Minimizing Test Suites with Unfoldings of Multithreaded Programs

OLLI SAARIKIVI, Aalto University and Helsinki Institute for Information Technology HIIT
HERNÁN PONCE-DE-LEÓN, Aalto University and Helsinki Institute for Information Technology HIIT
KARI KÄHKÖNEN, Aalto University and Helsinki Institute for Information Technology HIIT
KEIJO HELJANKO, Aalto University and Helsinki Institute for Information Technology HIIT
JAVIER ESPARZA, Fakultät für informatik, Technische Universität München

This paper focuses on computing minimal test suites for multithreaded programs. Based on previous work on test case generation for multithreaded programs using unfoldings, this paper shows how this unfolding can be used to generate minimal test suites covering all local states of the program. Generating such minimal test suites is shown to be NP-complete in the size of the unfolding. We propose an SMT-encoding for this problem and two methods based on heuristics which only approximate the solution, but scale better in practice. Finally we apply our methods to compute the minimal test suites for several benchmarks.

CCS Concepts: •**Theory of computation** → **Concurrency;** *Proof complexity;* •**Software and its engineering** → **Software verification and validation;** *Search-based software engineering;*

ACM Reference Format:
Olli Saarikivi, Hernán Ponce-de-León, Kari Kähkönen, Keijo Heljanko and Javier Esparza, 2016. Minimizing Test Suites with Unfoldings of Multithreaded Programs. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 24 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Verification of 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 executing all of them is not practical. There exist at least three well known techniques that have been used to deal with those problems: dynamic symbolic execution (DSE) [Godefroid et al. 2005; Sen 2006], partial order reductions (POR) [Godefroid 1996; Valmari 1996] and unfoldings [McMillan 1995]. DSE handles data values in a symbolic way allowing many concrete inputs values to be covered with a single execution; PORs techniques establish an equivalence relation between executions of the programs and explore a subset of all possible interleavings preserving at least one representative per equivalence class; unfolding-based techniques model executions with partial orders together with a conflict relation to distinguish between different executions of the system.

In this paper we build on our earlier work on verifying terminating multithreaded programs using net unfoldings [Kähkönen et al. 2012; Kähkönen and Heljanko 2014b] which generate a small number of executions that is often even smaller than those

This work is supported by the Academy of Finland project 277522 and the Research Training Group PUMA of the German Research Council.
Author's addresses: Olli Saarikivi, Hernán Ponce-de-León, Kari Kähkönen, Keijo Heljanko: Helsinki Institute for Information Technology HIIT and Department of Computer Science, Aalto University, Helsinki, Finland; Javier Esparza: Fakultät für informatik, Technische Universität München, Germany.

DOI: 0000001.0000001

generated by other partial order reduction methods. The approach also contains DSE, thus it both minimizes the number of interleavings with unfoldings and the number of concrete input values with dynamic symbolic testing. As a side effect this technique also produces a representation of the terminating multithreaded program as a net unfolding which can be exploited for further test suite optimization tasks, as shown in this article. Additionally we show how the same optimization can be applied on event structures representing the unfolding semantics of a multithreaded program.

1.1. Related Work

One popular approach to systematically verify single threaded programs is dynamic symbolic execution [Godefroid et al. 2005; Sen 2006] which allows all execution paths of a program to be covered without explicitly executing all input combinations. The input state is partitioned into equivalence classes triggering the same program behavior and one input is tested per equivalence class. This approach can also be extended to multithreaded programs by using a runtime scheduler that controls the execution of threads [Farzan et al. 2013]; 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 or deadlocks, it is not necessary to explore every possible interleaving of the program because some of its operations 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 [Diekert and Rozenberg 1995]. 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 [Godefroid 1996]. Recently, an improvement to this method have been proposed to explore exactly one execution for each Mazurkiewicz trace [Abdulla et al. 2014].

In Petri net unfoldings [McMillan 1995], each Mazurkiewicz trace of the system is represented by the notion of maximal configuration. The advantages of both POR and unfoldings have been recently combined in [Rodríguez et al. 2015] where the authors present another optimal POR algorithm which traverses an event structure rather than a computation tree as traditional POR approaches. This technique can handle programs with cyclic state space by using cut-off events; however it cannot handle the nondeterminism arising from the program accepting inputs from the environment and reacting to them.

Adding cut-off events when the programs can handle inputs is not a trivial task since the technique either needs for subsumption checks that use constraint solvers or storing complete global states rather than using a symbolic representation; such approaches are considerably heavyweight. A lightweight approach to capture abstract state information was presented in [Kähkönen and Heljanko 2014a] where states that are observed during the executions are modeled as a Petri net. This model is then used to determine if some execution paths lead to an already explored state. This 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.

Even using techniques such as dynamic symbolic execution or partial order reduction to alleviate the state space 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 to test local properties such as assertion violations. The testing algorithm presented in [Kähkönen et al. 2012] 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 [Kähkönen and Heljanko 2014b] allowing in general a more succinct representation of the execution paths. The algorithm constructs a Petri net unfolding on-the-fly and runs new executions while this unfolding can still be expanded; however it does not execute all possible maximal configurations (i.e. Mazurkiewicz trace) and thus it can generate less executions than even optimal POR at the price of only testing local states.

Another POR technique that allows further reductions than optimal ones (but does not preserve Mazurkiewicz traces) is local first search (LSF) [Niebert et al. 2001]. The technique was originally designed to optimize the search for local properties in transitions systems by characterizing a restricted subset of traces (called prime traces) that need to be explored to check such properties. Since the POR algorithms do not have complete information about the whole state space of the program (this is constructed while the program is being verified), LFS performs an analysis to detect non prime traces as soon as possible to avoid their exploration. This is based on a combinatorial aspect of the independence alphabet of the program. In [Bornot et al. 2002], LFS was combined with cut-offs events, allowing still further reductions, but not sacrificing the completeness of the algorithm.

1.2. Contributions

The techniques from [Kähkönen et al. 2012; Kähkönen and Heljanko 2014b] were implemented in a tool called SEDD which allows further reductions than traditional POR techniques, but still covers all the local states of a multithreaded program. In [Ponce de León et al. 2015] we focused on the question *how far is actually* SEDD *from the optimal number of executions needed to cover every local state?* We showed that minimizing the number of executions to cover the unfolding representation of the program is a NP-complete problem and proposed an SMT-encoding to solver this problem. Unfortunately such encoding seems not to scale for large programs.

