DPOR-DS: Dynamic Partial Order Reduction in Distributed Systems

Maysam Yabandeh and Dejan Kostić
School of Computer and Communication Sciences, EPFL, Switzerland
email: firstname.lastname@epfl.ch

Abstract

In this paper, we present DPOR-DS, an algorithm for dynamic partial order reduction in model checking of distributed systems. This work is inspired by the techniques introduced in the seminal work of DPOR [1] which is designed for multi-threaded systems. Different characteristics between distributed systems and multi-threaded systems raises new challenges for implementing the idea in distributed system domain. By developing techniques to address those challenges, we prove the soundness and completeness of DPOR-DS. The performance of DPOR-DS is then compared to exhaustive search and state-of-the-art heuristics. The experimental results show that even though dynamic partial order reduction can alleviate the exponential explosion problem of state space exploration algorithms, the exponential growth still shows up after some certain steps.

1 Introduction

Distributed systems are notoriously hard to debug. Several variety of approaches have been developed to address this problem. Model checking or systematic exploration of state space for the actual implementation of distributed systems is one important branch of approaches for tackling the difficulties in debugging distributed systems. However, the well-known problem of state space explosion makes the model checkers ineffective when it comes to the large state space of distributed system's implementations.

To address state space explosion problem, several potential solutions can be considered which can be categorized in two major groups: i) Partial Order Reduction (POR) techniques which remove the redundant states from the search tree without jeopardizing the completeness of the search, and ii) Heuristics which guide the search process towards more relevant states [11]. Although the state space explored by heuristics is not complete, they can be very effective in tackling the exponential state space explosion problem. On the other hand, the techniques based on POR are complete but the obtained performance is not good enough to be applied in model checking of large systems.

One other major problem of POR-based techniques is that they need to be supplied by dependency relations and persistent sets. This must be done by either the developer which is impractical for large systems or by static analysis of the source code. The latter also turns out to be very inefficient in practice as static analysis at compile time needs to be very conservative and hence the few inferred independency relations are not enough to combat the exponential explosion problem [1].

Recently, dynamic partial order reduction (DPOR) has been developed for model checking of multi-threaded systems which dynamically applies the POR technique at run-time. Leveraging the run-time information from actual interaction of threads, DPOR effectively applies POR techniques. More importantly, DPOR comes with proofs that the search algorithm is sound and complete for multi-threaded systems. The success of DPOR in multi-threaded systems has raised some interest to apply a similar technique on distributed systems as well [12]. However, because the system model in DPOR is different from distributed systems, any algorithm for model checking of distributed systems inspired from DPOR would lack the proof of completeness. Consequently, the inspired algorithms would degrade to the level of heuristics and would not be as trustworthy as DPOR.

In this paper, inspired from DPOR, we propose DPOR-DS, a new algorithm for model checking of distributed systems. DPOR-DS is sound and complete and these properties are guaranteed to hold through proofs. The rest of the paper is as follows: Section 2 demonstrates that why the proofs of DPOR can not be directly applied to distributed systems and hence why we need to design a new algorithm for distributed systems. Section 3 presents the general proposed algorithm, called DPOR-

DS. In Section 4 we present a precise version of the algorithm optimized for efficient implementation.

2 Background

In this section, we first give an overview of the DPOR algorithm. Then, we discuss the reasons that the proofs which come with DPOR can not be directly applied to the similar algorithms in distributed systems. Then, we explain the model of distributed system that our proposed algorithm assumes.

2.1 DPOR

In [1], the authors propose a new approach to partial-order reduction (i.e., DPOR) that avoids the inherent imprecisions of static alias analysis. DPOR starts by taking an arbitrary execution path of the program until completion. While executing executing each path, it dynamically collects information about shared memory locations that are read or written by threads and the order of these operations. Then, alternative transitions that need to be explored are identified by analyzing the collected data. Based on that, it adds backtracking points along the trace to explore the alternative transitions in the next round. The procedure is repeated until all alternative executions have been explored and no new backtracking points need be added. When the search stops, all deadlocks and assertion failures of the system are guaranteed to have been detected [1].

