Improving Dynamic Partial Order Reductions for Concolic Testing

Olli Saarikivi, Kari Kähkönen and Keijo Heljanko
Department of Information and Computer Science
Aalto University
FI-00076 AALTO, FINLAND
{Olli.Saarikivi, Kari.Kahkonen, Keijo.Heljanko}@aalto.fi

***Abstract*—Testing multi-threaded programs is hard due to the state explosion problem arising from the different interleavings of concurrent operations. The dynamic partial order reduction (DPOR) algorithm by Flanagan and Godefroid is one solution to reducing this problem. We present a modification to this algorithm that allows it to exploit the commutativity of read operations and provide further reduction. To enable testing of multi-threaded programs that also take input we show that it is possible to combine DPOR with concolic testing. We have implemented our modified DPOR algorithm in the LCT concolic testing tool. We have also implemented the sleep set algorithm, which can be used along with DPOR to provide further reduction. As the LCT tool was designed for distributed use we have modified the sleep set algorithm for use in a distributed testing client-server setting.**

I. Introduction

Testing programs to find software bugs is typically accomplished by executing the program with different inputs to search for executions that end up in an error state. Often a program has many inputs that result in the same paths through the program's instructions, which can produce test executions that are repeatedly testing the same externally observable behavior of the program over and over again. As exhaustively testing all potential test cases is often impractical even for small programs, various dynamic analysis methods have been devised to prune the search space of the test cases by grouping them into equivalence classes and trying to test only one representative test case from each equivalence class.

For single-threaded programs *concolic testing* [1]–[3] can be used to systematically explore all possible execution paths without testing all input combinations. This is achieved by running the program concretely, that is with real input values, and monitoring it symbolically. The information gathered from the symbolical execution is used to generate new inputs for exploring previously unexplored execution paths.

The inputs for single-threaded programs consist of command line parameters, anything read from the environment, etc. However, for multi-threaded programs the execution may also be affected by the schedule, i.e., the order in which operations from different processes are executed. Exhaustively testing all schedules suffers from the same problems as exhaustive exploration of inputs, namely that there can be too many schedules to test.

In multi-threaded programs some operations might be *independent* meaning that their executions do not affect each other. For example, after executing any two read operations a program will be in the same state regardless of the order in which the operations were executed. *Partial order reduction methods* aim to exploit these independencies to reduce the size of the search space [4]. In this paper we present a modification to the dynamic partial order reduction algorithm of Flanagan and Godefroid [5].

The execution path of a multi-threaded program that takes input is determined by the schedule as well as the input values. The schedule and input values also affect each other: a program may only be multi-threaded on some inputs or a multi-threaded program may read some input in only some of its schedules. To effectively test these programs we have in this work combined concolic testing with dynamic partial order reduction.

The main contributions of this work are as follows:

- A modification to the dynamic partial order reduction algorithm that allows it to exploit the independence of read operations.
- Modifications to the dynamic partial order reduction algorithm and the sleep set algorithm for implementing them in a distributed testing client-server setting.
- The first open source tool combining dynamic partial order reduction with concolic testing.

The sections of this paper are organized as follows. In Section II we describe the dynamic partial order reduction algorithm and present our modification to it. Section III provides a description of how to combine concolic testing with dynamic partial order reduction. In Section IV we present our implementation of the dynamic partial order reduction algorithm and the modified sleep set algorithm. Section V has a brief experimental evaluation of our modified algorithm and we provide some concluding remarks in Section VI.

II. Dynamic Partial Order Reduction

The aim of a partial order reduction algorithm is to exploit the knowledge of causal relationships between operations to avoid exploring redundant executions. In this section we provide an overview of the dynamic partial order reduction (DPOR) algorithm of Flanagan and Godefroid [5] and present a modification to the algorithm. Two versions of the original

algorithm are described in Sections II-B and II-C. In Section II-D we present our modification, which can be viewed as an intermediate between the two previously presented algorithms.

A. Definitions

A concurrent program consists of a finite set P of *processes* or *threads*. Each *process* executes a deterministic program made up of atomic *operations*. The execution of an operation acts on the programs *state*, which is made up of the processes' local states and the program's shared state. The program's state space S is the set of states that are reachable by executing sequences of operations from the program's initial state.

The operations that processes execute are divided into *visible operations* and *invisible operations*. Visible operations are those that operate on the shared state, while invisible operations operate only on the process' local state. Thus visible operations are the only operations that can directly affect the execution of operations in other processes. These include accesses to shared variables and usage of synchronization constructs. We write the set of all visible operations in the program as V. For a process p the next visible operation from the state s is $\mathrm{next}(s, p)$.

An operation is *enabled* in a state if it is the next operation of the process it belongs to and it is not blocked. For example, lock acquires for owned locks are blocked. Invisible operations are never blocked. For a visible operation v that is enabled in the state s we denote the execution of v from s with $s \xrightarrow{v} s'$, where $s' \in S$ is state the program is in after the visible operation v and any subsequent invisible operations from the same process have been applied. Note how this definition groups the execution of any invisible operations together with the previous visible one, thus omitting interleavings of invisible operations from the state space, while still being sufficient for finding all deadlocks and assertion errors in a program [6]. For a sequence of visible operations $w \in V^*$ we write the execution of the visible operations in w as $s \xrightarrow{w} s'$.

Concurrent programs often encounter situations where the order in which some visible operations from different processes are executed does not change the externally observable behavior of the program. For example, operations on different shared variables will always produce the same result regardless of the order in which they are executed. To reason about such operations we define a *dependency relation*. The following definition is adapted from Flanagan and Godefroid [5].

