A Sound Dynamic Partial Order Reduction Engine for Java Pathfinder

Kyle Storey
Brigham Young University
Provo, Utah
kyle.r.storey@gmail.com

Eric Mercer
Brigham Young University
Provo, Utah
egm@cs.byu.edu

Pavel Parizek
Charles University
Prague, Czechia
parizek@d3s.mff.cuni.cz

ABSTRACT

When model checking a multi-threaded program, it is often necessary to enumerate the possible ordering of concurrent events to evaluate the behavior of the program. However, enumerating every possible order of events quickly leads to state-space explosion. Dynamic Partial Order Reduction (DPOR) is a method to dynamically determine a subset of schedules that need to be evaluated to observe all the relevant behavior of a program. A sound implementation of DPOR in Java Pathfinder (JPF) can be tricky without incurring unacceptable amounts of overhead, because JPF does not support subdividing existing transitions. Conservatively inserting choice generators to end transitions at each possible scheduling point causes JPF to save a large amount of state. We present an extension to JPF, which is an efficient implementation of DPOR that attempts to minimize spacial complexity. It handles the directing of the search and uses a simple interface to allow the user to define the set of events to operate on and to determine which of those events are dependent. It keeps its own internal representation of all possible scheduling points without inserting choice generators at each point. It then restarts portions of the search, if necessary, to insert only the needed choice generators.

Keywords

Model Checking, Dynamic Partial Order Reduction, Java Pathfinder

1. INTRODUCTION

To discover all of the behavior of a multi-threaded program, an analysis needs to consider the possible order of concurrent events. Each possible order, or schedule as it is often called, may cause variables to take on different values. This may in turn change control flow, dramatically changing the program's behavior.

The number of possible schedules of a program grows at a factorial rate with the number of concurrent events. This means considering all the possible schedules of even a fairly small number of concurrent events is often intractable. Fortunately, the relative order of only some events will affect the behavior of the program.

Dynamic Partial Order Reduction (DPOR), such as the method presented by Flanagan and Godefroid [Flanagan and Godefroid 2005], dynamically discovers which events must be re-ordered and drastically reduces the number of schedules that must be analyzed to prove properties of a program.

JPF implements a form of DPOR using a sharedness mechanism called a sharedness policy. The sharedness policy observes the search and determines which objects are shared between threads. JPF enumerates all non-deterministic thread scheduling choices using choice generators. When the sharedness policy determines that an object is shared, JPF can be configured to insert a choice generator before an access to the object so the search can evaluate all the possible schedules of these events.

JPF's default DPOR mechanism can be a convenient way to observe many program behaviors, but if an analysis needs to observe schedules of events other than accesses to shared variables, there is little support for DPOR within JPF. More critically, if an analysis plans to dynamically discover which events in a program must be reordered, JPF cannot return to insert choice generators at events that an analysis does not discover must be reordered until later in the search. The inability to add thread choices at these dependent events prevents the soundness of analyses because it causes the search to miss behaviors. This motivates the need for a system within JPF that implements DPOR to explore all necessary schedules over any set of events, that also can support the dynamic discovery of dependent events. Our work, which we call a DPOR Engine, addresses these problems. It uses an internal representation of the potential search space, the DPOR Graph, to implement DPOR in a way that is more closely connected with Flanagan and Godefroid's [Flanagan and Godefroid 2005] work, while taking advantage of JPFs ability to save state so it does not always have to restart from the beginning of the program like the original algorithm. The source code for the DporEngine can be found at `http://bitbucket.org/byu-vv/dporengine/src/jpf19`.

2. DYNAMIC PARTIAL ORDER REDUCTION

This work attempts to follow closely the work of Flanagan and Godefroid [Flanagan and Godefroid 2005] in implementing DPOR. For a detailed description of their DPOR algorithm the authors would recommend reading their paper, but we will provide an overview and application of their algorithm in a JPF context here.