This article extends our previous work by using heuristics to minimize the number of executions needed to cover all the local states of the program. We show how this problem can be encoded as two different optimization problems: maximal satisfiability and finding maximal cliques in a graph. Experiments run on a set of benchmarks show that those encodings scale better than the SMT-encoding of our previous work at the price of obtaining an over-approximation rather than an exact solution. Our experiments suggest that in many of the cases, SEDD generates a number of executions that might be optimal. We additionally show how programs can be modeled not only with Petri net unfoldings, but also suing event structures; this allows us to compare our heuristics against a recently proposed Optimal POR technique.

The rest of the article is structured as following: Section 2 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 3 explains how to model a given program with different kinds of nets and event structures; Section 4 states the problem of covering all events of an unfolding and gives solutions based on SMT-encodings; in Section 5 we explain how to obtain approximate solution to this problem using Max-SAT and maximal cliques of a graph; we conclude in Section 6.

2. BACKGROUND

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

$$
\begin{array}{rcll}
Thread & ::= & Stmt* & (thread) \\
Stmt & ::= & lv := e \mid SV := lv \mid SV := c \mid & (statement) \\
 & & \textbf{while}\ (b)\ \{Stmt\} \mid & \\
 & & \textbf{if}\ (b)\ \textbf{then}\ \{Stmt\}\ \textbf{else}\ \{Stmt\} \mid & \\
 & & \textbf{lock}(lc) \mid \textbf{unlock}(lc) \mid \textbf{error} \mid & \\
 & & \textbf{end} \mid Stmt\ ;\ Stmt & \\
e & ::= & lv \mid SV \mid c \mid lv\ op\ lv \mid \textbf{input}() & (expression) \\
b & ::= & \textbf{true} \mid \textbf{false} \mid & (boolean\ expression) \\
 & & lv = lv\ |lv \neq lv\ | & \\
 & & lv < lv \mid lv < lv \mid lv \leq lv\ |lv \geq lv &
\end{array}
$$

lv is a local variable,
SV is a shared variable,
lc is a lock identifier, and
$op \in \{+, -, *, /, mod, \dots\}$,
c is a constant

Table I: Syntax of multithreaded programs.

2.1. The program under test

To simplify the presentation, we introduce a simple multithreaded language with integer-valued variables. The syntax of this language is shown in Table I and can be seen as a subset of imperative programming languages such as C or Java. There are two types of variables in the language: variables local to a thread and shared variables. To differentiate the variable types, we write local variables with lowercase letters and shared variables with capital letters. We assume that a thread can use a shared value only by assigning it first to a local variable. Similarly we assume that a thread can update a shared value only by assigning to it either a constant value or a value from a local variable. Programs written with proper programming languages can be automatically modified to satisfy these assumptions. For example, an if-statement that depends on a value of a shared variable can be replaced with statements that read the value of the shared variable to a temporary local variable and then branch the execution based on the value of the temporary variable.

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 (the **input**() expression in Table I). The termination assumption is made since even if unfoldings can deal with systems containing cyclic state spaces by using the so-called cut-off events, adding then is not a trivial task when the programs can handle inputs since the technique needs either to store complete global states (instead of using symbolic representation) or use subsumptions checks which are very expensive in practice. The non dynamically-created threads assumptions is due to the way programs are represented using regular unfoldings (see Section 3.1) where each thread location is represented by a Petri net place and thus the total number of threads must be known a-priori. This assumption can be dropped is one uses instead the contextual unfolding modeling [Kähkönen and Heljanko 2014b]; in this article we make such an assumption to compare all the proposed ways to model programs. We also assume that the operations accessing shared memory are sequentially consistent; this is a widely used assumption in verification and even if in recent years it has been relaxed (see for example [Abdulla et al. 2015; Abdulla et al. 2016]) the implications of such relaxation in the behavior of the program are beyond the scope of this article.

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 access the local state of a thread and visible operations which access the global state of the program and are the only operations that can directly affect the execution in other threads. 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, may compute new values using integer arithmetics and assign the final value 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. The following notions are central for the rest of the article.

Definition 2.1. A test is a set of input values and a schedule; an execution is a sequence of program operations and a test suite is a set of executions.

Notice that we use the term *test*, but our focus is on verification (a.k.a exhaustive testing) techniques.

2.2. Dynamic symbolic execution

Dynamic symbolic execution (DSE) or concolic testing [Godefroid et al. 2005; Sen 2006] 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 and the 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 it is not necessary to try them all. If an 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 try other branches in another 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 to be explored systematically.

2.3. Petri nets and their unfoldings

Some of the approaches presented in the next section consist of modeling the observable behavior of a multithreaded program with different kinds of net unfoldings. Different ways of modeling programs will be presented 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 $\leq$. The 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 symbolically 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 cyclic) branching process is called the unfolding of a Petri net.

Given an unfolding (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 2.2. Fig. 1 presents two unfolding 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 (since both events need to consume the permission token) while read operations are considered independent in the second unfolding; this is done by replicating the conditions representing the variable x to avoid their preset to intersect; for each variable there is one condition representing it for each thread. Events r_1 and r_4 are causally related ($r_1 < r_4$) since r_1 produces a token consumed by r_4; events r_1 and r_2 are in conflict ($r_1\#r_2$) since both events consume the same token in the condition in the intersection of their presets. Events r_5 and r_6 are neither causally related nor in conflict and are called concurrent (r_5 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 only maximal configuration $\{r_5, r_6\}$ of the second unfolding shows that the operations can be done independently in (b) and thus both possible orderings are linearizations[1] of the configuration.

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 [Montanari and Rossi 1995]. 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

[1] A linearization is a totally ordered extension of the order imposed by $\leq$.

```
Thread 1:         Thread 2:
   b = X;            c = X;
     end               end
```

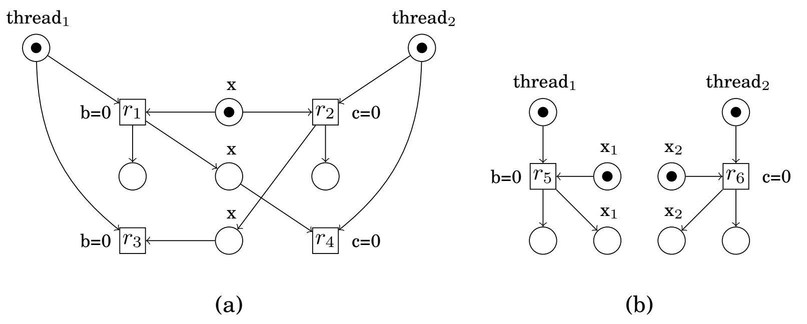


Fig. 1: An example program with its unfolding representation with (a) serialized access to shared variables and (b) place replication.

(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 sets of events.

Example 2.3. 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 (drawn as dashed lines in the figures). 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.