$D \subseteq V \times V$ is a valid dependency relation if and only if for all visible operations $v_1, v_2 \in V$ such that $(v_1, v_2) \notin D$ and states $s, s' \in S$ it satisfies the following properties:

1) If v_1 is enabled in s and $s \xrightarrow{v_1} s'$, then v_2 is enabled in s' if and only if v_2 is enabled in s.
2) If v_1 and v_2 are enabled in s, then there is a unique state s' such that $s \xrightarrow{v_1 v_2} s'$ and $s \xrightarrow{v_2 v_1} s'$.

Two visible operations are *independent* if they are not in the dependency relation. Note that the definition of the dependency relation only limits which visible operations can be considered independent. A dependency relation can be larger than it needs to: for example, $D = V \times V$ is a valid, albeit trivial, dependency relation.

An *execution sequence* is a sequence of visible operations $E = v_1 v_2 \dots v_n \in V^*$ such that given the initial state of the program s_0 the visible operations can be executed as:

$$s_0 \xrightarrow{v_1} s_1 \xrightarrow{v_2} \cdots s_{n-1} \xrightarrow{v_n} s_n$$

A single visible operation v_i is referred to with E_i. We write the extension of E with a new visible operation v as $E.v$. The empty sequence is written as ϵ. To refer to the state a visible operation E_i was executed from we use $\mathrm{pre}(E, i)$. For the state s_n we write $\mathrm{last}(E)$. The process that executed a visible operation E_i is referred to with $\mathrm{proc}(E_i)$.

In an execution sequence any two adjacent independent visible operations can be swapped without changing the final state that sequence leads to. Thus, an execution sequence is part of an equivalence class of execution sequences, which can be obtained from each other by repeatedly swapping adjacent independent visible operations [5]. Each such equivalence class corresponds to a partial order, which we capture with a *happens-before* relation between the indices of an execution sequence E. The happens-before relation $\rightarrow_E$ is the smallest relation such that:

1) If $i \leq j$ and E_i is dependent with E_j then $i \rightarrow_E j$.
2) If $i \rightarrow_E j$ and $j \rightarrow_E k$ then $i \rightarrow_E k$, i.e., the relation is transitive.

Given an execution sequence E and two indices a and b, if it holds that $a \not\rightarrow_E b$ and $b \not\rightarrow_E a$ then the two visible operations E_a and E_b are said to be *concurrent*.

As a shorthand we also define the happens-before relation between an index i of an execution sequence E and a process p as follows. It holds that $i \rightarrow_E p$ if and only if there exists a k such that $i \rightarrow_E k$ and $p = \mathrm{proc}(E_k)$. This means that a visible operation happened before a process if it happened before any visible operation executed by the process.

The following section shows how the happens-before relation can be implemented with *vector clocks*. These are maps from processes to indices in an execution sequence, i.e., functions from the space $VC = P \to \mathbb{N}$. Given two vector clocks $u, v \in VC$ we adopt the following notation:

- The pointwise maximum is $\mathrm{vmax}(u, v) = f(p) = \max(u(p), v(p))$.
- Updating a vector clock with a new value c for some process q is written as:

$$u[q := c] = f(p) = \begin{cases} c & \text{if } p = q, \text{ and} \\ u(p) & \text{otherwise.} \end{cases}$$

- The zero vector clock is written $\bot = f(p) = 0$.

B. Basic DPOR

A basic version of the DPOR algorithm of Flanagan and Godefroid [5] is presented in Figure 1. The algorithm consists of a procedure `Explore(E, CP, CE)`, which receives as its arguments the current execution sequence E and maps to the vector clocks for the processes and the visible operations, CP

```
start: Explore(ϵ, λx.⊥, λx.⊥)
1  procedure Explore(E, CP, CE)
2      s ← last(E)
3      forall the processes p do
4          v ← next(s, p)
5          if ∃i = max({i ∈ {1...|E|} | E_i is dependent
           and may be co-enabled with v and
           i ⋠ CP(p)(proc(E_i))}) then
6              if v ∈ enabled(pre(E, i)) then
7                  add v to backtrack(pre(E, i))
8              else
9                  add enabled(pre(E, i)) to
                   backtrack(pre(E, i))
10             end
11         end
12     end
13     if ∃v_0 ∈ enabled(s) then
14         backtrack(s) ← {v_0} // Initialize the
               backtracking set
15
16         done ← ∅
17         while ∃v_next ∈ (backtrack(s) \ done) do
18             add v_next to done
19             E' ← E.v_next
20             cv ← vmax({CE(i) | i ∈ 1..|E| and E_i
               dependent with v_next})
21             cv ← cv[proc(v_next) := |E'|]
22             CP' ← CP[proc(v_next) := cv]
23             CE' ← CE[|E'| := cv]
24             Explore(E', CP', CE') // Execute
                   the visible operation v_next
25
26         end
27     end
28 end
```

Fig. 1. DPOR using vector clocks

and CE respectively. The same notation is used for updating these maps as is for updating vector clocks. To start the algorithm the procedure `Explore` is called with an empty execution sequence and zeroed vector clocks as the arguments. Each state that the algorithm explores corresponds to one call to `Explore`. The algorithm consists of two parts: (1) on lines 3 – 12 backtracking points are identified and added and (2) on lines 13 – 27 visible operations are explored. The algorithm is presented here in a recursive fashion so that visible operations are executed with calls to `Explore`. In an actual implementation the algorithm might be structured differently. Section IV details one such implementation.

In addition to the execution sequence and vector clocks, which are passed as arguments to `Explore`, the algorithm has access to all the processes and their visible operations in the current and previous states. Also, a global variable *backtrack* associates a *backtracking set* with each state, which is used to store identified backtracking points. For a state s each visible operation in $backtrack(s)$ is one that will eventually be explored from that state.