The operation of the algorithm is depicted by an example in Figure 1. The example includes events from two processes represented by arrows: the red arrow pointing to bottom-right and the blue arrow pointing to bottom-left. After exploring a path, it detects that two variables are changing the same location of memory and hence they are dependent. After that it adds a branch at point S, to explore the alternative transition as well.

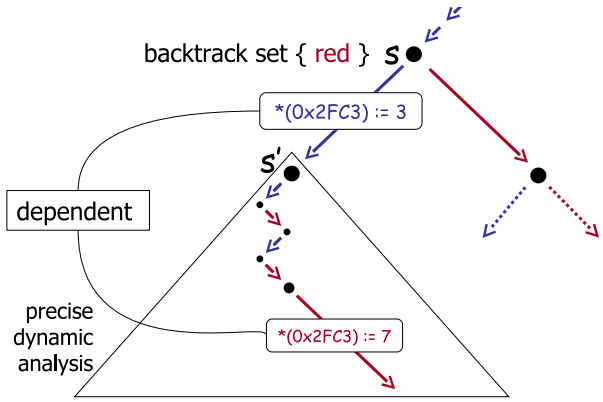

Figure 1: This picture illustrates the operation of DPOR. The example includes events from two processes represented by arrows: the red arrow pointing to bottom-right and the blue arrow pointing to bottom-left. After exploring a path, it detects that two variables are changing the same location of memory and hence they are dependent. After that it adds a branch at point S, to explore the alternative transition as well. *This picture is taken from www.eecs.umich.edu/acal/swerve/docs/117-2.ppt*

2.2 DPOR challenges for Distributed Systems

In this section, we explain why DPOR and its proofs are not directly applicable to distributed systems.

1. DPOR is designed for multi-threaded systems. In multi-threaded systems, there exists a concrete definition of shared objects between threads. In distributed systems, we need to first settle on a definition of the communication object. One candidate is the whole communication network. The large granularity of the communication object makes the POR techniques to be ineffective in practice. Alternatively, we can think of a receive buffer of each node as a communication object. The finer granularity is better but we need to be careful that it covers all the dependent events. For example, the above definition of communication object fails to consider the dependency of a local event (e.g. triggering a timer) with the network events (e.g. receipt of a message). The bottom-line is that the definition of communication object in distributed systems is neither straightforward nor unique and anyone can come up with a different definition.

2. DPOR deals with multi-threaded systems where each thread represents a sequential program. As a result, at each step, each thread can only run one event, i.e. running the next sequential command. Therefore, the whole algorithm of DPOR and its proofs are based on this assumption. In distributed systems however, at each point in time each process can run several events: trigger a scheduler, handle an application request, receive one of the ready network messages, and etc. This makes the DPOR algorithm directly unusable for distributed systems. Furthermore, the proofs provided for DPOR would not be valid for any other similar algorithm designed for distributed systems. Later, in Section 3 we show that this property of distributed systems brings up a completely new challenge in design of DPOR-DS.

3. As we explained above, in distributed systems there are several events that can run at each step and considering the bounded depth of search, it is not guar-

anteed that all of them have a chance to appear in the bounded path. Hence, adding the backtrack point does not guarantee that the algorithm explores the different orders of the current events at the next round. For example, assume that event e_1 appears in the bounded path before e_2 and we add a backtrack point before e_1 to explore the other order. However, after exploring e_2 in the added backtrack point, there might be other events in the ready event list and there is no guarantee that the algorithm picks e_1 from them before reaching the bounded depth. In Section 3 we present a novel technique to address this problem.

4. In DPOR, there are two types of operations: visible and invisible. Visible operations are the ones which are performed on shared objects. Each transition is defined to start with a visible operation, and continued by a set of invisible operations. However, this does not comply with state machine-based distributed systems, where a transition is defined by operations run by a handler. For example, if we consider the visible operation of receiving a message, the receive handler can also include commands for sending some other messages. In other words, the transition starts with a visible operation and ends with another visible operation (send message) as well.

5. The happens-before definition in DPOR is different from happens-before definition in distributed systems. For example, in DPOR if two events are dependent, they are then considered as having happens-before relation. However, in distributed systems two events e_1 and e_2 can be received by the same node (i.e. are dependent) but there is not necessarily any strict happens-before relation between them. Later in this section, we present a precise definition of happens-before in our model.

