Master research Internship

Internship report

Directed DPOR for the verification of asynchronous distributed systems

---

Domain: Distributed, Parallel, and Cluster Computing

Author:
Mathieu Laurent

Supervisor:
Martin Quinson (Myriads)
Thierry Jéron (Sumo)

Abstract: Software model checking is a formal method to verify the correctness of programs. In the recent years, it has been used to verify asynchronous distributed systems. However, dealing with the state space explosion is still a challenge. To do so, different approaches have been proposed. Among them, reduction techniques aim at narrowing the size of the exploration while staying sound. In an orthogonal manner, guiding techniques orient the search in hopes of finding a bug and end the exploration early. In this work, we present a novel approach that combines both reduction and guiding techniques. We also introduce new guiding strategies specifically designed for asynchronous distributed software. To demonstrate the effectiveness of our approach, we implemented the new algorithms in McSimGrid and evaluate their performances using Message Passing Interface (MPI) codes.

Contents

1 Introduction **1**

2 Context and state of the art **2**
- 2.1 Transition system and model checking 2
  - 2.1.1 Traces and action independence 3
  - 2.1.2 Principle of reduction 4
- 2.2 How to build the reduction dynamically 4
  - 2.2.1 Dynamic Partial Order Reduction 5
  - 2.2.2 Optimal Dynamic Partial Order Reduction 7
  - 2.2.3 Unfolding Dynamic Partial Order Reduction 9
- 2.3 Orienting the search with Directed Model Checking 11
  - 2.3.1 Guiding Strategies 12
  - 2.3.2 Valuations 15
- 2.4 Context 17

3 Contributions **17**
- 3.1 Multi-head dynamic partial order reduction 18
- 3.2 New guiding strategies 23
- 3.3 New valuations 24

4 Implementation **26**

5 Experimental Evaluation **29**
- 5.1 Test cases 29
- 5.2 Discussion 32

6 Future Work **34**

7 Conclusion **35**

1 Introduction

Distributed computation is now widely used. Sometimes because the data one has to treat are by nature distributed among many nodes. Cloud infrastructures, for example, are shared among different nodes and partially replicated. In such cases, it is crucial to avoid centralizing all the data on a single node due to considerations of time and resource efficiency. Other times it is the opposite: the data is originally centralized, but can be split temporarily and synchronized later. This is done to facilitate computationally intensive tasks that require lengthy calculations on each element of a matrix, such as solving differential equations in physical simulations. The common point in both situation is that the global algorithm run is distributed. Distributing the algorithm comes with an increased complexity of the global system. Locally everything is simpler: the data are smaller, the control flow only depends on a few variables; globally, not only do one have to ensure every local node is doing fine, one also have to assure everyone is doing it at the right time, communicating correctly with its neighbors and waiting if required. Synchronizing the processes is crucial, and it can be achieved in different ways. In our research, we will focus on verifying distributed programs, and more specifically, those using the Message Passing Interface (MPI) [1] library.

To do so, different techniques have been developed. Model checking is one of them [2]. It allows to formally verify whether a given property holds on a system, represented by a model. Software model checking is the extension of this technique to verify software programs. The soundness of the method helps to discover bugs in cases where even intensive testing fails. With software model checking, one constructs a direct model by considering all the possible configurations and builds the transitions corresponding to the semantics of their program. Doing so already leads us to a state space explosion: A ten-line program can result in a model with thousands of states. Moreover in distributed computing, there are multiple nodes, running different lines of code with complex communication patterns. Hence an other factor of state space explosion.

To mitigate the challenges posed by state space explosion, the literature offers various types of solutions. One commonly employed approach in software model checking is the use of partial order-based reduction techniques. These techniques leverage the independence between specific actions to significantly reduce the size of the state space, while ensuring the soundness of the approach. Another orthogonal approach is guided model checking. Instead of reducing the state space, this technique focuses the search towards specific paths that appear more promising. If the model is free of flaws, the exploration process may not be faster. However, if there are issues in the program, the search can be greatly accelerated by directing it towards potential problematic behaviors.

In this work, we consider an approach not fully explored yet that consists in combining both reduction and guiding techniques. While there have been some examples of this approach in the context of static verification, its application to dynamic reduction techniques and guiding strategies specific to asynchronous distributed algorithms remains unexplored. This association presents several challenges. Firstly, ensuring the correctness of the reduction procedure while enforcing specific choices is a significant challenge. Additionally, obtaining sufficient information from abstract states to make relevant choices poses another difficulty.

This internship report provides an overview of the current state of the art in reduction and guiding techniques for model checking, along with a concise description of the underlying model. Then, it presents the new ideas we have explored to combine these methods effectively with a discussion of our preliminary results. Finally, we conclude by examining potential leads for further research and the continuation of our work.

2 Context and state of the art

This section introduces the concept of labelled transition systems and software model checking. With that, we will be able to present state-of-the-art dynamic algorithms doing partial order reductions of the state space. Lastly, we will introduce guiding techniques used to speed up the exploration.

2.1 Transition system and model checking

We want to model the behaviour of our programs as graphs, or more precisely with a structure close to an automaton which are Labeled Transition Systems (LTS) [2]. An LTS $\mathcal{M} = (S, Act, S_0, \mathcal{T}, AP, \mathcal{L})$ is defined by a set of states S, a sub-set $S_0 \subseteq S$ the set of initial states, Act the set of possible actions in the model, a transition relation $\mathcal{T} \subseteq S \times Act \times S$, a set of atomic propositions AP that will help to identify desired behaviours, and a labelling function $\mathcal{L} : S \to 2^{AP}$ that will give a value to each atomic proposition in each state. When working with distributed programs, different process are running at the same time. To encode that into the LTS, we will use the action in Act. Any action $a \in Act$ will contain the information about the process executing it. We only allow one process to execute at each step (each action only refers to a single process). For $a \in Act$ (resp. $t \in \mathcal{T}$), we will note $proc(a)$ (resp. $proc(t)$) the process associated to the action (resp. the transition). Moreover, we suppose that our processes are deterministic meaning that from a given state $s \in S$, there is, at most, one action a per process. This supposition makes sens with MPI programs we are studying but it not always true with other models. Finally, a program always starting with the same memory layout, we will consider that S_0 is a singleton $\{s_0\}$.

For sake of generality, we should now speak about how we want to represent the property to verify. One way to do that is to encode the property in Linear Temporal Logic (LTL), a logic based on propositional formulas, enriched with operators describing temporal behaviors. From such a formula, we derive an automaton accepting infinite sequences satisfying the given property. Such automaton is known as a Büchi automaton and is expressed as a regular finite automaton, the main difference being that to be accepted, an execution has to pass by a final state an infinite number of times. When using LTL, one actually constructs the automaton corresponding to the negation of the desired property, so that when taking the synchronous product with the LTS, the model checking algorithm only have to check whether the language of the obtained automaton is empty. If not, it means there exists an execution of the model that violates the desired property.

In the end, there are two types of properties one may want to verify: safety or liveness. The first one can be seen as: "we never want this to happen". This kind of property is both very common (think about deadlocks, out-of-bound access, use before declaration, etc.) and easy to check. Indeed, it is a simple check of reachability within the state space of the program possible behaviors. If one expresses $\mathcal{B}$ the set of states having the undesired behavior, usually called **Bad States**, we can use the general Model Checking algorithm expressed in Algorithm 1 to verify a safety property. In our definition, this simply consists in having $AP = \{b\}$ such that: $\forall s \in S, \mathcal{L}(s) = b \iff s \in \mathcal{B}$. It proceeds by maintaining a set of opened states that have been reached. When taking a state from $Open$, we first verify it does not violate our property (it is not in $\mathcal{B}$), then we add its successor not yet reached to $Open$. Finally, we close the state just explored.

Liveness is a bit trickier to express, and to check. Liveness properties speak about things that have to be done repeatedly e.g., every time the elevator button is pressed, it will come to the corresponding floor at one point. If we want to verify this, we will have first to reach the condition,

and then, decide if it is possible to fulfill a property from this condition. As stated before, we will focus on safety properties and will not come back to the LTL formulation of the problem.

Algorithm 1: General Model Checking algorithm for safety properties

Data: LTS $\mathcal{M} = (S, Act, S_0, \mathcal{T}, AP, \mathcal{L})$, set of bad states $\mathcal{B}$
Result: true if property is satisfied, a counterexample if not

```
1  Closed ← ∅; Open ← S_0;
2  while Open ≠ ∅ do
3      S ← Select(Open);
4      Closed ← Closed ∪ S; Open ← Open \ S;
5      if S ∩ B ≠ ∅ then
6          return GeneratePath(S ∩ B)
7      end
8      Succ ← {s' | s ∈ S, s → s'};
9      Succ ← Succ \ Closed; Open ← Open ⋃ Succ;
10 end
11 return true
```

2.1.1 Traces and action independence

To express how the different reduction algorithms work, we need to specify some notations and definitions. For this section, we consider a LTS $\mathcal{M} = (S, Act, S_0, \mathcal{T}, AP, \mathcal{L})$. A **transition sequence** $E \in \mathcal{T}^*$ is a finite sequence of transitions $t_0 t_2 \dots t_{n-1} = (s_0, a_0, s_1) \dots (s_{n-1}, a_{n-1}, s_n)$ such that $s_0 \xrightarrow{a_0} s_1 \dots \xrightarrow{a_{n-1}} s_n$ and $s_0 \in S_0$. Intuitively, E is a possible execution of our model from the beginning. Given E a transition sequence, we will denote by:

- $E_i := t_i$, the ith transition,
- $E.t := t_0 \dots t_{n-1} t$, the concatenation of E with transition $t = (s_n, a_n, s_{n+1})$,
- $dom(E) := \{0, \dots, n-1\}$, the range of possible transitions in E,
- For $i \in dom(E)$, $sub(E, t_i) := t_0 \dots t_{i-1}$, the prefix of E, up to t_i, not included,
- For $i \in dom(E)$, $pre(E, t_i) := s_i$, the state reached after executing $sub(E, t_i)$,
- $last(E) := s_n$, the last state of the sequence.

In our model, processes are deterministic. At a given state $s \in S$, there is at most one transition per process. For this reason, we will sometimes use the process to speak about the corresponding transition in a given state. Two transitions t_1, t_2 of $\mathcal{M}$ are said to be **independent** if they verify the following properties:

- if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s',
- if t_1 and t_2 are enabled in s, then there exists two intermediate states v, w and a unique state s' such that $s \xrightarrow{t_1} v \xrightarrow{t_2} s'$ and $s \xrightarrow{t_2} w \xrightarrow{t_1} s'$.

If not, t_1 and t_2 are said to be **dependent**, noted $D(t_1, t_2)$. From the notion of dependence, we need to introduce the concept of **happens-before** which is a bit stronger. $\rightarrow_E$ is a relation between transitions of a transition sequence that forms a partial order over them. Formally, given an execution sequence E, $\rightarrow_E$ is the smallest relation over $\{t_1, \ldots, t_n\}$ such that:

- if $i \leq j$ and $D(t_i, t_j)$ then $t_i \rightarrow_E t_j$,
- $\rightarrow_E$ is transitively closed.

Two transition sequences describing the same happens-before relation are equivalent regarding the properties we want to verify, and we say they have the same **Mazurkiewicz trace**: Mazurkiewicz traces form equivalent classes for the relation happens-before. All equivalent sequences can be obtained by swapping adjacent independent transition from a given trace; if E, E' are two such sequences, we note $E \simeq E'$ the fact that they are equivalent.

2.1.2 Principle of reduction

As we saw before, the size of the state space obtained from a given parallel program can be too large to explore. To tackle this issue, we are going to introduce reductions. Intuitively, reductions are ways to define transitions that can be forgotten without loss of generality in the behaviors of the program. Mathematically speaking [3], if $\mathcal{M} = (S, Act, S_0, \mathcal{T}, AP, \mathcal{L})$ is an LTS, we say that an action $a \in Act$ is **enabled** in $s \in S$ if there exists $s' \in S$ such that $(s, a, s') \in \mathcal{T}$, and we call **reduction** a function $r : S \rightarrow 2^{Act}$, that-is-to-say, a function mapping every state of $\mathcal{M}$ to a sub-set of enabled actions. With such a function r, one can naturally define a reduced model $\mathcal{M}_r = (S_r, Act, S_0, \mathcal{T}_r, AP, \mathcal{L})$. It is a model verifying: $S_r \subseteq S$ and $(s, a, s') \in \mathcal{T}_r \iff (s, a, s') \in \mathcal{T} \wedge\ a \in r(s)$. That second condition means we are restricting the possible transitions to the one given in our reduction function r. Finally, we want the reduction to preserve the property we are checking, more precisely, if we have a property ϕ, we want that $\mathcal{M} \vDash \phi \iff \mathcal{M}_r \vDash \phi$. While constructing the function r, or the model $\mathcal{M}_r$ directly, all techniques are permitted. **Partial order reduction** [2] is one of those techniques. It leverages the fact that in real life systems, most actions are independent to one another. The reduction algorithms aim at keeping the fewest execution sequences possible while assuring to visit at least one per Mazurkiewicz trace.

2.2 How to build the reduction dynamically

Most of the time when considering a program, we do not want to build the whole model. Therefore, we do not have access to the whole state space for the reduction computation. To solve that, we will instead start the exploration, and decide, when considering possible transitions, whether we can cut and not explore it. Furthermore, since we are here working with programs and not simpler models, whether two given state are the same is not trivial. In practice, we will assume every encountered state is a new one and not try to record anything about it. This principle is called **stateless model checking**. This can be an issue when considering cyclic programs since those could be blocked in a livelock. We will for now assume that we work only with acyclic state spaces.

In this section, we present different methods to construct reductions dynamically.

2.2.1 Dynamic Partial Order Reduction

The aim of the dynamic partial order reduction (DPOR) Algorithm [4] is to construct for each explored node a **persistent set** of enabled transitions, and explore it. We say that a set $P \subseteq Act$ of enabled transitions in a state s is persistent in s iff for all nonempty sequences $t_0 \dots t_{n-1}$ starting in s such that $t_0, \dots, t_{n-1} \notin P$, t_{n-1} is independent with all transitions in P. The interesting property is that exploring a persistent set of every reached state during the exploration is sufficient to explore every Mazurkiewicz trace of the system. To understand this, we need to have a look at the actions that are not part of a persistent set. Let us take s a reached state, and $P(s)$ the persistent set we explored. First, note that $P(s) = enabled(s)$ is a possibility; but the persistent set can be smaller. In that case, if there exists such t enabled but not in the persistent, then by definition of the persistence, in particular, t is independent with every transition $t_p \in P(s)$. That gives us the existence of the transition sequence $t_p t$ from s (by definition of independence). And more generally, for any sequence E not directly reachable with $P(s)$, the persistent definition gives us the reachability of any $t_p E$ sequence.