To identify backtracking points, when a state s is explored the next visible operation $\mathrm{next}(s, p)$ is examined for all processes p. For each such visible operation v the algorithm, on line 5, searches for the last visible operation E_i in the current execution sequence that satisfies the following conditions:

1) E_i is dependent with v
2) E_i may be co-enabled with v
3) $i \not\leq CP(p)(\mathrm{proc}(E_i))$

Condition 1 excludes visible operations that are independent with v and as such do not need to be reordered. Condition 2 filters out visible operations that are never co-enabled with v. For example, if v is an acquire on a lock and some other visible operation r is a release on the same lock then these two visible operations are never co-enabled. In this situation it is safe to ignore r as a candidate backtracking point. Condition 3 ensures using vector clocks that the two visible operations are concurrent [5]. Together these conditions identify a race between v and E_i.

If a visible operation E_i satisfying these conditions exists, then visible operations will be added, on lines 6 – 10, to the backtracking set of its preceding state $\mathrm{pre}(E, i)$. If the visible operation v is enabled in $\mathrm{pre}(E, i)$ then it is added. Otherwise the algorithm defaults to adding all the visible operations that are enabled in $\mathrm{pre}(E, i)$, which can happen for example, when the process p has since the backtracking point executed some other unrelated visible operation before reaching v.

Once backtracking sets for previous states have been updated the algorithm will, on lines 13 – 15, select a single visible operation that is enabled in the current state and initialize the current state's backtracking set with the selected operation. The visible operation may be selected randomly, although the selection may significantly affect the amount of reduction achieved [7]. The algorithm will then proceed to execute visible operations from the backtracking set until no new operations remain. On the first iteration of the loop on lines 17 – 26 the previously selected visible operation is executed. On subsequent iterations the operations executed will be ones added to the backtracking set by some call to `Explore` further in the call stack.

Before executing the selected visible operation v_{next} the algorithm will, on lines 20 – 23, maintain the vector clocks. A new vector clock cv is computed as the maximum of the vector clocks of all previously executed visible operations dependent with v_{next}. Also the vector clock's entry for $\mathrm{proc}(v_{\mathrm{next}})$ is updated to the index of v_{next} in the new execution sequence E'. Then new vector clock maps, CP' for processes and CS' for indices of the execution sequence, are created by updating the old maps to point to the new vector clock cv for both the process and for the top of the new execution sequence.

C. Avoiding Stack Traversals

The algorithm in Figure 1 may use, on lines 5 and 20, time proportional to the size of the execution sequence E. Flanagan

start: `Explore` (ϵ, $\lambda x.\bot$, $\lambda x.\bot$, $\lambda x.0$)

```
1  procedure Explore(E, CP, CO, L)
2      s ← last(E)
3      forall the processes p do
4          v ← next(s, p)
5          i ← L(α(v))
6          if i ≠ 0 and i ⋠ CP(p)(proc(E_i)) then
7              if v ∈ enabled(pre(E, i)) then
8                  add v to backtrack(pre(E, i))
9              else
10                 add enabled(pre(E, i)) to
                   backtrack(pre(E, i))
11             end
12         end
13     end
14     if ∃v_0 ∈ enabled(s) then
15         backtrack(s) ← {v_0}
16         done ← ∅
17         while ∃v_next ∈ (backtrack(s) \ done) do
18             add v_next to done
19             E' ← E.v_next
20             p ← proc(v_next)
21             o ← α(v)
22             cv ← vmax(CP(p), CO(o))[p := |E'|]
23             CP' ← CP[p := cv]
24             CO' ← CO[o := cv]
25             L' ← L[o := |E'|]
26             Explore(E', CP', CO', L')
27         end
28     end
29 end
```

Fig. 2. DPOR without execution sequence traversals

and Godefroid [5] present a modified algorithm, shown in Figure 2, that avoids these traversals of the execution sequence.

As before, the modified algorithms `Explore` procedure takes as parameters the execution sequence E and the processes' vector clocks CP. However, instead of vector clocks for each index of the execution sequence the procedure receives vector clocks, in CO, for each communication object (e.g. a shared variable or a lock) that a previous visible operation has used. Finally, the last parameter L is a map from the communication objects to the index of the last visible operation in the execution sequence that used that communication object.

The switch to per-object vector clocks is justified by assuming that each visible operation v uses exactly one communication object $\alpha(v)$. It is also assumed that two operations are dependent if and only if they use the same communication object. Flanagan and Godefroid [5] show that with these assumptions accesses to a single communication object are totally ordered in the happens-before relation and that it is sufficient to only keep a vector clock $CO(o)$ for the last access to each communication object o. When maintaining these vector clocks, on lines 22 – 24 of Figure 2, the execution sequence no longer needs to be searched for dependent operations.

To avoid the stack traversal on line 5 of Figure 1, for each communication object o the index of the last visible operation in the execution sequence that accessed o is stored in $L(o)$. When identifying a potential backtracking point for the next visible operation v of a process it is sufficient to only check the visible operation at index $L(\alpha(v))$, which by our assumptions must be dependent with v. If the visible operation happens before v then it is the correct backtracking point. There cannot be any other backtracking point as the accesses to $\alpha(v)$ are totally ordered in the happens-before relation [5]. The last accesses are maintained on line 25 of Figure 2.

In Figure 1 potential backtracking points are excluded if they are never co-enabled with the next visible operation of the process under consideration. However, in Figure 2 it is assumed that all visible operations may be co-enabled, which is safe but may result in less reduction. Flanagan and Godefroid [5] present an optimization, omitted here for simplicity, that exploits the fact that an acquire and a release on the same lock are never co-enabled.