2.1 Overview of DPOR

DPOR is a method for exploring a reduced state space of a concurrent system by controlling the schedule of only certain events. We call these events *scheduling points*. By scheduling only over this subset of events, DPOR reduces the number of explored schedules dramatically. DPOR further reduces the number of explored schedules required by only scheduling over events that are mutually *dependent*. Intuitively, dependent events are events where the relative order of their execution affects the behavior of the program in some meaningful way. For example, scheduling points typically represent points of non-determinism such as accesses to shared memory. And two accesses to shared memory will only be dependent if they access the same memory location and at least one of them is a write. In order for events to be dependent they must also be able to execute concurrently and reversing the order of these events must produce two distinct program states. DPOR utilizes a *dependency relation* to determine whether two scheduling points are dependent. For any valid definition of scheduling

points, and any valid dependency relation over them, the DPOR algorithm explores a persistent set of transitions from every state, guaranteeing that safety properties can be completely verified during the search.

2.2 Difficulty of implementing DPOR within JPF

To implement a DPOR algorithm in JPF a choice generator must be inserted at each scheduling point. However, in general, it can be difficult to determine if a given instruction is a scheduling point. Consider an analysis which does not consider data race to be a bug, but instead must determine all the behaviors of a program due to data race. To do so, it must schedule over all accesses to shared variables where at least one of the accesses is a write. This analysis cannot determine a priori if any given variable is shared or not, so it maintains a set of variables that have been accessed by more than one thread. Often it is not discovered that a variable is shared until much later in the search. This means that any access to any variable could later be discovered to be a scheduling point. In JPF, transitions are defined to begin and end when a new choice generator is inserted. Once set, JPF does not support subdividing these transitions. This means that we cannot insert choice generators for previous accesses to a shared variable when it is discovered that those accesses are scheduling points. So, to keep our analysis sound, we would have to conservatively insert a new choice generator for every access to any variable. For each of these choice generators, JPF will consolidate and create a state object. As the number of variable accesses tends to be large, this quickly creates unacceptable amounts of overhead.

3. DPOR ENGINE

Our main contribution is the DPOR Engine: a system for easily including DPOR in a JPF search. Through a few interfaces, the engine is configurable to operate on any relevant set of events, and it correctly handles situations where it is not known a priori that an event is a scheduling point. To circumvent the large overhead that comes with inserting choice generators at every possible scheduling point, it maintains its own internal representation of the potential search space which we call a DPOR Graph. To control the scheduling of events, the DPOR Engine inserts its own custom choice generators which each hold an associated node from the DPOR Graph; allowing the DPOR Engine to connect each custom choice generator with its position in the search. Each custom choice generator then delegates decisions on whether there are more choices to consider and which threads should be run from that point to the DPOR Engine providing their node from the DPOR Graph to inform these decisions. The DPOR Engine also uses the DPOR Graph to determine whether there exists one or more scheduling points without an associated choice generator. If there is, it re-executes a portion of the search so that the choice generators can be included.

3.1 DPOR Graph

The DPOR Graph is the main data structure the DPOR Engine uses to direct the scheduling of the search. Figure 1 shows an example DPOR Graph. It was generated by the DPOR Engine from DporGraphExample.java in Figure 2. Topographically, the DPOR Graph is similar to a JPF state space graph where a choice generator was inserted at every potential scheduling point. It consists of three types of nodes: *Yes*, *Maybe*, and *Replay*. A *Yes* node indicates a point in the search that was confirmed to be a scheduling point where a choice generator was inserted. A *Maybe* node indicates that at that point in the search, the analysis could not determine if the event is a scheduling point or not. For example, if we want to schedule over accesses to shared variables, a *Maybe* node may be inserted when there was an access to a variable but

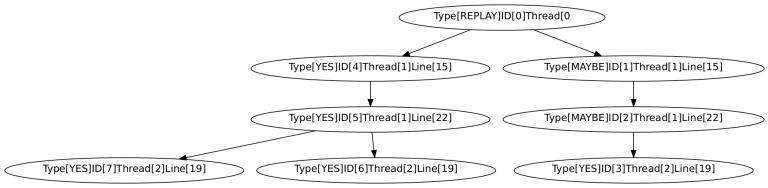

Figure 1: An example DPOR Graph

```
1  public class DporGraphExample extends Thread {
2    static int i; //sharedness unknown a priori
3    public static void main(String[] args) {
4    new DporGraphExample().start;
5      i = 2; //runs in main thread
6    }
7    public void run() {
8      i = 1; //runs in new thread
9    }
10 }
```

Figure 2: DporGraphExample.java

it is unknown whether the variable is shared or not. Both *Yes* and *Maybe* nodes will also contain a set of threads that are runnable from that point (runnables), and a set of threads that should be scheduled to run from that point (choices). Initially, the choices contain only the currently running thread when the node was created. The DPOR Engine adds choices to these nodes as necessary. *Replay* nodes mark points in the search from which we can replay. If it is discovered there is a scheduling point without an associated choice generator we can restart the search from this point so we can insert a choice generator; reducing the amount of redundant re-execution. Details on this process will be discussed in Section 3.4. Each DPOR Graph has a *Replay* node as its root node which is associated with the root choice generator. A DPOR Graph also may have additional *Replay* nodes at places defined by the DporInstructionMarker.