2.4. Event structures

The unfolding of a net is normally represented by an occurrence net, however one can replace conditions by two relations representing the causality and conflict information; this gives rise to event structures which are isomorphic to occurrence nets [Nielsen et al. 1981]. A labeled event structure over an alphabet L is a 4-tuple $\mathcal{E} := (E, \leq, \#, \lambda)$ where *(i)* E is a set of events; *(ii)* $\leq \subseteq E \times E$ is a partial order (representing causality) satisfying the property of finite causes: for all $e \in E$, we have $\lfloor e \rfloor := \{e' \in E \mid e' \leq e\}$ is finite; *(iii)* $\# \subseteq E \times E$ is an irreflexive symmetric relation (representing conflict) satisfying the property of conflict heredity: for all $e, e', e'' \in E$, if $e \# e'$ and $e' \leq e''$ then $e \# e''$; and *(iv)* $\lambda : E \rightarrow L$ is a labeling function. As in the case of occurrence nets, configurations are causally-closed and conflict-free set of events representing executions. The set of configurations of $\mathcal{E}$ is denoted by $\mathcal{C}(\mathcal{E})$. Given two event structures $\mathcal{E} := \langle E, \leq, \#, \lambda \rangle$

```
Thread 1:          Thread 2:          Thread 3:
   b = X;             X = 5;             c = X;
     end                end                end
```

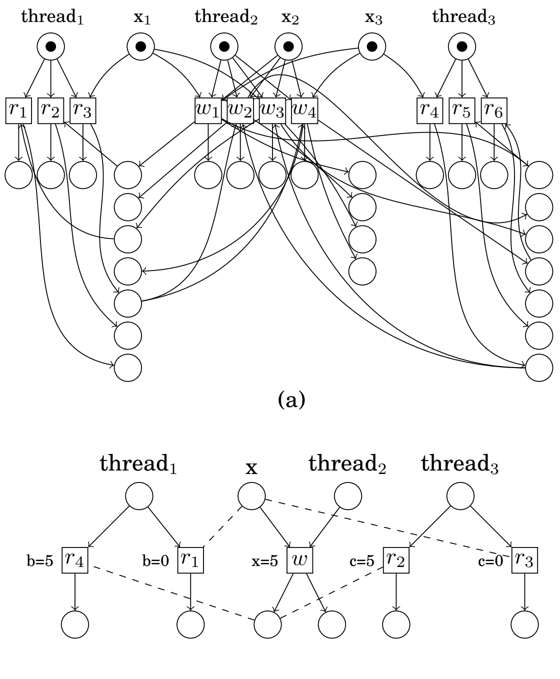


Fig. 2: Regular and contextual unfolding of a program.

and $\mathcal{E}' := \langle E', \leq', \#', \lambda' \rangle$ we say that $\mathcal{E}$ is a prefix of $\mathcal{E}'$, denoted by $\mathcal{E} \preceq \mathcal{E}'$, when $E \subseteq E'$ and $\leq, \#, \lambda$ are projections of $\leq', \#', \lambda'$ over E.

In the same way occurrence nets are isomorphic to event structures, contextual occurrence nets are isomorphic to asymmetric event structures, an extension of event structures where the symmetric conflict is replaced by an asymmetric one [Baldan et al. 2001]. The encodings for contextual nets presented in this article can be easily translated to asymmetric event structures.

Example 2.4. Fig. 3 shows three event structures representing the same behaviors as the occurrence nets of Fig. 1 (a) and (b) and Fig. 2 (a). Direct causalities and direct conflicts[2] are drawn respectively by arrows and dashed lines. Observe that there is a causal relation between two events e_1, e_2 if there is a condition b in the occurrence net such that $b \in e_1{}^\bullet$ and $b \in {}^\bullet e_2$. Similarly, there is a direct conflict in the event structure between e_1 and e_2 if there exists a condition $b \in {}^\bullet e_1 \cap {}^\bullet e_2$ in the occurrence net. The event structure (c) from the right-most part of the figure has three configurations $\{r_3, r_4, w_2\}$, $\{r_4, w_4, r_6\}$, $\{r_3, w_3, r_1\}$ and $\{w_1, r_2, r_5\}$ representing the four non-equivalent ways to interleave the operations of the program in Fig. 2 (the program contains six executions but those with consecutive reads lead to equivalent states).

[2]Direct causality is the transitive reduction of $\leq$ and direct conflict is the smallest relation inducing $\#$ through the property of conflict heredity.

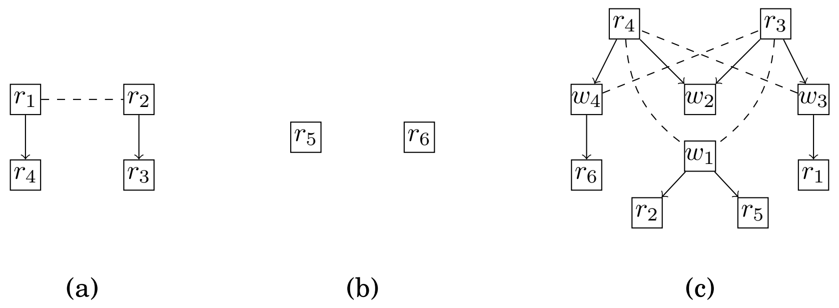


Fig. 3: Event structures.

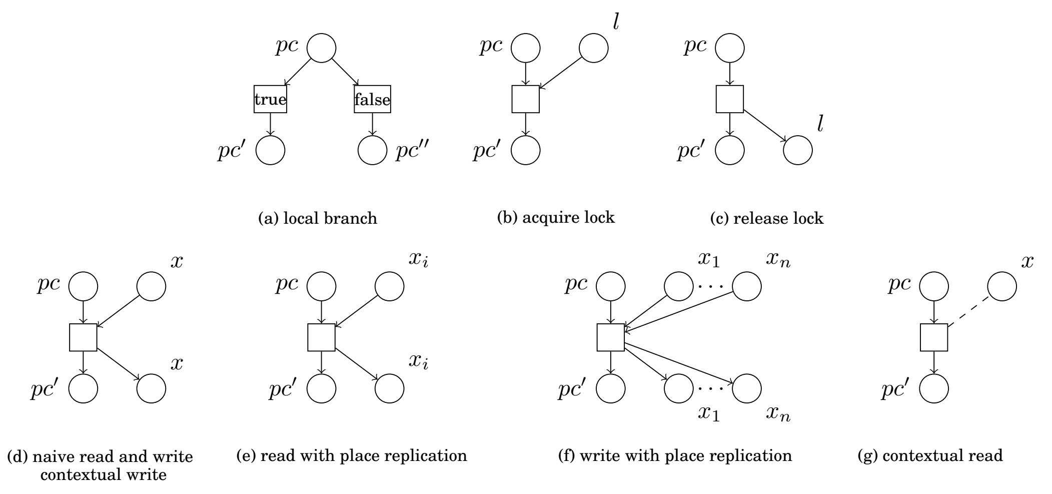
3. MODELING MULTITHREADED PROGRAMS

Most POR techniques 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, unfoldings (either as occurrence nets of event structures) can be used to obtain a more succinct representation in many cases. In the occurrence net representation, shared variables, locks and local states of threads are represented with conditions while operations are represented 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.