As stated, persistent sets are sufficient to explore the model, but it can still be way larger than what is necessary. Another dynamic reduction idea developed in [4] is the notion of **sleep set**. The idea behind sleep sets is not to avoid taking independent paths from a given node, but to remember for the future of the exploration that we already swap those two actions. A sleep set is a set of processes from which we do not have currently (during the exploration) to pick an action because we already covered the resulting sequence previously. Initially, the sleep set is empty. Then we perform a recursive search, just like in Algorithm 1. We go down a first branch, and when getting back to the origin, before testing another branch, we add the transition t of the first branch we just covered to our sleep set. As long as the exploration takes independent transitions with the one in the sleep set, it remains in it. Since they are independent, executing t now leads to a sequence that is equivalent to one explored in the previous subtree. If the transition is dependent, there might exist a **race condition** in the execution, and we absolutely want to test the sequence in which the two actions are reversed. Sleep sets, just like persistent sets, are proven to be sufficient to explore the whole set of Mazurkiewicz traces, but still not necessary. In fact, one can combine the two approaches in the same algorithm so that at a given state, they only explore transitions not in the sleep set that form a persistent set.

Algorithm 2 gives a general outline of dynamic partial order reduction algorithms. The algorithm is a recursive procedure $Explore(E, parameters)$ where E is the current transition sequence from which we want to explore successors, and `parameters` is a way to encode how different algorithms will treat the transition to explore (as a function here). The first call to this would be for instance $Explore(\emptyset, DPOR)$. The general idea is to have a structure that will retain for each sequence which transition we decide to explore on the state reached after this sequence. Initially it is empty for every sequence. When reaching a sequence for the first time, we choose some arbitrary transition and add it to the exploration. While exploring the children of the transition, we might add some other transition that we will need to explore to search for the entire state space. Last thing to notice in the procedure is the way we decide to add some transition to the structure. This procedure is highly dependent on the reduction algorithm and the construction used. For instance in [4], the conditional `add` is described in a way that when we are done exploring the sequence E (and so with any of its sub-sequence), $to_visit(E)$ is forming a persistent set of $enabled(last(E))$. The authors do it in two steps:

Algorithm 2: General DPOR procedure: $Explore(E, parameters)$

Data: E the current transition sequence, $parameters$ a function encoding informations relatives to the reduction algorithm, global data structure to_visit over sequences

Result: Exploration of every Mazurkiewicz trace of transition sequence from E

```
1  if to_visit(E) = ∅ then
2  |  choose some t ∈ enabled(last(E)) and add it to to_visit(E);
3  end
4  done = ∅;
5  while ∃t ∈ parameters(to_visit(E)) \ done do
6  |  if t is dependent with some transition t' in E and a sequence with t.t' has not been
   |    considered then
7  |  |  add a transition to to_visit(sub(E, t')) to explore a sequence starting with transition
   |  |    t;
8  |  end
9  |  Explore(E.t, parameters');
10 |  add t to done;
11 end
```

- They look for the closest ancestor t' in E verifying $D(t', t)$, and t is not on the same process as t' or another transition dependent with t' ($t' \not\rightarrow_E proc(t)$). Intuitively, this is because we do not want to try to invert transitions that are anyway forced by the order of execution inside the same process.
- For every $u \in enabled(pre(E, t'))$, we check if $u \rightarrow_E proc(t)$ i.e., if transition u is required to execute t, and therefore should be executed to reach an execution with t before t'. If there are such u, we choose one and add it to $to_visit(sub(E, t'))$. If we could not determine such u, it means that the dependence might be found later in the execution, but we can not determine already where. To remain sound, we must take $to_visit(sub(E, t')) = enabled(pre(E, t'))$.

Figure 1 gives an execution of this algorithm on a small example: DPOR will explore a first sequence until reaching a *terminal state* i.e., a state with no transition enabled. When encountering q_2, it detects the dependency with r_2 and so adds q_1 to the set of transition to be explored at the state before r_2 was taken. The algorithm proceeds to obtain a second execution, detects the same problem, and add q_2 to a set to be explored. When explored, this last addition allows doing the inversion between q_2 and r_2. In the end, three executions were explored ($r_1 r_2 q_2 q_2$, $r_1 q_1 r_2 q_2$ and $r_1 q_1 q_2 r_2$) while only two were necessary (note that the first two executions belong to the same Mazurkiewicz class).

Finally, sleep sets are added to the algorithm 2 by modifying $parameters$: when looking for a $t \in parameters(to_visit(E))$, we are setting aside any transition in the current sleep set and when calling $Explore(E.t, parameters')$, we update $parameters$ to add and remove transitions from the sleep set as described before: we add every process of transitions in *done* to the sleep set and we remove any u whose transition is dependent with the newly explored t. In the example given in Figure 1, sleep sets are enough to obtain optimality: in the second step, when exploring $r_1 q_1$, the transition r_2 will be a member of the sleep set (since we already explored it at this level, and it is independent with q_1). Therefore, DPOR will not try to explore any sequence starting with $r_1 q_1 r_2$

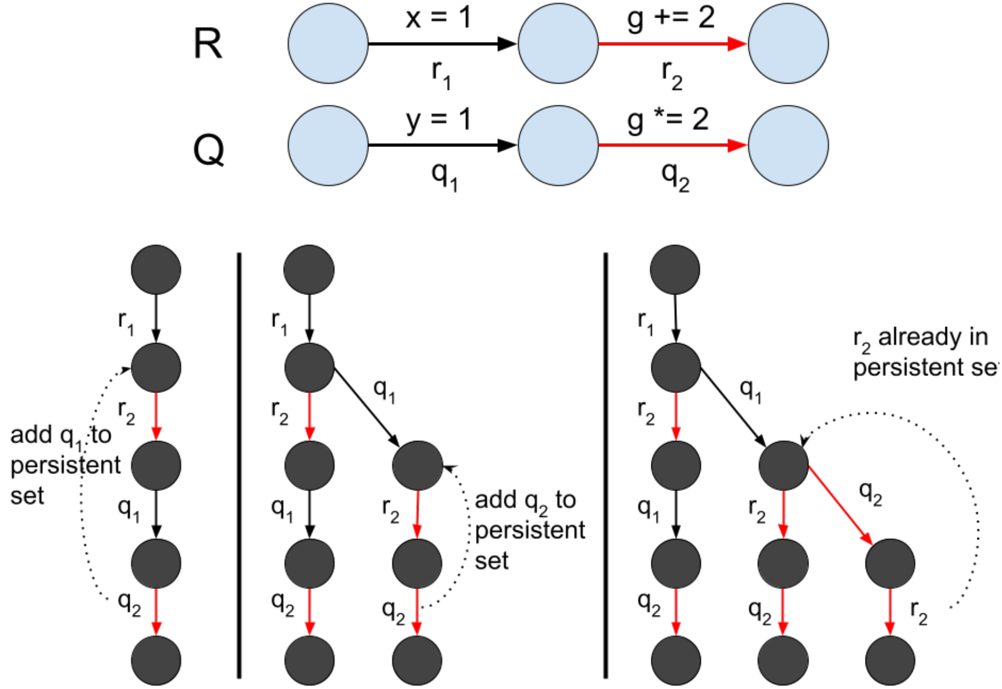

Figure 1: An example of the DPOR algorithm on a small concurrent program with two processes R and Q. r_2 and q_2 are the only dependent transitions.

and will directly explore $r_1 q_1 q_2$ by picking q_2, the only enabled transition at this point, not in a sleep set.

2.2.2 Optimal Dynamic Partial Order Reduction

As said before, DPOR is sufficient but not necessary. More explicitly, it guarantees to explore at least one execution per Mazurkiewicz class of equivalence, but it may explore multiples. Refining how we describe the set of transitions we want to visit at each node is a way to obtain the necessary part. Exploring fewer transitions without paying too much overhead allows to explore the state space faster. **Source sets** [5] are one way to go. Intuitively, if E is a transition sequence and W a set of sequences starting in $last(E)$, $Source(E, W) \subseteq enabled(last(E))$ is intended to contain a starting happens-before transition of every sequence in W. Formally, we call **weak initials** of w after E, the set of transitions verifying: $WI_E(w) = \{t \mid \exists w', v : \ E.w.v \simeq E.t.w'\}$. Therefore, the authors define that $P \subseteq enabled(last(E))$ is a source set for W after E if for each $w \in W, WI_E(w) \cap P \neq \emptyset$. When implementing this, the authors use a stronger condition on initials defined as followed: $I_E(w) = \{t \mid \exists w' : \ E.w \simeq E.t.w'\}$. The interest is that this condition is easier to verify. They implement source-DPOR by modifying Algorithm 2 accordingly to these definitions. Source-DPOR is using sleep sets as explained in the previous section, and it takes the choice to add transitions to to_visit as follows:

- for each t' in sequence E dependent with the next chosen transition t, the algorithm extracts w a sub-sequence of E composed of the transitions in E that are independent with t'.

- from the obtained sequence, we want to be sure that the search is able to visit one Mazurkiewicz trace, therefore we check if $I_E(w.t) \cap to_visit(sub(E,t')) \neq \emptyset$. If that is not the case, we add some arbitrary element in $I_E(w.t)$ to $to_visit(sub(E,t'))$.

Source-DPOR is proven to be both sufficient and necessary in the sense that it fully explores exactly one execution per Mazurkiewicz trace of the model. Though it suffers from the **sleep set blocking** problem: there are some execution sequences that are equivalent to one already explored, and for which we will stop the exploration thanks to sleep sets. But this stop can happen after some transitions are taken while we could have already decided, thanks to other sequences explored, that we will be blocked by sleep set very soon. To deal with that, the authors propose to combine source sets with a tree structure called **wakeup tree**. Wakeup trees are trees labelled with sequences of processes. The root is always the empty sequence $\langle\rangle$, and a node w can only have as children a node of the form $w.p$, with p a process. Finally, we also need an order $<$ over processes, that we then extend to sequences of processes such that if $p_1 < p_2$ then for any w, $w.p_1 < w.p_2 < w$. Formally, B is a wakeup tree after a transition sequence E and a set of process P, if it verifies the following:

- for every leaf w of B, $WI_E(w) \cap P = \emptyset$.
- if $u.p$ and $u.w$ are nodes in B with $u.p < u.w$, $u.w$ being a leaf, then $p \notin WI_{E.u}(w)$ (for simpler notations, we use p here and later to describe the transition in the given state associated with process p).

The first condition has to be considered regarding the set P of processes we are choosing. To prevent the possibility of starting to explore executions that will be blocked by the sleep set P, we require that every weak initial of a sequence in the wakeup tree is not in the sleep set. The second condition states that if we want to add a new sequence $u.p$ at node u inside our wakeup tree due to the sequence we visited with $u.w$, then p should not be a weak initial of the sequence we just explored: in other words, we do not want to revisit a sequence that is equivalent to the one we just visited. To work with this structure, the authors explain how to write a simple primitive that allows to insert a sequence into an existing wakeup tree and preserve the properties (they use the fact that with the order $<$, we now have a canonical representative for each Mazurkiewicz trace). With this new structure, we can adapt once again the algorithm 2 to obtain Optimal DPOR:

- *to_explore* will now contain a wakeup tree for each transition sequence; initially, it will contain the empty wakeup tree for each sequence. We also maintain a sleep set as explained before.
- when exploring a new node for the first time, instead of always choosing a random enabled transition, we first try to use the wakeup tree that would have been constructed previously during the search. If the tree is still empty, then we pick a transition to add to the tree.
- when finding a race condition, instead of adding a transition from $I_E(w.t)$ to $to_visit(sub(E,t'))$, if it does not conflict with the sleep set (meaning if $sleep(sub(E,t')) \cap WI_E(w.t) = \emptyset$) we insert the whole sequence $w.t$ inside the wakeup tree at $pre(E,t')$.

If we have a look back at our example in Figure 1, when reaching the sequence $r_1r_2q_1q_2(= E)$ in the first step, we find that $q_2(= t)$ is dependent with $r_2(= t')$ previously taken. Therefore, we eventually need to add something to visit to the wakeup tree after sequence r_1. Currently, this wakeup tree is composed of the root $\langle\rangle$ and only has one child R that was added by the choice of an initial transition. Of the sequence $r_2q_1q_2$ (i.e., the sub-sequence containing both dependent

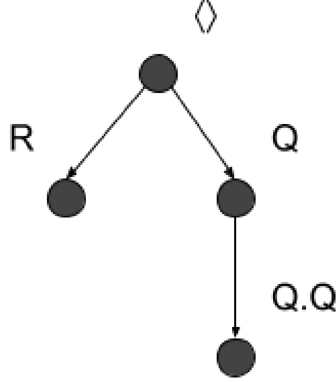

Figure 2: Wakeup tree at sequence r_1 after the execution.

transitions), we take transitions that are independent with r_2 to form the sub-sequence w of our algorithm: here $w = q_1$. We now add the processes of $q_1q_2(= w.t)$ to the wakeup tree after r_1 (we can verify that it is allowed: $sleep(sub(E, t')) = sleep(sub(r_1r_2q_1q_2, r_2)) = sleep(r_1) = r_2$ while $WI_E(w.t) = WI_{r_1r_2q_1q_2}(q_1q_2) = q_2$).

More recent papers on ODPOR propose to go even further on the optimization [6]: if we have a look at our computational model, there are some Mazurkiewicz classes that are distinct, but yet yield the same result regarding the properties we want to check. As an example, consider a C program that does 10 conditional `write`, and one single read at the end. We could consider all possibilities of write or not, giving us 2^{10} traces to explore, while the only `write` that really matters is the last one happening: it is the only `write` that will impact the rest of the program. To solve this issue, the authors introduce what they call **observers**. Intuitively, observers are specific transitions inside a sequence that observe the dependency between two other previous transitions in the sequence. If there are no transitions to observe the dependency, it is as if the transitions would be independent: if we swap them, it does not impact the correctness of the execution regarding our property. Observers are defined with the definition of happens-before, and are computed with the semantics of the operations of our program. The main idea now in the ODPOR algorithm is that when adding a sequence to a wakeup tree, if it is observed by some transition o, we are adding $w.t$ with all the transitions dependent with t that are not observed by o (e.g. the read before that does not change the final value). This way, when exploring the node that received this wakeup tree, we will be forced to explore only one fixed choice for all those non-observed transitions.

