test suite does not depend on the net itself, but on the prefix of the unfolding we consider. A weaker notion of cut-off may generate bigger prefixes but those may allow to cover every transition with fewer test executions.

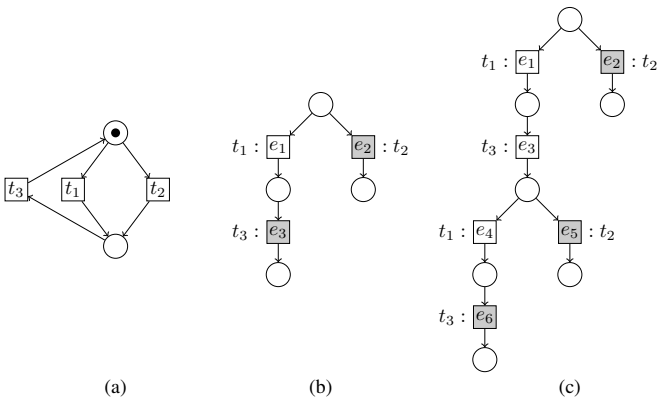


Fig. 6: A Petri net (a) and two prefixes (b) and (c) of its unfolding.

VI. Conclusions and Future Work

In this paper we have shown how to use different unfolding representations to generate test suites of a multithreaded program. We showed that the problem of covering all the events in the unfolding is NP-complete and proposed an SMT-encoding to solve it. Since different unfolding representations of the program generate different minimal test suites, we modified the encoding to cover all the statements of the problem; for such problem the minimality of the test suite does not depend on the chosen modeling. We run several experiments on a set of benchmarks which show that the encodings may not scale to cover every events for programs with a lot of branching, but it does for the statement coverage problem.

We also showed that covering all the transitions of a safe and terminating Petri net is also NP-complete in the size of its unfolding and show that bigger prefixes of the unfolding (when the original Petri net does not terminate) may generate smaller test suites.

In order to remove the acyclicity termination assumption of the program, we need to define a notion to cut-off the unfolding procedure. In addition of the consequences of doing so explained in the last section, our method need to be modified to keep track of global states of the program that have already been visited. Doing this requires keeping track of the local symbolic states of threads and can be very expensive especially when it needs to be determined if a symbolic state subsumes another. Lightweight approaches [9] may therefore be more practical.

Acknowledgment: we would like to thankfully acknowledge the funding by the Academy of Finland projects 139402 and 277522 and the Research Training Group PUMA of the German Research Council.

References

[1] K. Kähkönen, O. Saarikivi, and K. Heljanko, "Using unfoldings in automated testing of multithreaded programs," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, M. Goedicke, T. Menzies, and M. Saeki, Eds. ACM, 2012, pp. 150–159.

[2] K. Kähkönen and K. Heljanko, "Testing multithreaded programs with contextual unfoldings and dynamic symbolic execution," in *14th International Conference on Application of Concurrency to System Design, ACSD 2014*, 2014.

[3] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 213–223.

[4] K. Sen, "Scalable automated methods for dynamic program analysis," Doctoral Thesis, University of Illinois, 2006.

[5] A. Farzan, A. Holzer, N. Razavi, and H. Veith, "Con2colic testing," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 37–47.

[6] V. Diekert and G. Rozenberg, Eds., *The Book of Traces*. World Scientific Publishing Co., Inc., 1995.

[7] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. Lecture Notes in Computer Science. Springer, 1996, vol. 1032.

[8] P. A. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas, "Optimal dynamic partial order reduction," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 373–384.

[9] K. Kähkönen and K. Heljanko, "Lightweight state capturing for automated testing of multithreaded programs," in *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings*, ser. Lecture Notes in Computer Science, M. Seidl and N. Tillmann, Eds., vol. 8570. Springer, 2014, pp. 187–203.

[10] U. Montanari and F. Rossi, "Contextual nets," *Acta Inf.*, vol. 32, no. 6, pp. 545–596, 1995.

[11] A. Farzan and P. Madhusudan, "Causal atomicity," in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 315–328.

[12] J. Esparza and K. Heljanko, *Unfoldings - A Partial-Order Approach to Model Checking*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[13] C. Rodríguez, "Verification based on unfoldings of Petri nets with read arcs," Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, Dec. 2013.

[14] K. Kähkönen, "Automated systematic testing methods for multithreaded programs," Doctoral Dissertation, School of Science, Aalto University, 2015.

[15] V. Khomenko, A. Kondratyev, M. Koutny, and W. Vogler, "Merged processes: a new condensed representation of petri net behaviour," *Acta Inf.*, vol. 43, no. 5, pp. 307–330, 2006.

[16] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[17] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, J. Palsberg and M. Abadi, Eds. ACM, 2005, pp. 110–121.

[18] K. L. McMillan, "A technique of state space search based on unfolding," *Formal Methods in System Design*, vol. 6, no. 1, pp. 45–65, 1995.

[19] J. Esparza, S. Römer, and W. Vogler, "An improvement of McMillan's unfolding algorithm," *Formal Methods in System Design*, vol. 20, no. 3, pp. 285–310, 2002.

[20] V. Khomenko, M. Koutny, and W. Vogler, "Canonical prefixes of petri net unfoldings," in *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, ser. Lecture Notes in Computer Science, E. Brinksma and K. G. Larsen, Eds., vol. 2404. Springer, 2002, pp. 582–595.