Using Unfoldings in Automated Testing of Multithreaded Programs

Kari Kähkönen, Olli Saarikivi, Keijo Heljanko
Department of Information and Computer Science, School of Science, Aalto University
PO Box 15400, FI-00076 AALTO, Finland
{kari.kahkonen, olli.saarikivi, keijo.heljanko}@aalto.fi

ABSTRACT

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

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Testing and Debugging**]: Symbolic execution

General Terms

Verification, Algorithms, Reliability

Keywords

Dynamic symbolic execution, unfoldings

1. INTRODUCTION

Designing correct multithreaded programs is a very challenging task, mostly due to the large number of ways how different threads can interleave their execution. For example, if there are n independent operations being executed concurrently, there are $n!$ possible interleavings. Different interleavings can lead to possibly different states in the program and therefore a programmer needs to make sure that no interleaving leads to an erroneous state. It is, however, easy for the programmer to erroneously miss some of the possible interleavings.

It is easy to generate an execution tree that represents all the possible interleavings of a program. This execution tree can be finite or infinite depending whether there is a cycle in the state space of the program. A large number of the interleavings can be irrelevant for checking properties like the reachability of a control state in the program. This is because quite often some of the operations executed by a thread are independent with operations in other threads and therefore the set of possible interleavings can be partitioned into equivalence classes that are often called Mazurkiewicz traces [3]. A sequence of executed operations is one linearization of a trace and the rest of the linearizations belonging to the same Mazurkiewicz trace can be obtained by swapping adjacent independent operations in the sequence.

As the number of possible interleavings can grow very quickly, ways to fight the state explosion caused by this are needed. Partial order reduction methods (e.g., persistent sets [8]) can be seen as reducing the execution tree representation containing all possible interleavings by ignoring provably irrelevant parts of the tree such that at least one representative from each Mazurkiewicz trace equivalence class gets explored. An alternative way to fight state explosion is to use a "compression approach" by constructing a symbolic representation of the possible interleavings that is more compact than the full execution tree. Unfoldings, first introduced in the context of verification algorithms by McMillan [12], is an example of such a representation (see [4] for an extensive survey on the topic).

In this work we present an unfolding approach to construct a compact representation of the interleavings of multithreaded programs. This allows our new testing algorithm to avoid irrelevant interleavings. In particular, we can sometimes cover all of the local control states of the system using less test runs than there are Mazurkiewicz traces of the system. We will also show how this approach can be combined with dynamic symbolic execution (DSE) to test multithreaded programs that use input values as part of the execution.

Symbolic execution of sequential programs can also be seen as a compression approach. It is easy to consider all possible combinations of input values, however, this becomes quickly infeasible because just two 32-bit integer input val-

ASE '12, September 3-7, 2012, Essen, Germany

ues would generate $(2^{32})^2$ possible combinations. Symbolic execution constructs a symbolic representation that partitions the input values to equivalence classes such that all the possible input values are still presented by the symbolic structure. Given this similarity, we feel that the unfolding approach integrates naturally to DSE in an intuitive way.

It is also possible to combine partial order reduction methods with DSE. Persistent set based dynamic partial order reduction [7] and race detection and flipping [16] are examples of algorithms presented recently that are suitable for this. In this work we compare our approach with these algorithms and discuss the advantages and disadvantages of using a compression approach over reduction methods in testing multithreaded programs. Especially, we show that our algorithm allows better reduction in the number of needed test runs than the current dynamic partial order reduction based approaches. The main contributions are the following: (i) a new algorithm combining dynamic symbolic execution and unfolding methods that allows multithreaded programs containing input values to be tested, (ii) an approach to unfold a multithreaded program using only the information collected while executing the program, and (iii) a comparison of the new approach with dynamic partial-order reduction and race detection and flipping that are similar to algorithms based on partial-order reductions.

The rest of this paper is organized as follows. Section 2 provides the necessary background definitions and concepts used in this work. Section 3 presents the new approach to explore multithreaded programs. A comparison of this approach with related approaches is given in Section 4. An experimental evaluation of the new algorithm is given in Section 5 and Section 6 concludes the paper.

2. BACKGROUND

This section introduces background technical definitions and concepts needed for understanding the new algorithm.

2.1 Basic Definitions

2.1.1 Petri Nets

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

2.1.2 Causality, Conflict and Concurrency

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

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

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

UnfoldNet(Σ: a Petri net)
1: *Unf* := empty net
2: *pe* := `PossibleExtensions`(*Unf*, Σ)
3: **while** *pe* $\neq \emptyset$ **do**
4: add an event $e \in$ *pe* and its output conditions to *Unf*
5: *pe* := `PossibleExtensions`(*Unf*, Σ)
6: **end while**

Figure 1: Basic unfolding algorithm

2.1.3 Occurrence Nets and Branching Processes

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

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

Labeled occurrence net (also called *branching process*) is a tuple $(O, l) = (B, E, G, l)$ where $l : B \cup E \mapsto P \cup T$ is a labeling function such that (adapted from [10]):

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

It is possible to obtain different branching processes by stopping the unfolding process at different times. The maximal branching process (possibly infinite) is called *the unfolding* of a Petri net.