2.2.3 Unfolding Dynamic Partial Order Reduction

The principle of Unfolding dynamic partial order reduction [7] (UDPOR) is quite different from the other DPOR algorithms: it does not focus on transitions like previous DPOR algorithms but rather on **configurations** and **events**. In this section, we will take $\mathcal{M} = (S, Act, S_0, \mathcal{T}, AP, \mathcal{L})$ an LTS, with $D \subseteq \mathcal{T} \times \mathcal{T}$ its dependency relation. A **$\mathcal{T}$-labelled event structure** (LES) is a tuple $\mathcal{E} = (E, <, \#, h)$, where E is a set of events, the causality relation $<\subseteq E \times E$ is a strict partial order on E, $h : E \to \mathcal{T}$ labels events with a corresponding transition, and the **conflict relation** $\# \subseteq E \times E$ is symmetric, irreflexive and transitive. An LES must satisfy these two following properties: (1) for all $e \in E, \{e' \in E \mid e' < e\}$ is finite, and (2) for all $e, e', e'' \in E$, if $e\#e'$ and $e' < e''$ then $e\#e''$. (1) means that for a given event, there are only a finite number of events

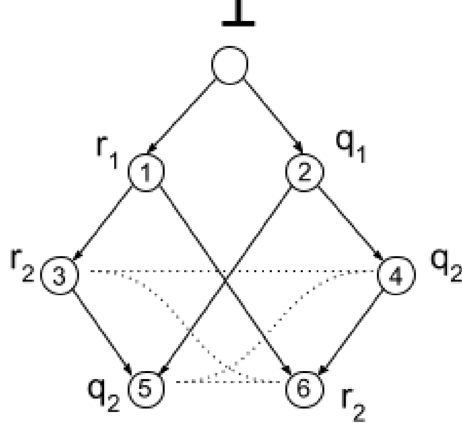

Figure 3: Unfolding semantic of the Figure 1 with events numbered from 1 to 6. Arrows represent causality; dotted lines conflict

happening before it (in other words, that the execution has a beginning); (2) states that conflict is inherited by causality. If e is an event, we call **causes** of e the set $\lceil e \rceil = \{e' \in E \mid e' < e\}$, i.e., the set of events that need to happen before e can happen. Finally, a configuration of $\mathcal{E}$ is a finite set $C \subseteq E$ such that: $\forall e, e' \in C, \lceil e \rceil \subseteq C \wedge \neg(e \# e')$. A configuration is closed by causality, free from conflict. With these definitions, we want to describe collections of them that intend to represent possible executions of our system.

Given $t \in \mathcal{T}$ a transition of our system, we now define the possible histories $\mathcal{H}_{\mathcal{E},D,t}$ of t as a set that contains the configuration H of $\mathcal{E}$ such that: t is enabled at the state reached after transitions in H, and either $H = \{\bot\}$ or for all $<$-maximal events $e \in H$, $D(h(e), t)$. $\bot$ is a special event that we add to our LES to be able to encode the start of the execution: it is unique and each event causally depends on it. This definition of **history** simply means that we want H to contain the required happens-before events for e to happen. We can now build the **unfolding** $\mathcal{U}_{\mathcal{M},D}$ of our model. It is a LES containing inductively defined events with a canonical name of the form $e := (t, H)$ where $t \in \mathcal{T}$ and H will be a configuration of $\mathcal{U}_{\mathcal{M},D}$. This is done inductively by adding events until saturation. We start with the LES $(\{\bot\}, \emptyset, \emptyset, h)$ with $h(\bot) = \varepsilon$. Then if we have $\mathcal{E} = (E, <, \#, h)$ an LES such that there exists $t \in \mathcal{T}$ and $\mathcal{H}_{\mathcal{E},D,t}$, we create $\mathcal{E}' = (E \cup \{(t, H)\}, < \bigcup_{e' \in H}(e', e), \# \bigcup_{e' \in \mathcal{K}_{\mathcal{E},D,e}}(e, e'), h')$ where $\mathcal{K}_{\mathcal{E},D,e}$ is the set of conflicting events with e in $\mathcal{E}$, $\forall e' \in E, h'(e') = h(e')$ and $h'(e) = t$. The idea behind the unfolding is that exploring it is by construction exploring one sequence per Mazurkiewicz trace.

Due to the structure of the unfolding, the partial order reduction algorithm derived from it (UDPOR), algorithm 3, is quite different from the previous general algorithm 2. To explain it, let $\mathcal{U} = \mathcal{U}_{D,\mathcal{M}}$ be the unfolding of $\mathcal{M}$ with D. Initially, we call $Explore(\{\bot\}, \emptyset, \emptyset)$. Parameter $\mathcal{D}$ is used to store events that have already been explored; it can be seen as the sleep sets we had before. A is the set of guiding events that will have to be taken first before trying other new branches; this is a sort of *to_explore* set. When called, the algorithm first computes the **extensions** of the configuration C $(ex(C))$. These are the events not yet in C but whose causes are already there: $ex(C) = \{e \in E \mid e \notin C \wedge \lceil e \rceil \subseteq C\}$. It forms the candidates from which we will pick the next transitions. If there are no enabled transition in the configuration that are not disabled by $\mathcal{D}$, then it means we reached a final branch and we backtrack. Else we pick a new event to be added to the

Algorithm 3: UDPOR procedure: Explore(C, D, A)

Data: C a configuration, $\mathcal{D}$ a set of disabled events, A a set of guiding events and U a global structure containing currently explored events

Result: Exploration of the unfolding from configuration C, without events in $\mathcal{D}$

```
1  Compute ex(C) and add all events in it to U;
2  if enabled(C) ⊆ D then
3  |  return
4  end
5  if A = ∅ then
6  |  choose e ∈ enabled(C) \ D;
7  end
8  else
9  |  choose e ∈ enabled(C) ∩ A;
10 end
11 Explore(C ∪ {e}, D, A \ {e});
12 if ∃j ∈ Alt(C, D ∪ {e}) then
13 |  Explore(C, D ∪ {e}, J \ C);
14 end
15 U = U ∩ Q_{C,D};
```

configuration, and if possible, we pick it in A. We recursively explore the configuration with the new event. Then comes the difficult task: we already explored every configuration containing C and e, so to explore all configuration containing C, we need to look for those not containing e. These are called **alternatives**. An alternative to $D' = D \cup e$ after a configuration C in a set of events U is a sub-set $J \subseteq U$ such that $J \cap D' = \emptyset$, $C \cup J$ is still a configuration and $\forall d \in D', \exists j \in J : \ d\#j$. If there exists such alternative, we explore it. Finally, we only keep in U events in C, D and those conflicting with events in either C or D (represented as $Q_{C,D}$).

The computation of $Alt(C, \mathcal{D} \cup \{e\})$ in the general case is NP-complete [8]. Solutions to this issue have been proposed, such as the **k-partial alternatives** which computes alternatives where events of J are only conflicting with k events of $\mathcal{D}$ instead of all. This allows for a fine-tuning between quasi-optimality and efficiency. Furthermore, computing $ex(C)$ is not trivial either: for instance the problem is also NP-complete when considering Petri nets. The authors of [8] show that in the case of a specific model of asynchronous distributed systems that can encode MPI programs, they can build a tuned version of algorithm 3 that works in time $O(n^2 \log(n))$ thanks to the fact that the amount of causal predecessors of a given event is bounded.

2.3 Orienting the search with Directed Model Checking

When using model checking, there are two possible outcomes: we can either prove the given property holds, or find a counterexample showing why it does not. Since our algorithm stops on the first bug found (that is what we are here for), we want to visit as few states as possible before reaching a counter-example. Some programs may have a state space that is too wide to be searched entirely. But we may have some hint about where to look, and that place might be small enough. And sometimes, when trying to find the bug faster we end up with a smaller counterexample, which can

be important for the programmer to understand more easily the reason of the bug and, hopefully, solve it faster.

Speaking about directing, there are two main things that can be described. First, you may want to have a global idea of where you are going. These can be complex strategies taking advantage of the whole exploration knowledge. These are what we will call **guiding strategies**. Then you may want to have the opportunity to decide between two given choices, and therefore you will need to score those choices according only to things from which the choices have been created. The way you want to evaluate the choices, in our case transitions to explore, are called **valuation functions**. This section presents both guiding strategies and valuation functions, as well as some usual combinations of both.

2.3.1 Guiding Strategies

The idea behind a guiding strategy is to choose which transition should be considered next in the execution tree being explored. For instance, **Depth First Search** (DFS) is an example of a usual guiding strategy: at any point, take the most recent reached state that has not been totally explored, and pick an arbitrary outgoing transition from it. It has interesting properties, such as ensuring that all opened states in the search are contained in a single sequence, and it is the natural design of the reduction algorithm we saw before. But one is not forced at all to preserve a single sequence of opened states as we will see. The strategies we will look at can be grouped in two distinct categories depending on whether it requires a valuation function to properly work. Let us first have a look at some strategies that do not require such function.

A basic and usual strategy after DFS is **Random**: simply choose the next transition randomly among all the currently opened ones. Not only among all the deeper state, but among any not yet visited transition. This has a few good properties. It can simulate real executions better than a more fixed strategy such as DFS. Furthermore it servers as a good baseline when some other strategies are failing. There is **no free lunch** [9] when optimizing. The idea is simple: it is impossible to find a strategy that would always be better than an other one. We can try to find something that will be better in some particular cases (and that is what we will in fact try to do), but there will always be cases where the strategy is worse than others, and maybe way worse. In cases where no strategy sound interesting and random has a few good results, we can try to learn from those and understand where the other strategies failed.

Following the idea that we want to reach specific transition sequences, we can refer to the work of FlyMC [10]. In this paper, the authors present some algorithmic strategies to enhance the scalability of testing techniques of data-centers and cloud systems. Their idea is that some interleaving are more promising than others, and should therefore be tested before. This aligns to our idea of guiding strategy. As well as implementing DPOR to their field, the authors present two distinct strategies. We will call the first one **FlipParrallel**. The idea is the following: when we look at the reduction algorithms, we can see that from a given transition sequence, the next sequence we are trying is only differing from the previous one by a single flip of event. So to reach a very different sequence, it may take a lot of flips. To better understand, let us have a look at Figure 4. We have two kinds of actions: there is no independence relation between them but no b_i can happen before any a_1 or a_2. So we must consider all permutations of b_i and a_i between them. If we only do single flips such as in DFS, we will wait to explore all the $3! = 6$ combinations of b_i before exploring any execution sequence starting with actions $a_2 a_1$. The authors of FlyMC propose that from a first sequence, the search should continue by exploring a sequence that has as many flips

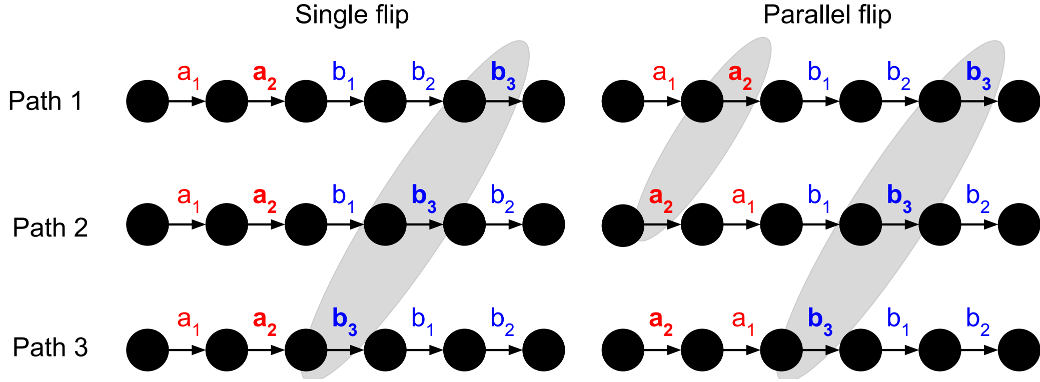

Figure 4: Example of a parallel flip: actions a_i are independent of b_i, so we can achieve a flip of actions of both type from path 1 to 2.

of difference as possible. On a system composed of N computational nodes, their algorithm will try to realize N simultaneous flips across all the N nodes. Sequences that differ only by fewer inversions are kept for latter search. So we end up by trying as a second sequence $a_2a_1b_1b_3b_2$ doing two flips in one shot. This is not always possible: flipping multiple actions from path 2 brings us back to a single flip of a previous case. In that case, the algorithm simply explores the remaining single flips. According to the authors, it also captures well the fact that mature distributed programs are usually robust for common interleaving order, while bugs are found in uncommon execution paths.

In distributed computing, many programs share a same kind of workload among different nodes: in FlyMC [10] they speak about data nodes, or follower nodes in cloud infrastructures, but we can think about master-slave patterns in message passing. This leads to different transition sequences reaching a similar global state. To avoid this repetition, the author of FlyMC implements **StateSymmetry** by recording past transitions with an abstract representation of the states starting the transition. This is possible in their model because they have access to every operation of the system, and therefore can have significant abstract representations of the program state. In the world of stateless model checking, this can not be used as a reduction technique because we miss information to have significant abstract representations. Instead of eliminating equivalent sequences based on potential symmetry, we can choose to defer the exploration of transitions leading to such symmetries until later stages. By adopting this approach, we maintain soundness while still capitalizing on the knowledge we have gained.

When speaking about guiding strategies that requires a valuation function, the first coming to mind are the shortest path algorithms. Usually, such algorithms work on directed graphs with a notion of weight over the edges. Directed model checking uses a more generic version allowing it to work on any **cost algebra** [3]. The general theory of cost algebra is wide. We will focus only on a simple case. Here we take $\langle \mathbb{R}^+ \cup \{+\infty\}, +, \leq, +\infty, 0\rangle$ as our cost algebra. This simply means that:

- our distances will be positive real numbers, eventually $+\infty$,
- we will be adding (resp. comparing) them using the usual $+$ (resp. $\leq$) operator over reals,
- and the minimum (resp. maximum) possible distance will be 0 (resp. $+\infty$).