3.2 Interface

To use the DPOR Engine, two interfaces must be implemented: DporInstructionMarker, and DporDependenceRelation. Optionally the DporResettable interface can be implemented to help an analysis synchronize its local state with the state of the search.

The DporInstructionMarker specifies to the DPOR Engine which events may be scheduling points. Before each instruction is executed, the DPOR Engine calls the mark method in the DporInstructionMarker, as indicated in line 2 of Algorithm 1. The mark method must determine if the given instruction should be considered a scheduling point. It does so by returning *Yes*, *No*, *Maybe*, or *Replay*. If the DporInstructionMarker returns *Yes*, the instruction is a scheduling point. A choice generator is inserted, and a new *Yes* node is added to the DPOR Graph as indicated in lines 3 through 5. Lines 7 and 8 show that if the DporInstructionMarker returns *Maybe*, the instruction could potentially be a scheduling point but the analysis is not yet sure. In this case no choice generator is inserted, and a *Maybe* node is added to the DPOR Graph. Instead of returning *No*, the DporInstructionMarker may return *Replay* for a given instruction that is not a scheduling point. If it does a choice generator is inserted. Instead of restarting the search

from the beginning to insert choice generators, a smaller section of the search can be replayed from that point. Lines 10 through 12 describe this behavior. If the DporInstructionMarker returns *No*, execution continues as normal.

The DporDependenceRelation specifies to the DPOR Engine which pairs of scheduling points are dependent. To do so it must implement the dependent method which given two nodes from the DPOR Graph must return a boolean indicating if the two scheduling points associated with those nodes are dependent. Specifically, the DporDependenceRelation should return true if and only if the two associated scheduling points are dependent as defined by the given analysis, they can be co-enabled, and they are concurrent.

To aid in this task, each node contains an object called the DependenceInfo which can store whatever information is necessary for the analysis to determine if two scheduling points are dependent. For instance, the analysis may place an object that holds the name of the variable that was accessed at that point, to see if both scheduling points accessed the same variable, and some data structure to help determine if the two scheduling points are concurrent, such as a vector clock. To provide an opportunity for the analysis to store this info in the node, each time a new node is created in the DPOR Graph the onCreateNode method is called providing the new node being created as well as the parent of that node. At that time, an object can be stored in the parent node as the creation of a new node marks the end of that transition. A mutable object can also be stored in the child node so it can be retrieved when needed and updated during the transition.

It is anticipated that many analyses using the DPOR Engine will need to keep track of information about the state of the program. As the DPOR Engine backtracks and tries new paths, it can be difficult to keep this information synchronized as the program state is reset during the model checking process. The DporResettable interface simplifies this. Similar to how JPF resets the program state on a backtrack, the DporEngine will reset the state of an analysis registered as a DporResettable. To do so, an analysis must implement getImmutableState as well as resetState. When a new choice generator is created the DPOR Engine calls getImmutableState on each registered DporResettable and stores the state object for the corresponding scheduling point and DporResettable. Then later, when the search backtracks to that choice generator, the corresponding object is retrieved for each DporResettable and resetState is called, passing the object received at that point. The DporResettable can then use the state object to reset its data to the match the program state.