D. Exploiting the commutativity of reads

The algorithm in Figure 2 assumes that two visible operations are always dependent if they operate on the same communication object. However, two reads from the same communication object do not interact and can safely be excluded from the dependency relation. Flanagan and Godefroid [5] mention that the algorithm can be modified to exploit this independence by using two vector clocks for each communication object, but do not present the details of such a modification. As one of the main contributions of this work Figure 3 presents our modification to the algorithm.

In this algorithm the map for the per-object vector clocks has been replaced with two maps: the write clocks, CW, and the access clocks, CA. On lines 26 – 40 these and the processes' vector clocks are updated according to the following rules:

- If the visible operation is a write then the process' vector clock $CP(p)$ is updated from the access clock $CA(o)$. For reads the update is made from the write clock $CW(o)$. The process' vector clock entry for itself is updated as normal to point to the index of the visible operation being executed.
- If the visible operation is a write then the write clock is updated to match the process' vector clock.
- The access clock is always updated to match the process' vector clock.

In effect, when a process executes a read it skips the vector clock updates from all the preceding reads up to the last write. For writes this works as before, as on writes the process' vector clock updates from the read clock, which is always updated. This method of updating the vector clocks correctly implements causality with reads and writes [8].

The way backtracking points are identified is adapted to the modified vector clocks and, therefore, a modified happens-

```
start: Explore(ε, λx.⊥, λx.⊥, λx.⊥, λx.0, λx.ε)
1  procedure Explore(E, CP, CW, CA, LW, LR)
2      s ← last(E)
3      forall the processes p do
4          v ← next(s, p)
5          if v may write and ∃k = max({ k | LR(α(v))_k ≰
           CP(p)(proc(E_i)) }) then
6              i ← LR(α(v))_k
7          else
8              i ← LW(α(v))
9          end
10         if i ≠ 0 and i ≰ CP(p)(proc(E_i)) then
11             if v ∈ enabled(pre(E, i)) then
12                 add v to backtrack(pre(E, i))
13             else
14                 add enabled(pre(E, i)) to
                   backtrack(pre(E, i))
15             end
16         end
17     end
18     if ∃v_0 ∈ enabled(s) then
19         backtrack(s) ← {v_0}
20         done ← ∅
21         while ∃v_next ∈ (backtrack(s) \ done) do
22             add v_next to done
23             E' ← E.v_next
24             p ← proc(v_next)
25             o ← α(v)
26             if v may write then
27                 cv ← vmax(CP(p), CA(o))[p := |E'|]
28                 CP' ← CP[p := cv]
29                 CW' ← CW[o := cv]
30                 CA' ← CA[o := cv]
31                 LW' ← LW[o := |E'|]
32                 LR' ← LR[o := ε]
33             else
34                 cv ← vmax(CP(p), CW(o))[p := |E'|]
35                 CP' ← CP[p := cv]
36                 cv ← vmax(cv, CA(o))
37                 CA' ← CA[o := cv]
38                 LW' ← LW
39                 LR' ← LR[o := LR(o).|E'|]
40             end
41             Explore(E', CP', CW', CA', LW', LR')
42         end
43     end
44 end
```

Fig. 3. DPOR exploiting the commutativity of reads

before relation. The last accesses map is replaced with two new maps: (1) the index of the last write is stored in LW and (2) the indices of the last reads since the last write are stored in LR. As LR is a sequence of indices we adopt the same notation for working with it as we use for execution sequences. Both of these are maintained on lines 26 – 40. Accordingly, we modify the way backtracking points are identified on lines 5 – 9. If the current process' p next visible operation v is a write then the sequence of last read operations $LR(\alpha(v))$ is searched to find the last read that did not happen before p. If such a visible operation exists then it is the backtracking point. Otherwise or if v is not a write the backtracking point is $LW(\alpha(v))$, provided it did not happen before p.

We will now argue that like the DPOR algorithm with per-object vector clocks in Figure 2, this modified algorithm is a specialization of the basic DPOR algorithm in Figure 1. For a detailed proof of why DPOR can be used for finding deadlocks and assertion errors refer to Flanagan and Godefroid [5], who show that when a call to `Explore` returns it will have explored a *persistent set* [9] from the corresponding state.

On lines 20 – 23 of Figure 1 the vector clocks are maintained so that the next process' vector clock is updated to be the maximum of the vector clocks of all preceding dependent visible operation. In the algorithm in Figure 3 the maximum of all preceding dependent visible operations is always found in either CW or CA. Specifically, for a read operation the maximum of all preceding dependent visible operation is found in CW, while for a write operation it is found in CA.

In Figure 1 on line 5 to identify a backtracking point for process p the algorithm searches for the last dependent visible operation in the execution sequence that did not happen before $\text{next}(s, p)$. In the modified algorithm all write operations for a single communication object are totally ordered in the happens-before relation. In the pseudocode this can be seen on lines 27 – 30, where the write clock CW is updated from the access clock CA, which in turn aggregates vector clock updates from all visible operations. Therefore, for processes which have a read operation next it is sufficient to consider only the last write. If the last write happened before the process then all preceding writes did too. For processes that have a write operation next any reads since the last write are checked in addition to the last write. Reads preceding the last write do not have to be checked as they necessarily happened before the last write. This property of the happens-before relation in use is reflected on lines 34 – 37, where for read operations the process and the CA vector clock are updated from the read clock CW.

Compared to the algorithm in Figure 2, which requires constant time to find backtracking points for each process, the modified algorithm in Figure 3 does add a potential traversal of the execution sequence on line 5. A trivial worst case for finding a backtracking point is $O(|E|)$. This happens, for example, when a program consists of a single process executing multiple successive reads from a single communication object followed by a write to the same communication object. To find a backtracking point for the process when the write is enabled all preceding reads will be checked. In this case the search can not end early as no suitable backtracking point will be found among the preceding reads. In the best case the algorithm requires constant time to find a backtracking point for a single process. This happens, for example, when only

write operations are executed, which keeps the sequences of last reads in LR empty (on line 32 in Figure 3).