Using this cost algebra, if we have an LTS $\mathcal{M} = (S, Act, S_0, \mathcal{T}, AP, \mathcal{L})$, we now decide of a cost

function over all the transitions, i.e., a function $c : \mathcal{T} \to \mathbb{R}^+ \cup \{+\infty\}$. And this function can be anything meaningful in our search. You could e.g., give a higher value to transitions that require a lot of computation time, so you avoid them, or just give a uniform weight. Once this is done, the problem of guiding the search is simply a problem of shortest path distance from the initial state s_0 to any bad state $s \in \mathcal{B}$. The issue with that is the same we tried to face in the reduction section: we can not compute the whole state space, and therefore, if we use a simple algorithm such as **Dijkstra's algorithm**, we may never finish in a reasonable time (unreasonable times can range from days, to years or even centuries). To try to solve this (huge) issue, we will rely not only on the cost function we decided to use but also on a second valuation function we will call an **heuristic**. An heuristic is an approximation of the real distance induced by the cost function. If for $s \in S$, we note $\delta(s_0, s) = \min\{\Sigma_{i=1}^{n} c(E_i) \mid E \text{ is a transition sequence from } s_0 \text{ to } s\}$ the distance from s_0 to s induced by the cost function, then we say that an heuristic $h : S \to \mathbb{R}^+ \cup \{+\infty\}$ is **admissible** when $\forall s \in S, h(s) \leq \delta(s_0, s)$, that is to say, when it always under-approximates the real distance. Interesting property about heuristic is that it can be deduced from something else than the complete exploration graph (which we can not compute). We will have a deeper look at some heuristics in the valuation section. **A*** is a search algorithm initially used to compute the distance problem in graphs thanks to a heuristic. It is even proven to be optimal as long as the used heuristic is admissible. Optimality means that the distance found from our starting state s_0 to a point of interest is minimal, and therefore, the path found is also a shortest one. In our context, it works by modifying how we pick a state in the set *Open* in algorithm 1. We now store a map d associating to each encountered state the smallest distance with which we reached it yet and instead of being arbitrary, we fix $\mathit{Select}(\mathit{Open}) = \min_{(s,d) \in \mathit{Open}} d + h(s)$. In practice, it is not clear how to combine both reduction and A* search: the first ones are written directly with a forcing order of DFS while the second one needs to pick the smallest state among multiple branches.

The last guiding strategy we will talk about is **Monte Carlo Tree Search** (MCTS) [11]. MCTS is a method used to cover a state space that incrementally optimizes its choices based on knowledge it acquired by sampling random explorations. It aims at finding a better option among multiple ones and has had a deep impact on fields like two player-games AI such as chess or Go. The algorithm is a global loop that will construct the exploration tree node by node. Adding a node to the tree involves four steps, as illustrated in Figure 5:

- *Selection* selects the most promizing node with at least one unexpanded child currently existing in the tree. This is done by applying a Tree Policy selecting children recursively from the root node;
- *Expansion* adds an unexplored child of the selected node to the tree;
- *Simulation* performs a simulated execution from the new node using a default policy, until a terminal state or bound is reached;
- *Backpropagation* evaluates the result of the previous execution and propagates that information along the path from the root to the newly added node.

To use MCTS, three components must be defined: a tree policy, a default policy and a valuation function for the states reached during simulation. Potential valuation functions will be discussed in the next subsection, but in typical scenarios, valuations may represent outcomes such as player A winning or player B being beaten by a hundred points. So defining a valuation for MCTS in the

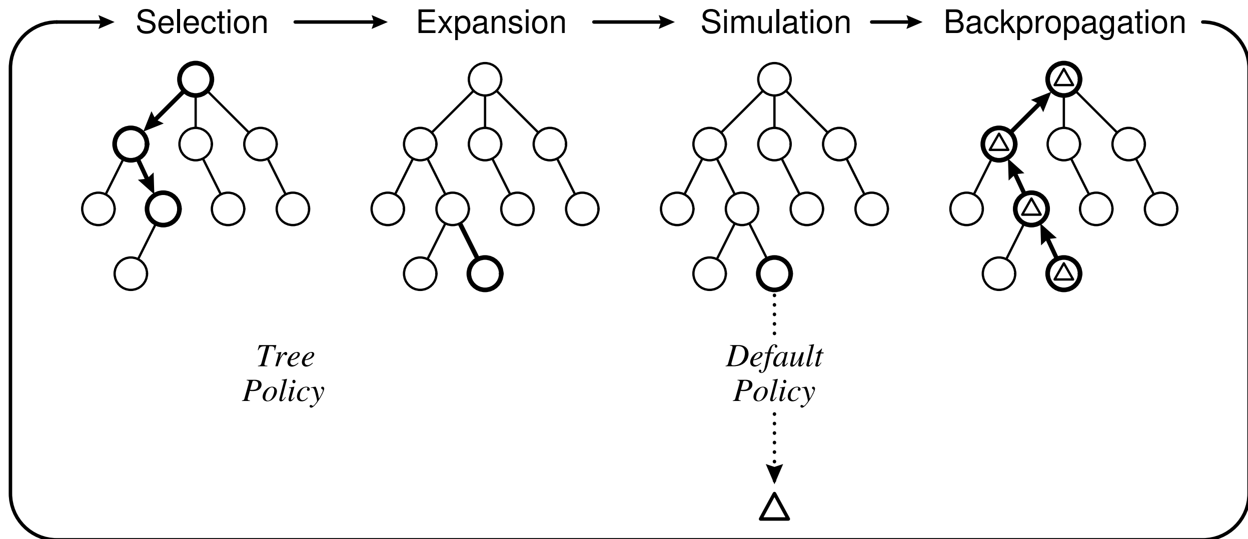

Figure 5: One iteration of the general MCTS approach [11].

perspective of bug hunting is unclear: you can not only state bug versus no-bug as when the bug is reached, the exploration is completed. The default policy can be anything from pure randomness to tweaked one, as long as it does not require computational intensive algorithm: this policy might be called a thousand times during a single simulation! Tree policy is a bit harder, and literature has shown many ways of providing with statistical policies. The basic idea is to mix the choice between apparently the best solution and some others that are a bit worse. This is mandatory because the knowledge gained from the simulation is very sparse: it only concerns a single execution while there could be millions.

2.3.2 Valuations

As demonstrated by A* search and MCTS, some guiding strategies require the aid of a valuation of states or transitions based on information specific to those states and rather than the entire exploration process. Furthermore, one can also use valuation to do strategic choices when asked for arbitrary ones in the different algorithms. To refine the classification of these evaluations, we can consider the type of information they use for computation. The first type only uses information about exactly known things, often included in the execution sequence or the state itself. On the other hand, the second can have access to some abstract hints about what might occur in the future.

The first valuation we will discuss is called **ActorSwitch**. Its purpose is to consider the number of times the active actor has changed throughout the execution. Both maximizing or minimizing this value offer distinct benefits. Maximizing it corresponds to a form of fairness between actors and tries to mimic real executions with interleaving of actors, such as in round-robin scheduling. On the opposite, minimizing it aims at exploring corner case executions that might be missed by programmers and therefore be more error-prone. Both choices can be achieved by storing information about the actor who took part in the execution sequence leading to the state and then giving a value to enabled transition based on their process.

The second valuation is based on objects shared between processes and the semantic associated

in the model. For instance, if one has a waiting queue for accessing specific resources, one can over-value states that help the queue growing. This way one can do a sort of stress test of some condition. This idea of stress-testing can be applied to many objects including mutexes, mailboxes, queues, simple shared variables, etc. In all these cases, the associated valuation should be guided by the model and the semantics of the cases one wishes to focus on.

As stated before, while searching for bugs using the MCTS approach, one of the difficulties is to find a good valuation function. In the work of [12], the authors are tackling video game testing problems with the MCTS algorithm. The goal of the test is not only finishing the game but trying the possibilities offered in the game-play. To achieve this, they introduce the concept of **features** which are elements they want the exploration to encounter. For each feature, they define a **criterion**, represented by positive rational numbers, indicating the desired frequency of encountering the feature. This way, the execution is rewarded for every featured seen, but to a lesser degree when the criterion is already fulfilled. Moreover, multiple goals, consisting of different sets of features and criteria, can be combined and prioritized. Goals have been created from data collected from human testers, but also from specific behaviors based on the semantics of the game actions. This approach proves valuable in situations where the final result of the execution lacks substantial meaning.

Let us now have a look at valuation functions that can take advantage of knowledge from a potential future of the execution. One such source of information is the **control flow graph** (CFG) of the process. A CFG is an oriented graph in which a node represents an operation and a vertex between n_1 and n_2 means that after n_1, the process can do n_2. CFG can be extracted from the source code of programs with static analysis. Using those graphs, one can determine precisely which operations can be executed in the future by the process. Therefore, it is possible to derive heuristics on the number of steps remaining to reaching a specific function or execution point for each process. In [13], the authors identify assertions as potential sources of problems in the code. An assertion is a function taking as input a boolean formula that the programmer expects to be true when the assertion is encountered at runtime. To guide the search towards those assertions, they specify a valuation function that corresponds to A* heuristics to any assertion in the code. In order to be a little more precise, they also keep a context during the execution representing knowledge on the current execution branch. For instance, when entering an `if` branch, the boolean condition value is known. So it is possible to add it to the knowledge while in that branch. With the knowledge, one can refine and prune some paths leading to `assert` that will be necessarily true, hence not corresponding to a bug.

When dealing with more complex properties allowed by LTL, the structure of the property itself can serve as a guiding advantage. This has been accomplished, for instance, in the context of Petri nets [14]. To do so, the authors keep the automaton describing the property while model-checking the whole LTS. When taking steps in the LTS, they also take corresponding ones in the automaton. And on this automaton, they define a notion of distance to an accepting state. This is done recursively on the structure of the property by giving values corresponding to the difference between the current marking of the Petri net and the desired one, and computing nodes distance accordingly. Thanks to that notion of valuation, the authors obtain significant results, and even proved to be able to combine their results with reduction techniques, such as partial order reduction.

Theorem 4.4.3 *An AsyncSend is independent of an AsyncReceive of another actor.*

$$\forall\, act1,\, act2 \in Actors : act1 \neq act2 \wedge TypeInv \implies I(AsyncSend(act1, _, _, _),\ AsyncReceive(act2, _, _, _))$$

Figure 6: An example of an independence theorem given in [16].

2.4 Context

As outlined in the introduction, the focus of our work is on ensuring the correctness of programs written with the widely used MPI library [1]. MPI is a C library commonly employed in high-performance computing, enabling the development of distributed code executed across multiple nodes. It provides a set of primitives, such as Send, Recv, Broadcast, Reduce, and others, for internode communication. Based on the principle of single instruction multiple data (SIMD) parallelism, multiple processes are created with the same source code, differing only in a specific variable known as the rank. Rewriting an entire MPI analyzer would be a daunting task. Instead, we leverage SimGrid [15], a framework designed to run simulation of distributed applications on distributed platforms. SimGrid includes McSimGrid, an integrated model checker that already implements a few techniques such as DPOR. By incorporating our findings into McSimGrid, we not only have the opportunity to compare different results but also extend these ideas to other distributed fields covered by SimGrid, such as pthread-based codes using mutexes and locks as synchronization tools.

In practical terms, MPI primitives are encoded using an abstract model of distributed applications, consisting of only five distinct operations [16]. Communication between processes is supported by the use of mailboxes, which serve as rendezvous points for sends and receives. When a send or receive operation is issued on a mailbox that already contains a complementary receive or send, they are paired together, forming a communication that is stored in a set of matched communications. The first two operations in the model are $AsyncSend$(m, data) and $AsyncRecv$(m, d), representing asynchronous send and receive operations, respectively. To provide information about these communications, the model introduces two additional operations. $TestAny$(Com) is a non-blocking operation that returns true if any communication in the set Com is already paired, and false otherwise. On the other hand, $WaitAny$(Com) is a blocking operation that only returns when at least one communication in Com is already paired. Additionally, processes may need to perform local computations that do not involve network communication. These computations are captured by the $LocalComp()$ operation. In [16], the author formalize those operations, as well as synchronization ones, before proving the independence theorems (such as the one in Figure 6) that are, for instance, required to use reduction algorithms. Since we do not want to look at the state representation in memory, we will ignore the operations of type $LocalComp()$ and consider only resulting states after any other one. Formally, in our LTS, an action $a \in Act$ will be a combination of a process ID and an operation among $\{AsyncSend, AsyncRecv, TestAny, WaitAny\}$.

3 Contributions

In this section, we present our findings about the combination of reduction techniques and guiding strategies. To be able to combine both of these orthogonal approaches, we must first explain the modifications we made to the usual reduction algorithm. In a second time, we will have a look at the different new guiding strategies and valuations we propose, that are taking advantage of some

MPI specific knowledge.

3.1 Multi-head dynamic partial order reduction

As discussed, a different version of traditional reduction algorithms must be proposed to take full advantage of guiding strategies. In fact, in both DPOR and SDPOR algorithms, we identify, at first, three steps that are asking for a choice. First when first encountering a node (line 2), then when backtracking to an already visited node (line 5) and finally when a reversible race has been found (line 7) (lines are given in algorithm 2, page 6). The second option assumes that multiple transitions have been added to the *to_visit* set at that node in the meantime. Those three specific moments can already take advantage of guiding strategies, but as suggested by the A* algorithms, it could be interesting to also have the opportunity to pick the next state we want to consider among opened transitions and not only perform a depth first search. To do so, we present algorithm 4: **Multi-head DPOR**. It is here implemented using persistent sets as described in [4] but works identically with source sets of [5].

Algorithm 4: Multi-head DPOR(s_0)

Data: s_0 an initial state, data structures *to_visit* and *done* mapping sequences to sets of transitions, a set of sequences *exploration_heads* initially empty.

