Noname manuscript No.
(will be inserted by the editor)

Partial Order Reduction for Abstraction-based Verification of Concurrent Software

Csanád Telbisz · **Levente Bajczi** ·
Dániel Szekeres · **András Vörös**

the date of receipt and acceptance should be inserted later

Abstract TBA

Keywords model checking · concurrency · partial order reduction · abstraction

*Funding.*This research was partially funded by the EKÖP-25-3 New National Excellence Program under project number EKÖP-25-3-BME-22, and the Doctoral Excellence Fellowship Programme under project number DKÖP-23-1-BME-5; funded by the National Research Development and Innovation Fund of the Ministry of Culture and Innovation of Hungary.

1 Introduction

Partial order reduction (POR) is an effective technique for handling concurrency, and abstraction is an efficient approach to handling data in model checking. I present a general theoretical framework for combining these model checking paradigms where the advantages of using the two techniques simultaneously are also exploited. I also present a concrete verification algorithm using POR and abstraction together as a case study of my general approach. The novelty of my method lies in the general formulation of POR used during abstract state space exploration. Existing approaches combining these techniques are typically specific to POR or abstraction algorithms [12, 20, 24, 28]. A general approach for combining abstraction and commutativity checking is proposed by *Farzan et al.* [13]. However, that paper does not investigate POR algorithms, simply the general properties of abstract commutativity relations. My work shows how the idea of abstract commutativity applies generally to POR in abstract state space exploration algorithms.

The core concept of POR is to identify equivalent executions [15]. Then, it is enough to check a single representative from each equivalence class. Identifying equivalent interleavings is based on the interaction of threads: dependency is defined between the interacting program operations. Traditionally, a syntactic over-approximation is used

Department of Artificial Intelligence and Systems Engineering
Budapest University of Technology and Economics
Budapest, Hungary
E-mail: {telbisz,bajczi,szekeres,vori}@mit.bme.hu

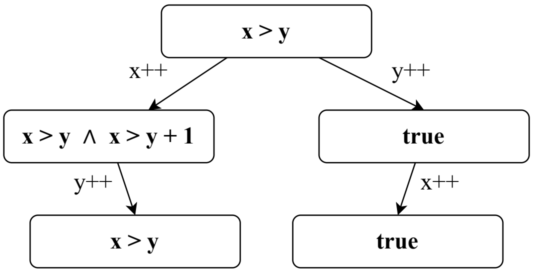

Fig. 1.1: Syntactically independent actions that are not commutative in the abstract state space

as a method of calculating dependency for partial order reduction: two actions are independent if they do not use common shared variables (and the two actions belong to different processes) [15]. For example, let us have a state in the (concrete) state space of the program with two enabled actions from different threads: `x++` and `y++`. No matter, in what order we explore the two actions (`x++`, `y++` or `y++`, `x++`), we will reach the same state since the actions operate on different variables. So we can skip the exploration of one of the two paths.

When it comes to combining POR with abstraction, we face the problem that traditional approaches may calculate invalid dependency relations: it is not trivial to apply POR in an abstract state space where the values of variables are not tracked explicitly [12,24]. Example 1.1 shows a situation where syntactically independent actions are not commutative in an abstract state space.

Example 1.1 Assume that we have an abstraction where we only track the predicates $x > y$ and $x > y + 1$ about our variables. Figure 1.1 demonstrates that two actions that are independent in the traditional sense (no shared variable) may be dependent in the abstract state space. Actions `x++` and `y++` are dependent since they are not commutative in the abstract state space due to the difference in the labeling of the abstract states. Later, Figure 3.2 shows a case where actions with different variables can even disable each other.

Even though the syntactic over-approximation of dependency is not a valid dependency relation in the abstract state space, I show that using it for partial order reduction will always find an error state in the abstract state space if an error is reachable in the concrete state space. Furthermore, I restrict the calculation of dependency, to consider fewer actions dependent in an abstract state space. Intuitively, if the source of dependency between two actions is ignored due to abstraction, it is needless to consider these actions dependent. That is, two actions using the same shared variable are only considered dependent if we track any information about any of their shared variables in the abstraction. As a basic example, take the actions $x = 0$ and $x = 1$. Existing methods consider these actions dependent since they both write x. However, my approach allows considering these actions independent when we do not track any information about x in the abstraction.

To further motivate my approach, it is possible to achieve exponential gains in terms of the number of explored interleavings by using my abstraction-based algorithm for partial order reduction. Consider the example from Figure 1.2 with $2N + 1$

initially: `x:=y:=z:=0`

Process p_0

```
repeat N times:
  z := z+2*y
if z mod 2 = 0:
  x := 0
else:
  x := 1
```

Processes $p_1, ..., p_N$

```
y := y+1
```

Processes $p_{N+1}, ..., p_{2N}$

```
y := y*y
```

finally: `assert(x*y=0)`

Fig. 1.2: Motivational example for possible exponential gain

processes. The safety of the program can be proven with abstraction by only tracking the predicates $z \bmod 2 = 0$ and $x = 0$ about our variables. As $z \bmod 2 = 0$ is an invariant of the loop of p_0, x will get the value 0 which satisfies the assertion. To prove this, traditional methods would explore all interleavings of instructions using y since these actions would be considered dependent due to y: processes $p_1 - p_{2N}$ have $2N!$ interleavings (potentially resulting in different values of y) not considering the loop of p_0; together there are even more possible interleavings. However, my algorithm notices that we do not track any information about y, so it will not consider the actions using y dependent: this way, my method explores a single execution and guarantees the safety of the program.

Summarizing my contributions: I introduce a general abstraction-based partial order reduction approach which is independent of both the underlying POR algorithm and the abstract domain and which uses the information coming from the abstraction to boost partial order reduction. By proving the correctness of using the new abstraction-based dependency relation, I also prove that using the traditional dependency relation for partial order reduction in an abstract state space is correct as well (which is also a non-trivial statement as testified by Example 1.1). I have implemented and evaluated my method in the abstraction-based verification tool Theta [26].

The organization of this chapter is as follows. First, Section 2 introduces the theoretical background. Section 3 presents the novel general approach to combine POR and abstraction in an abstract state space exploration. Section 4 demonstrates an algorithm implementing abstraction-based POR. Finally, I evaluate the approach in Section 5 using the algorithm introduced in Section 4.

2 Preliminaries

We need to discuss some further preliminaries before we can dive into the contributions of this chapter. First, I introduce some extra notation used in this chapter, then I present the basic concept of partial order reduction.

For the states s_1, s_2, and the set of variables $U \subseteq V$, I write $s_1 = s_2$ *on* U to denote that $s_1(v) = s_2(v)$ for each $v \in U$. The notation $w \backslash t$ means the transition sequence obtained from w by removing the first occurrence of t from w.