III. Combining Dynamic Partial Order Reduction and Concolic Testing

Concolic testing has been previously combined with other dynamic partial order reduction algorithms than the one by Flanagan and Godefroid [5]: the race detection and flipping algorithm in [10] includes concolic testing. In this section we describe how the DPOR algorithm of Flanagan and Godefroid [5] can be used together with concolic testing.

A. Concolic Testing

Concolic testing [1]–[3] is a technique for systematically exploring a program's execution paths by running it with different inputs. To this end, concrete execution of the program is paired with symbolic execution so that concrete operations are interleaved with their symbolic equivalents. The symbolic execution yields a set of constraints which are used to generate new concrete input values for further testing.

To prepare a program for concolic testing its input values are marked for the testing tool. Any variables that may be affected by these input values are considered *symbolic variables*. A program is then instrumented so that for each concrete operation that uses a symbolic variable a symbolic counterpart is added. These operations generate constraints on the program's input values. In particular branching operations that use symbolic variables produce *path constraints*.

On the first test run the program is executed with random input values. From the constraints generated by the execution a single path constraint is selected, which is then negated, producing the constraint for the unexplored branch. The constraints from the beginning of the execution up to and including the negated constraint are then fed into a constraint solver. If the set of constraints is satisfiable we get a new set of input values that will cause the program to deviate from the previous execution path at the branching operation corresponding to the negated path constraint. An unsatisfiable set of constraints means that a concrete execution of the program can never follow the corresponding path. Once a new set of input values is obtained they are used as the input in the next test run. By repeating this process we can explore all the possible execution paths that the program can take.

A more in depth explanation of concolic testing is given in [11], [12] also describing the LCT concolic testing tool that was extended in this work.

B. Adding DPOR

Concolic testing can be thought of as systematically exploring a tree of a program's paths through its branching operations. DPOR, on the other hand, explores a tree where each new branch corresponds to an identified backtracking point. To employ both these algorithms simultaneously these trees must be combined. This can be achieved by constructing an execution tree where nodes for symbolic operations and nodes for scheduling decisions are interleaved in the order they are encountered during execution. To think of it in another way the resulting tree will contain nodes for scheduling decisions followed by a "concolic subtree" for each visible operation. The subtree is in turn formed by symbolic operations from the executing process of the corresponding visible operation. The leaves of the subtree are the scheduling points at the next visible operations that the process encounters.

The DPOR algorithm presented in Section II backtracks to previously explored states by returning from the `Explore` procedure. In an actual implementation this can be performed by, for example, re-executing the program with the same schedule, storing backtracking states or a combination of these [13]. In this case re-execution is a natural choice as the same technique is used by concolic testing. Furthermore, storing previous states might not be straightforward as in concolic testing a node in the execution tree is a set of constraints on the input and thus corresponds to many concrete states.

Recall that executing a visible operation was defined to include applying the visible operation as well as any invisible operations until the next visible operation is encountered. However, when combining with concolic testing the next visible operation may no longer be unique, as the invisible operations after a visible operation is executed can vary depending on the input to the program. Thus when a visible operation is explored the first path through the invisible operations will be arbitrary. The other paths through the invisible operations will be explored by concolic testing in subsequent test executions. This change does not greatly affect the implementation of the DPOR algorithm. Because the past of the execution and the next visible operation for each process is still known during each call to `Explore`, the logic for updating the vector clocks and identifying the backtracking points stays the same.

From the concolic testing algorithm's point of view this change to the execution tree adds some extra nodes that need to be handled correctly. Normally when a node that corresponds to an open branch is selected for exploration the input constraints are gathered from the nodes on the path from the root node to the selected node. In the combined execution tree this path will include extra nodes from the DPOR algorithm. However, these do not affect the input constraints and thus can safely be ignored.

IV. Implementation

We have implemented the DPOR algorithm with support for reads as an extension to the Lime Concolic Tester (LCT) [11], [12], which is an open source concolic testing tool for Java programs. To implement the DPOR algorithm we first added support for testing multi-threaded programs. We will now provide an overview of LCT, describe the support for multi-threaded programs, and finally show how the DPOR algorithm is implemented in this setting.

To prepare a Java program for testing with LCT it is modified to retrieve its inputs from the LCT runtime library. This can be, for example, achieved by writing a separate test driver that retrieves values from the LCT runtime library and then calls the original program with these values. The

```
1 while threads running do
2 |   wait until all running threads are blocked in the
  |   scheduler
3 |   V ← set of next visible operations for all threads
4 |   v ← Select(V)
5 |   apply v
6 end
```

Fig. 4. The main scheduler loop

modified program is then *instrumented* with a tool that adds for each operation that might be affected by the input values a call to a symbolic counterpart in the LCT runtime library. The instrumentation tool uses the Soot Java optimization framework [14] to perform the required transformations.

For the actual testing LCT uses a client-server model with potentially multiple concurrent clients executing tests in different parts of the execution tree. The *testing server* keeps track of the execution tree explored thus far. When a client connects the testing server selects a new path to explore and sends the relevant constraints to the client. The client then solves a set of inputs from these constraints (or asks for another path to be tested if they are unsatisfiable) and starts a new test execution, where these input values are used as the return values for the calls the test harness or modified program makes to the LCT runtime library to retrieve input. During its execution it sends details of each symbolic operation it executes to the server, which the server in turn uses to follow the client in the execution tree and update it as necessary. The clients do not persist any information between executions.

The version 2.2.0 of LCT that includes our extensions is available at http://www.tcs.hut.fi/Software/lime/.