6. DPOR assumes that the state space is an acyclic graph. However, we can never make such an assumption in distributed systems.

7. It assumes that in the first round, the algorithm finishes a path from start to end; i.e. after executing the last transition there is no other transition enabled. In other words, it implicitly assumes that exploring the whole state space by the algorithm is feasible. However, we know that the complexity of the protocols in distributed systems, makes these kinds of model checking impossible; i.e. considering the time limit, because of the large state space, it is not feasible to reach to bottom of the search tree. Actually, that is why we have to implement bounded depth search in model checking of distributed systems.

It is also worth-noting that DPOR is a state-less algorithm, however wherever it is possible to have stateful search, one can redesign the algorithm to take advantage of this feature. One example is MaceMC [4] which implements a stateful search algorithm.

2.3 System Model

We next present a simple model of distributed systems and present a precise definition of happens-before relation based on this model. We concentrate on distributed systems implemented as state machines, as this is a widely-used approach [3, 5, 6, 9, 10].

Figure 2 describes a simple model of a distributed system. We use this model to describe system execution at a high level of abstraction.

System state. The state of the entire distributed system is given by 1) the local state of each node, and 2) the in-flight network messages. We assume a finite set of node identifiers N (corresponding to, for example, IP addresses). Each node $n \in N$ has a local state $L(n) \in S$. Local state models all node-local information such as explicit state variables of the distributed node implementation, the status of timers, and the state that determines application calls. Network state is given by in-flight messages, I. We represent each in-flight message by a pair (N, M) where N is the destination node of the message and M is the remaining message content (including sender node information and message body).

Node behavior. Each node in our system runs the same state-machine implementation. The state machine is given by two kinds of handlers: a message handler executes in response to a network message; an internal handler executes in response to a node-local event such as a timer and an application call.

We represent message handlers by a set of tuples H_M. The condition $((s_1, m), (s_2, c)) \in H_M$ means that, if a node is in state s_1 and it receives a message m, then it transitions into state s_2 and sends the set c of messages. Each element $(n', m') \in c$ is a message with target destination node n' and content m'. Internal node action handler is analogous to a message handler, but it does not consume a network message. Instead, $((s_1, a), (s_2, c)) \in H_A$ represents handling of an internal node action $a \in A$. (In both handlers, if c is the empty set, it means that the handler did not generate any messages.)

System behavior. The behavior of the system specifies one step of a transition from one global distributed system state (L, I) to another global state (L', I'). We denote this transition by $(L, I) \rightsquigarrow (L', I')$ and describe it in Figure 2 in terms of handlers H_M and H_A. The handler that sends the message directly inserts the message into the network state I, whereas the handler receiving the message simply removes it from I. To keep the

model simple, we assume that transport errors are particular messages, generated and processed by message handlers.

basic notions:
N – node identifiers
S – node states
M – message contents
$N \times M$ – (destination process, message)-pair
$C = 2^{N \times M}$ – set of messages with destination
A – local node actions (timers, application calls)

system state : $(L, I) \in G,\ \ G = 2^{N \times S} \times 2^{N \times M}$
local node states : $L \subseteq N \times S$ (function from N to S)
in-flight messages (network) : $I \subseteq N \times M$

behavior functions for each node :
message handler : $H_M \subseteq (S \times M) \times (S \times C)$
internal action handler : $H_A \subseteq (S \times A) \times (S \times C)$

transition function for distributed system :

node message handler execution :

$$\frac{((s_1, m), (s_2, c)) \in H_M}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I_0 \uplus \{(n, m)\}) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I_0 \uplus c)\end{array}}$$

internal node action (timer, application calls) :

$$\frac{((s_1, a), (s_2, c)) \in H_A}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2\}, I \uplus c))\end{array}}$$

Figure 2: A Simple Model of a Distributed System

Happens-before. As mentioned before, the event in our model can be an application call, running a timer, and receiving a message. Each of these events can change the local state of the node beside sending some new messages to the network.

Inspired by [5] we define happens-before relation which matches the presented system model. The relation "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following three conditions: i) If e_1 and e_2 are events in the same process, and e_1 comes before e_2, then $e_1 \rightarrow e_2$. ii) if by handling event e_1 the following transition happens: $(L, I) \rightsquigarrow (L', I \uplus c))$, and $e_2 \in c$, then $e_1 \rightarrow e_2$. iii) If $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ then $e_1 \rightarrow e_3$.