A transition system is action-deterministic if $|I| \leq 1$ and $|\alpha(s)| \leq 1$ for any state $s \in S$ and action $\alpha \in A$ [7]. The state space of a program is not action-deterministic due to non-deterministic assignments, and uninitialized variables. However, *unknown* is a possible value for variables when using abstraction (see details in **??**): an uninitialized variable or a variable after a non-deterministic assignment gets the specific value *unknown*. This way, the state space becomes action-deterministic. In an action-deterministic transition system, I use $\alpha(s)$ for the single state s' with $(s, \alpha, s') \in T$. Partial order reduction algorithms are classically formulated for action-deterministic transition systems [6, 7, 14]. In some cases, instead of using the term action-deterministic, it is said that control non-determinism is allowed [3]. Some works investigate partial order reduction algorithms in non-deterministic state spaces, also considering types of abstraction introducing non-determinism [18]. In those settings, partial order reduction algorithms need to satisfy different properties. However, this research direction is out of this work's scope, so I assume that the abstract state space is action-deterministic.

A further assumption that we need for the algorithms in this chapter is that the state space is finite. For data variables, this is not a real restriction in most cases, as the domain of program variables are typically bounded (e.g., 32-bit integers in most languages) or abstraction can take care of a finite domain (e.g., the size of the domains of each predicate in predicate abstraction is 2). The restriction concerns concurrent programs where new threads are created dynamically in an infinite loop. Naturally, in real scenarios, we do not need an unbounded number of threads (and in fact, the limitations of the execution environment also limits the possible number of running threads at the same time), so this is also only a theoretical limitation. Besides, it is a common assumption to take in the papers presenting partial order reduction algorithms [2, 15, 27].

2.1 Partial Order Reduction

Partial Order Reduction (POR) is a well-known technique for avoiding the exploration of redundant thread interleavings in the verification of a multi-threaded program [15]. Its key idea is to define an equivalence relation on traces and explore a single representative (or as few as possible) from each equivalence class. Traces are defined to be equivalent if they can be obtained from each other by successively swapping adjacent *independent* actions. An equivalence class is called a *Mazurkiewicz trace* [21]. Intuitively, if adjacent independent actions are swapped, the outcome will remain the same: by exploring a single trace from each equivalence class, we still cover all behaviors of the system. For the above interpretation of equivalence, we need a definition of independence.

Dependency plays a key role in partial order reduction. The general formulation of dependency is as follows [15]:

Definition 2.1 (Valid Dependency Relation) Let $TS = (S, A, T, I)$ be an action-deterministic transition system. Let $D \subseteq A \times A$ be a binary, reflexive, and symmetric relation. D is a *valid dependency relation* if for all $\alpha, \beta \in A$, $(\alpha, \beta) \notin D$ (α and β are *independent*) implies that the following two conditions hold for all $s \in S$:

- if $\alpha \in \mathit{enabled}(s)$, then $\beta \in \mathit{enabled}(s)$ iff $\beta \in \mathit{enabled}(\alpha(s))$, *and*
- if $\alpha, \beta \in \mathit{enabled}(s)$, then $\beta(\alpha(s)) = \alpha(\beta(s))$.

α and β are *dependent* if they are not independent.

The first condition means that independent actions can neither disable nor enable each other. The second property states that independent actions commute. Sometimes, *dependency of transitions* is used in this paper: by the dependency of transitions, I mean dependency of their actions. Note that I will introduce relations in this work that are not *valid* dependency relations; however, for semantic reasons, I will refer to them as dependency relations - without the label *valid*.

Since the goal of POR is to avoid exploring multiple traces leading to the same state, the definition cannot be used directly for determining dependency (two actions should be explored in both orders to decide whether they commute). An appropriate approximation of the dependency relation in an abstract state space is a main contribution of this work.

Many algorithms have been introduced for partial order reduction in the last decades. My abstraction-based extension is orthogonal to the underlying POR algorithm. In this work, I build my presentation on the concept of source sets which is the core of optimal dynamic partial order reduction [1].

3 Abstraction-Aware Partial Order Reduction

This chapter describes how partial order reduction can be integrated into an abstraction-based model checking algorithm. Since the core concept of the approach is to consider extra information about the used abstraction when applying partial order reduction, I call the algorithm *abstraction-aware partial order reduction.* My approach is independent of the underlying POR algorithm as well as the applied abstract domain. The only requirements are that CFA locations of all processes must be explicitly tracked, and the abstract state space must be action-deterministic, that is the transfer function must return a single successor for each state and operation.

3.1 Dependency relations

In my algorithms, different dependency relations are used for partial order reduction. It is important to note that these are not necessarily valid dependency relations in all transition systems. First, I define the syntactic dependency relation.

3.1.1 Syntactic Dependency Relation

A syntactic dependency relation denoted by D_S is the classically used syntactic overapproximation of dependency [15]. That is, two actions $(\alpha, \beta) \in D_S$ (α and β are dependent) iff:

- α and β are actions of the same process, *or*
- $vars(\alpha) \cap vars(\beta) \neq \emptyset$, and at least one variable in $vars(\alpha) \cap vars(\beta)$ is written by α or β.

The syntactic dependency relation is a valid dependency relation in the concrete state space [15]. However, in general, it is not a valid dependency relation in the abstract state space; see the motivating example Example 1.1.

3.1.2 Abstraction-Based Dependency Relation

I introduce a new dependency relation for abstract state spaces (similar conditions are proposed in [13]). An abstraction-based dependency relation D_{Π} is also a syntactic approximation. However, it is defined in an abstraction with respect to the precision of the abstraction.

Definition 3.1 Let us have an abstract state space built with precision Π, and let D_{Π} be a binary, reflexive, and symmetric relation. Two actions $(\alpha, \beta) \in D_{\Pi}$ (α and β are dependent with respect to precision Π) iff:

- α and β are actions of the same process, *or*
- $vars(\alpha) \cap vars(\beta) \cap vars(\Pi) \neq \emptyset$, and at least one variable in $vars(\alpha) \cap vars(\beta) \cap vars(\Pi)$ is written by α or β.

In other words, the second condition means that α and β may still be independent if they use common variables. They are only dependent when the abstraction stores any information about any variable that they both access. Note that D_{Π} is a subset of the syntactic dependency relation D_S for any precision Π: the first condition is the same for both relations, and the second condition of D_{Π} implies the second condition of D_S. As a consequence, D_{Π} is not a valid dependency relation in the abstract state space either (same counterexample from Example 1.1). Furthermore, D_{Π} is not necessarily a valid dependency relation in the concrete state space.