POR techniques use an independence relation to define equivalence classes of executions; the independence relation allows to represent executions as partial orders instead of sequences. Since a collection of partial orders (or executions in our setting) can be compactly represented by event structures [Ponce de León and Mokhov 2015], the semantics of a multithreaded program can be given by an event structure where independent operations generate concurrent events and dependent ones generate either causally related or conflicting events. These semantics for multithreaded programs were introduced in [Rodríguez et al. 2015] and are parametric in the independence relation. For example if one sets reads operations to be dependent, Fig. 3 (a) will be generated for the program in Fig. 1; in this event structure, events representing reads are either in conflict or causally dependent. However if one sets reads as independent, Fig. 3 (b) would be obtained where events are concurrent.

We present four different ways to model a program, three using Petri nets and one using event structures: 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 in occurrence nets (both techniques were introduced in [Kähkönen et al. 2012]); the third approach uses contextual nets which may reduce the size of the unfolding by introducing read arcs (this is the technique used in [Kähkönen and Heljanko 2014b]); the final representation (given by [Rodríguez et al. 2015]) uses event structures with events labeled by operation of the program and is parametric on the independence relation over those operations.

For the Petri net representation, 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



pc
true
false
pc′
pc″
(a) local branch
l
(b) acquire lock
(c) release lock
x
x_i
x_1
x_n
(d) naive read and write contextual write
(e) read with place replication
(f) write with place replication
(g) contextual read

Fig. 4: Modeling programs with unfoldings.

and thus thread cannot be dynamically created. We also assume that there is a set of conditions for each lock in the program. The constructs of Fig. 4 can be used to model a program as an occurrence net 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) in Fig. 4 represent symbolic branching of the program depending on inputs and acquiring or releasing locks for any of the Petri net modeling approaches. Sections 3.1 and 3.2 explain how reading from and writing to shared variables can be modeled with different kinds of nets and assumptions. Finally in Section 3.3 we show how the behavior of a multithreaded program can be modeled by an event structure.

3.1. 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. 4 (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 even if they are reads, 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. 4 (e) and (f)). This approach is known as place replication [Farzan and Madhusudan 2006] 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 operations become independent and the two executions of the program, i.e. $r_5 \cdot r_6$ and $r_6 \cdot r_5$, can be obtained as linearizations of its unique configuration $\{r_5, r_6\}$. This unfolding contains only two events instead of four as in the naive case.

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 as shown in [Kähkönen and Heljanko 2014b].

3.2. 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. 4. 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. 4 (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 replaced by a single write event, but the four non-equivalent ways to interleave the executions of the program are still represented by the four configurations of the c-net.

3.3. Modeling programs with event structures

Given a multithreaded program with its operations in T and an independence relation $\diamond \subseteq T \times T$ over the operations (the complement of $\diamond$ is called a dependence relation and defined as $\diamondtimes := (T \times T) \setminus \diamond$), we define a labeled event structure $\mathcal{E}$ where each event represents firing an operation in the program and each execution corresponds to a linearization of a configuration; by abuse of notation we denote by $state(C)$ the state reached after executing the operations related to the events in C (preserving the causal order, if any) and we say that an operation $t \in T$ is enabled at $state(C)$ if the program can perform such operation in the corresponding state.

Each event of $\mathcal{E}$ is identified inductively as $e := \langle t, H \rangle$ where t is an operation of the program (i.e. the label of e over λ) and H a configuration of $\mathcal{E}$; event e represents the occurrence of operation t after the history H. For an operation t and an event structure $\mathcal{E}$, we define the set $\mathcal{H}_t \subseteq \mathcal{C}(\mathcal{E})$ of candidates histories for t as the maximal subset such that if $H \in \mathcal{H}_t$ then

— operation t is enabled at $state(H)$, and
— either $H := \{\bot\}$ or for every $\leq$-maximal event $e \in H$ we have $\lambda(e) \diamondtimes t$.

Once the event e has been added to the event structure we need to check that its operation is not dependent with the operation of some event e' already present in $\mathcal{E}$ which was not in the history of e. Since both operations are dependent, their order is important and we need to prevent them to be part of the same configuration; for this we introduce a conflict between both events. The set of conflicting events for e is denoted by $\mathcal{K}_e$ and contains any event e' such that $e \notin \lfloor e' \rfloor, e' \notin \lfloor e \rfloor$ and $\lambda(e) \diamondtimes \lambda(e')$. Intuitively, they represent dependent operations not happening in the same execution.

Given a program and an independence relation $\diamond$, the set of its finite unfolding prefixes is the smallest set of event structures such that

(1) $\langle \{\bot\}, \emptyset, \emptyset, \lambda \rangle$ with $\lambda(\bot) := \epsilon$ is an unfolding prefix.
(2) Let $\mathcal{E}$ be an unfolding prefix with a history $H \in \mathcal{H}_t$ for some operation t, the event structure resulting from extending $\mathcal{E}$ with a new event $e := \langle t, H \rangle$ and satisfying
 — for all $e' \in H$ we have $e' < e$,

— for all $e' \in \mathcal{K}_e$ we have $e' \# e$, and
— $\lambda(e) := t$
is also an unfolding prefix.

The unfolding of the program is the unique $\preceq$-maximal element in the set of unfolding prefixes under $\diamond$. It has been proven in [Rodríguez et al. 2015] that such unfolding always exists and it is unique. Moreover for every non-empty execution σ of the system there exists an unique configuration C such that σ is an interleaving of C.

In Fig. 3, the event structures (a) and (b) are obtained from the program in Fig. 1 setting the read operations as dependent and independent respectively. The event structure in (c) is the one obtained for the program in Fig. 2 when the reads are considered independent.

3.4. Program unfolding

We have shown how to model a program using regular and contextual occurrence nets or event structures; all of them representing all the possible ways in which operations of the program can be interleaved. Each unfolding represents all the possible executions of the program since any linearization of a configuration (i.e. any total order between its events that respects their causal order) represents an execution of the program. Even if a program has different representations, any 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. If one considers read operations as independent in the program of Fig. 2, this program contains four Mazurkiewicz traces representing the final states

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

These traces correspond to the four maximal configurations of each of the unfoldings:

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

for the regular unfolding in Fig. 2(a) and the event structure in Fig. 3(c), and

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

for the contextual unfolding in Fig. 2(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 algorithm would explore at least four executions paths for this program. In the next section we show how to reduce the number of executions in the program while covering every local state of the threads.