3.3 DPOR Algorithm using a DPOR Graph

As JPF executes each program step, the DPOR Engine passes the instruction that will be executed next to the DporInstructionMarker and collects the result. Then the DPOR Graph is updated and choice generators are inserted as described. The DPOR Engine keeps track of the node that was created for use in the DPOR Algorithm. It is represented by N as seen in Algorithm 1 line 4. If the DporInstructionMarker returned *Yes* or *Maybe* the DPOR algorithm is executed as seen on lines 6 and 9. Starting with the parent node of the newly created *Yes* or *Maybe* node, the DPOR Engine traverses up the graph through the ancestor nodes of the new node, searching for the most recent node which the DporDependenceRelation reports is dependent with the new node. If that node is found, the DPOR Engine will add choices to the ancestor to schedule over those events. The process of searching for the most recent dependent node is described in lines 17 through 24 of Algorithm 1. Lines 26 through 32 describe the process of scheduling over the dependent events. To do so, it starts by attempting to add the current thread for the new node to the choices of the ancestor node. If the current thread is not in the runnables, the DPOR Engine conservatively adds all the runnables for the ancestor node to the ancestors choices. These steps implement the DPOR algorithm as described by Flanagan and Godefroid [Flanagan and Godefroid 2005].

However, if the ancestor node that was found to be dependent with the new node is a *Maybe* node, this indicates that the instruction connected with that ancestor node was indeed a scheduling point and the DPOR Engine must arrange for a portion of the search to be replayed so a choice generator can be inserted.

For illustration, consider Figures 1 and 2. Figure 1 shows the final state of the DPOR Graph, but consider a previous state of the graph where only the nodes with IDs 0 and 1 have been created. When the graph was in this state, JPF was about to execute a write on line 8 of DporGraphExample.java. The DPOR Engine looked ahead at this instruction and passed it to the DporInstructionMarker which in this case returned *Yes*. The DPOR Engine then created the node with ID 2 as seen in Figure 1. It inserted a new choice generator at that point and then began executing the DPOR Algorithm. It searched through the ancestors of the new node beginning with Node 1. It passed Nodes 1 and 2 to the DporDependenceRelation. In this case, the relation returned true, indicating that the nodes are dependent. The DPOR Engine then added choices to schedule over these two nodes. To do so, it checked if thread 1 could be run from the scheduling point associated with Node 1. In this case it could, so thread 1 was added to the choices of Node 1. Node 1 also is a *Maybe* node so the DPOR Engine needed to arrange to replay this section to insert the necessary choice generators. We describe this process in Section 3.4.

3.4 The Replay Algorithm

If it is discovered that a *Maybe* node was indeed a scheduling point, the DPOR Engine will traverse up the DPOR Graph beginning at the parent of the *Maybe* node. When it reaches a *Replay* node (which can be at one of the additional places requested by the DporInstructionMarker, or the replay that is the root of each graph) a flag will be set in the *Replay* node indicating that the search must replay from that point. When JPF backtracks to a choice generator associated with a *Replay* node, the DPOR Engine checks if the flag was set. The replay-flag being set indicates that some scheduling point later in the search from that point did not receive a choice generator and must be replayed. However, if the *Replay* node has an ancestor *Replay* node that also must replay a larger portion of search, replaying this smaller section will be redundant. To avoid these redundancies, the DPOR Engine again searches up the DPOR Graph. If it finds a *Replay* node with its flag set, it will return to JPF indicating that the *Replay* node at the current search point has no more choices and JPF will continue to backtrack. If none of the current *Replay* node's ancestors have the replay-flag set, the DPOR Engine returns a choice for the initial thread again, effectively restarting the search from that point.

Continuing the previous illustration from Section 3.3, when the DPOR Engine discovers it needs to schedule a replay for Node 1, it traverses up the ancestors of Node 1 to find the most recent replay node. In this case immediately finds the root node, Node 0. The DPOR Engine sets a flag in Node 0 so that when JPF backtracks to that point it will replay this section. The left branch of Node 0 in Figure 1 shows the results of this replay. Note that on the

Algorithm 1 DPOR using DPOR Graph

```
1: procedure ExecuteInstruction(I)
2:     M = InstructionMarker.mark(I)
3:     if M = Yes then
4:         N = DporGraph.addNode(Yes)
5:         InsertChoiceGenerator(N)
6:         ExecuteDpor(N)
7:     else if M = Maybe then
8:         N = DporGraph.addNode(Maybe)
9:         ExecuteDpor(N)
10:    else if M = Replay then
11:        N = DporGraph.addNode(Replay)
12:        InsertChoiceGenerator(N)
13:    end if
14: end procedure
15: procedure ExecuteDpor(N)
16:    P = N;
17:    while P.hasParent do
18:        P = P.getParent
19:        d = DependenceRelation.dependent(P,N)
20:        if P.type ≠ Replay ∧ d then
21:            AddScheduleFor(P,N)
22:            break
23:        end if
24:    end while
25: end procedure
26: procedure AddScheduleFor(P,N)
27:    T = N.getCurrentThread
28:    if T ∈ P.getRunableThreads then
29:        P.addChoice(T)
30:    else
31:        P.addAllRunnableThreads
32:    end if
33:    if P.type = Maybe then
34:        ArrangeReplay(P)
35:    end if
36: end procedure
```