Example 3.1 To illustrate the difference between the syntactic and the abstraction-based dependency relation, consider the actions `x=1` and `[x>0]`. They are dependent based on the syntactic dependency relation since they both use the variable x, and the first action writes it. For the abstraction-based dependency relation, we need an abstraction with a precision. First, let the precision be $\Pi = \{x < y,\, z = 1\}$: then the two actions are dependent in D_{Π} since we have information about x in this abstraction. However, when the precision is $\Pi = \{0 < y,\, z = 1\}$, then our actions are independent in D_{Π} as the value of x is completely ignored in this abstraction.

As we have seen, the introduced relations are not valid dependency relations. However, I show in the next sections that using these relations (D_{Π} in particular) for partial order reduction in an abstract state space with precision Π, feasible errors are still found.

3.2 Partial Feasibility

First, I generalize the concept of abstract trace feasibility by introducing partial feasibility. Intuitively, a partial concretization of an abstract trace is a partial variable assignment for each abstract state of the trace with only a subset of variables assigned so that the partial variable assignment does not contradict the abstract state expressions. The motivation for introducing partial feasibility is that variables ignored in the abstraction may spoil feasibility.

Example 3.2 Let us have a program with the following three independent actions (with the variables being initially zero):

α: `x=1` | β: `[y=0] reach error` | γ: `[z=1] z=0`

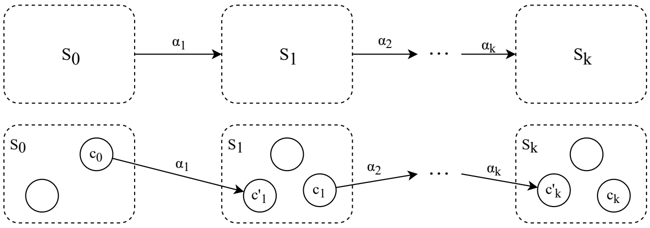

Fig. 3.1: Abstract trace with a partial concretization

Clearly, γ is not enabled in the initial concrete state due to z being zero. Let us have the abstraction where only the variables x and y are tracked explicitly. Now, γ is enabled in the abstract initial state as we have no information about z. A partial order reduction algorithm may explore the trace $w = \gamma.\alpha.\beta$ and no other traces as all actions are independent. Even though w is not feasible, it will be enough for us in a sense that is formalized by partial feasibility.

Definition 3.2 An abstract trace $w = \alpha_1 ... \alpha_k$ from an abstract state s_0 ($s_0 \xrightarrow{\alpha_1} ... \xrightarrow{\alpha_k} s_k$) is *partially feasible* for the set of variables $P \subseteq V$ if there are concrete states $c_0, ..., c_k$ such that:

- $c_i \models s_i$ for each $0 \leq i \leq k$, and
- for each $0 \leq i < k$, $\exists c'_{i+1}$ such that $c_i \xrightarrow{\alpha_{i+1}} c'_{i+1}$, $c'_{i+1}(p) = c_{i+1}(p)$ for each process p, and $c'_{i+1} = c_{i+1}$ on P.

We refer to $c_0, ..., c_k$ as a *partial concretization* of w for P.

If w consists of a single action α, I say that α is a partially feasible action from the abstract state. Note that w being partially feasible for V means that w is feasible. The connection between feasibility and partial feasibility is straightforward: if w is not partially feasible for a $P \subseteq V$, then w is not feasible; and if w is feasible, then w is partially feasible for any $P \subseteq V$. In practice, I will use $P = vars(\Pi)$ for a precision Π: thus, partial feasibility will only depend on variables that we have information about in the current abstraction.

Example 3.3 Figure 3.1 illustrates partial feasibility: we have the abstract trace $\alpha_1 ... \alpha_k$ from the abstract state s_0 and it has a partial concretization $c_0, ..., c_k$ (for the sake of simplicity, I omitted the values of variables from the figure).

We can see an example of an abstract trace that is not partially feasible for $\{x, y\}$ in Figure 3.2. The figure shows an abstract state space built with a precision consisting of the predicates $x + y \leq 2$ and $y \geq 2$. The state expression of the initial state is *true*, and it remains *true* even after executing $x = 0$ since none of the predicates or their negated form is a consequence of this action. However, $y = 2 - x$ implies $x + y \leq 2$. It still allows both $[x \geq 2]$ and $[y \geq 2]$ to be enabled. Even though $x = 0$, $y = 2 - x$, $[x \geq 2]$ is an abstract trace from the initial state, it is trivially not partially feasible since there is no possible partial variable assignment for $vars(\Pi) = \{x, y\}$ meeting the requirements of Definition 3.2 due to x. Also note that the actions $[x \geq 2]$ and $[y \geq 2]$ (belonging to different processes) disable each other even though they are independent based on the syntactic dependency relation.

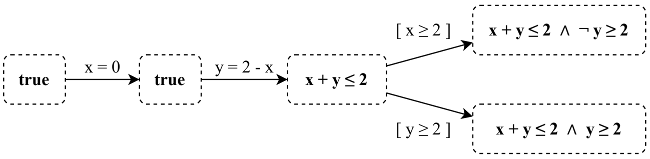

Fig. 3.2: Example of a partially infeasible abstract trace

The following lemma states that if we have a sequence of concrete states $c_0, ..., c_k$ that satisfy the conditions in Definition 3.2, then there is indeed an abstract trace in the abstract state space whose partial concretization is the sequence $c_0, ..., c_k$.

Lemma 3.1 *Let Π be the precision of the abstraction and $P = vars(\Pi)$; and $c_0, ..., c_k$ be concrete states such that for each $0 \leq i < k$, $\exists c'_{i+1}$ with $c_i \xrightarrow{\alpha_{i+1}} c'_{i+1}$, $c'_{i+1}(p) = c_{i+1}(p)$ for each process p, and $c'_{i+1} = c_{i+1}$ on P. Let s_0 be any abstract state with $c_0 \models s_0$. Then: $\alpha_1 ... \alpha_k$ is an abstract trace that exists in the abstract state space from s_0.*

Proof. The abstract state space being an over-approximation of the concrete state space means that if there is a transition $c_i \xrightarrow{\alpha_{i+1}} c'_{i+1}$ in the concrete state space, then there is a transition $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ in the abstract state space where $c_i \models s_i$ and $c'_{i+1} \models s_{i+1}$. The expression function of abstract states only uses the variables in the precision. Thus, from $c'_{i+1}(p) = c_{i+1}(p)$ for each process p, and $c'_{i+1} = c_{i+1}$ on P, it follows that $c'_{i+1} \models s_{i+1}$ implies that $c_{i+1} \models s_{i+1}$. Now, with induction for i from 0 to $k-1$, we get that $\alpha_1 ... \alpha_k$ is an abstract trace in the abstract state space from s_0. □

3.3 Relaxed Partial Order Representation

We can connect the partial order representation defined by a dependency relation with partial feasibility. Let $w_1 \approx^{\Pi} w_2$ denote that abstract traces w_1 and w_2 can be transformed into each other by successively swapping adjacent actions that are independent in D_{Π}. Thus, the relation $\approx^{\Pi}$ defines equivalence classes (Mazurkiewicz traces [21]) on abstract traces. The following theorem states that either all abstract traces are partially feasible in such an equivalence class or none.