2.2 Constructing Unfoldings

Here we first consider a simple algorithm for unfolding Petri nets, and later adopt it for testing multithreaded programs. The approach we take is based on truncating an

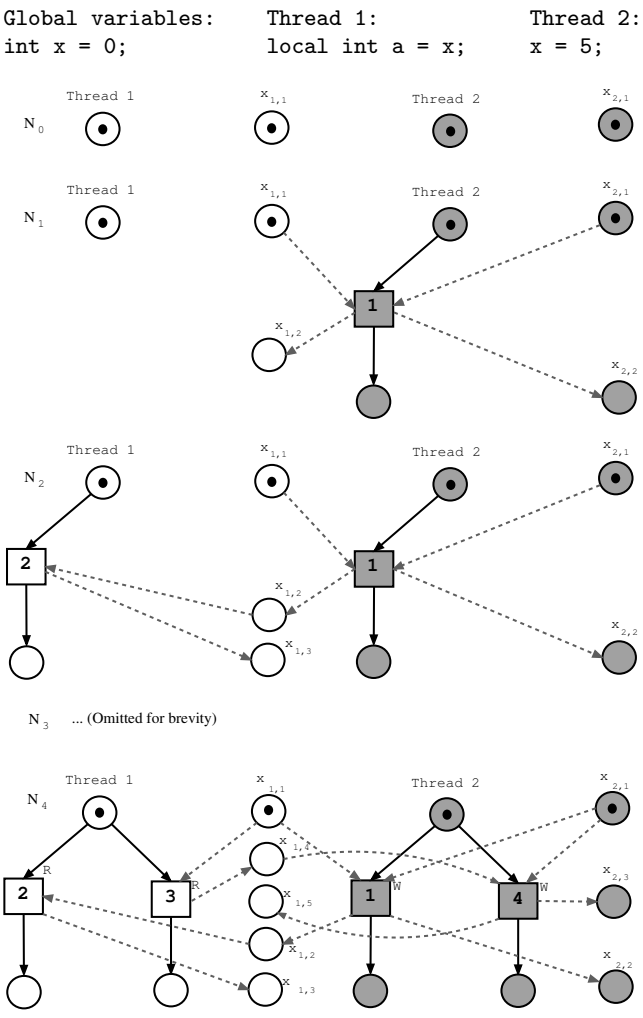

Figure 2: Unfolding of the simple example

infinite unfolding, similar to dynamic symbolic execution approaches that truncate the infinite computation tree to guarantee termination. Using advanced techniques such as cut-off events, see [4], is non-trivial and left for further study.

Algorithm in Fig. 1 shows the basic approach to generate unfoldings of Petri nets. The algorithm starts with an empty net and in each iteration computes a set of events that can be added to the current copy of the unfolding and extends the unfolding with an event from this set. As explained in [5], to improve the performance of the algorithm it is not necessary to compute the the whole set of possible extension in each iteration but instead update the set with events that are enabled by the last event added to the unfolding. Computing the possible extensions is computationally the most expensive part of unfolding algorithms and in fact a decision version of the problem can be shown to be NP-complete, for a more detailed analysis, see [4].

As a first example of unfolding multithreaded programs, consider the simple program in Fig. 2 with two threads where the first thread simply reads a shared variable and the second thread writes to the shared variable. There are two possible ways to execute this program: either the read operation is executed before the write or the other way around. To unfold this program, we start with an initial set of conditions that contains a condition for the initial program counter values for each thread and a copy for the shared variable x for each thread. This is shown in Fig. 2 as the net N_0. From this initial state the net can be extended by either adding an event corresponding to the read or the write operation. N_1 is the resulting net after adding the write event. Note that the write accesses a copy of the variable x of all threads. For this new net, there are again two possible ways to extend it. Either by a read event from the initial state or by a read event that reads the value written by the write event already added to the net. In N_2 the latter read is added to the net. Finally, the net N_4 is the resulting unfolding of the program after all events have been added. The construction of such unfoldings follows the general idea of the algorithm in Fig. 1 and is formalized in Section 3. Also note that the only race in the program is the race in the initial state between write in event 1 and read in event 3. This can be seen in the final unfolding where the condition $x_{1,1}$ has two outgoing edges such that the following events belong to different threads.

2.3 Dynamic Symbolic Execution

Dynamic symbolic execution [9, 14, 1] (also known as concolic testing) is a method where a given program is executed both concretely and symbolically at the same time in order to explore the different behaviors of the program. The main idea behind this approach is to, at runtime, collect symbolic constraints at each branch point that specify the input values causing the program to take a specific branch. As an example, a program `x = x + 1; if (x > 0);` would generate constraints $input_1 + 1 > 0$ and $input_1 + 1 \leq 0$ at the if-statement given 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. All input values that satisfy a path constraint will explore the same execution path for sequential programs. For multithreaded programs the schedule affects the execution path as well. The nondeterminism caused by the thread interleavings can be handled in dynamic symbolic execution by taking control of the scheduler and considering the schedule as an input to the system. For more details, see e.g., [14].

2.4 Representing Multi-threaded Programs

To formalize our algorithm, we use a simple language to describe the programs that can be tested. To keep the presentation simple, this language does not not contain dynamic thread creation or dynamically varying number of shared variables, instead these are fixed at program start time. Handling these features in the context of Java programs is discussed in Section 3.4. The syntax of the language for describing threads in the programs is shown in Fig. 3 and can be seen as a subset of programming languages such as C, C++ or Java.