Algorithm 2 Replay Algorithm

```
1: procedure ArrangeReplay(P)
2:     R = P;
3:     while R.hasParent do
4:         R = R.getParent
5:         if R.type = Replay then
6:             R.replayFlag = TRUE
7:             break
8:         end if
9:     end while
10: end procedure
11: procedure MustReplay(R)
12:    if R.replayFlag = FALSE then
13:        return FALSE
14:    end if
15:    S = R;
16:    while S.hasParent do
17:        S = S.getParent
18:        if S.type = Replay ∧ S.replayFlag = TRUE then
19:            return FALSE
20:        end if
21:    end while
22:    return TRUE
23: end procedure
```

replay, the DporInstructionMarker now marks the write on line 5 as *Yes*. This is because it now knows that i is a shared variable. This allows the DPOR Engine to insert the necessary choice generators and schedule over those scheduling points. You can see that Node 3 and Node 1 both correspond to line 5 of the program but on the replay Node 3 has two children indicating that two schedules were executed.

4. RESULTS

We have implemented a few examples that utilize the DPOR Engine to test its functionality and compare empirically the trade offs of using the system. We implemented the following examples: DataRaceBehavior (DRB) which dynamically discovers sharing and enumerates over shared variables utilizing the *Maybe* and replay features; EveryAccess (EA) which implements the InstructionMarker to return *Yes* if the instruction is a field access. This is the naive approach of inserting a choice generator for every potential scheduling point; and the default JPF DPOR mechanism (JPF) which runs the benchmark using JPF's GlobalSharednessPolicy and HjSyncPolicy, a minimal SyncPolicy. It was fairly simple to implement each of these examples. The most difficult was the full featured DRB and it was implemented in just 79 lines of code.

We took the average time it took to model check each benchmark in milliseconds averaging over 5 runs. We also indicate whether, by our evaluation, it enumerated all the behavior of the program. The DPOR Engine can be configured to emit the DPOR Graph as a dot file whenever a thread terminates. For DRB and EA we include the number of graphs emitted as it gives a measure of how many schedules were enumerated. See Table 1 for the results of our tests. All of the source of the benchmarks shown can be found in the repository.

In some cases, using the DPOR Engine creates more work and overhead then it gives benefit. For instance in MaybeExample note that DRB produced 8 graphs where EA produced only 5. This is because the example returned *Maybe* on an event that turned out to be a scheduling point. This resulted in a replay which caused some redundant execution. However, as the number of scheduling points increases the benefit of utilizing all of the DPOR Engine's features becomes apparent. As with DporEngineExample where where DRB clearly outperforms EA and JPF.

5. RELATED WORK

DPOR was first introduced in [Flanagan and Godefroid 2005] as a way to use runtime information to refine partial order reduction techniques based on persistent sets. Several alternative implementations of DPOR have been implemented in JPF before [Noonan et al. 2014, Shafiei and Mehlitz 2014, Rizzi et al. 2014, Brat and Visser 2001], but they are tied to specific use cases and do not generalize to any dependency relation. Ongoing work on DPOR algorithms may benefit from a framework like the DPOR Engine [Isabel 2019, Chalupa et al. 2017, Abdulla et al. 2014, Zhang et al. 2015, Albert et al. 2019]. Many of these projects use SYCO, a tool like JPF for the ABS concurrent objects language [Albert et al. 2016]. Like SYCO, JPF with the DPOR Engine allows for research and development of new partial order reduction techniques, but targeted at Java programs.

6. FUTURE WORK

[1] Currently the DPOR Engine depends on the InstructionMarker to retain enough information about the search to correctly label

[1]Between the time of peer-review and publication this work has been completed. It can be found at `http://bitbucket.org/byu-vv/dporengine/src/sdpor`.