Theorem 3.1 *Let us have an abstract state space S built with precision Π, let $s \in S$ be an abstract state, w_1 and w_2 be transition sequences with $w_1 \approx^{\Pi} w_2$, and $P = vars(\Pi)$.*

Then, w_1 is a partially feasible abstract trace from s for P iff w_2 is a partially feasible abstract trace from s for P.

Proof. I prove that swapping two adjacent actions independent based on D_{Π} in a partially feasible abstract trace w will result in an abstract trace w' that is also partially feasible. The case is symmetric for w_1 and w_2: I can assume that w_1 is partially feasible. Since $w_1 \approx^{\Pi} w_2$ means that the partially feasible abstract trace w_1 can be transformed into w_2 by successively swapping adjacent independent actions, and partial feasibility is preserved in each step, it follows that w_2 is partially feasible.

Let $w = q.\alpha.\beta.r$ for some traces q and r ($s \xrightarrow{q} s_q \xrightarrow{\alpha} s_\alpha \xrightarrow{\beta} s_{\alpha\beta} \xrightarrow{r} s_w$) and $w' = q.\beta.\alpha.r$ from state s with w being partially feasible from s for P, and $(\alpha, \beta) \notin D_{\Pi}$. I check that w' is also partially feasible from s for P. Let $A = vars(\alpha)$, and $B = vars(\beta)$.

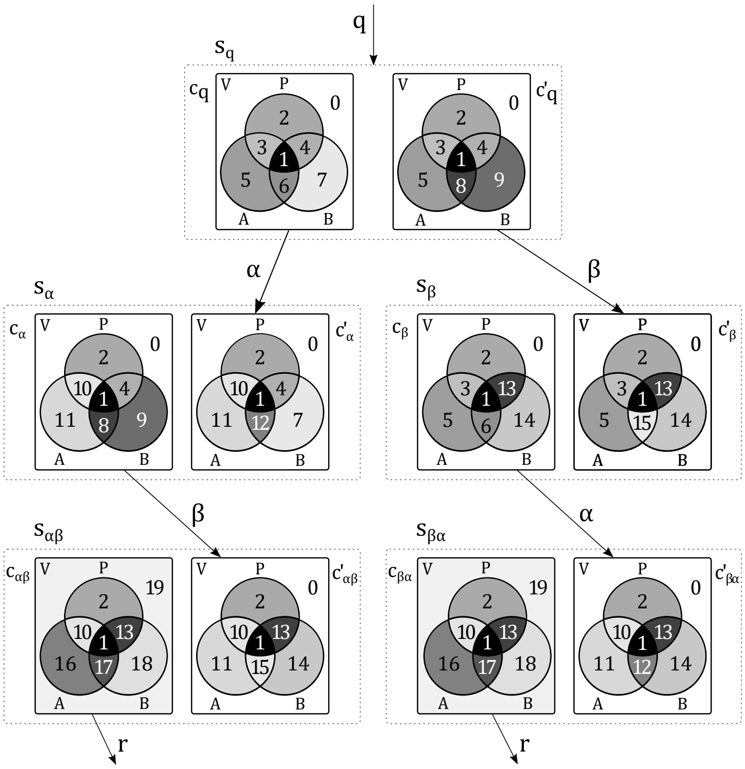

Fig. 3.3: Variables of states in the proof of Theorem 3.1.

Since w is partially feasible, we have a partial concretization $c, ..., c_q, c_\alpha, c_{\alpha\beta}, ..., c_w$ of w with $c_q \models s_q$, $c_\alpha \models s_\alpha$, $c_{\alpha\beta} \models s_{\alpha\beta}$. Ignoring the end of w, we get that $c, ..., c_q, c_\alpha, c_{\alpha\beta}$ is a partial concretization of $q.\alpha.\beta$ for P. My goal is to show that there is a partial concretization $c, ..., c'_q, c_\beta, c_{\beta\alpha}$ of $q.\beta.\alpha$ such that $c_{\alpha\beta} = c_{\beta\alpha}$.

Throughout the proof, I will compare the values of variables in different concrete states. For this, as a reference, let us make the following observations:

Observation 1. *Since $(\alpha, \beta) \notin D_\Pi$, $A \cap B \cap P = \emptyset$ or neither α nor β modifies any variable in $A \cap B \cap P$. In both cases, $c_1(v) = c_2(v)$ for each variable $v \in A \cap B \cap P$ for any concrete states c_1, c_2 with $c_1 \xrightarrow{\alpha} c_2$ or $c_1 \xrightarrow{\beta} c_2$.*

Observation 2. *The action α can only change the values of variables in A, so if we have concrete states c_1 and c_2 with $c_1 \xrightarrow{\alpha} c_2$, then $c_1 = c_2$ on $\overline{A}$. Similarly for β.*

In Figure 3.3, I visualize the concrete and abstract states appearing in the proof. The variable values are represented by Venn diagrams in each concrete state. Sets with the same colors (and numbers) indicate that variables belonging to them have the same values. The colored numbers in the text should be interpreted such that the set of variables mentioned just before the colored numbers is the union of sets represented by the numbers.

Using Definition 3.2 of partial feasibility for the partial concretization $c, ..., c_q, c_\alpha, c_{\alpha\beta}$: there is a concrete state c'_α such that $c_q \xrightarrow{\alpha} c'_\alpha$ and $c'_\alpha = c_\alpha$ on P $\boxed{1}\boxed{2}\boxed{4}\boxed{10}$; similarly, there is a concrete state $c'_{\alpha\beta}$ such that $c_\alpha \xrightarrow{\beta} c'_{\alpha\beta}$ and $c'_{\alpha\beta} = c_{\alpha\beta}$ on P $\boxed{1}\boxed{2}\boxed{10}\boxed{13}$.