4. MINIMAL TEST SUITES FOR LOCAL REACHABILITY

The goal of this section is, given an unfolding representation of a multithreaded program, compute the minimal test suite covering every event. This is equivalent to covering all local states of the program and can thus be seen as a coverage criteria. Notice also that by the unfolding constructors in Fig. 4, branch decisions are represented by events and thus covering every event guarantees also branching coverage.

We show that the problem of covering every event is NP-complete in the size of the unfolding and propose a solution to compute such test suite using an SMT-encoding. If in addition to the unfolding, 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.

The results of this and the following section are given in terms of regular and contextual occurrence nets; however due to the isomorphism between them and event structures or asymmetric event structures, such results also hold for the modeling of programs presented in Section 3.3.

4.1. 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 executions.

Definition 4.1 (*EVENTS-COVER*). Given an unfolding $\mathcal{U} := (B, E, G, C)$ and an integer k, decide if there exists a set $\{C_1, \dots, C_k\}$ 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 4.2. *EVENTS-COVER is in NP.*

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

— each C_i is a configuration,
— $\bigcup_{i \le k} C_i = E$

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 $e_1 \le e$ and $e_2 \le e'$ 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 4.3. *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 $\mathcal{U} := (B, E, G, C)$ in the following way:

— for 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 $c'_v \in e_v{}^\bullet$, and
— for each edge $a := (v_1, v_2) \in A$ there is a condition c_a in B with $c_a \in {}^\bullet e_{v_1} \cap {}^\bullet e_{v_2}$.

The resulting unfolding has not causal dependencies, i.e. $\le := \emptyset$, 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.

ALGORITHM 1: EVENTS-COVER is in NP

```
Input: An unfolding net $\mathcal{U} := (B, E, G, C)$ and an integer $k$
Output: accept/reject.
Guess a set of sets $\{C_1, \ldots, C_k\}$ where $C_i := \{e^i_1, \ldots e^i_{n_i}\} \subseteq E$ ;
for $i := 1 \ldots k$ do
    for $j := 1 \ldots n_i$ do
        if $\neg CAUSALLY\text{-}CLOSED(e^i_j, C_i)$ then reject;
        ;
        for $l := 1 \ldots n_i$ do
            if $CONFLICT(e^i_j, e^i_l)$ then reject;
            ;
for $e \in E$ do
    $found := False$;
    for $i := 1 \ldots k$ do
        if $e \in C_i$ then $found := True$;
        ;
    if $\neg found$ then
        reject
accept
```

$\Rightarrow$) 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 that 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. 5 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 $\{e_1, e_4\}, \{e_2, e_5\}, \{e_3\}$ are configurations covering the net.

Example 4.4. Consider the program of Fig. 2. Clearly if we represent the program with a regular unfolding and the naive 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

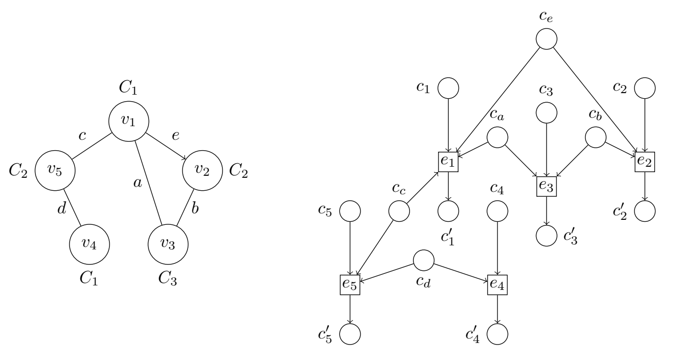


Fig. 5: Reduction of graph coloring to EVENTS-COVER.

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 executions needed to cover the unfoldings.

4.2. 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 [Esparza and Heljanko 2008; Rodríguez 2013; Kähkönen 2015] 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 copying k times the configuration encoding) such that every event belongs to at least one configuration.

Given an unfolding $\mathcal{U} := (B, E, G, C)$ 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 represents 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. 4 (a)), the variables representing inputs of the program need to be renamed. Let g_i be the constraint g where each variable $input$ has 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 occurrence nets 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 their 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 [Rodríguez 2013; Khomenko et al. 2006], 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 4.5. Example 4.4 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. Theorem 4.4 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.

4.3. 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. This is due to the difference in the expressive power of each formalism: for example in Fig. 2 (a), every read event must consume from a variable condition and generates a new one which in turns generates the creation of several write events. This does not happen in Fig. 2 (b) since contextual nets allow to model the read of a variable without creating new conditions. The difference in the test suites sizes is due to the fact that the EVENTS-COVER problem is defined in terms of one particular representation (one unfolding) and not in terms of the program itself.

One interesting problem defined directly on program is *what is it the minimal number of executions needed to cover all the statements?* Notice that if in addition of the unfolding we have information about which statements are covered by each event, we can minimize the executions not to cover each event, but rather each statement of the program. 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:

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

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 executions are needed to cover every statements than to cover every event.

Example 4.6. Suppose the following labeling relates every event in Fig. 2 (a) with the statements of the program:

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

Using formulas (C1)-(C3),(SC) for $k = 1$ we obtain the encoding of Fig. 6 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 execution. The same result can be obtained using the naive representation of the program and the contextual one.

Causal clauses: Conflict-freeness: Statement covering:

$$\begin{array}{rcl}
\varphi_{w_2} & \Rightarrow & \varphi_{r_3} \wedge \varphi_{r_4} \\
\varphi_{w_3} & \Rightarrow & \varphi_{r_3} \\
\varphi_{w_4} & \Rightarrow & \varphi_{r_4} \\
\varphi_{r_1} & \Rightarrow & \varphi_{w_3} \\
\varphi_{r_2} & \Rightarrow & \varphi_{w_1} \\
\varphi_{r_5} & \Rightarrow & \varphi_{w_1} \\
\varphi_{r_6} & \Rightarrow & \varphi_{w_4}
\end{array}
\qquad
\begin{array}{rcl}
\varphi_{w_1} & \Rightarrow & \neg\varphi_{r_3} \wedge \neg\varphi_{r_4} \\
\varphi_{w_3} & \Rightarrow & \neg\varphi_{r_4} \\
\varphi_{w_4} & \Rightarrow & \neg\varphi_{r_3} \\
\varphi_{r_3} & \Rightarrow & \neg\varphi_{w_1} \wedge \neg\varphi_{w_4} \\
\varphi_{r_4} & \Rightarrow & \neg\varphi_{w_1} \wedge \neg\varphi_{w_3}
\end{array}
\qquad
\begin{array}{l}
\varphi_{r_2} \vee \varphi_{r_3} \vee \varphi_{r_6} \\
\varphi_{r_1} \vee \varphi_{r_4} \vee \varphi_{r_5} \\
\varphi_{w_1} \vee \varphi_{w_2} \vee \varphi_{w_3} \vee \varphi_{w_4}
\end{array}$$

Fig. 6: SMT-encoding for statement covering using the place replication representation.