Table 1: Results of tests comparing different DPOR Approaches

Benchmark	DRB Time (ms)	DRB Graphs	DRB All Behavior	EA Time (ms)	EA Graphs	EA All Behavior	JPF Time (ms)	JPF All Behavior
MoreSimple	4642	7	Yes	4601	7	Yes	4589	No
LessSimple	5126	17	Yes	4887	19	Yes	4589	No
MaybeExample	4772	8	Yes	4574	5	Yes	4388	No
DporEngineExample	6527	62	Yes	7446	125	Yes	>300000	Yes

instructions on the replay. Specifically, it is expected that the InstructionMarker return *Yes* to any instructions that in a previous run the InstructionMarker returned *Maybe* that were indeed scheduling points. If it continues to return *Maybe* for instructions that are scheduling points, the DPOR Engine will simply continue to replay those sections and the search will never finish. We intend to remove this burden from the user by keeping track of where the analysis is in the program using the DPOR Graph. To do this, we need to develop a way to accurately determine if the instruction that is being executed on the replay is the same instruction that created a given node in the DPOR Graph. This way, the DPOR Engine can determine which instructions are scheduling points on the replay without having to consult the InstructionMarker. This would have the added benefit of reducing the amount of redundant executions on the replay. In many cases during the replay the DPOR Engine would not have to re-execute the first thread chosen at a scheduling point and could skip to subsequent choices.

Additionally, the DPOR Engine is only sound if state matching is disabled in JPF. Yang *et. al.*[Yang et al. 2008] presented a method to adjust DPOR to support state matching. Their solution involves keeping deltas to keep track of state, but as JPF already handles state saving and resetting we propose adjusting the state match conditions and the DPOR Engine to implement a stateful DPOR as described by Yang *et. al.*[Yang et al. 2008].

7. CONCLUSION

We present an extension to JPF which simplifies the inclusion of Sound DPOR in the search. It is an implementation of known DPOR algorithms with some extension to overcome limitations in JPF. DPOR is most beneficial when an analysis also dynamically detects dependency of events. Because JPF does not support breaking transitions when set, a necessary feature when implementing DPOR when events cannot be known to be scheduling points a priori, a naive approach would need to insert choice generators at every possible event dependency of events leading to a large amount of overhead from JPF storing state. The DPOR Engine circumvents this large overhead by maintaining a DPOR Graph: a structure that is a projection of the JPF state space that is used to determine which portions of the program must be replayed to break transitions. The DPOR Engine allows the user to ignore the details of directing the search to more quickly implement their analysis.

8. REFERENCES

[Abdulla et al. 2014] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 373–384.

[Albert et al. 2019] Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J Stuckey. 2019. Optimal context-sensitive dynamic partial order reduction with observers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 352–362.

[Albert et al. 2016] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. 2016. SYCO: a systematic testing tool for concurrent objects. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 269–270.

[Brat and Visser 2001] Guillaume Brat and Willem Visser. 2001. Combining static analysis and model checking for software analysis. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE, 262–269.

[Chalupa et al. 2017] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158119

[Flanagan and Godefroid 2005] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315

[Isabel 2019] Miguel Isabel. 2019. Conditional Dynamic Partial Order Reduction and Optimality Results. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. ACM, New York, NY, USA, 433–437. https://doi.org/10.1145/3293882.3338987

[Noonan et al. 2014] Eric Noonan, Eric Mercer, and Neha Rungta. 2014. Vector-clock Based Partial Order Reduction for JPF. *SIGSOFT Softw. Eng. Notes* 39, 1 (Feb. 2014), 1–5. https://doi.org/10.1145/2557833.2560581

[Rizzi et al. 2014] Eric F. Rizzi, Mathew B. Dwyer, and Sebastian Elbaum. 2014. Safely Reducing the Cost of Unit Level Symbolic Execution Through Read/Write Analysis. *SIGSOFT Softw. Eng. Notes* 39, 1 (Feb. 2014), 1–5. https://doi.org/10.1145/2557833.2560580

[Shafiei and Mehlitz 2014] Nastaran Shafiei and Peter Mehlitz. 2014. Extending JPF to Verify Distributed Systems. *SIGSOFT Softw. Eng. Notes* 39, 1 (Feb. 2014), 1–5. https://doi.org/10.1145/2557833.2560577

[Yang et al. 2008] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. Efficient Stateful Dynamic Partial Order Reduction. In *Model Checking Software*, Klaus Havelund, Rupak Majumdar, and Jens Palsberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 288–305.

[Zhang et al. 2015] Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 250–259.