Result: Exploration of every Mazurkiewicz trace from s_0

1 choose some $t \in \texttt{enabled}(s_0)$ and add it to $to_visit(\langle\rangle)$;
2 add $\langle\rangle$ to *exploration_heads*;
3 **while** $exploration_heads \neq \emptyset$ **do**
4 | choose some E from *exploration_heads*;
5 | **if** $\exists t \in to_visit(E) \setminus done(E)$ **then**
6 | | **if** $\exists i = \max\{i \in \texttt{dom}(E) \mid D(E_i, t) \wedge (E_i) \nrightarrow \texttt{proc}(t)\}$ **then**
7 | | | **if** $\{t' \in \texttt{enabled}(\texttt{pre}(E, E_i)) \mid \texttt{proc}(t') = \texttt{proc}(t) \vee t' \rightarrow \texttt{proc}(t)\} \neq \emptyset$ **then**
8 | | | | add some t' to $to_visit(\texttt{sub}(E, E_i))$;
9 | | | **end**
10 | | | **else**
11 | | | | $to_visit(\texttt{sub}(E, E_i)) \leftarrow \texttt{enabled}(\texttt{pre}(E, E_i))$;
12 | | | **end**
13 | | | add $\texttt{sub}(E, E_i)$ to *exploration_heads*;
14 | | **end**
15 | | choose some $t' \in \texttt{enabled}(\texttt{last}(E.t))$ and add it to $to_visit(E.t)$;
16 | | add $E.t$ to *exploration_heads*;
17 | | add t to $done(E)$;
18 | **end**
19 | **else**
20 | | $exploration_heads.remove(E)$
21 | **end**
22 **end**

The main difference with classical reduction algorithms lies in the set *exploration_heads*. At any point of the algorithm, it contains the executions that could still be augmented by taking a

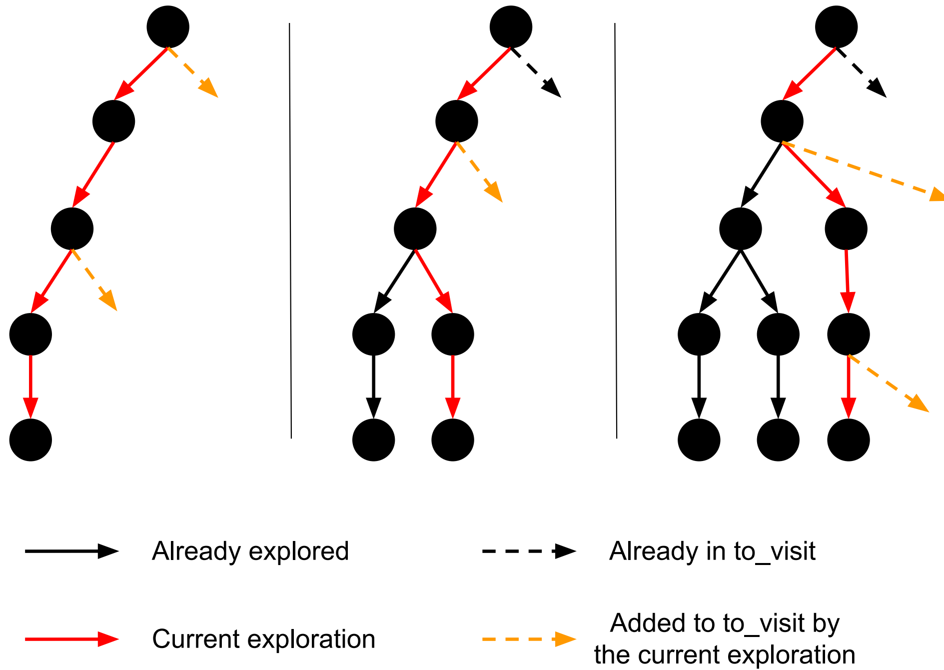

Figure 7: Example of the exploration of a state space by DPOR.

transition that has been enabled by the reduction. In the beginning, the reduction enables some initial transition, and therefore we put the empty sequence $\langle\rangle$ in *exploration_heads*. Then, as long as there is a sequence in *exploration_heads* we pick one, and from that sequence, we try to take a transition. If one has been enabled by the reduction but has not yet been taken, we pick it. On that transition, we perform the regular DPOR operations, and if any persistent set was augmented, i.e., if we found a reversible race, we add the corresponding transition sequence to the opened exploration heads. This way, this newly enabled transition could already be chosen at the next iteration. Finally, we explore the chosen transition by creating a new node, passing the transition to *done* and adding the extended sequence to the one yet to be explored.

Figure 7 and Figure 8 illustrate the difference between DPOR and Multi-Head DPOR. In Figure 7, we can see how the depth first search specificity of the algorithm forces the order in which we visit the tree of possibilities. Even if the algorithm knows for a while that there exists another branch at the root, it can not explore it before fully exploring the currently selected child. Furthermore, it can not either explore a partial transition sequence. DPOR must keep exploring a sequence until reaching a terminal node with no transition enabled. On the other hand, in Figure 8, we see that Multi-Head DPOR can try to explore the other choice at the root before completing the exploration of the left subtree. To achieve this, some nodes are saved in the set *exploration_heads*. Multi-head DPOR can also stop the exploration of a transition sequence at any point. Again, it takes advantage of *exploration_heads* to do so.

Intuitively, we simply moved the timing at which we populate the sets. Since two children in the exploration tree do not impact each other, it does not matter if we explore pieces of both at the same time. On the other hand, it is mandatory for A* to maintain optimality to be able to choose

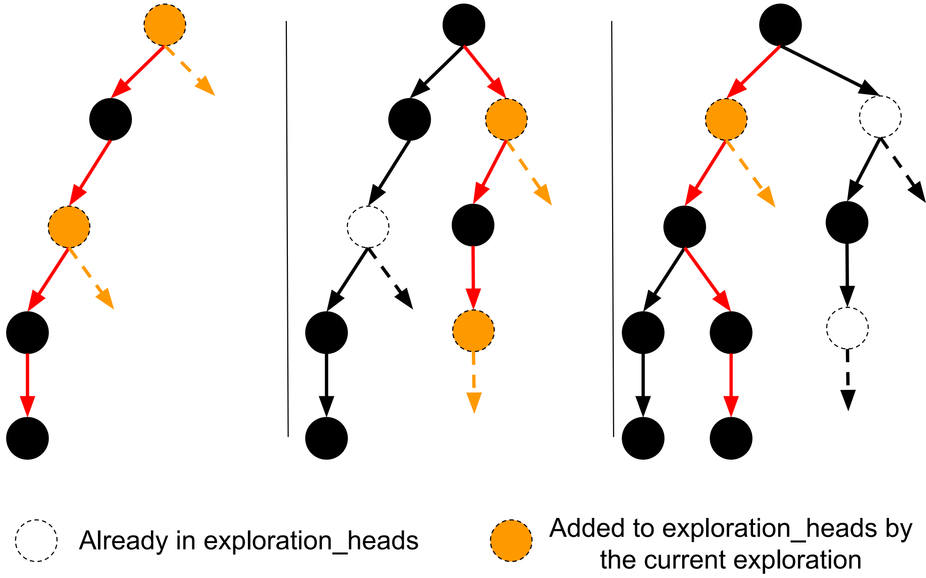

Figure 8: Example of a possible exploration of the same state space as Figure 7 by Multi-Head DPOR

at every new step. With that new algorithm, we now have three clear arbitrary decisions: the new one being made when picking an open exploration head. To prove the correctness of our algorithm, we will prove that DPOR and Multi-Head DPOR explore the same executions, and therefore, the correctness of Algorithm 4 is implied by the correctness of DPOR. To do so, we need to introduce the notion of *explored sequences*. We will say that a sequence E is being *reached* by DPOR when a call Explore(E, DPOR) is made, and it is reached by Multi-Head DPOR when it is added to *exploration_heads*. We will note $\mathcal{R}_{DPOR}$ and $\mathcal{R}_{MH}$ the sets of sequences reached by DPOR and Multi-Head DPOR. In all the following proofs, we will suppose that the arbitrary choices made while running Multi-Head aligns with those made in DPOR: that is to say, choices on lines 2 (first encounter), 5 (backtrack) and 7 (reversible race) in DPOR will be respected if possible at lines 15 (first encounter), 5 (backtrack) and 8 (reversible race) in Multi-Head. We first show why it is sufficient to consider the sets $\mathcal{R}$. Then we establish the link between $\mathcal{R}$ and the sets of our algorithm. Finally, we prove the theorem.

Theorem 3.1. *Multi-Head DPOR explores at least one execution per Mazurkiewicz class.*

To prove the theorem, we will rely on the correction of DPOR. DPOR is proven to explore at least one execution per Mazurkiewicz class [4]. If we can show that the two algorithms reach the same set of sequences (i.e., show that $\mathcal{R}_{DPOR} = \mathcal{R}_{MH}$), then we have proved our theorem.

This little lemma will be useful to prove $\mathcal{R}_{DPOR} = \mathcal{R}_{MH}$ since it connects the sets of reached sequences and the actual set used in the algorithm. Intuitively, it states that *to_visit* sets describe exactly the children of any sequence in $\mathcal{R}_{MH}$.

Lemma 3.2. $\forall t \in \mathcal{T}, \forall E \in \mathcal{T}^*, (E \in \mathcal{R}_{MH} \land t \in \mathit{to_visit}_{MH}(E)) \iff E.t \in \mathcal{R}_{MH}$.

Proof. Let $t \in \mathcal{T}$, let $E \in \mathcal{T}^*$,

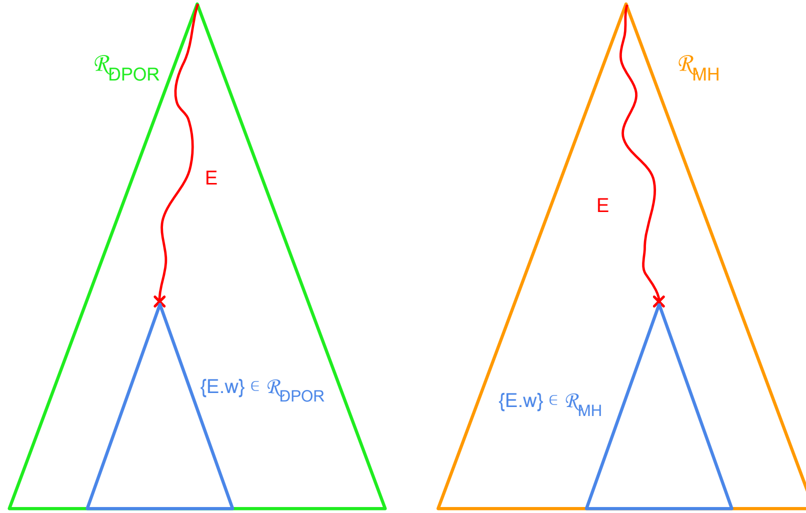

Figure 9: Simple representation of Lemma 3.3 statement. The reached set, represented as a tree, for DPOR is in green, in orange for Multi-Head DPOR; a shared execution sequence E is in red and the matching continuations of that sequence are in blue. The lemma states that the blue subtree are equal.

- $\Longrightarrow$
  Since $E \in \mathcal{R}_{MH}$, by definition it means that E was in *exploration_heads* at some point. Furthermore, anytime a transition is added to a set of $to_visit(E')$ (either at line 15 or lines 8/11), E' is added to *exploration_heads*. Therefore, E was in *exploration_heads* after t was added to $to_visit(E)$. From that time, E will only be removed from *exploration_heads* when $to_visit(E) \setminus done(E) = \emptyset$. If t is chosen in $to_visit(E) \setminus done(E)$ then the proof is over. So let us suppose that we reached a point where $to_visit(E) \setminus done(E) = \emptyset$ and t was never chosen. Since $t \in to_visit_{MH}(E)$, it means that $t \in done(E)$. But the only way to add t to the set $done(E)$ is to reach line 17. Hence, t was chosen at some point at line 5. So line 16 was reached, and therefore $E.t$ was added to *exploration_heads* which is the definition of $E.t \in \mathcal{R}_{MH}$.

- $\Longleftarrow$
  If $E.t \in \mathcal{R}_{MH}$, let us consider the first time it was added to *exploration_heads*. That necessarily happened at line 16 since line 13 only adds prefix of the current sequence (i.e., sequences that have already been reached). But when reaching line 16, E was picked from *exploration_heads* (i.e., $E \in \mathcal{R}_{MH}$) and t was picked from $to_visit(E) \setminus done(E)$ (in particular $t \in to_visit(E)$).

□

We will suppose that the same lemma is true for DPOR since it is used to prove the correction of the algorithm in [4]. We will now prove a lemma intuitively stating that if a sequence is reached by both algorithms, then the sequences reached from this initial sequence are the same in both algorithms. The idea of the lemma is represented by Figure 9. In other words, if there is a sequence

E reached by both algorithms (in red), then the subtree explored by both algorithms (in blue) is the same. Using that lemma on sequence $\langle\rangle$ that is reached by both algorithms proves the first lemma, and therefore the theorem.

Lemma 3.3. *Let $E \in \mathcal{T}^*$, if $E \in \mathcal{R}_{DPOR} \cap R_{MH}$, then $\forall E' \in \{E.w \mid w \in \mathcal{T}^+\},\ E' \in \mathcal{R}_{DPOR} \iff E' \in \mathcal{R}_{MH}$.*

Proof. We will note $N = \max\{|E| \mid E \in \mathcal{R}_{DPOR} \cap \mathcal{R}_{MH}\}$, the size of the longest sequence reached by both algorithms, and $\mathcal{H}(n)$ the property: " $\forall E \in \mathcal{R}_{DPOR} \cap \mathcal{R}_{MH}$ s.t. $N - |E| \leq n$, $\forall E' \in \{E.w \mid w \in \mathcal{T}^+\},\ E' \in \mathcal{R}_{DPOR} \iff E' \in \mathcal{R}_{MH}$". With $E \in \mathcal{R}_{DPOR} \cap \mathcal{R}_{MH}$, let us prove $\mathcal{H}_n$ by induction on n.

- *Base case: $n = 0$.*
  By definition of N, if $N \leq |E|$, then $N = |E|$. In particular, $\{E.w \mid w \in \mathcal{T}^+\} \cap \mathcal{R}_{DPOR} = \emptyset$. Therefore, *enabled*(*last*(E)) $= \emptyset$. So $\{E.w \mid w \in \mathcal{T}^+\} \cap \mathcal{R}_{MH} = \emptyset$. Hence, $\mathcal{H}_0$ is true.

- *Induction step: suppose there exists n such that $\mathcal{H}_n$ is true. Let us show $\mathcal{H}_{n+1}$.*
  Let us take $E \in \mathcal{R}_{DPOR} \cap \mathcal{R}_{MH}$ such that $N - |E| \leq n + 1$ and show that $\forall E' \in \{E.w \mid w \in \mathcal{T}^+\},\ E' \in \mathcal{R}_{DPOR} \iff E' \in \mathcal{R}_{MH}$.
  If there is no continuation of the sequence E reached by DPOR, then it means that *enabled*(*last*(E)) $= \emptyset$ and the proof is the same as the base case. Without loss of generality, let us then suppose that $\{t \mid t \in \mathcal{T} \wedge E.t \in \mathcal{R}_{DPOR}\} = \{t_1, \ldots, t_n\}$ with transitions numbered in the order they are added to $to_visit_{DPOR}(E)$ (the sets are the same according to Lemma 3.2 for DPOR). We will now show that $to_visit_{DPOR}(E) = to_visit_{MH}(E)$. Let us first prove that $to_visit_{DPOR}(E) \subseteq to_visit_{MH}(E)$. We note $\mathcal{H}'_k$ the property: "The k first transitions of $to_visit_{DPOR}(E)$ are in $to_visit_{MH}(E)$". Let us prove $\mathcal{H}'_k$ by induction on k.