Following the same idea, if one considers an occurrence net being the unfolding of a Petri net, there is a mapping between events and transitions (each event is an instance of a transition in the original net). We have proven in [Ponce de León et al. 2015] that covering the executable transitions of any terminating safe Petri net is also NP-complete in the size of its unfolding and we have shown how to encode such a problem into SMT.

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.

4.4. Experiments

We compare the test suites obtained by the SEDD tool [Kähkönen et al. 2012; Kähkönen and Heljanko 2014b] 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 [de Moura and Bjørner 2008] 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 [Flanagan and Godefroid 2005]. 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 II. As the number of executions performed by the algorithms SEDD 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 results of SEDD 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 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 executions are needed. The computational time for $k > 8$ usually grows 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.

The above experiments suggest that even the proposed encoding computes an exact solution to find the minimal test suite covering the unfolding representations of a program, copying the encodings of configuration k times results in general in very large formulas which cannot be easily solved by the solver.

Table II: Experimental results with the SMT-encoding.

Benchmark	Statement coverage		Contextual unfolding		Event coverage		PR unfolding		Event coverage	
	Tests	Time	Execs	Time	Execs	Time	Execs	Time	Execs	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

5. MINIMIZING TEST SUITES BASED ON HEURISTICS

The SMT-encoding presented in the last section gives an exact answer to the question *is it possible to cover all the events with k executions* and thus the optimal number of executions can be implemented by generating encodings for different k in an iterative way. However, as it has been shown by the experiments, this approach does not scale in practice. In this section we propose to use heuristics to find a *preferably small* set of executions that covers the unfolding. We encode this as two different optimization problems using Max-SMT and a maximal clique problem in graphs. These heuristic compute an over-approximation of the solution, but work much better in practice.

5.1. The Max-SMT encoding

Given a Boolean formula (over some theory) ϕ in CNF (conjunctive normal form), the Max-SMT problem consists in finding an assignment to its variables such that it maximizes the number of clauses in ϕ. The formula is split into clauses which are mandatory (or hard) and clauses which are relaxable (or soft). The goal is to find an assignment that satisfies all the hard constraints while maximizes the number of soft clauses; variations exist where the soft constraints are weighted and the goal is to maximize the weight of the satisfied soft constraints.

We propose to find, in an iterative way, configurations that maximize the number of events that have not yet been covered by any configuration. In Section 4.2 we showed how to encode a configuration into SMT using constraints (C1)-(C3) and (R1)-(R3). Those are the constraints that are mandatory (since they represent the fact that we need to find a configuration) and are thus marked as hard. On each iteration the number of covered events that have not been covered before is maximized by adding a soft constraint for each event that has not been covered by any previous configuration. In other words we mark the still-uncovered subset of (EC) as soft.

The approach is summarized by Algorithm 2: it takes as an input an unfolding and returns a number of sufficient configurations to cover all the events. The set U keeps track of the events that still need to be covered. As an optimization it is initialized to the set of maximal events $max_{\leq}(E)$. On each iteration we create a new formula where the configuration encoding is marked as hard and soft constraints are added for each event still in U. Additionally, we add a hard constraint for one event e_{force} in U that forces it to be in the cover. Adding such a constraint can not make the formula unsatisfiable, since for any event e the configuration $\lfloor e \rfloor$ can cover it. This constraint greatly speeded up some of our larger benchmarks. On the other hand it does make the algorithm slightly less greedy, since the maximal cover on the current iteration might not have included the forced event.

ALGORITHM 2: Minimizing Test Suites with Max-SMT

```
Input: An unfolding net $\mathcal{U} := (B, E, G, C)$
Output: a number $k$ of executions which is enough to cover $E$
$k := 0$;
$U := max_{\leq}(E)$;
while $\exists e_{force} \in U$ do
    $\phi := \varphi_{e_{force}}$ ;
    for $e \in E$ do
        $\phi \;:=\; \phi \;\wedge\; HARD(\varphi_e \Rightarrow \bigwedge_{e' \in {}^\bullet({}^\bullet e)} \varphi_{e'}) \quad \wedge\; HARD(\varphi_e \Rightarrow g_e)$
            $\wedge\; HARD(\varphi_e \Rightarrow \bigwedge_{e' \in {}^\bullet \underline{e}} \varphi_{e'}) \quad \wedge\; HARD(\varphi_e \Rightarrow \bigwedge_{e' \in {}^\bullet({}^\bullet e) \cup {}^\bullet \underline{e}} n_{e'} < n_e)$
            $\wedge\; HARD(\varphi_e \Rightarrow \bigwedge_{{}^\bullet e' \cap \underline{e} \neq \emptyset} n_e < n_{e'})$;
        for $c \in {}^\bullet e$ do
            $\phi := \phi \wedge HARD(\varphi_e \Rightarrow \bigwedge_{e' \in c^\bullet \setminus \{e\}} \neg\varphi_{e'})$;
    for $e \in U$ do
        $\phi := \phi \wedge SOFT(\varphi_e)$;
    assert $SOLVE(\phi) == SAT$;
    for $\varphi_e \in MODEL(\phi)$ do
        $U := U \setminus \{e\}$ ;
    $k := k + 1$ ;
return $k$
```

We use the notation $HARD(\phi)$ and $SOFT(\phi)$ to instruct the Max-SAT solver that the constraint ϕ is hard and soft respectively. One can use an off-the-shelf Max-SMT solver to check if the formula is satisfiable ($SOLVE(\phi) == SAT$). Finally, each event such that its corresponding variable is true in the model ($\varphi_e \in MODEL(\phi)$) is removed from U, the number of used configurations is increased and if U is still non-empty a new iteration is started.

5.2. The Max-Clique Encoding

In graph theory, the maximal clique problem consists in finding a maximal subset of nodes that are pair-wise adjacent. This problem arises in several real-word settings such as social networks or bioinformatics. Even if the decision problem is NP-complete and even hard to approximate, there exists several algorithms that work very well in practice. We show now how to minimize the number of executions to cover the unfolding representation of a program; as in the case of Section 5.1, we try to maximize the number of events covered by a single configuration and use an iterative approach until all the events are covered.