Based on Observation 2 for α, $c_q = c'_\alpha$ on $\overline{A}$ $\boxed{0}\boxed{2}\boxed{4}\boxed{7}$. Putting it together, $c_q = c_\alpha$ on $P \setminus A$ $\boxed{2}\boxed{4}$. Let c'_q such that $c'_q(p) = c_q(p)$ for each process p[1], and $c'_q = c_q$ on $\overline{B} \cup P$ $\boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}$. Also, let $c'_q = c_\alpha$ on $B \setminus P$ $\boxed{8}\boxed{9}$. Defining c'_q this way, with Observation 1 we can conclude that $c'_q = c_\alpha$ on B $\boxed{1}\boxed{4}\boxed{8}\boxed{9}$.

From partial feasibility we know that β is enabled in c_α. The guard condition of β only depends on the values of variables in B. Therefore, by the fact that $c'_q = c_\alpha$ on B, β is enabled in c'_q: there is a concrete state c'_β with $c'_q \xrightarrow{\beta} c'_\beta$.

Let c_β be a concrete state with $c_\beta = c'_\beta$ on $\overline{A} \cup P$ $\boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{13}\boxed{14}$. Also, let $c_\beta = c_q$ on $A \setminus P$ $\boxed{5}\boxed{6}$. Using the definitions of c_β and c'_q, and Observation 2 for β: $c_\beta = c'_\beta = c'_q = c_q$ on $(A \cap P) \setminus B$ $\boxed{3}$. By the definition of c_β and Observation 1, $c_\beta = c_q$ on A $\boxed{1}\boxed{3}\boxed{5}\boxed{6}$. This way, the fact that $\alpha \in \mathit{enabled}(c_q)$ implies that α is enabled in c_β.

Now let $c'_{\beta\alpha}$ be such that $c_\beta \xrightarrow{\alpha} c'_{\beta\alpha}$. Let $c_{\beta\alpha}$ be such that $c_{\beta\alpha} = c'_{\beta\alpha}$ on P $\boxed{1}\boxed{2}\boxed{10}\boxed{13}$. Also, let $c_{\beta\alpha} = c_{\alpha\beta}$ on $\overline{P}$ $\boxed{16}\boxed{17}\boxed{18}\boxed{19}$. I show that $c_{\beta\alpha} = c_{\alpha\beta}$ even on P $\boxed{1}\boxed{2}\boxed{10}\boxed{13}$.

Again, consider that $c_q = c_\beta$ on A $\boxed{1}\boxed{3}\boxed{5}\boxed{6}$, thus α from c_q and c_β will result in the same new values for $v \in A$: $c'_\alpha = c'_{\beta\alpha}$ on A $\boxed{1}\boxed{10}\boxed{11}\boxed{12}$. From Observation 2 for β, and the definitions of the following states, $c_{\alpha\beta} = c'_{\alpha\beta} = c_\alpha = c'_\alpha = c'_{\beta\alpha} = c_{\beta\alpha}$ on $A \cap P$ $\boxed{1}\boxed{10}$. Symmetrically, $c_{\alpha\beta} = c'_{\alpha\beta} = c'_\beta = c_\beta = c'_{\beta\alpha} = c_{\beta\alpha}$ on $B \cap P$ $\boxed{1}\boxed{13}$. Finally, $c_{\beta\alpha} = c_q = c_{\alpha\beta}$ on $P \setminus (A \cup B)$, since neither α, β, nor the definition of any concrete state changed the value of such a variable v $\boxed{2}$.

Thus, $c_{\beta\alpha} = c_{\alpha\beta}$ on V (all variables). It can be easily seen that process locations are the same in $c_{\beta\alpha}$ and $c_{\alpha\beta}$, so $c_{\beta\alpha} = c_{\alpha\beta}$. Based on the above definitions of c_β and $c_{\beta\alpha}$, the above proven property that $c_{\beta\alpha} = c_{\alpha\beta}$, and using Lemma 3.1 for the sequence of concrete states $c, ..., c_q, c_\beta, c_{\beta\alpha}, ..., c_w$, we get that $w' = q.\beta.\alpha.r$ is a partially feasible abstract trace for P from the abstract state s which proves the theorem. □

Also, take the following lemma which states that a partially feasible trace w can be extended to a partially feasible trace with an action that is enabled in the first concrete state of any partial concretization of w; it will be useful later.

Lemma 3.2 *Let Π be the precision of the abstraction, s be an abstract state, and let the trace $w = w_1 ... w_n$ be a partially feasible trace from s for $vars(\Pi)$. Let $c_0, ..., c_n$ be a partial concretization of w from s, and let $\alpha \in enabled(c_0)$ such that $(\alpha, w_i) \notin D_\Pi$ for each $1 \leq i \leq n$. Then, $w.\alpha$ is also a partially feasible trace from s for $vars(\Pi)$.*

[1] For better readability, I will omit CFA locations from now on. Based on the notations, all states with the same name differing only in an apostrophe have the same CFA locations for each process.

Proof. Throughout the proof, $1 \leq i \leq n$ and $0 \leq j \leq n$. Let $W = \bigcup_i vars(w_i)$, that is, the variables used by any action of w; and let $c_\alpha = \alpha(c_0)$.

I define the concrete states c'_j such that $c'_j = c_\alpha$ on $vars(\alpha) \setminus W$, and $c'_j(v) = c_j(v)$ otherwise. As for the CFA locations, $c'_j(p) = c_j(p)$ for each process $p \neq p_\alpha$, and $c'_j(p_\alpha) = c_\alpha(p_\alpha)$.

It can be easily checked that the conditions of Lemma 3.1 are met by the concrete states $c_0, c'_0, ..., c'_n$. For c_0 and c'_0, there is c_α with $c_0 \xrightarrow{\alpha} c_\alpha$, and $c_\alpha = c'_0$ on $vars(\varPi)$ (see the definitions of these states and Observation 1 for α and w_i). From c'_0, the conditions are met because $c_0, ..., c_n$ is a partial concretization of w for $vars(\varPi)$, and $c'_j = c_j$ on W, and $c'_j(p_{w_i}) = c_j(p_{w_i})$ for each process p_{w_i} (as $p_\alpha \neq p_{w_i}$ based on $(\alpha, w_i) \notin D_\varPi$).

Then, by Lemma 3.1, the trace $\alpha.w$ is partially feasible from s for $vars(\varPi)$ with the partial concretization $c_0, c'_0, ..., c'_n$. Since $(\alpha, w_i) \notin D_\varPi$, $\alpha.w \approx^\varPi w.\alpha$ which implies by Theorem 3.1 that $w.\alpha$ is partially feasible from s for $vars(\varPi)$. □

So far, I have defined equivalence classes on abstract traces with the relation $D_\varPi$. Evidently, if a partially feasible abstract trace reaches an error state, then all other traces in its equivalence class reach an error state. It comes from the fact that error states are defined as states where any process is in an error location. Since all traces of an equivalence class contain the same actions (cf. they can be obtained from each other by successively swapping certain actions), if an action leads to an error location, it will reach the error in all traces regardless of the order of actions.