  - *Base case: $k = 1$.*
    The first transition added to $to_visit_{DPOR}(E)$ was an arbitrary choice among *enabled*(*last*(E)). Since we suppose that both algorithms made the same arbitrary choice when possible, and Multi-Head DPOR also selects the first transition among *enabled*(*last*(E)), then $t_1 \in to_visit_{MH}(E)$.
  - *Induction step: suppose there exists k such that $\mathcal{H}_k$ is true. Let us show $\mathcal{H}_{k+1}$.*
    If t_{k+1} was added to $to_visit_{DPOR}(E)$, there exist $1 \leq i \leq k$, $1 \leq j \leq k$ s.t. $i \neq j$, $t' \in \mathcal{T}$ and $w \in \mathcal{T}^*$ such that t_i and t' are in reversible race and $E.t_j.w.t' \in \mathcal{R}_{DPOR}$. That is to say, t_{k+1} was added due to the fact that we found a race between an already explored transition in $to_visit_{DPOR}(E)$ and a transition found in a sequence starting in E. Since we supposed $\mathcal{H}'_k$, we have that $t_i, t_j \in to_visit_{MH}(E)$, so $E.t_i, E.t_j \in \mathcal{R}_{MH}$ with Lemma 3.2. Moreover, $N - |E.t_j| = N - |E| - 1 \leq N$ by hypothesis on E. Therefore, we can apply $\mathcal{H}_n$ to the sequence $E.t_j$. This gives us in particular that $E.t_j.w.t'$ is also in $\mathcal{R}_{MH}$. Hence, the same reversible race will also be spotted by Multi-Head DPOR before adding $E.t_j.w.t'$ to *exploration_heads*, and t_{k+1} will be chosen to be added to $open_states_{MH}(E)$.

  The proof for $to_visit_{MH}(E) \subseteq to_visit_{DPOR}(E)$ is similar. Now that we have $to_visit_{DPOR}(E) = to_visit_{MH}(E)$, let us prove that $\forall E' \in \{E.w \mid w \in \mathcal{T}^+\},\ E' \in \mathcal{R}_{DPOR} \iff E' \in \mathcal{R}_{MH}$.

Let E' be a sequence in $\{E.w \mid w \in \mathcal{T}^+\}$. We will note $E' = E.t.w'$ with t a transition, and w' a sequence (potentially empty).

- *If* $t \notin to_visit(E)$: then $E.t \notin \mathcal{R}_{DPOR}$ which means that $E' \notin \mathcal{R}_{DPOR}$, and $E.t \notin \mathcal{R}_{MH}$ which means that $E' \notin \mathcal{R}_{MH}$ (those implications are easily verified inductively with Lemma 3.2).
- *If* $t \in to_visit(E)$: then $t \in \mathcal{R}_{DPOR}$ and $t \in \mathcal{R}_{MH}$. So $E.t \in \mathcal{R}_{DPOR} \cap \mathcal{R}_{MH}$. If $w' = \langle\rangle$ then the proof is done. Else, since $N - |E.t| \leq n$, we obtain by applying $\mathcal{H}_n$ to $E.t$ that $E' \in \mathcal{R}_{DPOR} \iff E' \in \mathcal{R}_{MH}$.

□

Intuitively, the proof is working thanks to the fact that DPOR does not allow different children to interact with each other's data. A node will only populate persistent and source sets among his parents, and once something is added to that parent, it will never be removed. For this reason, we remain doubtful on the possibility of adapting this idea to ODPOR. Indeed, the problem is that a child can add a whole sequence to a wakeup tree of one of his parents. This roughly corresponds to adding a sub-sequence to a data in another children node. The problem is also open when considering UDPOR. The computation of $Alt(C, \mathcal{D} \cup \{e\})$ requires using a set of events U that will precisely be known only when the left subtree is fully explored. Therefore, trying to compute the right subtree (i.e., computing the alternative) early does not sound possible.

If we now take a step back to look at the choices we have, there are two types of choices:

- we will call *inserting choice* a choice of a transition to be added to *to_visit*, either when the state is first encountered, or when a reversible race is detected.
- we will call *picking choice* a choice of transition to consider among multiple ones already inserted. This is the case when we backtrack to a node with multiple possibilities, but now, it is also the case when picking the node we want to backtrack to in *exploration_heads*.

With those particular choices, we can now explain how the new guiding strategies work.

3.2 New guiding strategies

In this subsection, we explain the two new guiding strategies we have designed.

The first guiding strategy is **SimilarityAvoidance**. It comes from the idea that in stateless model-checking, we have to rely on our knowledge of the dependence relation to decide how we build our reduction. Despite all the effort we put in understanding the semantics, we may have to over-approximate it sometimes. That is the example of multiple `write` on the same variable with a single `read` in the end. Statically, we will state that two `write` on the same variable are dependent. But in that execution sequence, only the two `write` happening before the `read` really are. Furthermore, in our model, we are only observing communications between processes. Hence, there are a lot of things internal to each process that we do not know about. We may miss some important things. To sum up, we may consider a lot of states as different when in fact they are identical.

SimilarityAvoidance aims at mitigating this phenomenon. We can not take as granted that two different states are equal and stop the exploration because it could ruin the soundness of the approach. But what we can do is delay the exploration of states that seem similar to ones already

explored. The idea to do so is to consider the local history of processes and consider that states reached with the same local history may be identical. When finding a similar state, we delay the exploration of its children until only similar states are explored. If we reach this point, it means that maybe those similar states are really different. If they are identical, the transition chosen to be explored from them does not matter: the reduction will do the same exploration whatever. But if they end up being different, by choosing the least visited transition among similar states, we implement the FlipParrallel of FlyMC at the same time.

More precisely, for each reached node, we associate a corresponding abstract state consisting of only the number of transitions taken by each process. For each abstract state, we save the number of times the different processes have been chosen. When doing an inserting choice, we choose the transition associated to the process that has been the least taken in the corresponding abstract state. When doing a picking choice, we choose the transition for which the number of time it was explored in the abstract state is the lowest.

The second guiding strategy is quite similar to **SimilarityAvoidance**, but also tries to tackle a specific pattern of programming used in distributed systems. **ActorSymmetry** utilizes the concept of abstract state and avoids doing the same choices for identical abstract states just like **SimilarityAvoidance**. The difference comes from the way they build the abstract state. With **ActorSymmetry** we want to deal with applications programmed by a master/slave pattern. In those applications, one process is distinguished from the other as the master. The master usually deals with organization tasks, gatherings, or sharing of data. The slaves are asked to do tasks and are used as work force by the application, regardless of their specific identity. In other words, the slave processes are identical. A good example is a matrix computation. The master is given two whole grids. Different slaves are given sub-matrices, and computing the same thing, regardless of what the master has attributed.

When model checking those specific cases, the idea is that interleaving one slave with the master is the same as interleaving another slave with the master. There are processes that are symmetric in their communications, tasks, computations... They are symmetric in their role. Therefore, the algorithm will try not to visit an interleaving already explored with another symmetrical process. The problem is that SimGrid does not have access to the computations that are being done inside the application. In order to detect potential symmetry without an important overhead, we then just look for processes that have executed the same number of transitions. For instance, if there are three processes P, Q, R and if we write (p, q, r) the number of transitions taken by each of them, we will consider equivalent abstract states $(1, 4, 3)$ and $(1, 3, 4)$ because processes Q and R may be symmetric in their execution, and therefore the corresponding real state may have already been explored.

We have to keep in mind that those guiding strategies are simple in the way they make choices, and a lot of time will not be the best, but they are efficient in terms of computation required to decide. Moreover, those strategies do not aim at being effective all the time: we even saw that in the general case it is impossible. Sometimes **ActorSymmetry** will identify two processes as symmetric when they are not, and could be then worse than any other strategy, but this is not a problem as long as there are cases in which detecting the symmetry is critical for the exploration.

3.3 New valuations

In addition to those two new guiding strategies, we introduce two new valuations. These valuations take advantage of some usual MPI patterns that either lead to more errors, or inefficient exploration.

```
While(True){
  c = MPI_ISend(PreviousResult);
  While(not MPI_Test(c)){
    DoNextComputation()
  }
}
```

Figure 10: MPI_Test example.

The two valuations we present here are punitive in the sense that they discourage the exploration to consider some patterns. Those two valuations are only based on exact information from the past and the present of the execution.

The first valuation is a way for the exploration to dodge infinite loops. The one we want to avoid is presented in Figure 10. It can be found for instance in any producer/consumer pattern. A process has a computation to do and must share the result. But to gain time, it will compute while waiting for the previous transmission to complete. Hence, the testing on the result of the communication. If we use a depth first search approach to explore the possibilities here, we will fire the transition corresponding to the test on the communication. And we will do this forever, since it is always enabled, and the result will not change until a corresponding `recv` is executed by another process. One classical solution to this kind of problem is to use bounded model-checking. This relies on setting a threshold for the exploration depth, i.e., a number of steps to be explored before considering the current transition sequence is a dead-end. In some cases it is reasonable to do so, but most of the time, one also loose potential behaviors by doing so. Instead of doing that, we use valuation to dodge those pitfalls. If we detect that a test transition has been executed and is still enabled, we penalize the corresponding transition in order to force the exploration of the rest of the state space.

A second valuation is used to tackle a possibility offered by the MPI library that is error-prone. Looking back at Figure 10, we need to write the master part that will receive all those data. For that, MPI provides with receive primitives that will receive messages from a given process. The first option is to write a `recv` for each slave process. This is tedious in terms of cases to handle, having to consider every case of matching communication. Furthermore, this solution is not scalable, requiring to modify a huge amount of code any time the number of process is modified. Second and chosen option is to use a `MPI_ANY_SRC` tag. It simply means that this receive will match a `send` from any process to this one. It has interesting and necessary usage but can also lead to hard-to-detect problems due to unplanned interleaving. A simple example of this is illustrated in Figure 11. In this example, there is a race condition between the two `send`($P0$) operations of process P1 and P2. If P1 `send`($P0$) is matched with P0 first `recv`(any), then P0 will execute the `send`(1) that will also be matched since P1 can execute `recv`($P0$) and life goes on. On the other hand, if P2 `send`($P0$) is matched with the `recv`(any), then P1 will be blocked, waiting for its own `send`($P0$) to be matched, while P0 will wait for a `send`($P1$) to be matched. Here we have a potential deadlock. This example is quite simple, but in practice, there are harder communication patterns that can still lead to a problem.

To help capturing those situations, we propose the **DelayAny** valuation. The idea is simple, we will undervalue transitions that correspond to a `recv` action using an `any` tag. This way, we maximize the number of potential matching communications that may have been missed by the

```
P0                    P1                P2
recv(any)             send(P0)          send(P0)
send(P1)              recv(P0)
recv(any)
```

Figure 11: MPI any source tag example.

```
P0                    P1                P2
g = recv(any)         send("1", P0)     send("0", P0)
g = recv(any)
assert(g!=1)
```

Figure 12: A small MPI example with an assertion.

programmer. Those matching communications can be the source of potential bugs.

4 Implementation

In this section we discuss the implementation we performed of the different novelty we explained earlier as well as classical solutions presented in the state-of-the-art section, adapted to our model.

As stated before, we use McSimGrid as the baseline for our model checking algorithm. SimGrid [15] works by simulating MPI processes as different threads. As in an operating system, processes can run and execute user code as they want, but when they need to run MPI primitives, they need to switch in kernel mode by calling the maestro (a Simgrid specific thread). Maestro is in charge, just like a real operating systems handling system calls, to orchestrate the demands to the MPI library. This is especially useful when model checking. McSimGrid puts another layer on top of this. A checker is in charge to discuss with the maestro and will tell which MPI call should be until terminating or reaching another call. This dynamic allows us to visit an execution sequence at our will. Better than that, McSimGrid also offers the opportunity to restart the application, and execute some sub-sequences of transitions again, therefore creating the possibility to backtrack in the exploration. With these basic explanations about our tool, we can now explain how we implemented our ideas.