We assume that the only nondeterminism in threads is caused by concurrent access of shared variables or by input data from the environment. We also assume that the memory model is sequentially consistent. Operations that access shared variables are called visible operations as they can be used to share information between the threads. The state of a multithreaded program consists of the local state of each of the threads and the shared state consisting of the shared

T	::=	Stmt*	(thread)
Stmt	::=	l: S	(labeled stmt.)
S	::=	lv := e | sv := lv | **if** b goto l' | **lock**(sv) | **unlock**(sv)	(statement)
e	::=	lv | sv | c | lv op lv | input	(expression)
b	::=	**true** | **false** | lv = lv | lv ≠ lv | lv < lv | lv > lv | lv ≥ lv | lv ≤ lv	(boolean expr.)
		op ∈ {+, -, *, /, %, ...}, lv is a local variable, sv is a shared variable and c is a constant	

Figure 3: Simplified language syntax

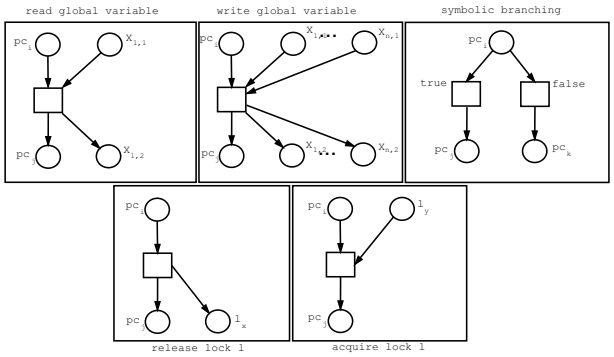

Figure 4: Modeling execution events

variables. The visible operations considered in this work are read and write of shared variables and acquire and release of locks. We assume that a read operation reads a value from a shared variable and assigns it to a variable in the local state of the thread performing the operation. Write assigns either a constant or a value from a local variable to a shared variable. Non-visible operations, such as if-statements, are evaluated solely on the values in the local state of the executing thread and therefore cannot access shared variables directly. In real programs the statements can be modified automatically to satisfy these assumptions.

3. COMBINING DYNAMIC SYMBOLIC EXECUTION AND UNFOLDINGS

In this section we describe how a multithreaded program can be tested using DSE and unfolding techniques without constructing an explicit model of the program (*e.g.*, a very large high-level Petri net) that would be unfolded. Instead, the approach constructs an unfolding of the program executions based solely on the information collected dynamically during test runs.

To build an unfolding to capture the control and data flow of a multithreaded program, we model the visible operations executed during testing with nets shown in Fig. 4. These constructs are the ones employed in [6]. When testing a program with n threads, the unfolding is initially set to contain one condition for each thread and n conditions for each shared variable. The conditions for shared variables

```
UnfoldProgram(P: program)
 1: Unf := initial unfolding
 2: pe := set of events enabled in the initial state of P
 3: while pe ≠ ∅ do
 4:    choose target ∈ pe
 5:    event_sequence := Execute(P, target, k)
 6:    for all e ∈ event_sequence do
 7:       if e ∉ Unf then
 8:          add e and its output conditions to Unf
 9:          pe := pe \ {e}
10:          UpdatePotExt(pe, Unf, e)
11:       end if
12:    end for
13: end while
```

Figure 5: Main loop of the new testing algorithm

can be seen as local copies of the shared variable for each thread (labeled $x_{n,i}$ for thread n in Fig. 4). Having local copies of shared variables allows, for example, two reads of a same variable by different threads to be independent by construction. Each event added to the unfolding has one condition that corresponds to a control location of a thread at preset and another such condition in the postset. Similarly, events corresponding to visible operations have conditions for shared variables in their pre- and postsets. For example, reading a shared variable accesses the local copy of the variable of the reading thread, whereas writing accesses the copies of all threads.

The conditions in Fig. 4 can be divided into three categories. One set of conditions represent the control location (or a program counter value) of a thread, the second set represents shared variables and the third represents locks. To simplify the discussion, we refer to these types of conditions as thread conditions, shared variable conditions and lock conditions, respectively. As any program has a unique next operation for each program counter value, it is easy to see that for each thread condition all the events in the postset of the condition must be of the same type (e.g., if a thread condition has a read event in the post set, all other events in the post set must be read events as well).

When an event is added to the unfolding, we need to compute its possible extensions. By considering the combination of each thread condition and its following event type as a result of unfolding a transition in a Petri net, we could use existing standard Petri net possible extension algorithms in the construction of the unfolding. However, these algorithms are designed for arbitrary Petri nets in mind and are computationally expensive. In our case the structure of the unfolding we want to generate is quite restricted and we will show in the following that these restrictions allow the possible extensions to be computed more efficiently in practice.

3.1 The Algorithm

The new algorithm, shown in Fig. 5, starts by adding events that are enabled in the initial state of the program to a set of possible extensions (denoted as *pe* in line 2 of the algorithm). The algorithm then uses the standard DSE approach where an unvisited location in the execution structure is selected (in this case an event from the set of possible extensions in line 4) and the program is executed with a schedule and input values obtained by solving the path constraints collected during earlier executions to explore the target location. The information obtained from this exe-

cution is then converted into a sequence of events (line 5) and any errors during the execution, such as uncaught exceptions, are reported. This sequence can be seen as a net containing events corresponding to the operations observed during the execution. A net containing only events 1 and 2 and their pre- and postsets in Fig. 2 would be an example of an event sequence. The construction of event sequences is further illustrated in the example at the end of this section.

Notice that the events will not be added to the unfolding when an event sequence is constructed. Instead, the events in the sequence are processed one by one in their execution order (the for-loop in line 6). In this loop an event will be added to the unfolding unless it was already added by an earlier execution. When a new event is added, the set of possible extensions is updated (line 10) with new events that could be enabled after firing the event just added. To guarantee termination, the execution depth of each thread is limited by bound k and the test run is terminated if the number of executed operations exceeds this limit.