Given an unfolding $\mathcal{U} := (B, E, G, C)$, we construct a graph $G := (V, A)$ where $V := max_{\leq}(E)$, and $(e_1, e_2) \in A$ iff e_1 **co** e_2, i.e. the graph contains one node for each maximal event and there is an arc between two nodes if their corresponding events are concurrent. Since we only consider maximal events, they can be either concurrent or in conflict (if they would be causally dependent, one of them would not be maximal). By finding a maximal clique in G, we obtain a maximal set of pairwise concurrent maximal events. We can remove the events in the clique from V and repeat the procedure

ALGORITHM 3: Minimizing Test Suites with Max-Cliques

```
Input: An unfolding net $\mathcal{U} := (B, E, G, C)$
Output: a number $k$ of executions which is enough to cover $E$
$k := 0$;
$V := max_{\leq}(E)$ ;
while $V \neq \emptyset$ do
    $A := \{(e_1, e_2) \in V \times V \mid e_1$ co $e_2\}$ ;
    Find a maximal clique $C$ in $(V, A)$ ;
    $V := V \setminus C$ ;
    $k := k + 1$ ;
return $k$
```

Table III: Experimental results for greedy event coverage.

Benchmark	Traces	Contextual unfolding			PR unfolding				
		SEDD	Max-SMT		SEDD	Max-SMT		Max-Clique	
		Execs	Execs	Time	Execs	Execs	Time	Execs	Time
Filesystem 1	8	3	**2**	0m 0s	3	**2**	0m 0s	**2**	0m 0s
Filesystem 2	32	3	**2**	0m 0s	3	**2**	0m 0s	**2**	0m 0s
Parallel Pi 1	24	**24**	**24**	0m 0s	**24**	**24**	0m 0s	**24**	0m 0s
Parallel Pi 2	120	**120**	**120**	0m 7s	**120**	**120**	0m 7s	**120**	0m 1s
Parallel Pi 3	720	**720**	**720**	7m 26s	**720**	**720**	9m 30s	**720**	1m 5s
Synthetic	1316	762	**590**	1m 59s	921	**654**	1m 58s	**654**	0m 1s
Dining	831	798	**786**	4m 46s	798	**786**	2m 48s	**786**	0m 12s
Fib 1	19605	4950	**3521**	49m 38s	**19605**	-	>60m	**19605**	17m 23s
Fib 2	59908	**14546**	-	>60m	**59908**	-	>60m	-	>60m

to find another configuration covering the remaining events. Algorithm 3 summarizes the procedure. We use this approach only for the place replication unfolding, since in the contextual unfolding a clique in the pair-wise co-relation might have a $\nearrow$-cycle and thus it might not correspond to a configuration.

5.3. Experiments.

We evaluate the test suite minimization algorithms from above on the set of benchmarks used in Section 4.4 and benchmarks coming from [Rodríguez et al. 2015]. We have implemented the Max-SMT encoding (Algorithm 2) using the Z3 solver [de Moura and Bjørner 2008] and the Max-Clique encoding (Algorithm 3) using the Cliquer tool [Niskanen and Östergård 2002; Östergård 2002]. For the Max-Clique encoding we additionally detect when the maximum clique size is at most two and switch over to a simpler routine that on each iteration simply selects any pair of adjacent nodes (if any) or an arbitrary node. Table III compares the test suites generated using the heuristic approaches with those from SEDD. The entries with "-" indicate that the running time exceeded one hour, at which point we terminated the execution.

When using contextual unfoldings, for the Filesystem, Synthetic, Dining and Fib1 benchmarks the Max-SAT approach was able to improve upon the test suites from SEDD, which gives a new upper bound on the size of the minimal test suite. For example, now we can bound the size of the minimal test suite for Synthetic (with a contextual unfolding) to $8 < opt \leq 590$. The results for Filesystem are optimal as verified by the results in Table II. For the Parallel Pi benchmark, no conclusion can be made since Max-SAT does not compute the minimal number of executions but instead an over-approximation. However this shows that in practice SEDD work as good as optimization techniques that know a priori the structure of the whole unfolding.

For the place replication unfolding we can compare the solutions of the Max-Clique and the Max-SMT encodings. The Max-Clique encoding has lower or equal running time on all benchmarks and was able to solve the Fib 1 benchmark, on which the Max-SMT encoding exceeded our time limits. A distinguishing feature of the Fib 1 benchmark is that the maximum clique size is two, which results in only one call to the Cliquer tool being made (to find one maximum clique), while the other cliques are found using the simpler procedure for finding cliques of size at most two. In our experiments this greatly contributed to the low running time of the Max-Clique approach on the Fib 1 benchmark. In the Max-Clique approach, the construction of the co-relation dominates the running time: for example for Fib 1 constructing the co-relation took 14m 24s. Notice that for all the cases were Max-SMT and Max-Clique found a solution, both solutions coincide. No final conclusion can be made from this but since the three techniques (SEDD and both optimization methods) coincides, this may suggest that the approximate solutions are in fact optimal.

To evaluate the Max-SMT encoding on event structures, we modified Algorithm 2 so that instead of encoding the configurations of an occurence net, we encode those of an event structure. For an event structure $\mathcal{E} := (E, \leq, \#, \lambda)$ the following constraints are used:

$$\bigwedge_{e \leq e'} \varphi_{e'} \Rightarrow \varphi_e \tag{ES1}$$

$$\bigwedge_{e \# e'} \neg(\varphi_e \wedge \varphi_{e'}) \tag{ES2}$$

For all e, variable φ_e corresponds to the event e being fired. (ES1) encodes causality, while (ES2) encodes conflicts. Apart from this change the algorithm remains the same.

Table IV compares the test suites generated by the POET tool [Rodríguez et al. 2015] (one execution per Mazurkiewicz trace) and the Max-SMT encoding on event structures. For the CCNF family of programs the Max-SMT encoding is able to reduce the test suite to two (which is actually the optimal) regardless of the number of threads involved. This is due to the fact that the CCNF programs are composed of sets of non-communicating threads, each of which can be covered with two executions. In contrast, POET sees an exponential blowup as the number of threads grows due to the fact that it is exploring Mazurkiewicz traces. This highlights how for some programs minimizing test suites such that only local properties are maintained can result in an exponential reduction in the size of the test suite. The authors are currently working in a method that performs LFS on the unfolding semantics of the program (i.e. an event structure); initial results show an exponential reduction for this family of examples, although optimality is not yet reached.