We implemented the CFG distance solution using application source code. For that, our tool takes as input the *LLVM intermediate representations* (LLVM ir), which is a specific representation used inside LLVM. LLVM [17] is a framework used to provide compilation analysis and optimizations. It is capable to work on many languages and is a powerful tool when doing static analysis. In Figure 12, we have an example of a small MPI program with an assertion. When looking at it in LLVM ir, we obtain blocks, which are composed of multiple simple execution statements. Figure 13 shows the CFG obtained with the LLVM ir. It has multiple defaults for the usage we want. The biggest issue is linked to the fact that we are only observing the MPI communication. There is a lot of information in this graph we will not be able to use at runtime because we will not know it. For instance, from the checker, we can not determine at runtime if a process in block labelled %26 is going to exit with a true or a false conditional jump. We do not have access to that. Neither maestro (in SimGrid) nor the checker discussing with maestro know about the internal state of the MPI process. That is not a problem for our other algorithms, we are doing stateless model

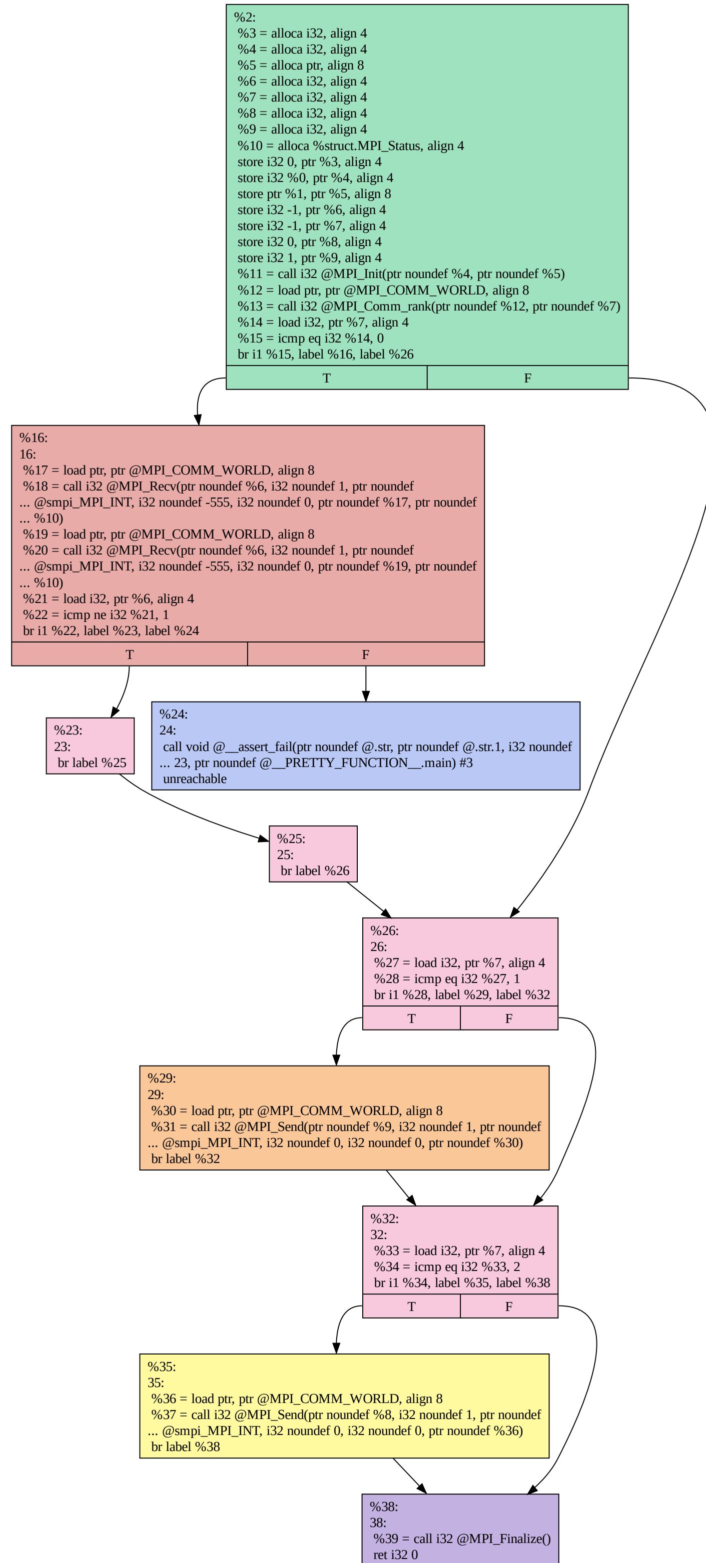

Figure 13: The CFG of Figure 12 in LLVM intermediate representation.

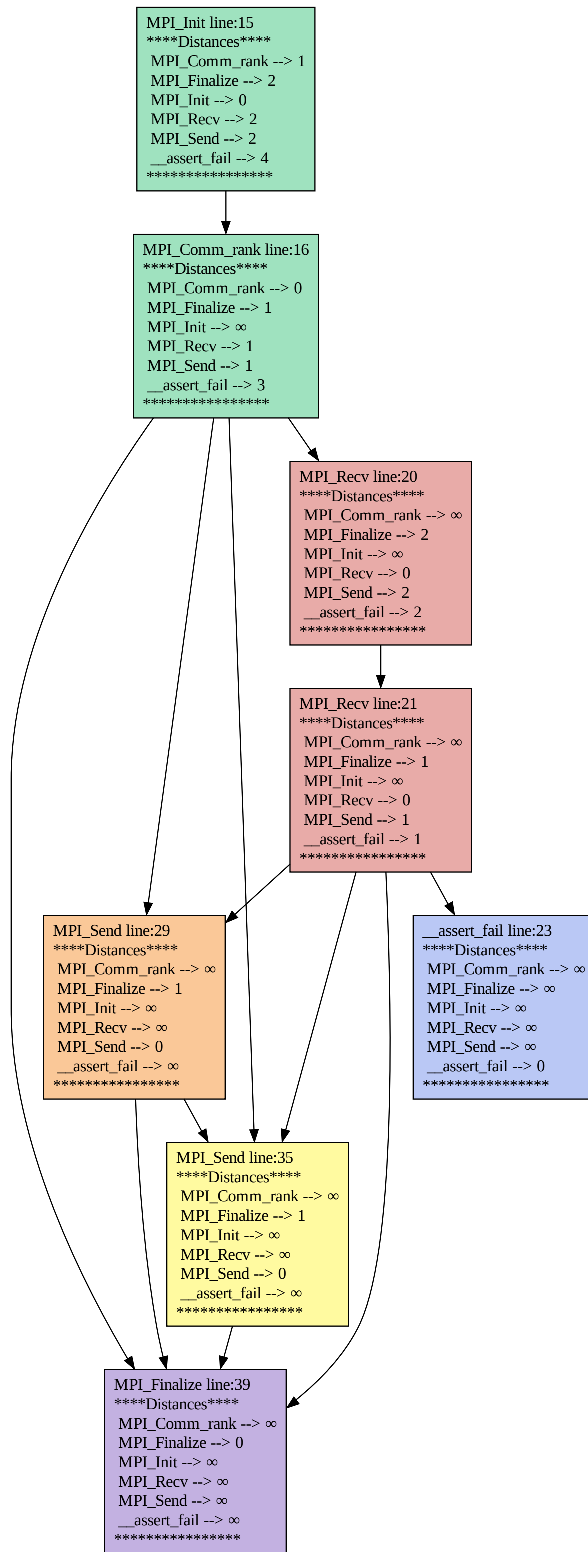

Figure 14: The extracted CFG from Figure 13 with computed distances.

checking. But here, it simply means we need to cut off all this information that will pollute the computation of our distance.

Instead of using the CFG from the intermediate representation, we instead read for the LLVM ir of the code and extract what we can see in Figure 13. Compared to the previous representation, some blocks have been split, others simply disappeared (as seen with the color pattern). This is because we simply kept the MPI calls, the things that are observed by McSimGrid, as well as our target, the assertion. The links between the node of the new CFG still respect the flow of the program: for each MPI call, the outgoing vertices represent the possible calls the process can execute just after. It is also enriched with the information required to localize precisely the calls inside the code. For instance, there are two `MPI_Recv` in the code: one at line 20 and one at line 21. With that information, when receiving a `MPI_Recv` in the checker, we can identify precisely where in the code, i.e., in the graph, we are. Computing the distances is now easier: to have an admissible heuristic, we make optimistic choices and take the smallest distances when branches exist. This method gives a heuristic for each process in terms of smallest number of MPI operations to execute before reaching an assertion. To compute the final value for the state, we simply sum the value for each process. This gives us an optimistic approximation on the number of MPI calls to be executed from that state before reaching an assertion.

In addition to the A* algorithm, we also implement:

- the **SimilarityAvoidance** strategy discussed earlier.
- a uniform random strategy, now possible thanks to the Multi-Head version of DPOR.
- a **MinMatch/MaxMatch** that aims at saturating the usage of mailboxes that are shared between processes. MinMatch tries to avoid matches communications in order to maximize the possibility of finding potential deadlocks. On the opposite, MaxMatch tries to match the communications as soon as possible. This way we can hope to keep counter-examples simple to read for the user, and therefore efficient to correct the bug found.

Figure 15 gives a quick summary of the different combinations of strategies and valuations we implemented.

5 Experimental Evaluation

In this section, we present some preliminary results we obtained with our few examples. From those results, we start the discussion about the interest of the different strategies and valuations introduced.

5.1 Test cases

To evaluate our propositions, we would need an important number of erroneous codes that are already a bit huge in terms of either number of MPI operations or number of involved processes. Literature proposes different benchmarks of MPI erroneous applications. In [18], the authors are providing with a classification of MPI possible bugs as well as an important number of erroneous codes and expected behavior to be found by a verification tool. But all these codes are quite small, and some of them are even deterministic in the sense that they have a single possible execution

Strategy	Valuation	Inserting Choice	Picking Choice
DFS	Depth	Maximum depth	Order of process rank
DFS	MaxMatch	Minimum number of Non-paired communication	Transition matching a communication > non-communication transition > Transition creating an unmatched communication; ties ordered by process rank
DFS	MinMatch	Maximum number of Non-paired communication	Transition matching a communication < non-communication transition < Transition creating an unmatched communication; ties ordered by process rank
DFS	SimilarityAvoidance	Least seen abstract state; ties ordered by depth	Least taken transition in equivalent abstract state; ties ordered by process rank
Random	Uniform	Random uniform choice	Random uniform choice
A*	CFG	Smallest distance from origin + CFG heuristic valuation	Smallest CFG heuristic valuation

Figure 15: Summary of implemented strategies with valuations and how they handle the different choices.

sequence. In [19], the authors provide fewer codes, but they also created variants of existing mini-apps that are bugged. The bugs are introduced by modification of code they give. This time, the considered applications are of a reasonable size. The problem comes from the deterministic property of the considered bugs. They always appear regardless of the explored transition sequence. Therefore, we can not use any of these approach to compare our strategies and valuations.

Instead, for now, we propose to test our findings on some small examples we designed and `heat`, a test taken from another tool benchmark [20] that aims at computing the result of the `heat` propagation equation on a small grid. All those tests lead to a problem, either in the form of a violated assertion (in our codes) or of a deadlock (for `heat`). Our examples are variations of the same pattern. Those variations are obtained by changing the order of MPI process, the type of communications, and/or augmenting the complexity by adding synchronizing operations. The pattern has been chosen because it was used to highlight an initially existing bug in the implementation of the DPOR algorithm in McSimGrid. It is described in Figure 16 and roughly corresponds to a read/write race. The interesting part is that it is easy to detect the race between the two `send` of P1 and P2, but it is not as easy to understand that to reverse the race, one must execute process P3 or P4. When adding synchronization around the pattern, it is easy to make the number of possibilities explode, partially because the translation of simple collective communications may take multiple `send`/`recv` executions in terms of mailboxes.

We tested all the combinations of Figure 15 on each pattern except for A* with the control flow graph valuation that for which `heat` is not relevant. `Heat` leads to a deadlock (i.e., an inter-blocking situation) between processes. It contains no meaningful assertion. Therefore, the strategy consisting in guiding towards assertion makes no sense in this case. Finally, most of the experimentation are run with the specificity that backtracks are only permitted when reaching a terminal node. In other words, when exploring a sequence E, the search will continue choosing the extended sequence $E.t$ in $exploration_heads$ until $enabled(last(E)) =$. At that point, the choice for the next sequence is

```
P0               P1               P2               P3             P4
g = recv(any)    send(P3)         send(P4)         recv(P1)
recv(P2)
g = recv(any)    send("1", P0)    send("0", P0)
assert(g!=1)
```

Figure 16: A pattern of non-trivial reversible race.

free. This is done in order to mitigate the overhead of re-executing transitions, which is required by McSimGrid when backtracking.

Our results are presented in Figure 5.2. The values are given both in terms of visited states and executed transitions. Visited states are the number of states reached in the corresponding LTS. Executed transitions is a specificity of McSimGrid functioning. To backtrack, we need to reload a specific state and execute a prefix of the transition sequence. Executed transition is the total number of transitions the simulated program ran. The values with a "greater than" indication means that we stopped the execution before the bug was found. The Figures given are those at the time the execution was stopped. Threshold difference between Heat and synchronization is due to the overhead of computation in Heat. Some of the transitions do real calculus while our pattern is only a communication outline. The Figures for the strategy involving randomness are given as mean(±standard deviation) over 100 executions with different random seeds.

There is almost always something better than a depth first search. As we could expect, choices based on the order of processes are highly impacted by the reordering. This is seen for DFS+Depth, but also for every other combination that resolves ties using the process ordering. This impact can have important consequences on the performance (SimilarityAvoidance on the pattern + synchronization is one example). On the opposite, Random+Uniform is much less impacted by a simple reordering. These results guide us toward the use of randomness to revolve ties for future testings.

The synchronization impact on results is important. As indicated in the table, even if it consists in only a few more MPI calls (two barrier synchronizations and one communicator split), it adds more than 50 transitions to the system. In order to emulate a single rendezvous between 5 processes with only sends and receives on a mailbox is costly. Since all those send and receive are using the same resources, there is a lot of dependency between actions. Hence, the search can try multiple interleavings for a same single collective communication. This is the reason why both A* and MinMatch are much worse with synchronization. DFS+Depth explores deeper interleaving first. Therefore, it is not impacted by the earlier synchronizations and is able to find the bug. MaxMatch matches communications as soon as possible and backtracks to state with the most matched communications, which serves as a sort of Depth choice here. SimilarityAvoidance is made to detect the equivalence between states reached after different interleavings of synchronization. Finally, when exploring a new sequence, Uniform does not necessarily try the same ordering as before. Every DFS-based strategy will do the same initial choices after executing the synchronizing transitions in a different order. Therefore, if it did not find the bug the first time, it will not either the second time. Uniform dodges that issue. After trying a first sequence, even if it chooses to interleave a synchronizing transitions, it will eventually make new choices for the end of the execution. Hence, at every try it has a chance to find the bug. This is a second motivation for using Random-based strategy over DFS-based ones.

5.2 Discussion

Code Description	Strategy	Valuation	Visited States	Executed Transitions
pattern	DFS	Depth	83	224
	DFS	MaxMatch	42	114
12 transitions	DFS	MinMatch	20	23
	DFS	SimilarityAvoidance	19	22
	Random	Uniform	23(±15)	37(±39)
	A*	CFG	19	24
pattern	DFS	Depth	115	290
+ reordering	DFS	MaxMatch	19	25
	DFS	MinMatch	44	93
12 transitions	DFS	SimilarityAvoidance	39	69
	Random	Uniform	23(±17)	38(±46)
	A*	CFG	20	25
pattern	DFS	Depth	1246	12416
+ synchronization	DFS	MaxMatch	3613	23677
	DFS	MinMatch	147611	713273
66 transitions	DFS	SimilarityAvoidance	211	400
	Random	Uniform	175(±106)	439(±399)
	A*	CFG	>200K	>500K
pattern	DFS	Depth	1447	16881
+ synchronization	DFS	MaxMatch	2726	16876
+ reordering	DFS	MinMatch	160748	817156
	DFS	SimilarityAvoidance	2466	6262
66 transitions	Random	Uniform	197(±122)	527(±473)
	A*	CFG	>200K	>500K
Heat	DFS	Depth	>100K	>1.1M
	DFS	MaxMatch	>100K	>860K
93 transitions	DFS	MinMatch	320	475
	DFS	SimilarityAvoidance	583	629
	Random	Uniform	559(±1529)	2166(±9173)
	A*	CFG	X	X

Figure 17: Number of explored states visited before finding the bug using Multi-Head DPOR with sleep sets. The pattern uses 5 processes, Heat uses 4.