It should be noted that even if the event sequence is computed in line 5 by re-executing the program, the algorithm works also when the event sequence is obtained by backtracking the execution to the point where the target event is executed and continuing from there. This could be done, for example, by forking the execution at each point where new possible extensions are discovered.

Example 1. To illustrate the algorithm further, consider the program in Fig. 6. The unfolding shown in the figure has been generated by executing first the thread 1 fully, followed by thread 2, in the first test run. The event sequence for this execution corresponds to the events 1, 2, and 3. Notice that the if-statements of the threads do not generate events as they do not depend on input values in this particular execution (i.e., they are not evaluated after the `x = input();` statement has been executed). Adding event 1 to the net does not result in any new possible extensions. However, when event 2 is added, the possible extensions algorithm notices that the write event of x can be performed either before or after the event 1. Therefore two possible extension events are generated that correspond to events 3 and 4. The possible extension corresponding to event 3 will already be explored by the current execution and therefore is taken out of the set of possible extensions when the event 3 is added to the unfolding.

For the second test run, an event from the set of possible extensions is selected as a new test target. In this case the set contains only the event 4. This event can be reached by a schedule that forces the test run to first execute two visible operations of thread 2 (and then allows the test run to follow an arbitrary schedule). With this schedule (after adding the event 5) the test run evaluates the if-statement of thread 1 which now depends on the input value assigned to x by thread 2. Assuming that the test run follows the false branch, an event corresponding to the true branch along with a path constraint is added as a possible extension (event 7). For the final test run the path constraint $input > 0$ is solved and the resulting concrete value is used as the input. This allows the error location of the program to be reached.

If the part of an unfolding that belongs to a single thread is considered in isolation, it can be seen as a symbolic execution tree constructed by DSE together with additional

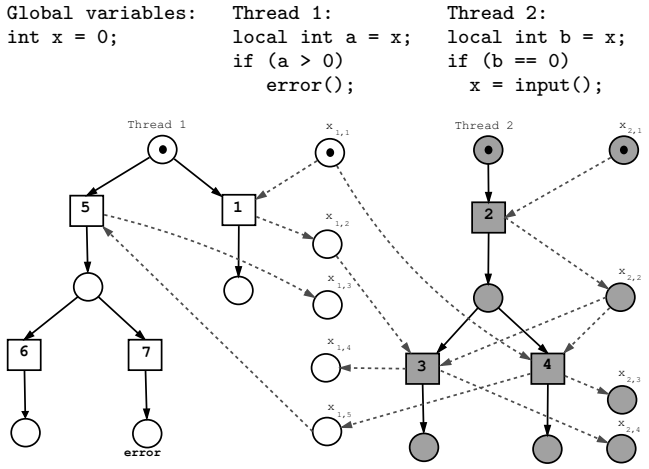

Figure 6: Complete unfolding of the example

branching for scheduling related operations. Note also that if some operation in the program can be reached by multiple execution paths or non-equivalent schedules, the respective events in the unfolding are distinct as the set of events that causally preceed a given event exactly describe an execution path through the control flow graph of the program and a set of schedules belonging to the same Mazurkiewicz trace. We next describe the UpdatePotExt subroutine in Section 3.2 and Execute in Section 3.3.

3.2 Computing Possible Extensions

As discussed before, we could use the standard possible extension algorithms to implement UpdatePotExt. For example, to compute possible extensions for a thread condition that has read as the next operation we coukd check every shared variable condition of the thread executing the read operation to see if it is concurrent with the thread condition. If it is, then there is a read event as a possible extension that has the shared variable condition and the thread condition in its preset. The number of shared variable conditions that need to be considered using this approach can be large and it might be the case that only a small subset of them are actually concurrent with the thread condition in question. In this section we describe properties of the type of unfoldings that are generated by our algorithm that allows us in many cases reduce the number of conditions that need to be considered when computing possible extensions. The approach presented here is inspired by the DPOR algorithm [7] that looks for transitions that are in race and can be seen as being adjacent in some test execution. To make the discussion precise, we use the following definitions.

Definition 1. If a place p in a Petri net is marked initially and every transition that has p in its preset has it also in its postset, the place p is *permanently marked*.

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

Definition 3. Let C_1 and C_2 be co-sets of conditions that represent the same set of places S. The co-sets C_1 and

C_2 are *adjacent* if there exists a sequence of events $e_0...e_n$ and a reachable marking M containing all the conditions in C_1 such that firing the event sequence from M leads to a marking where all conditions in C_2 are marked and there is no prefix of the event sequence $e_0...e_{i<n}$ leading to a marking that contains a co-set representing S.

Example 2. Let us consider the co-sets $a = \{x_{1,2}, x_{2,2}\}$, $b = \{x_{1,1}, x_{2,1}\}$ and $c = \{x_{1,2}, x_{2,1}\}$ in Fig. 6. The sets a and b are not adjacent as there are two events connecting them. However, b is adjacent with c and c is adjacent with a.

Definition 4. Two co-sets C_1 and C_2 that represent the same set of places are *alternatives* if there exists a co-set C_3 that is adjacent to both C_1 and C_2 and there exists conditions $a \in C_1$, $b \in C_2$ and $x \in C_3$ such that $x < a$ and $x < b$.