For the other programs, the Max-SMT encoding does not result in reduction since the programs are such that the main thread joins with all other threads at the end of the program. This communication results in the event structure having a separate maximal event for each Mazurkiewicz trace. The ProdCons(2) program is an exception, with Max-SMT achieving a reduction of one execution due to the fact that this event structure is generated using cut-off events and thus some event representing the joins mentioned above is not modeled.

Table IV: Experimental results for covering event structures.

Benchmark	POET	Max-SMT	
		Execs	Time
CCNF(9)	16	**2**	0s
CCNF(17)	256	**2**	0s
CCNF(19)	512	**2**	0s
Peterson	**20**	**20**	0s
PGSQL	**4**	**4**	0s
ProdCons	**386**	**386**	8s
ProdCons(2)	15	**14**	0s
Spin08	**84**	**84**	1s
SSB	**4**	**4**	0s
SSB(1)	**23**	**23**	0s
SSB(3)	**90**	**90**	1s
SSB(4)	**142**	**142**	2s
STF	**6**	**6**	0s
Szymanski	**159**	**159**	2s

6. CONCLUSIONS AND FUTURE WORK

In this paper we show how to use different unfolding representations to generate test suites for multithreaded programs. We show that the problem of covering all the events in the unfolding is NP-complete, and present an associated SMT-encoding for finding a minimal test suite. Additionally we presented a modified encoding to cover all the statements in the unfolded program. This decouples the minimality of a test suite from the way the program is modeled. We run several experiments on a set of benchmarks which show that the encodings for covering all events does not scale to program with larger unfoldings, but the encoding for statement coverage do.

In light of the apparent infeasibility of finding minimal event covers, we present two heuristic approaches to minimizing test suites: an approach using a Max-SMT encoding that can handle all types of unfoldings, and an approach using Max-Clique that is faster, but is only suitable for the place replication unfolding. We describe optimizations for both approaches. We evaluate the approaches on our set of benchmarks and achieve minimization for several programs. In particular, for the Synthetic program the heuristic approaches achieved a significant reduction, while the encoding for minimality timed out with a very low lower bound. We further evaluate the approaches on a set of event structures from the POET tool [Rodríguez et al. 2015] and show that preserving only local properties (such as assertion errors) can allow exponentially smaller test suites.

Future work includes a technique that applies local first search to programs modeled with event structures; this technique does not preserve Mazurkiewicz traces, but it can have up to exponential reductions in the number of executions for local properties. Another interesting topic is adding cut-off events in the modeling of non-terminating program which can handle inputs from the environment.

REFERENCES

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Vol. 9035. Springer, 353–367.

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. 2014. 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*. ACM, 373–384.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. Stateless Model Checking for POWER. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 9780. Springer, 134–156.

Paolo Baldan, Andrea Corradini, and Ugo Montanari. 2001. Contextual Petri Nets, Asymmetric Event Structures, and Processes. *Inf. Comput.* 171, 1 (2001), 1–49.

Sébastien Bornot, Rémi Morin, Peter Niebert, and Sarah Zennou. 2002. Black Box Unfolding with Local First Search. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS'02, Proceedings (Lecture Notes in Computer Science)*, Vol. 2280. Springer, 386–400.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. 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 (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340.

Volker Diekert and Grzegorz Rozenberg (Eds.). 1995. *The Book of Traces*. World Scientific Publishing Co., Inc.

Javier Esparza and Keijo Heljanko. 2008. *Unfoldings - A Partial-Order Approach to Model Checking*. Springer.

Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. 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*. ACM, 37–47.

Azadeh Farzan and P. Madhusudan. 2006. Causal Atomicity. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science)*, Vol. 4144. Springer, 315–328.

Cormac Flanagan and Patrice Godefroid. 2005. 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*. ACM, 110–121.

Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science, Vol. 1032. Springer.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. 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*. ACM, 213–223.

Kari Kähkönen. 2015. *Automated Systematic Testing Methods for Multithreaded Programs*. Doctoral Dissertation. School of Science, Aalto University.

Kari Kähkönen and Keijo Heljanko. 2014a. 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 (Lecture Notes in Computer Science)*, Vol. 8570. Springer, 187–203.

Kari Kähkönen and Keijo Heljanko. 2014b. Testing Multithreaded Programs with Contextual Unfoldings and Dynamic Symbolic Execution. In *14th International Conference on Application of Concurrency to System Design, ACSD 2014*. IEEE Computer Society, 142–151.

Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. 2012. 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*. ACM, 150–159.

Victor Khomenko, Alex Kondratyev, Maciej Koutny, and Walter Vogler. 2006. Merged processes: a new condensed representation of Petri net behaviour. *Acta Inf.* 43, 5 (2006), 307–330.

Kenneth L. McMillan. 1995. A Technique of State Space Search Based on Unfolding. *Formal Methods in System Design* 6, 1 (1995), 45–65.

Ugo Montanari and Francesca Rossi. 1995. Contextual Nets. *Acta Inf.* 32, 6 (1995), 545–596.

Peter Niebert, Michaela Huhn, Sarah Zennou, and Denis Lugiez. 2001. Local First Search - A New Paradigm for Partial Order Reductions. In *12th International Conference on Concurrency Theory, CONCUR'01, Proceedings (Lecture Notes in Computer Science)*, Vol. 2154. Springer, 396–410.

Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. 1981. Petri Nets, Event Structures and Domains, Part I. *Theoretical Computer Science* 13 (1981), 85–108.

Sampo Niskanen and Patric Östergård. 2002. Cliquer - routines for clique searching. (2002). http://users.aalto.fi/~pat/cliquer.html

Patric Östergård. 2002. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics* 120, 1-3 (2002), 197–207.

Hernán Ponce de León and Andrey Mokhov. 2015. Building Bridges Between Sets of Partial Orders. In *Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings (Lecture Notes in Computer Science)*, Vol. 8977. Springer, 145–160.

Hernán Ponce de León, Olli Saarikivi, Kari Kähkönen, Keijo Heljanko, and Javier Esparza. 2015. Unfolding Based Minimal Test Suites for Testing Multithreaded Programs. In *15th International Conference on Application of Concurrency to System Design, ACSD 2015, Brussels, Belgium, June 21-26, 2015*. IEEE Computer Society, 40–49.

César Rodríguez. 2013. *Verification Based on Unfoldings of Petri Nets with Read Arcs*. Thèse de doctorat. Laboratoire Spécification et Vérification, ENS Cachan, France.

César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. Unfolding-based Partial Order Reduction. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. 456–469.

Koushik Sen. 2006. *Scalable automated methods for dynamic program analysis*. Doctoral Thesis. University of Illinois.

Antti Valmari. 1996. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*. 429–528.

Received February 2007; revised March 2009; accepted June 2009