3.4 Source Sets for Abstraction-Aware Partial Order Reduction

We need an algorithm that explores only a single trace (or as few as possible) per equivalence class. Any POR algorithm could be chosen from the literature. Since I strive for generality, I use the concept of source sets [3] as necessary conditions on the correctness of a POR algorithm can be formulated with source sets [2]. I slightly modify its formulation to demonstrate that using the abstraction-based (or the traditional) dependency relation for partial order reduction in the abstract state space yields correct results.

Using the relation $\approx^\varPi$ on abstract traces, I relax the definition of *weak initials* and *source sets* from [3]:

Definition 3.3 (Weak Initials) Let s be an abstract state, $\varPi$ be the precision of the abstraction, and w be a transition sequence from s such that w is partially feasible from s for $vars(\varPi)$. For w, the set $WI^\varPi_s(w)$ of weak initials in s is a set of actions: $\alpha \in WI^\varPi_s(w)$ iff there are transition sequences v_1 and v_2 such that $\alpha.v_1 \approx^\varPi w.v_2$, and $w.v_2$ is partially feasible from s for $vars(\varPi)$.

Definition 3.4 (Source Sets) Let s be a state, and W be a set of transition sequences such that w is an abstract trace from s for each $w \in W$. A set $P \subseteq enabled(s)$ is a *source set for* W *in* s if for each transition sequence $w \in W$, $WI^\varPi_s(w) \cap P \neq \emptyset$.

For abstract traces w and w' from the abstract state s, I use $w \sqsubseteq_s w'$ to denote that $w.v \approx^\varPi w'$ for some transition sequence v.

Example 3.4 To demonstrate weak initials and source sets, take the program of Figure 1.2, and let s_0 be the abstract initial state. Consider two precisions Π_1 and Π_2. In the first case, let us explicitly track all variables of the program, so $\mathit{vars}(\Pi_1) = \{x, y, z\}$. In the other case, let us only track the predicates $z \bmod 2 = 0$ and $x = 0$, so $\mathit{vars}(\Pi_2) = \{x, z\}$.

Let w_1 be the following trace of the program: first, the actions of processes p_1, ..., p_{2N} in order (actions writing y), then the actions of p_0 (with first branch of the *if*). Let w_2 another trace, where the actions of p_0 come first, and then the actions of processes p_1, ..., p_{2N}. Evidently, both w_1 and w_2 are feasible traces from s_0 in both abstract state spaces (explored with precision Π_1 and Π_2).

In the first case, the weak initials $WI^{\Pi_1}_{s_0}(w_1)$ only contains $y := y + 1$ (of p_1). To see this, note that no action α of the program is independent in D_{Π_1} with all preceding actions of α in w_1 due to y being used by the first actions of all threads: it is impossible to consecutively swap adjacent independent elements to obtain another trace w'_1 starting with α ($\nexists w'_1$ such that w'_1 starts with α and $w'_1 \approx^{\Pi_1} w_1$). Similarly, $WI^{\Pi_1}_{s_0}(w_2) = \{z := z + 2 * y\}$. Since w_1 and w_2 are both transition sequences from s_0, neither $P_1 = \{y := y + 1\}$ nor $P_2 = \{z := z + 2 * y\}$ is a source set for all transition sequences in s_0, since $P_1 \cap WI^{\Pi_1}_{s_0}(w_2) = P_2 \cap WI^{\Pi_1}_{s_0}(w_1) = \emptyset$. In fact, the only source set in s_0 is $\mathit{enabled}(s_0)$ itself.

In the second case, actions using y are not dependent in D_{Π_2}. So $WI^{\Pi_2}_{s_0}(w_1)$ and $WI^{\Pi_2}_{s_0}(w_2)$ both contain all enabled actions in s_0 (the first actions of all threads can be swapped with independent swaps). In fact, $WI^{\Pi_2}_{s_0}(w) = \mathit{enabled}(s_0)$ for any trace w starting from s_0. Therefore, any action $\alpha \in \mathit{enabled}(s_0)$ forms a source set alone since α is part of the weak initials of all traces starting from s_0: e.g., $\{z := z + 2 * y\}$ is a source set in s_0.

We can use this relaxed definition of source sets to formulate the correctness of abstraction-aware partial order reduction. The goal is to reduce the size of the abstract state space by exploring only a subset of enabled actions from each state. Generally, it is required from such a state space reduction that if an error state can be reached in the original state space, then an error state is also reachable in the reduced state space; this is to ensure that reachability analysis performed in the reduced and the original abstract state space yield equivalent results.

However, in our context of abstraction, it is enough to have an error state in the reduced abstract state space if there is a feasible trace from the initial abstract state to an abstract error state in the original abstract state space. That is, if abstract error states can only be reached with spurious traces (meaning that there is no error state in the concrete state space), there is no need to include an error state in the reduced state space.

Lemma 3.3 *Let s be an abstract state, Π be the precision of the abstraction, and w be a trace from s such that w is partially feasible from s for $\mathit{vars}(\Pi)$. If $\alpha \in WI^{\Pi}_{s}(w)$ and $\alpha \notin w$, then*

1. *$w.\alpha$ is also partially feasible from s for $\mathit{vars}(\Pi)$, and*
2. *$\alpha.w \approx^{\Pi} w.\alpha$.*

Proof. From Definition 3.3 of weak initials, there are transition sequences v_1 and v_2 such that $\alpha.v_1 \approx^{\Pi} w.v_2$. This means that $w.v_2$ can be obtained from $\alpha.v_1$ by successively swapping adjacent independent actions. Since $\alpha \notin w$, $\alpha \in v_2$ necessarily. Also, for any action $\beta \in v_1$ but $\beta \notin w$, $\beta \in v_2$. Thus, any $\beta \notin w$ (including α) preceding any action $\gamma \in w$ in $\alpha.v_1$ must be independent with such γ actions: $(\beta, \gamma) \notin D_\Pi$.

So we can first move all such β after w by successive independent swapping steps without swapping α with any other such β. Thus, we get the transition sequence $w.\alpha.v_2'$ such that $\alpha.v_1 \approx^\Pi w.\alpha.v_2' \approx^\Pi w.v_2$. Since $w.v_2$ is partially feasible from s based on the definition of weak initials, $w.\alpha.v_2'$ is partially feasible as well by Theorem 3.1. Thus, its prefix $w.\alpha$ is partially feasible from s which proves the first statement.