LEMMA 1. *Let c be a condition in an unfolding and S be a set of initially marked places. Let G be a graph constructed as follows: there is a vertex in G for every co-set representing S such that all conditions in the co-set are concurrent with c and there is an edge between two vertices if the respective co-sets are either adjacent or alternatives. The graph G is connected.*

PROOF. See Appendix. □

Once we have found one possible extension for some event, Lemma 1 shows that the graph G gives the search space from where the other possible extensions can be found. We can then do a backtrack search in this space to compute the possible extensions. By restricting the search to graph G instead of searching the whole unfolding, the search space for possible extensions is in practice in many cases greatly reduced. In worst case, however, the search spaces in both cases are the same. It will be shown in the following, that one possible extension that acts as the starting point for the search can always be found efficiently.

In order to do a backtrack search in the graph G, we need to be able to compute adjacent and alternative co-sets efficiently. For a co-set C containing only shared variable conditions it is easy to determine the adjacent co-sets as the shared variables are modeled with permanently marked places. Specifically, the event sequences in Definition 3 contain only a single read or write event as firing any such event transforms C to a new co-set representing the same shared variable conditions. All these events can therefore be found in the pre- and postset of the conditions in C. Checking if two conditions are concurrent can be done in linear time in the size of the unfolding. Adjacent lock conditions can be efficiently computed by by maintaining a mapping from each lock event to the set of following unlock events and a mapping from unlock event to the causally preceding lock event. The only case where there are alternative co-sets is when there are multiple execution paths from a lock event to different unlock events. These can again efficiently found by maintaining the mapping described above.

3.2.1 Possible Extensions from Thread Conditions

Computing possible extensions for thread conditions that have either unlock or symbolic branching as their next operation is straightforward. For unlocks there can be only one extension. The handling of symbolic branching is identical to normal dynamic symbolic execution: a branch with a symbolic constraint and its negation are added to the unfolding. For the other cases we need to compute a set of shared variable conditions or lock conditions that are concurrent with the given thread condition such that the set forms the preset of the possible extension event together with the thread condition. Assuming that one such set of conditions is known, we can use Lemma 1 to restrict the number of conditions that need to be considered in order to find the the rest of the sets.

If the next operation of the thread condition is either read or write, then one such set can be found by considering the marking obtained by firing the event sequence that is currently being explored by the algorithm in Fig. 5 up to the point of reaching the thread condition. The shared variable conditions in this this marking are then concurrent with the thread condition and because shared variables are modeled with permanently marked places, there is a marked condition for every shared variable copy in every marking. Therefore, the initial set of conditions needed by Lemma 1 can be directly obtained from the computed marking.

For a thread condition t that has lock as the next operation, it is not necessarily the case that in the marking obtained as above there is a lock condition marked that is concurrent with the thread condition. In this case, there either is no suitable lock condition that is concurrent with the thread condition or one such lock condition can be found either in the preset of the last lock acquire event executed in the event sequence currently being explored or in one of the unlock events that release the lock acquired by this event. This is because the lock conditions in the postset of the unlock events are guaranteed to be concurrent with the thread condition t (i.e., the lock operation following t becomes enabled when the the current lock is released). If such unlock event exists in the unfolding, we have found one suitable lock condition. In the case that no such unlock events yet exist in the unfolding, they will be added to it later when the lock in question is unlocked. In this case, the only condition adjacent or alternate to these future conditions is the lock condition in the preset of the last lock acquire event that has been executed and therefore by Lemma 1 it is the only possible place to find a suitable concurrent lock condition.

3.2.2 Possible Extensions from Shared Variable and Lock Conditions

There can also be possible extensions from shared variable or lock conditions that are not part of the extensions computed from the thread conditions. For this we need to find all thread conditions having a next operation that accesses the shared variable or lock condition and that are concurrent with the conditions C in the postset of the event for which we are computing the possible extensions. For each thread condition t found we can then use Lemma 1 to restrict the search space to find possible extensions that all have t and C as part of their presets. In order to find the thread conditions, there are two types of thread conditions that need to be considered: thread conditions for which no possible extensions have yet been computed and thread conditions with known possible extensions.

By Lemma 1 we know that if a suitable thread condition has known possible extensions, one of the extensions must have a co-set that is alternative or adjacent to the condi-

tions in C. Therefore such thread conditions can be found by searching the events connected to the adjacent or alternative co-sets of C; more specifically the thread conditions preceding these events and checking if they are concurrent with C. As not all possible extensions are necessarily yet added to the unfolding, we need to do this search in a net that is the union of the current unfolding and the currently known possible extensions.

For thread conditions that have no known possible extensions yet, a list of them need to be maintained and each thread condition in the list must be checked in turn to see if it is concurrent with the conditions in C. Notice that the only thread conditions that can be in the list are thread conditions that have lock as their next operation. For all other operation types one possible extension is always known. Therefore the size of the list that need to be checked is small in practice in most cases.

3.3 Computing Schedules and Input Values to Execute Tests

The EXECUTE subroutine performs a single test run to generate an event sequence. For this a schedule and input values need to be computed. This can be done by collecting all the events that causally precede the target event. For each event we maintain the information in which order they were added to the unfolding (*i.e.*, the labeling of events with numbers in our examples) and which thread executed the event. This allows a schedule for a test run to be constructed simply by ordering the collected events in the order they were added and requiring the scheduler to schedule threads in this order. We also store the symbolic constraint information obtained by symbolic execution to these events, the path constraint describing the set of possible input values can be constructed and given to a constraint solver in the same way as in standard dynamic symbolic execution. That is, we take a conjunction of all the symbolic constraints in the collected events. If the path constrain is unsatisfiable, the target event is unreachable and it is removed from the set of possible extensions. A satisfiable path constraint gives input values that together with the computed schedule forces a test run to reach the target event.