3 DPOR-DS Design

In this section, we present DPOR-DS, dynamic partial order reduction for model checking distributed systems. DPOR-DS checks for happens-before and dependency relation between events in the explored path and based on that decides which other sequences need to be explored to have the search complete. We first, give an overview of happens-before relation and its implication for search algorithms. Then, we formally define the dependency relation in distributed systems. Finally, based on happens-before and dependency relations we present the design of DPOR-DS.

If $e_1 \rightarrow e_2$, it intuitively means that e_2 can not exist in the system if e_1 does not happen. Hence, if want to check an alternative ordering of events in which e_0 happens before e_1 in the sequence and e_2 happens before e_0, then it is necessary (but not sufficient) to handle e_1 before e_0, because it is e_1 that leads to generation of e_2 and e_2 can not show up in the trace before e_0 if e_1 is not explored yet. The other implication of happens-before relation is that if $e_1 \rightarrow e_2$ and also e_1 and e_2 are dependent, then it is does not make sense to check the other order of these two events, because e_2 can not exist before e_1.

3.1 Dependency relation

If two transition are independent, it means that their execution does not interfere with each other. In other words, the order of their execution does not change the final state of the system and hence it is sufficient for the search algorithm to explore only one ordered sequence of them. Next, we present the formal definition of dependency relation (and its dual, independency relation) which is adapted from [2].

Definition 1. Let τ be the set of transitions of a concurrent system and $D \subseteq \tau \times \tau$ be a binary, reflexive, and symmetric relation. The relation D is a valid dependency relation for the system *iff* for all $t_1, t_2 \in \tau, (t_1, t_2) \notin D$ (t_1 and t_2 are independent) implies that the two following properties hold for all states (L, I) in the state space of the system:

1. if t_1 is enabled in (L, I) and $(L, I) \overset{t_1}{\rightsquigarrow} (L', I')$, then t_2 is enabled in (L, I) iff t_2 is enabled in (L', I'); and
2. if t_1 and t_2 are enabled in (L, I), then there is a unique state (L', I') such that $(L, I) \overset{t_1 t_2}{\rightsquigarrow} (L', I')$ and $(L, I) \overset{t_2 t_1}{\rightsquigarrow} (L', I')$.

Although complete, this definition is not efficient for checking at runtime; to make sure that two events are independent we need to explore them on all the possible states, obtain the resulting states, and finally check their equality. Alternatively, we can define some conditions that are *sufficient* to make two events independent. These conditions can be manually defined by the user (which is impractical) or obtained through static analysis of the source code (which is inefficient as explained before).

In DPOR-DS we take the middle point and define conditions on the behavior of the events and at runtime we check these conditions on the events. A well-known example is manipulation of the shared objects; if two events manipulate two different set of objects, then they are independent in that particular state.

Note that we need to be conservative on definition of shared objects. In other words, the definition must announce all the dependent events as dependent and it might conservatively declare some independent events as dependent as well. The more accurate the dependence conditions, the more efficient the search algorithm is, because it ignores more sequences of partial orders. For example, we can take the whole network as the shared object. Then, the algorithm would be complete but inefficient, because all the events which send a message to the network are considered as dependent.

The shared object o that we took into consideration in this paper is the union of the message receive buffer for each node and its local state. Formally, we have:

$o \in O, O = \{(s_i, (i, m)) | i \in N, m \in M\}$,

where N is the set of nodes, M is the set of message contents, and the pair (i, m) is a message to destination i. Note that this definition is sufficient for two events to be independent because if two events e_1 and e_2 are handled by two different nodes, then there is nothing shared between them to make them dependent. Nevertheless, one can take a finer granularity (by taking advantage of static analysis tools, perhaps) and make the dependence set smaller and therefore the algorithm more efficient.