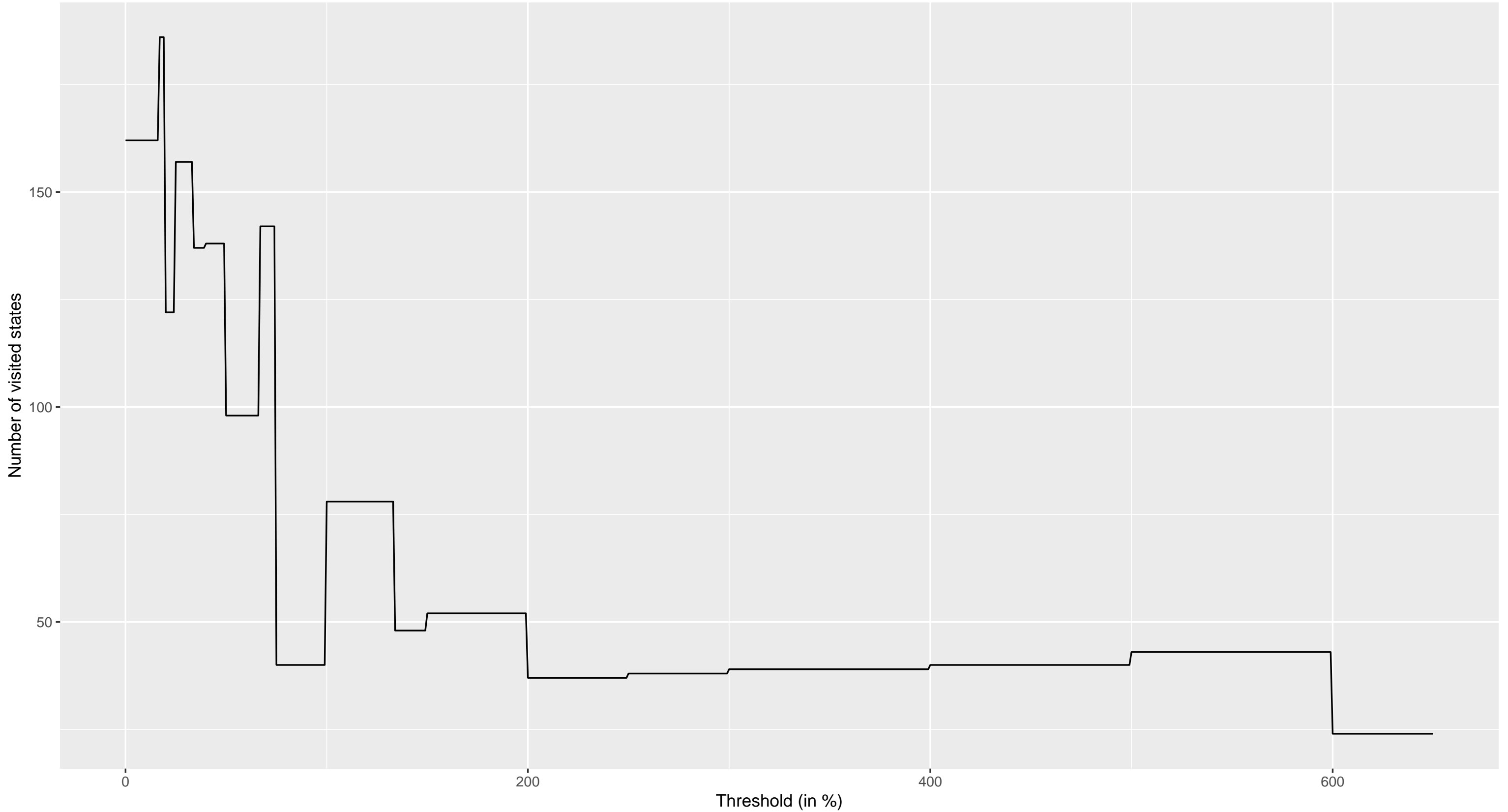

Figure 18: Evolution of the number of visited states over different thresholds for the pattern file with A*+CFG strategy.

The challenge proposed by Heat in terms both of number of interleaving and computation overhead due to computation mimics a small real application. It appears that classical DPOR using sleep set is not efficient enough to find the bug in a reasonable number of steps. To appear, the bug requires that a `send` is delayed for multiple transitions in order for the corresponding `recv` to match the wrong `send`. When this communication pairing happens, it can lead to a deadlock a few transitions later. DPOR has too many things to interchange before reaching this situation. That is the same for MaxMatch that will not consider such execution that does not match a communication right away before a long time. On the other hand, MinMatch delays such matching as much as possible. In that way, the wrong `send` can be matched more easily with the `recv`. SimilarityAvoidance achieves the task thanks to the fact that many patterns in the communication are identical. Hence, it describes a more realistic dependency relation. Finally, in average, random is still quite efficient on the matter. Again, the fact it does not force the order of execution from one sequence to another helps a lot. However, the standard deviation is much bigger. This is because some specific executions are especially worse than the average (worst case being 14672 visited states for 89616 executed transitions).

As stated before, results in 5.2 are those obtained with the original McSimGrid backtracking policy (i.e., backtracking is only done when reaching a terminal node). In theory, that does not interact well with A* algorithm. A* requires taking the best option available at *every* step,

not only the final ones. To take that into account, we implemented a threshold mechanism. If set, the threshold modifies the McSimGrid policy by allowing to backtrack to the best choice in *exploration_heads* if the ratio between the current choice valuation and the best one is above the chosen threshold. For instance, with a 50% threshold and a current best choice of 6, we will backtrack as soon as the current sequence gets a valuation above 9. A threshold of 0% corresponds to always pick the best option among *exploration_heads*. This is what is done in the standard A* algorithm.

Figure 18 shows the behavior of the A* algorithm used in McSimGrid with different thresholds. The evolution is made by levels. This is normal because we are dealing with integer values and discrete thresholds. When the threshold becomes large enough, the number of visited states turns back to the value without using threshold. Furthermore, values obtained for smaller thresholds are quite chaotic and seems to be way worse. This tends to show that using a threshold is not efficient with current implementation of backtrack in McSimGrid.

Not following strict threshold in A* algorithm means that we do not have the correctness of the algorithm anymore. The exploration is still sound, but the counter-example found (i.e., the path in the path-finding associated problem) is not guaranteed to be optimal anymore. In fact, in the pattern example, the counter-example returned by A*+CFG without threshold is of length 11 while the smallest one is of size 8. A more surprising result at first is that A*+CFG and a threshold of 0% (i.e., always taking the optimal, hence implementing a true A* algorithm) does not return an 8 long answer. This is due to the reduction. A* will only return the optimal on the graph it is allowed to explore. With the reduction, we only guarantee that a path equivalent to the counter-example is allowed. We have no guarantee regarding the size of that equivalent path available. Without Persistent sets reduction, the algorithm returns the optimal for counter-example for the program. This experimentation enlightens the following phenomenon: keeping the optimality component of A* while doing reduction is costly while sometimes being worthless. Finally, even if the algorithm without Persistent sets reduction returns the optimal answer, the cost of not using the reduction is too important (231 visited states and 1277 executed transitions, against the 19/24 Figure 5.2 results).

One issue we are having with the A* algorithm is the choice of heuristic. The current CFG heuristic has not been impressive in terms of result. This can be explained partially because of the graph we are considering. A MPI program is written as a single code, multiple data parallelism. Therefore, all the processes have the same static control flow graph, but may end up having very different execution flows. A refinement of the CFG could be done in order to improve the obtained heuristic. With a worse heuristic, the issue is that the algorithm tends to turn into a Breadth First Search (BFS) that is not a good option for the exploration of an exponentially growing tree. Furthermore, a BFS has an important impact in terms of memory. DFS had the advantage to only keep in memory states that are part of the current explored sequence. In BFS, the algorithm can, in the worst cases, keep the whole tree.

6 Future Work

As explained in the result discussion, the CFG analysis we currently perform is very light compared to our needs. Different avenues exist to improve it. One is a refinement based on the rank of each process. This can help to obtain a more precise CFG for a given process. A second idea is to perform a flow analysis on the variable at stakes in the assertion. In fact, we currently ignore the

content of the assertion. However, we could find the transitions that really have an impact on the assertion variables (i.e., that are part of the assertion formula) and focus on interleaving those. Another possible thing would be to transform the MPI CFG to a lower mailbox operation level. Since McSimGrid is executing this lower lever, it helps the precision of the analysis.

There exists a difference between the MPI protocol used by the application and the actual lower level implementation that is verified by McSimGrid. Those differences sometimes lead us to explore interleaving that would make no sense in the world of MPI calls. For instance, if an application calls a `MPI_Barrier` with all its processes, we will try to interleave any calls generated by the barrier. However, we could consider all the obtained low level transitions as single atomic blocks. This way, we would tend to consider interleaving that correspond to the MPI operations. This would require us to prove beforehand the correctness of the implementation of the MPI calls in terms of mailbox operations. But we will do this once, and then be able to gain a lot of exploration time. Practically, we could ensure that when a transition is taken, we execute as much transition corresponding to the same original MPI call as possible.

Finally, one unexplored idea is to inject dependencies between actors one after another. It appears that a vast majority of distributed bugs only concern two or three actors. If it is so, there is no need to interleave all processes every single time. To mitigate this phenomenon, we can leverage our guiding strategies to orient the search along paths interverting only specific processes. Such a guiding strategy is not easy to compose. On the other hand, we could try to adapt the DPOR algorithm, so that the dependency used to determine the happens-before relation is dynamic. During the exploration, we slowly add dependencies between processes so that more and more interleavings are explored until reaching soundness.

7 Conclusion

We have introduced Multi-Head DPOR, a new variation the classical dynamic partial order reduction algorithm for software model checking. This new algorithm aims at augmenting the freedom of the order of the search. Thanks to such freedom, guiding techniques can be applied at a deeper level than before. Techniques such as A* algorithm that could not be implemented before due to the depth first search characteristic of DPOR are now possible.

In addition to this new algorithm, we propose a few more guiding strategies and valuations. These leverage specific pattern of distributed application to fasten the search. We successfully implemented those new strategies as well as classical ones, but adapted to the known specificities of MPI programs.

Introduced strategies combined with Multi-Head DPOR show promising results compared to classical reduction without guided model checking. Different strategies show different results depending on the problem searched and the structure of the program. Those analyses leave for potential new valuations and adaptability of the existing ones for other models such as memory synchronization.

References

[1] "The MPI forum - MPI: a message passing interface," in *Proceedings Supercomputing '93* (B. Borchers and D. Crawford, eds.), pp. 878–883, ACM, 1993.

[2] C. Baier and J. Katoen, *Principles of model checking.* MIT Press, 2008.

[3] S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs, A. Fehnker, and H. Aljazzar, "Survey on directed model checking," in *Model Checking and Artificial Intelligence, 5th International Workshop, MoChArt 2008. Revised Selected and Invited Papers* (D. A. Peled and M. J. Wooldridge, eds.), vol. 5348 of *Lecture Notes in Computer Science*, pp. 65–89, Springer, 2008.

[4] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005* (J. Palsberg and M. Abadi, eds.), pp. 110–121, ACM, 2005.

[5] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14* (S. Jagannathan and P. Sewell, eds.), pp. 373–384, ACM, 2014.

[6] S. Aronis, B. Jonsson, M. Lång, and K. Sagonas, "Optimal dynamic partial order reduction with observers," in *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Proceedings, Part II* (D. Beyer and M. Huisman, eds.), vol. 10806 of *Lecture Notes in Computer Science*, pp. 229–248, Springer, 2018.

[7] C. Rodríguez, M. Sousa, S. Sharma, and D. Kroening, "Unfolding-based partial order reduction," in *26th International Conference on Concurrency Theory, CONCUR 2015* (L. Aceto and D. de Frutos-Escrig, eds.), vol. 42 of *LIPIcs*, pp. 456–469, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[8] T. A. Pham, T. Jéron, and M. Quinson, "Unfolding-based dynamic partial order reduction of asynchronous distributed programs," in *Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019* (J. A. Pérez and N. Yoshida, eds.), vol. 11535 of *Lecture Notes in Computer Science*, pp. 224–241, Springer, 2019.

[9] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[10] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa, A. Gupta, S. Lu, and H. S. Gunawi, "Flymc: Highly scalable testing of complex interleavings in distributed systems," in *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[11] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[12] S. Ariyurek, A. Betin-Can, and E. Surer, "Enhancing the monte carlo tree search algorithm for video game testing," in *2020 IEEE Conference on Games (CoG)*, pp. 25–32, 2020.

[13] N. Lee, Y. Kim, M. Kim, D. Ryu, and J. Baik, "Directed model checking for fast abstract reachability analysis," *IEEE Access*, vol. 9, pp. 158738–158750, 2021.

[14] P. G. Jensen, J. Srba, N. J. Ulrik, and S. M. Virenfeldt, "Automata-driven partial order reduction and guided search for LTL model checking," in *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Proceedings* (B. Finkbeiner and T. Wies, eds.), vol. 13182 of *Lecture Notes in Computer Science*, pp. 151–173, Springer, 2022.

[15] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, pp. 2899–2917, June 2014.

[16] T. Pham, *Efficient state-space exploration for asynchronous distributed programs: Adapting unfolding-based dynamic partial order reduction to MPI programs.* PhD thesis, 12 2019.

[17] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, (USA), p. 75, IEEE Computer Society, 2004.

[18] M. Laurent, E. Saillard, and M. Quinson, "The MPI bugs initiative: a framework for MPI verification tools evaluation," in *5th IEEE/ACM International Workshop on Software Correctness for HPC Applications, Correctness@SC 2021, St. Louis, MO, USA, November 19, 2021* (I. Laguna and C. Rubio-González, eds.), pp. 1–9, IEEE, 2021.

[19] J.-P. Lehr, T. Jammer, and C. Bischof, "Mpi-corrbench: Towards an mpi correctness benchmark suite," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '21, (New York, NY, USA), p. 69–80, Association for Computing Machinery, 2021.

[20] D. Khanna, S. Sharma, C. Rodríguez, and R. Purandare, "Dynamic symbolic verification of MPI programs," in *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings* (K. Havelund, J. Peleska, B. Roscoe, and E. P. de Vink, eds.), vol. 10951 of *Lecture Notes in Computer Science*, pp. 466–484, Springer, 2018.