3.4 Handling Dynamic Number of Shared Variables and Threads

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

Handling dynamic number of threads is more challenging. The main problem here is that when a thread performs a write operation, the corresponding write event accesses shared variable conditions from all threads in the program.

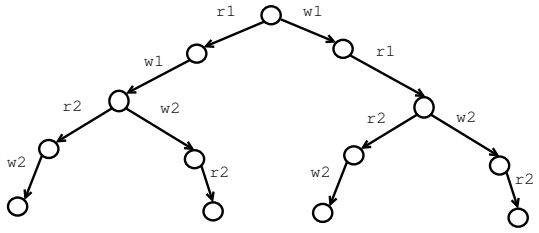

Figure 7: Exponential example

It is not enough to consider only those threads that are running at the time the operation is executed as it is possible that there exists an execution of the program where there are additional threads running and the write affects the behavior of those threads as well. One simple way to address this problem is to update each write operation in the unfolding to access the conditions of a new thread when the thread is added to the unfolding. This, however, is an expensive operation. An alternative way is to model the thread creation with an event that reads all the shared variables of the parent thread and writes the shared variables to the child thread. The problem with this approach is that thread creations can now be considered to be in race with some write operations and this can lead to unnecessary test runs. Currently dynamic thread creation is not supported and finding an efficient solution to the problem is part of future work.

3.5 Further Observations

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

To illustrate this property further, consider a program with $2n$ threads and n shared variables such that for all the shared variables there is one thread that reads the variable and one that writes to it. Therefore there are n pairs of threads that are independent to each other. For testing approaches based on partial-order reduction, the number of needed test executions to cover the program starts to grow exponentially as the number of threads increases. This is illustrated in Fig. 7. With unfoldings the possible extensions from each pair are concurrent and therefore it is possible with to explore the whole program with only two test runs regardless of the number of threads. This can lead to an exponential reduction compared to DPOR-like approaches.

4. RELATED APPROACHES

Dynamic partial-order reduction [7] and race detection and flipping [16] are algorithms that can be combined with dynamic symbolic execution to reduce the number of test

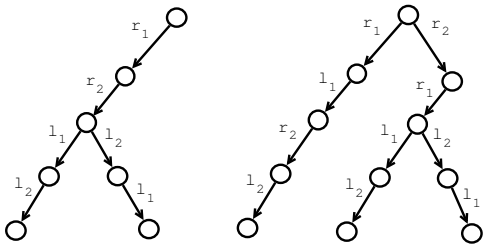

Figure 8: Order example

runs needed to test multithreaded programs. The intuition behind these algorithms is to find transitions that are in race in the current execution and then introduce backtracking points to the execution tree such that the different orderings of the transitions in race will eventually be explored. The main difference between these two algorithms is that DPOR computes persistent sets and race detection and flipping uses postponed sets (*i.e.*, when a race is detected, in another test run the first transition participating in the race is postponed as much as possible, causing the transitions to swap places in the trace). Another difference is that race detection and flipping remembers one operation participating in the race for the next test run and in some cases is able to use this information to avoid unnecessary backtracks.

Both of these algorithms reduce the execution tree of the program in question to avoid exploration of irrelevant interleavings of transitions. However, the obtained reduction depends on the order in which the events are added to the execution tree. In [11] multiple different heuristics are considered for choosing the order in which the transitions are explored and it is shown that the number of execution paths that need to be tested can vary greatly between the heuristics. As a simple example, consider a program with two threads where both threads read a shared variable and then lock and unlock the same lock. Figure 8 shows two possible execution trees that the algorithms explore depending on whether the both reads are added to the tree first or not. The unfolding algorithm presented in this paper does not have this problem. In fact, the new algorithms gives a canonical representation of the program under test regardless of the order in which transitions are explored whereas in dynamic partial-order reduction based approaches the execution trees and the number of required test runs can be different as in the example above.

One additional difference between the partial-order reduction algorithms and the new approach is that the reduction algorithms are guaranteed to find not only safety violations but all deadlocks as well. This is due to the fact that these algorithms explore all Mazurkiewicz traces of a program whereas the new algorithm constructs a representation that contains all the traces but does not necessarily perform a test execution on each of them. This can sometimes result in exponential reduction in the number of test cases. However, the new algorithm needs a separate post processing step with the generated unfolding as an input if all deadlocks (possible global final states) of the program need to be enumerated. There is, to our knowledge, no dynamic partial-order reduction like approach that can further restrict the search space when only local reachability properties of each thread are to be checked.

The operations in partial-order reduction based algorithms are also computationally less expensive than in the new algorithm. Therefore, in cases where both approaches explore the same number of execution paths, the existing algorithms are likely to be faster, but, on the other hand, the new algorithm can significantly reduce the number of execution paths that need to be tested and thus make the overall testing process more efficient especially if runtime of the test runs are long, for example, due to calls to a constraint solver.

5. EXPERIMENTAL EVALUATION

This section provides preliminary experiments to compare our algorithm with DPOR and race detection and flipping.