A. Scheduling

To control the execution of visible operations a *scheduler* is introduced into the client side of LCT. The threads that constitute the program under test make a call to the scheduler asking for permission to execute before each visible operation they encounter. For each such call the visible operation about to be executed is recorded and the calling thread enters a wait until it is given permission to run. These permissions are in turn issued by a *main scheduler loop* from a separate thread.

A simplified pseudocode listing for the main scheduler loop can be seen in Figure 4. On each iteration of the loop it first waits for all threads of the program to enter the scheduler and begin the wait for permission to execute. Threads that terminate without executing any visible operations are ignored. Next the main scheduler loop constructs a set of the next visible operations for all threads. These include the ones that are not enabled, e.g., acquires for locked objects. Finally, a visible operation is selected to run through the `Select` procedure and the returned operation is then given permission to run. The `Select` procedure contains the client side part of our DPOR implementation and is presented in Section IV-C.

B. Instrumentation

To enable threads to ask permission for executing visible operations we extended the instrumentation based approach already present in LCT. All basic reads and writes that operate on shared variables are instrumented so that scheduler is called right before the operation is executed. This is similar to the approach used by Godefroid [6]. The call blocks, i.e., it does not return, until the scheduler gives permission for that visible operation to be executed. For example, consider the following write to a shared variable.

```
someObject.fieldName = 1;
```

To instrument this operation a call to the scheduler is added:

```
Scheduler.preWrite(someObject, "fieldName");
someObject.fieldName = 1;
```

The parameters for the call to `Scheduler.preWrite` are: (1) the base object of the variable and (2) the variable name. In the actual implementation the call would also include the code file name and line number for debugging purposes. Operations on elements of arrays are handled similarly with the variable name being replaced by the array index. The `synchronized` blocks in Java compile down to the `monitorenter` and `monitorexit` bytecode instructions [15]. These are in turn instrumented similarly to read and write operations.

The `Object.wait` operation is instrumented differently from the previous operations. Instead of adding a call before the operation, the whole operation is replaced with a call to the scheduler. For example, the call `obj.wait()` would be replaced by `Scheduler.doWait(obj)`. The method `Scheduler.doWait` internally uses and has the same semantics as `Object.wait`. The instrumentation for the methods `Object.notify` and `Object.notifyAll`, and thread parking related methods in `java.util.concurrent.locks.LockSupport` are all implemented similarly to `Object.wait`, with semantically equivalent alternative implementations in the scheduler.

This covers the most common constructs used for communication and synchronization between threads in Java. While not used for communication, the `Thread.start` method also receives some instrumentation. A call to `Scheduler.informThreadStart(Thread)` is added after each call to `Thread.start`, which notifies the scheduler that it may have to start tracking a new thread.

C. DPOR in a Client-Server Setting

This section describes how we have implemented DPOR in the client-server setting of LCT. DPOR has been previously parallelized in [16]. For a parallel partial order reduction approach based on ample sets see [17].

Our implementation of DPOR resides in the `Select` procedure, referred to in Figure 4, as well as on the testing server. Figure 5 presents a rough pseudocode for the `Select` procedure. Two larger changes have been made to fit the algorithm into the LCT tool: (1) to reach previous states the execution sequence is replayed instead of returning from the

```
1  procedure Select(V)
2      s ← last(E)
3      if SE is not empty then
4          v ← pop(SE)
5      else
6          forall the visible operations v ∈ V do
7              p ← proc(v)
8              if v may write and ∃k =
               max({ k | LR(α(v))_k ≰ CP(p)(proc(E_i)) })
               then
9                  i ← LR(α(v))_k
10             else
11                 i ← LW(α(v))
12             end
13             if i ≠ 0 and i ≰ CP(p)(proc(E_i)) then
14                 e ← enabled(pre(E, i))
15                 if v ∈ e then
16                     SendBacktrack({v}, i)
17                 else
18                     SendBacktrack(e, i)
19                 end
20             end
21         end
22         v ← r ∈ V such that r is enabled
23     end
25     // Maintain the vector clocks and
          last accesses.
26
27     SendExecution(v, V)
28     return v
29 end
```

Fig. 5. Client side part of our DPOR implementation

`Explore` procedure, (2) backtracking points are reported to the testing server instead of being stored locally.

The `Select` procedure has access to a stack of visible operations SE, which is initialized from the testing server at the beginning of the execution. When `Select` is called visible operations will be taken and executed from the stack SE until it is empty. The stack is always such that the last visible operation executed from it is one that was previously added to a backtracking set.

The LCT client loses all state when the execution ends. As such backtracking decisions are, on lines 16 and 18, sent to the testing server instead of being stored locally. To make sense of the backtracking decisions the server must keep track of where the client is in the server's execution tree. For this purpose, on line 27, each scheduling decision that `Select` makes is also sent to the server. On the server the backtracking decisions received from the client are handled by adding new nodes to the execution tree at the point indicated by the received index. These nodes will then be explored in a later execution.

From the way we have implemented DPOR in Figure 5 it is apparent that multiple concurrent clients may be used for testing: when one client discovers a backtracking point another one may start a test execution to explore it before the first one has finished. While no client side modifications are required to enable this, the server will have to be properly synchronized. In [16] an alternative approach to distribute DPOR is presented where the testing is divided to multiple workers without relying on a synchronizing server as in LCT. This approach provides excellent scalability to the number of workers but in some cases results in exploring a same schedule multiple times. The client-server approach of LCT avoids this problem and based on initial results of distributing concolic testing of single-threaded programs [11], the tighter synchronization still scales well up to at least 20 clients.