3.2 The basic approach

After exploring a sequence of events, the approach is to look at the dependent events which can be reordered, i.e. there is no happens-before relation between them. Then, if two concurrent events e_i and e_j are independent and e_j is handled after e_i in the explored sequence, then we need to explore another sequence in which the e_j is handled before e_i. To this aim, we look for the event with the smallest index in the explored sequence after e_i where $e_k \rightarrow e_j$. Then, we add a branch before e_i to let e_k to be explored before that. The reason is that we know i) $e_k \rightarrow e_j$ and ii) we want e_j to be explored before e_i. Consequently, it is a necessary (but not sufficient) condition to explore e_k before e_i. We call this simple approach as DPOR-naive.

Figure 3 illustrates the applying above approach, i.e. DPOR-naive, on a simple example. In this example $e_2 \rightarrow e_3$ and $e_1, e_3 \in D$. Hence after exploring the first sequence that e_1 is handled before e_3, it is desirable that we explore another sequence in which e_3 is handled before e_1. According to DPOR-naive we simply add a branch for e_2 before exploring e_1. As shown in the picture, even though it finally leads to explore the desired sequence, it however explores the redundant path of $[e_2, e_1, e_3]$.

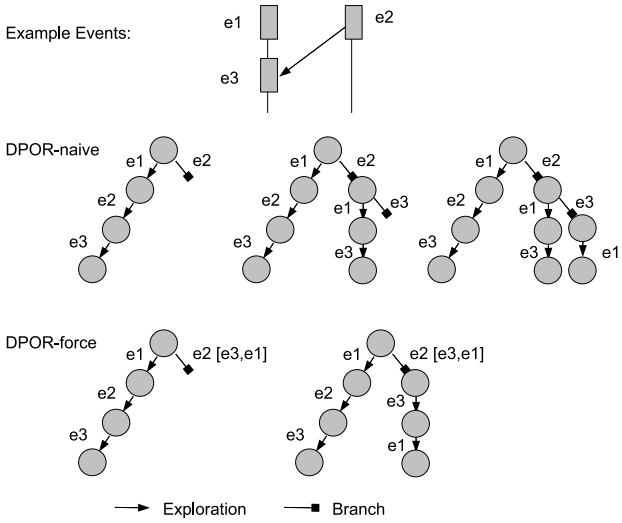

Figure 3: Applying DPOR-naive and DPOR-force on a simple example. In this example $e_2 \rightarrow e_3$ and $e_1, e_3 \in D$. Hence after exploring the first sequence that e_1 is handled before e_3, it is desirable that we explore another sequence in which e_3 is handled before e_1. According to DPOR-naive we simply add a branch for e_2 before exploring e_1. As shown in the figure, even though it finally leads to explore the desired sequence, it also explores the redundant path of $[e_2, e_1, e_3]$. By applying DPOR-force on the same example, we see the redundant path would be omitted because the desired path is forced during the exploration of the added branch.

Beside the redundant paths, DPOR-naive could suffer from incompleteness as it is explained in Section 2, point #3. In the example shown in Figure 3 suppose there is also another event e_4 in the ready event list. Because the search is bounded to depth 3, hence e_4 did not have a chance to be handled in the first explored sequence. In exploring the second sequence, the search algorithm might choose to explore e_4 instead of e_3 and hence neither the desired path would be explored nor there would be any dependency in the last explored path to cause adding more branches and exploring more.

We can improve the DPOR-naive algorithm by explicitly attaching the desired order in the added branch and force the algorithm to apply that order when it continues the search from branch point. For example, in Figure 3 the desired order after exploring the first branch is $[e_3, e_1]$ and we can add this information to the added branch. Later, when the algorithm expands the branch, it forces this order in the explored sequence. As illustrated in Figure 3, in this way we avoid the redundant explored path by DPOR-naive. Furthermore, it addresses the problem

```
1  proc DPOR−DS(n : int)
2    path = Explore an arbitrary path of size n;
3    tree = emptyTree();
4    tree.add(path);
5    addBranch(tree, path, 0, n−1, n);
6
7  //s is start of the current subtree,
8  //m is the branch point, n is the end
9  proc addBranch(tree, path, s, m, n)
10   subtree = tree.subtreeUnderEvent(path[m]);
11   foreach (e ∈ subtree)
12     if path[m] ↛ e and (path[m],e) ∈ D
13       subpath = tree.pathBetweenEvents(path[m], e);
14       subpath = subpath.reorderEvents(path[m], e);
15       //subpath is the desired order
16       tree.insertBranch(m, subpath);
17       newpath = exploreOpenBranches(tree);
18       addBranch(tree, newpath, m, n−1, n)
19     //to limit the branches to the new subtree
20   if (s < m)
21     addBranch(tree, path, s, m−1, n);
```