5.1 Implementation

We have implemented the algorithm described in this paper and the DPOR algorithm extended with a support for commutativity of reads and writes as described in [13] as well as sleep sets in a prototype tool that currently allows testing a small subset of Java programs. Both of these algorithms have been set to select the next test target randomly from the set of all known test targets. The tool uses Soot framework [17] to instrument the program under test with additional code that enables symbolic execution alongside concrete execution and allows the schedule of the threads to be controlled. As a constraint solver the tool uses Z3 [2]. To compare our algorithm with race detection and flipping we also use jCUTE [15], which is a tool that combines race detection and flipping with dynamic symbolic execution to test Java programs. For jCUTE we do not report runtimes as jCUTE restarts JVM for each test run which our tool does not and also uses a different constraint solver. Therefore the runtimes are not directly comparable and the runtimes of jCUTE would be noticeably longer. However, runtime overhead per test execution of race detection and flipping is expected be close to the DPOR implementation used in these experiments.

5.2 Benchmarks

The indexer and file system benchmarks are from [7] where they are used for experimental evaluation of the DPOR algorithm. Both of them are examples of programs where static partial-order reduction algorithms cannot accurately predict the use of the shared memory and therefore conservatively explore more states than dynamic approaches. Parallel pi-algorithm is an example of a typical program where a task is divided to multiple threads and the results of each computation are merged in a block protected by locks. The test selector benchmarks represent an abstracted and faulty implementation of a modification to a small part of our tool that allows multiple test executions to be run concurrently. The pairs program is an artificial example that is similar to the example discussed in Section 3.5. This could be considered as a close to optimal case for the unfolding approach when compared to partial-order reduction approaches. The single lock benchmark is a program where multiple threads access various shared variables in a way that all accesses are protected by the same lock. In this case all the approaches explore the same amount of execution paths (the interleavings of lock operations) and there are no such operations that can cause the DPOR algorithm to explore unnecessary paths due to scheduling choices (as explained in Section 4). The example also forces the unfolding algorithm to make

large number of checks whether two conditions are concurrent. This illustrates a case where the DPOR algorithm has an advantage when compared to the unfolding approach. The synthetic examples are handmade examples where the threads perform an arbitrarily generated sequences of operations, including branching operations on input values. The benchmarks used in the evaluation are available online [1].

5.3 Discussion

Table 1 shows the results of our experimental evaluation. To obtain the results, each benchmark was tested 25 times with both the unfolding algorithm and the DPOR algorithm. For these tests we report the best, worst and median runtime and number of execution paths explored by the algorithms. To get an idea of the memory requirements of the new algorithm, the results also show the number of events in the unfolding that is generated by the algorithm. The size of the pre- and post sets of any event is at most the number of threads plus one. Each event and condition requires a constant amount of memory. The memory requirements for for the lock-to-unlock mappings described in Section 3.2 are negligible in these experiments.

In the file system example there is only one execution path up to using 13 threads. After this the threads start to interact in pairwise manner such that the pairs do not affect each other. For the partial-order reduction based algorithms the number of test cases starts to grow rapidly whereas the new algorithm improves the situation significantly. In the single lock and synthetic-1 benchmarks there are no input variables and therefore no constraint solver calls and DPOR is able to take full advantage of the fact that it is more lightweight than our approach. For the optimal case for unfoldings, our algorithm expectedly performs very well.

Based on these results the unfolding approach typically requires less test executions but is slower per test execution. However, the reduction to the number of executions is in many case enough to make the new algorithm faster than DPOR. In the cases where jCUTE executes the same number of tests as our algorithm, race detection and flipping is expected to be faster than the unfolding algorithm.

6. CONCLUSIONS

We have presented a new approach combining dynamic symbolic execution and unfoldings for testing multithreaded programs. We have shown that this approach can result in some cases even in an exponential reduction to the needed test runs when checking reachability of local states of threads when compared with existing dynamic partial-order reduction based approaches. In particular, we can sometimes cover all of the local states of threads using less test runs than there are Mazurkiewicz traces of the system. The approach also does not need any ordering heuristics to improve the partial-order reduction effectiveness, unlike previous approaches that work on execution trees. We have discussed the advantages and disadvantages of the new testing approach over existing algorithms. Basically our approach is more efficient in the number of test cases but relies on more expensive algorithms (possible extensions calculation) to do so. Unfoldings naturally capture the causality and conflicts of the events in multithreaded programs in a way that allows test cases to be efficiently generated. Given this, we believe that unfolding based techniques are promising alternatives to dynamic partial-order reduction based approaches in the context of automated testing and provides interesting avenues for further research.

[1] `http://users.ics.tkk.fi/ktkahkon/ASE2012`

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS 2006)*, pages 322–335. ACM, 2006.

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

[3] V. Diekert. *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995.

[4] J. Esparza and K. Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer-Verlag, 2008.

[5] J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In J. C. M. Baeten and S. Mauw, editors, *CONCUR*, volume 1664 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 1999.

[6] A. Farzan and P. Madhusudan. Causal atomicity. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 315–328. Springer, 2006.

[7] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In J. Palsberg and M. Abadi, editors, *POPL*, pages 110–121. ACM, 2005.

[8] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[9] 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 (PLDI 2005)*, pages 213–223. ACM, 2005.

[10] V. Khomenko and M. Koutny. Towards an efficient algorithm for unfolding Petri nets. In K. G. Larsen and M. Nielsen, editors, *CONCUR*, volume 2154 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 2001.

[11] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *13th International Conference of Fundamental Approaches to Software Engineering*, pages 308–322, 2010.