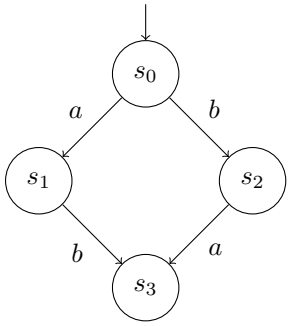

Fig. 6. Program with two independent visible operations

D. Sleep Sets in a Client-Server Setting

In this section we describe the sleep set algorithm, which was first described by Godefroid [18]. As one of our main contributions we also present our modification to it for implementation in a client-server setting. The following explanation of sleep sets is adapted from Godefroid [9].

Consider a program that consists of two processes, which execute the independent visible operations a and b. Figure 6 presents the state space of such a program, where from the initial state s_0 the program will move to the state s_1 or s_2 depending on the operation executed and will end up in the state s_3 after the execution of the second operation. The final state s_3 is shared for both interleavings of the operation due to their independence.

The sleep set algorithm is based on the observation that if a visible operation t_1 is already explored from some state s, then when any visible operation $s \xrightarrow{t_2} s'$, where t_2 is independent of t_1, is explored there is no need to explore t_1 from s'. To this end for each state we associate a *sleep set*, which is a set of visible operations that will not be explored from that state. For example, in Figure 6 if the first execution is $s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_3$ then when b is executed from s_0 the visible operation a will be added to the sleep set in state s_2. In this case we can avoid exploring $s_2 \xrightarrow{a} s_3$.

The sleep set for the initial state of a program $\mathrm{sleep}(s_0)$ is the empty set. When a visible operation is explored from any state, $s \xrightarrow{v} s'$, the sleep set for the successor state s' is

computed as follows. Let $\mathrm{explored}(s)$ be the visible operations that have already been explored from the state s. The *candidate sleep set* for s' is the union $\mathrm{sleep}(s) \cup \mathrm{explored}(s)$. This candidate sleep set is then filtered to only include visible operations that are independent with the executed operation v thus producing the final sleep set $\mathrm{sleep}(s')$.

Our setting presents two complications to implementing sleep sets: (1) only the server knows which visible operations have been explored from a given state and (2) only the client knows the dependencies between visible operations, because on the server a visible operation is identified only by its executing thread (or also the target thread in case of `Object.notify`). Therefore we split the implementation between the server and client as follows.

In the beginning of the execution the server sends the candidate sleep set for the state the client will reach once it has re-executed the visible operations sent by the server (as described in Section IV-C). Once the client has executed the last visible operation from the server the dependent operations are removed from the sleep set, which is then sent back to the server. For the rest of the execution no new visible operations need to be added to the sleep set as the client is always executing the first operation from the states it is in. Dependent visible operations are removed after each executed operation and the resulting sleep sets are sent to the server.

Combining sleep sets with DPOR is straightforward and the resulting `Select` procedure is shown in Figure 7. The sleep set is stored in the global variable SLP, which is initialized with the candidate sleep set from the server. On line 9 the selection of the next visible operation to execute is limited to the enabled operations that are not in the current sleep set. If all enabled visible operations are in the sleep set then the execution ends immediately. On lines 11 – 14 if the visible operations from the server have been exhausted then the visible operations that are dependent with the operation that was selected to be executed are removed from the sleep set. Then, on line 13, the resulting sleep set is sent to the server.

```
1  procedure Select(V)
2      s ← last(E)
3      if SE is not empty then
4          v ← pop(SE)
5      else
7          // Identify backtracking points.
8
9          v ← r ∈ V such that r ∈ enabled(s) \ SLP or
           end the execution if no such visible operation
           exists.
10     end
11     if SE is empty then
12         SLP ← {r ∈ SLP | r is not dependent with v}
13         SendSleepSet(SLP)
14     end
16     // Maintain the vector clocks and
          last accesses.
17
18     SendExecution(v, V)
19     return v
20 end
```

Fig. 7. Client side of our sleep set implementation

V. Experiments

In this section we provide some experimental results for the modified DPOR algorithm with support for commutative reads (DPOR-CR from now on). We tested it against the unmodified DPOR algorithm and the race detection and flipping algorithm in the closed source jCUTE [3] tool. The DPOR-CR algorithm was tested both with sleep sets and without. All algorithms, except for the race detection and flipping algorithm, were implemented in LCT. In these experiments we have used a single client for testing with LCT. We have left the evaluation of DPOR with multiple concurrent clients to future study.

In the experiments we compare the number of executions needed by each partial order reduction algorithm to test a program. All the programs under test have a finite state space and, therefore, the differences in the numbers depend on how much reduction the algorithms achieve.

The File System and Indexer programs are from Flanagan and Godefroid [5], where they were used to evaluate the DPOR algorithm. The Parallel Pi program implements a parallel algorithm for calculating the value of π. All three of these programs were tested using a varying number of threads. The Bounded Buffer and Synchronous Queue programs are two larger programs that test implementations of thread-safe containers. The container in the Bounded Buffer program is an example from [19] while the container in the Synchronous Queue program is from the Java Collections Framework.

The results can be seen in Figure 8. The number in parentheses beside the Indexer, File System and Parallel Pi programs is the number of threads used. The number of executions for the Synchronous Queue with jCUTE is missing due to a bug or incompatibility we encountered. As noted before, the amount of reduction achieved by the DPOR algorithm can vary depending on in which order visible operations are explored [7]. In these tests we explored visible operations in a random order while running each test a five times. We report the average number of test executions in these five tests. The jCUTE tool explores visible operations deterministically and thus the reduction it provides does not vary.

The DPOR-CR algorithm fares better than the unmodified DPOR algorithm on the Indexer, File System and Parallel Pi programs. This is a straightforward result from the presence of reads on shared variables in these programs. On the Bounded Buffer program the DPOR-CR algorithm provided no benefit over the unmodified DPOR algorithm. This is due to all operations on shared variables in the program being protected by `synchronized` blocks.