For the second statement, take again that $(\alpha, \gamma) \notin D_\Pi$ for each $\gamma \in w$ since α precedes every other action in $\alpha.v_1$. This, by definition of the relation $\approx^\Pi$ means that $\alpha.w \approx^\Pi w.\alpha$. □

Let $W_\Pi(s)$ denote the set of partially feasible traces from an abstract state s for $vars(\Pi)$ in an abstract state space with precision Π. The following theorem (a modified version of the corresponding theorem in [2]) guarantees the correctness of a partial order reduction algorithm with certain conditions.

Theorem 3.2 *Let S be the original abstract state space built with precision Π, and S_R be the reduced abstract state space obtained from S by restricting the set of actions that are explored from each state. If the following two conditions are satisfied:*

1. *for each state s in S_R, the set of explored actions is a source set for $W_\Pi(s)$ in s,*
2. *for each cycle in S_R, if an action α is enabled in all states of the cycle, then α is explored from some state of the cycle,*

then for each state s in S_R and abstract trace w from s in S such that w is partially feasible for $vars(\Pi)$, there is a transition sequence w' in S_R such that $w \sqsubseteq_s w'$.[2]

The proof proceeds similarly to the proof of the original theorem in [2] though some statements need more thorough justification.

Proof. I prove the theorem by induction on the length of w. The base case with $|w| = 0$ is trivial. For the inductive step, let us have the trace $w \in W_\Pi(s)$: by definition of $W_\Pi(s)$, w is partially feasible from s. By condition (1), some action $\alpha \in WI_s^\Pi(w)$ is explored from s in S_R. We have two cases:

1. $\alpha \in w$
   By definition, $\alpha \in WI_s^\Pi(w)$ means that $\alpha.(w \backslash \alpha) \approx^\Pi w$ (otherwise, $\alpha.v_1$ could not be transformed into $w.v_2$ by successive independent swapping steps when applying Definition 3.3 for α and w). This implies together with the assumption that w is partially feasible from s that $\alpha.(w \backslash \alpha)$ is also a partially feasible abstract trace from s based on Theorem 3.1. As a consequence, $(w \backslash \alpha)$ is a partially feasible abstract trace for $vars(\Pi)$ from $\alpha(s)$.
   From the induction hypothesis for state $\alpha(s)$ and the partially feasible trace $(w \backslash \alpha)$ from $\alpha(s)$, we know that the reduced state space S_R contains a trace w'' with $(w \backslash \alpha) \sqsubseteq_{\alpha(s)} w''$. This way, S_R also contains the trace $\alpha.w''$ from s. We now have that $\alpha.(w \backslash \alpha) \sqsubseteq_s \alpha.w''$. From this, along with $\alpha.(w \backslash \alpha) \approx^\Pi w$, we can easily infer from the definitions of $\approx^\Pi$ and $\sqsubseteq_s$ that $w \sqsubseteq_s \alpha.w''$, so we can take $\alpha.w''$ as w' in the theorem.

[2] Note that w' is not guaranteed to be partially feasible from s.

2. $\alpha \notin w$
   Let $\alpha_1 = \alpha$. Then, using Lemma 3.3, $\alpha_1 \in WI_s^{\Pi}(w)$, $\alpha_1 \notin w$, and the assumption about the partial feasibility of w imply that $w.\alpha_1$ is a partially feasible trace from s, and $\alpha_1.w \approx^{\Pi} w.\alpha_1$. This, together with Theorem 3.1 implies that $\alpha_1.w$ is also a partially feasible trace from s. Thus, w is a partially feasible trace from $\alpha_1(s)$. Again, by condition (1), some action $\alpha_2 \in WI^{\Pi}_{\alpha_1(s)}(w)$ is explored from $\alpha_1(s)$ in S_R. Continuing in this way, we have two cases:
   - There is a sequence of actions $\alpha_1, ..., \alpha_k$ such that for each $1 \leq i \leq k-1$, w is a partially feasible trace from $\alpha_{i-1}(...(\alpha_1(s)))$ with $\alpha_i \notin w$, and $\alpha_k \in w$, and for each $1 \leq i \leq k$, $\alpha_i \in WI^{\Pi}_{\alpha_{i-1}(...(\alpha_1(s)))}(w)$. Now, we can extend the reasoning in case 1 of the proof (with α_k being in a similar position as α in case 1): we have a trace w'' such that S_R contains the trace $\alpha_1...\alpha_k.w''$ from s such that $\alpha_1...\alpha_k.(w \backslash \alpha_k) \sqsubseteq_s \alpha_1...\alpha_k.w''$. Knowing that $\alpha_1...\alpha_k.(w \backslash \alpha_k) \approx^{\Pi} \alpha_k.(w \backslash \alpha_k).\alpha_1...\alpha_{k-1}$ implies $\alpha_k.(w \backslash \alpha_k) \sqsubseteq_s \alpha_1...\alpha_k.w''$. Since $\alpha_k.(w \backslash \alpha_k) \approx^{\Pi} w$, we have that $w \sqsubseteq_s \alpha_1...\alpha_k.w''$, so we can take $\alpha_1...\alpha_k.w''$ as w' in the theorem.
   - There is an unbounded sequence of actions $\alpha_1, \alpha_2, ...$ such that for each $i = 1, 2, ...$, w is a partially feasible trace from $\alpha_{i-1}(...(\alpha_1(s)))$, $\alpha_i \notin w$ but $\alpha_i \in WI^{\Pi}_{\alpha_{i-1}(...(\alpha_1(s)))}(w)$. Consequently, there must be a loop in this unbounded sequence of actions (since the state space if finite) with the first action w_1 of w enabled in all states of the loop. By condition (2), w_1 must be explored from at least one state of the loop, and we are back in case 1 of the proof.

□

If an error state is reachable in the concrete state space from the initial state with trace w, w is also an abstract trace from the abstract initial state s_0 leading to an abstract error state, since the abstract state space is an over-approximation of the concrete state space. As w is a feasible trace, it is also partially feasible for any subset of variables. As a consequence, Theorem 3.2 can be used for w: there is an abstract trace w' in the reduced abstract state space with $w \sqsubseteq_{s_0} w'$. Since error locations are sinks in the CFA, any state reachable from an error state is also an error state, so w' also reaches an abstract error state. Note that this w' is not necessarily feasible, only partially feasible for the variables in the precision. However, in such a case, the refinement step of the CEGAR algorithm will realize that w' is not feasible, and it will refine the abstraction.