[12] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous

Table 1: Comparison of the new algorithm with dynamic partial-order reduction

	Unfolding							DPOR						jCUTE
	number of paths			time (seconds)			size	number of paths			time (seconds)			paths
Benchmark	min	med	max	min	med	max		min	med	max	min	med	max	
indexer (12)	**8**	**8**	**8**	**2**	**2**	**2**	544	51	85	138	6	10	14	**8**
filesystem (14)	**2**	**2**	**2**	**0**	**0**	**0**	130	**2**	3	5	**0**	**0**	**0**	**2**
filesystem (16)	**2**	**3**	**4**	**0**	**0**	**0**	178	9	16	35	1	2	2	31
filesystem (18)	**2**	**4**	**5**	**0**	**0**	**0**	226	49	97	217	4	6	10	2026
parallel π (4)	**24**	**24**	**24**	**1**	**1**	**1**	294	36	95	925	**1**	2	7	**24**
parallel π (5)	**120**	**120**	**120**	**3**	**3**	**3**	1346	882	2698	10086	8	17	53	**120**
test selector (3)	**65**	**65**	**65**	2	**2**	**2**	955	**65**	87	191	**1**	**2**	4	**65**
test selector (4)	**2576**	**2576**	**2576**	**62**	**70**	**76**	33677	5667	8042	13163	71	97	158	**2576**
pairs (6)	**6**	**7**	**10**	**0**	**0**	**2**	228	512	512	512	7	8	10	580
single lock (4)	**2520**	**2520**	**2520**	40	42	44	36834	**2520**	**2520**	**2520**	**13**	**13**	**14**	**2520**
synthetic-1 (3)	**984**	**984**	**984**	15	15	15	3888	2562	3716	5624	**7**	**10**	**13**	2430
synthetic-2 (3)	**1940**	**1943**	**1947**	52	**54**	**55**	7590	6183	7768	10365	**46**	56	72	4860
synthetic-3 (4)	**682**	**682**	**682**	**13**	**14**	**14**	3959	2677	8550	16372	19	52	96	1757

circuits. In G. von Bochmann and D. K. Probst, editors, *CAV*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 1992.

[13] O. Saarikivi, K. Kähkönen, and K. Heljanko. Improving dynamic partial order reductions for concolic testing. In *Proceedings of the 12th International Conference on Application of Concurrency to System Design*, to appear.

[14] K. Sen. *Scalable automated methods for dynamic program analysis*. Doctoral thesis, University of Illinois, 2006.

[15] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006. (Tool Paper).

[16] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2006.

[17] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)*, page 13. IBM, 1999.

APPENDIX

A. PROOF OF LEMMA 1

Let us denote a vertex in G corresponding to a co-set C with $v(C)$ and let us assume that it does not hold that G is connected. This means that there are co-sets A and B such that $v(A)$ and $v(B)$ belong to different subgraphs of G.

Let G' be graph that has a vertex for every co-set representing S and an edge between two vertices if the respective co-sets are adjacent. The vertices of G are then a subset of the vertices in G'. Let $v(I)$ be the vertex in G' such that all the conditions in I are in the initial marking. For any vertex v there exists at least one path in G' from $v(I)$ to v. In the shortest path it holds that for any vertices $v(N)$ and $v(N')$ that are next to each other in the path, any condition $n \in N$ is either the same or causally related to the corresponding condition in N'.

Let us now consider such paths from $v(I)$ to both $v(A)$ and $v(B)$. In one of these paths there must exist a vertex representing a co-set that is not concurrent with c; otherwise A and B would belong to the same subgraph in G. Such co-set contains a condition z that is not concurrent with c and causally precedes the condition $a \in A$ or $b \in B$ representing the same place. If we select A and B such that both paths contain only one vertex corresponding to a co-set concurrent with c then z is causally related to both a and b. There are now three cases how z can be in relation to a and b: (i) $a < z < b$, (ii) $b < z < a$, or (iii) $z < a$ and $z < b$. Note that $z = a$ and $z = b$ cannot hold as a and b are concurrent with c. Because z is causally related to both a and b, it cannot be that z is in conflict with either of them.

Let us consider the first and second cases. Since z is not concurrent with c it holds that either $c < z$, $z < c$ or $c \# z$. These alternatives imply $c < b$, $a < c$ and $c \# b$, respectively, for the first case and $c < a$, $b < c$ and $c \# a$, for the second case. Each of these contradicts the assumption that a and b are concurrent with c. In the third case $c < z$ and $c \# z$ imply $c < a$ and $c \# a$, respectively, leading to a contradiction. We still need to consider the case $z < c$. If there exists a co-set containing z that is adjacent to both A and B, then A and B are alternatives by definition and therefore in the same subgraph in G. Therefore there must be a co-set Y adjacent to either A or B containing a condition z' that represents that same place as z and $z < z'$. We can also assume that in Y there is a condition y that is not concurrent with c. Otherwise the co-set Y would be concurrent with c which is against our selection of A and B to be only such sets in the path. Let us consider the case where the set Y is adjacent to A and let a' and b' be conditions in A and B, respectively, that represent the the same place as y. Since y is not concurrent with c, we know that either $y < c$, $c < y$ or $y \# c$. The first alternative implies that b' is not concurrent with c because it must hold that either $y \# b'$ or $b' < y$. This is in contradiction with the fact that every condition in B is concurrent with c. The second and third alternatives imply $c < a'$ and $a' \# c$, respectively, leading again to a contradiction. The case where Y is adjacent to B is symmetric to the case above and therefore leads to a contradiction. Since all the cases lead to contradictions, it must hold that G is connected.