Figure 4: The pseudo code of DPOR-DS algorithm

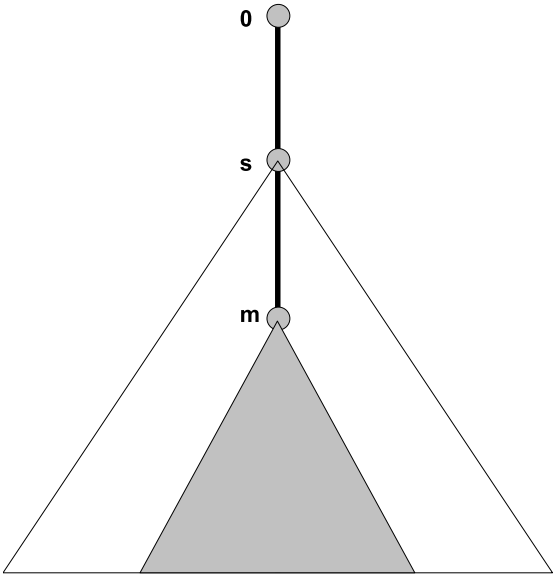

Figure 5: Schematic view of the precondition and postcondition of the procedure addBranch in Figure 4. The subtree under m^{th} event in the path is assumed to be explored and there is no further possible reordering of event which needs to be checked. The postcondition is represented by the subtree under s^{th} event in the path which will be explored in a way that it will not have any further possible reordering of event which needs to be checked.

of incompleteness mentioned in Section 2, point #3; the other unexplored events in the first sequence can happen in the expanded sequence by the branch as long as the desired order is forced in the new sequence. We call this approach DPOR-force.

3.3 DPOR-DS

In section, we present DPOR-DS based on the DPOR-force technique explained in the last subsection. The pseudo code is presented in Figure 4.

The procedure addBranch receives a path and assumes that the subtree under m^{th} event of the path has been explored before and there is no further possible reordering of event which needs to be checked. It then adds more branches and explores them until it makes sure that the subtree under s^{th} node does not have any further possible reordering of event which needs to be checked. The precondition and postcondition of the addBranch procedure is schematically depicted in Figure 5. The procedure DPOR-DS explores a random path and calls the addBranch procedure and passes this path as an argument to it. The explored subtree is the last event of the path (which obviously does not have any reordered alternative) and the subtree which needs to be explored is the whole tree.

In addBranch, first the explored subtree under m^{th} event is separated (Line 10). Then, for each event e in the subtree where e is dependent to m^{th} event and can be reordered with that (Line 12), the algorithm adds a branch before m^{th} event (Line 16). As explained before, we need to also attach the desired order which we want to be explored to the branch. This order is obtained by calling reorderEvents function on the subpath between m^{th} event and e (Line 14). Basically, this function looks for the event e_k with the smallest index in the subpath where $e_k \rightarrow e$. Because $path[m] \not\rightarrow e$ and also $e_k \rightarrow e$, hence $path[m] \not\rightarrow e_k$ as well. Thus, $path[m]$ and e_k can be reordered. The result of this reordering is attached to the added branch (Line 16). The open branch is explored in Line 17. The algorithm recursively calls addBranch for the new explored path. Afterwards, it recursively calls addBranch to extend its search by one event (Line 21).

The completeness of the above algorithm is established by the following theorem.

Theorem 1. Before calling the addBranch procedure in Figure 4, if the search in the subtree under the m^{th} event is complete, then the search in the subtree under s^{th} event will be complete after returning from the procedure.

Proof: See Appendix.