The results for the Synchronous Queue program show a very low number of executions for the DPOR-CR algorithm when

Program	Number of test executions				
	No reductions	DPOR	DPOR-CR	DPOR-CR, sleep sets	jCUTE
Indexer (12)	> 10000	8614.6	154	27	**8**
Indexer (13)	> 10000	> 10000	> 10000	722.4	**343**
File System (14)	> 10000	6.8	3.2	2.6	**2**
File System (16)	> 10000	568.4	26.8	**19.5**	31
File System (18)	> 10000	> 10000	250.2	**145.8**	2026
Parallel Pi (3)	> 10000	> 10000	3217.8	19.2	**6**
Parallel Pi (5)	> 10000	> 10000	> 10000	1220.6	**120**
Bounded Buffer	> 10000	64.4	67.2	16	**8**
Synchronous Queue	> 10000	> 10000	> 10000	**9**	N/A

Fig. 8. Average number of test executions explored for each program and algorithm

combined with sleep sets while all other combinations went over our testing limit of 10000 executions. We are unsure of the exact reason for the stark difference between the DPOR-CR algorithm with and without sleep sets. It would, however, seem that the two algorithms complement each other very well on the Synchronous Queue program. For example, on the File System program the benefit from using sleep sets is much less pronounced.

Compared to the unmodified DPOR algorithm, the race detection and flipping algorithm of jCUTE often achieves significantly better results. However, once our modification and and sleep sets are added the DPOR based approach is much more competitive. Particularly on the File System (18) program the DPOR-CR algorithm with sleep sets outperforms jCUTE by a wide margin. One difference between jCUTE and DPOR-based approaches is that when a race is detected, jCUTE 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 caused by the same race. DPOR does not have this extra information and therefore can backtrack unnecessarily. One direction for future work is to investigate if similar optimization can be implemented to DPOR.

VI. Conclusions

We have modified the dynamic partial order reduction (DPOR) algorithm of Flanagan and Godefroid [5] to exploit the commutativity of read operations and have combined the modified algorithm with concolic testing to enable systematic testing of multi-threaded programs that take input. We have implemented our modified algorithm and the sleep set algorithm in the Lime Concolic Tester (LCT) [11], [12], which is an open source concolic testing tool designed for distributed use. Our versions of the DPOR algorithms also allow parallelization of testing in a client-server setting. Our experimental evaluation shows that the modified DPOR algorithm greatly improves on the unmodified DPOR algorithm.

References

[1] 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)*. ACM, 2005, pp. 213–223.

[2] K. Sen, "Scalable automated methods for dynamic program analysis," Doctoral thesis, University of Illinois, 2006. [Online]. Available: http://osl.cs.uiuc.edu/~ksen/paper/sen-phd.pdf

[3] 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)*, ser. Lecture Notes in Computer Science, vol. 4144. Springer, 2006, pp. 419–423, (Tool Paper).

[4] A. Valmari, "The state explosion problem," in *Petri Nets*, ser. Lecture Notes in Computer Science, W. Reisig and G. Rozenberg, Eds., vol. 1491. Springer, 1996, pp. 429–528.

[5] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL*, J. Palsberg and M. Abadi, Eds. ACM, 2005, pp. 110–121.

[6] P. Godefroid, "Model checking for programming languages using VeriSoft," in *POPL*, 1997, pp. 174–186.

[7] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha, "Evaluating ordering heuristics for dynamic partial-order reduction techniques," in *FASE*, ser. Lecture Notes in Computer Science, D. S. Rosenblum and G. Taentzer, Eds., vol. 6013. Springer, 2010, pp. 308–322.

[8] K. Sen, G. Rosu, and G. Agha, "Runtime safety analysis of multi-threaded programs," in *ESEC / SIGSOFT FSE*. ACM, 2003, pp. 337–346.

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

[10] K. Sen and G. Agha, "A race-detection and flipping algorithm for automated testing of multi-threaded programs," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, E. Bin, A. Ziv, and S. Ur, Eds., vol. 4383. Springer, 2006, pp. 166–182.

[11] K. Kähkönen, T. Launiainen, O. Saarikivi, J. Kauttio, K. Heljanko, and I. Niemelä, "LCT: An open source concolic testing tool for Java programs," in *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'2011)*, 2011, pp. 75–80.

[12] K. Kähkönen, "Automated test generation for software components," Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland, Technical Report TKK-ICS-R26, Dec 2009.

[13] P. Godefroid, "Software model checking: The VeriSoft approach," *Formal Methods in System Design*, vol. 26, no. 2, pp. 77–101, 2005.

[14] 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)*. IBM, 1999, p. 13.

[15] J. Engel, *Programming for the Java virtual machine*. Addison-Wesley, 1999.

[16] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Distributed dynamic partial order reduction based verification of threaded software," in *SPIN*, ser. Lecture Notes in Computer Science, D. Bosnacki and S. Edelkamp, Eds., vol. 4595. Springer, 2007, pp. 58–75.

[17] L. Brim, I. Cerná, P. Moravec, and J. Simsa, "Distributed partial order reduction of state spaces," *Electr. Notes Theor. Comput. Sci.*, vol. 128, no. 3, pp. 63–74, 2005.
[18] P. Godefroid, "Using partial orders to improve automatic verification methods," in *CAV*, ser. Lecture Notes in Computer Science, E. M. Clarke and R. P. Kurshan, Eds., vol. 531. Springer, 1990, pp. 176–185.
[19] J. Magee and J. Kramer, *Concurrency - state models and Java programs (2. ed.)*. Wiley, 2006.