4 Implementation

In this section, we give an implementation of the DPOR-DS algorithm presented in Figure 4. Checking the happens-before and independency relation with all events in the subtree is not efficient because of duplicate

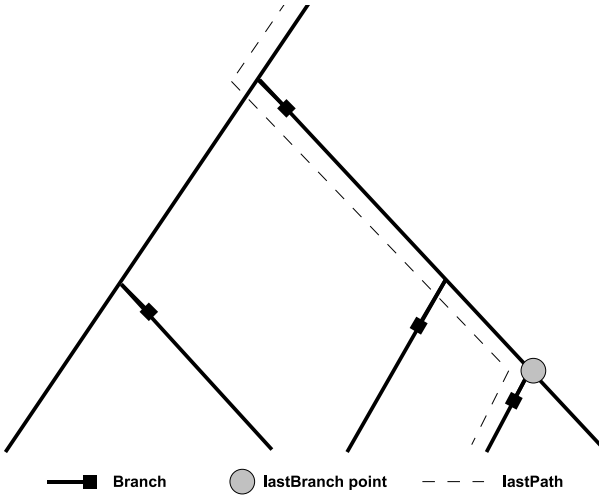

Figure 6: The notion of lastPath in DPOR-DS algorithm. The checking of dependency and happens-before relation needs to be done only on the lastPath (not on a whole subtree). Furthermore, lastPath has two parts; the sequence before the lastBranch and the sequence after that. The checks between events in the sequence before the lastBranch point is guaranteed to be done and hence can be skipped from the search.

```
1  proc DPOR−DS(n : int)
2    lastPath = Explore an arbitrary path of size n;
3    lastBranch = −1;
4    do {
5      //1) Add all branches
6      foreach (e_i ∈ lastPath[1..n−1])
7        s = max(i,lastBranch)+1;
8        foreach (e_j ∈ lastPath[s..n])
9          if (e_i ↛ e_j and (e_i,e_j) ∈ D)
10           hb_list = {e_k | i+1 ≤ k ≤ j and e_k → e_j};
11           subpath = hb_list.e_i;
12           //subpath is the desired order
13           lastPath.insertBranch(i, subpath);
14     //2) Update the last branch
15     lastBranch = last enabled branch in lastPath;
16     //3) Update the last path
17     lastPath = explore(lastPath, lastBranch);
18   } while (lastBranch > 0);
```

Figure 7: The efficient implementation of DPOR-DS algorithm

checks. To address that, we introduce the notion of last explored path (lastPath) and all the checking operations are done only on this path. This notion is illustrated in Figure 6.

Furthermore, lastPath has two parts separated by lastBranch; the sequence before the lastBranch and the sequence after that. The checks between events in the sequence before the lastBranch is guaranteed to be done and thus can be skipped from the search. Therefore, it is required to only check the relation between events on the second sequence and the events between the second sequence and the first sequence. Figure 7 presents the efficient implementation of DPOR-DS based on the explained notion of lastPath.

Initially, DPOR-DS explores a random path till depth n and set the value of lastBranch to -1 (Line 3). Then, the main loop of the algorithm starts at Line 4. In the main loop, it first checks every pair of events in the last explored path, i.e. lastPath (Line 6,8). However, as explained above, it is guaranteed that in lastPath, all the pairs of events before lastBranch are already checked. Hence, the algorithm skips them in the search (Line 7). If a given pair of events (e_i,e_j), where $i < j$, are dependent and they can be reordered, i.e. there is no happens-before relation between them (Line 9), then a branch is added to lastPath right before e_i (Line 13).

As explained before in Section 3, the desired order is also attached to the added branch. The desired order is obtained by selecting all the events e_k between e_i and e_j where $e_k \rightarrow e_j$ (Line 10). Because of the transitivity property of happens-before described in Section 2, $e_i \not\rightarrow e_k$ and they can be moved before e_i. The desired path is created then by concatenating e_i after the series of e_k (Line 11).

After adding the required branches, lastBranch is then updated to the last inserted branch of lastPath (Line 15). Also, lastPath is updated to the path obtained by expanding lastBranch (Line 17). The main loop of the algorithm continues till there is no unexpanded branch in the tree (Line 18).

5 Related Work

In this section, we list the related work to dynamically applying POR for software systems.

Multi-threaded programs Dynamic partial order reduction [1] is designed for multi-threaded systems to dynamically detect the dependency between operations and explore the alternative transitions according to them. It assumes that the state space of the program is loop-free and also small enough that can be explored completely by the algorithm. DPOR-DS is designed for distributed systems where at each step there are more than one event to explore. Moreover, DPOR-DS does not make any assumption on the size of the state space.

MPI programs There were several efforts in adopting the DPOR technique for MPI programs. Message Passing Interface (MPI) is a specification for an API that allows many computers to communicate with one another. It is used in computer clusters and supercomputers.

In the tool designed for model checking of MPI programs [8], a technique inspired from DPOR is applied. The tool focuses only on one-sided communication operations. MPI-2 defines three one-sided communications operations, Put, Get, and Accumulate, being a write to remote memory, a read from remote memory, and a reduction operation on the same memory across a number of tasks.

Others [7] try to formally define independency relation in MPI programs. They take a simplified model of the program and apply their modified DPOR on it. They also consider wild-card receive which is closer to semantics of communication in distributed systems. The lack of experimental results however, raises the questions about the effectiveness of the implemented version of DPOR.

DPOR-DS is designed for distributed systems in which at each step several events might be ready to run. The operations are more complex than simple one-sided operations of MPI programs. It also applies to actual implementation of distributed systems (not a model abstracted from the system).

Acknowledgments

This project is supported by the Swiss National Science Foundation (grant FNS 200021-125140).

References

[1] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[2] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101(2):337–359, 1992.

[3] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[4] Charles E. Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[5] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Com. of the ACM*, 21(7):558–565, 1978.

[6] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[7] R. Palmer, G. Gopalakrishnan, and R.M. Kirby. Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In *Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 43–53. ACM New York, NY, USA, 2007.

[8] S. Pervez, G. Gopalakrishnan, R.M. Kirby, R. Palmer, R. Thakur, and W. Gropp. Practical model-checking method for verifying correctness of mpi programs. *Lecture Notes in Computer Science*, 4757:344, 2007.

[9] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *NSDI*, 2004.

[10] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[11] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[12] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, April 2009.

6 APPENDIX: Proof of Theorem 1

In the addBranch procedure, the precondition is that the subtree under m*th* event is checked and there are no further events which are dependent and can be reordered. Assuming that, by induction we prove that the postcondition of the subtree holds: the subtree under s*th* will be completely checked in a way that there are no further events which are dependent and can be reordered.

The procedure addBranch has two parts: i) after the loop in Line 11 the subtree under $m-1$*th* event is checked, and ii) after the recursive call in Line 21, the rest of the tree up to s*th* event is checked. After the loop in Line 11, the relations between all events in the subtree and the m*th* event of the path are checked (Line 12) and the proper branches are added (Line 18). Therefore, we can conclude the following Lemma:

Lemma 1. If the precondition of the addBranch procedure holds, after the loop of Line 11, the relation between the m*th* event and the subtree under it is checked and there are no further events which are dependent and can be reordered.

Furthermore, because of the recursive call inside the loop in Line 18 for each added branch, the relations between each added branch, $path[m]_{b_i}$ and its corresponding subtree are checked.

Lemma 2. If the precondition of procedure addBranch holds, after the loop in Line 11, the relation between all added branches under $m-1$*th* event and their corresponding subtrees under them are checked and there are no further events which are dependent and can be reordered.

Lemmas 1 and 2 provide the first part of the proof: after the loop in Line 11 the subtree under $m-1$*th* event is checked. Also, the recursive call in Line 21 works like

a loop over all events from $m-1$*th* up to s*th* event of the path. Hence, if the precondition is true, the subtree under s*th* event of the path is checked by DPOR-DS.

Now we need to show that the precondition holds in all calls to addBranch function. There are three case: i) Line 5 where the subtree under m*th* event is a single event (i.e., $path[n]$ and hence there is no checking to be made; ii) Line 18 where the passed subtree is a single event (i.e. n*th* in the new explored path and hence there is no checking to be made; iii) Line 21 where the passed subtree is the subtree under $m-1$*th* event. According to Lemma 1 and 2, the relation between all children of the $m-1$*th* event and their corresponding subtrees are checked by the loop in Line 11. On the other hand, in the procedure addBranch, no new branch is added for the subtrees which are already checked (i.e., subtree under m*th* event). Consequently, the subtree passed as argument in Line 18 is completely checked.