4 Static Abstraction-Aware Partial Order Reduction Algorithm

Several algorithms have been presented in the literature for partial order reduction [1,14,15]. There are two main approaches: *static* and *dynamic* partial order reduction [7]. In the static version, the model (i.e., the CFA of the program) is analyzed, and the reduced state space is generated based on this static analysis. The dynamic approach constructs the reduced state space during the model checking. Although abstraction-aware partial order reduction uses on-the-fly information about the current abstraction, its core partial order reduction algorithm can be implemented in a static fashion.

Though my abstraction-aware extension can be applied to any partial order reduction algorithm, dynamic approaches tend to have much more complex formulations [2],

Algorithm 1: Calculating a Source Set from s with Π

```
Input: $s, \Pi$
Output: $P$                                   /* Source set in $s$ */
$P \leftarrow \{\alpha\}$ for some $\alpha \in enabled(s)$
while any new item is added to $P$ do
    $P \leftarrow P \cup \{\alpha \ : \ \alpha \in enabled(s) \setminus P, \ \exists \beta \in future\_actions(s, \alpha), \exists \gamma \in P$ with $(\beta, \gamma) \in D_\Pi\}$
```

so I will restrict myself to the presentation of a static approach in this paper. My static source set-based abstraction-aware partial order reduction algorithm is similar to Overman's algorithm [15,22]. Before presenting the algorithm that calculates source sets, the following notions are introduced (similar concepts can be found in [2]):

Definition 4.1 (May-enabled action in a state) An action α is *may-enabled* in a state s, if $\alpha \in enabled(s)$ or α can become enabled after a sequence of transitions from processes other than p_α.

An action $\alpha \notin enabled(s)$ can be *may-enabled* in a state s when the source location of α is $s(p_\alpha)$, but the guard condition of α evaluates to false in s.[3]

Definition 4.2 (Future actions) For an $\alpha \in enabled(s)$, $future\_actions(s, \alpha)$ is a set of actions: $\beta \in future\_actions(s, \alpha)$ iff there is a transition sequence $w = w_1 ... w_n$ from s, where $w_n = \beta$, and the first action β_0 of p_β in w is either α or $\beta_0 \notin enabled(s)$.

The condition on β_0 in the definition of *future_actions* implies that β_0 is may-enabled in s. If β is in the same process as α, then $\beta_0 = \alpha$. Otherwise, β_0 is not enabled in s.

We can easily compute an over-approximation of $future\_actions(s, \alpha)$ by analyzing the static model of the program. Initially, the actions of process p_α are collected with a graph search of the CFA of p_α. An action $\beta \notin enabled(s)$ from another process that is may-enabled in s can be enabled by an action γ reached in the CFA of p_α (when γ writes a variable that β uses in its guard condition). Then, $future\_actions(s, \beta)$ is computed recursively to collect more future actions.

With the help of *future_actions*, we can compute source sets in a state. The current state s and the precision Π of the current abstraction are the inputs of Algorithm 1. Initially, any enabled action (or practically all enabled actions of a single process) is put in the source set(-to-be) P. As long as any new action is added to P, the following is repeated: $future\_actions(s, \alpha)$ is calculated for each $\alpha \in enabled(s) \setminus P$. If there is any action $\beta \in future\_actions(s, \alpha)$ that is dependent with an action $\gamma \in P$, α is added to P.

Before proving the correctness of Algorithm 1, I note the following property of software. Branches in a program are defined in an if-elseif-else manner, that is the execution can proceed on a branch independent of the variable assignment.

Property 4.1 For any location l of a CFA the disjunction of guard conditions of all outgoing actions from l is *true*.[4]

[3] If processes can be created or terminated dynamically, actions can be may-enabled in other ways, too; e.g., first actions of processes and join operations.

[4] An action without a guard is technically an action whose guard is *true*.

Theorem 4.1 *A set P returned by Algorithm 1 for a state s and precision Π is a source set in s for $W_\Pi(s)$.*

Proof. Let us check the definition of source sets, that is, for each trace from s, one of its weak initials is in P. Let $w = w_1 ... w_n \in W_\Pi(s)$ be a partially feasible abstract trace from s. We have two cases:

1. $\exists w_i \in P$ for some $1 \leq i \leq n$ (that is w contains an action from P). Let $w_f \in P$ be the first occurrence of an element of P in w: $w_j \notin P$ for $1 \leq j < f$.
   Then, $(w_j, w_f) \notin D_\Pi$. To see this, assume the opposite: there is a w_d dependent with w_f based on D_Π, and $d < f$. Since w_f is the first action from P in w, w_d can be reached from s with actions from processes that do not have actions in P. That is, $w_d \in \mathit{future\_actions}(s, \alpha)$ for some $\alpha \in \mathit{enabled}(s) \setminus P$. In this case, Algorithm 1 would have added α to P in line 3 (with w_d as β, w_f as γ, and α as α using the notation of the algorithm). This did not happen, so our indirect supposition is wrong.
   Since w_f is independent with all actions preceding w_f in w, $w_f.(w \setminus w_f) \approx^\Pi w$ which means by definition that $w_f \in WI_s^\Pi(w)$. So one of the weak initials of w is in P, indeed.
2. $\nexists w_i \in P$ for $1 \leq i \leq n$.
   This supposition implies that all actions in w are independent with all actions in P based on D_Π. Assume the opposite: there is a w_d for some $1 \leq d \leq n$, and $\gamma \in P$ such that $(w_d, \gamma) \in D_\Pi$. Reasoning is similar to case 1: since $\nexists w_i \in P$, $w_d \in \mathit{future\_actions}(s, \alpha)$ for some $\alpha \in \mathit{enabled}(s) \setminus P$. Then, Algorithm 1 would have added α to P which did not happen.
   Since all actions in w are independent with all actions in P, there is at least one $\alpha \in P$ that is a weak initial of w. For this, take a partial concretization $c_0, ..., c_n$ of w from s ($c_0 \models s$). Take any action $\delta \in P$: if $\delta \in \mathit{enabled}(c_0)$, let $\alpha = \delta$. Otherwise, there is a $\delta' \in \mathit{enabled}(c_0)$ of process p_δ (starting from the same CFA location as δ) based on Property 4.1. Furthermore, $\delta' \in P$ since $(\delta', \delta) \in D_\Pi$ (as they belong to the same process) and $\delta' \in \mathit{future\_actions}(s, \delta')$, so Algorithm 1 adds δ' to P. As a consequence, δ' is independent with all actions in w. In this case, let $\alpha = \delta'$. Since $\alpha \in \mathit{enabled}(c_0)$, and $(\alpha, w_i) \notin D_\Pi$, we can infer that $\alpha.w \approx^\Pi w.\alpha$, and $w.\alpha$ is partially feasible from s based on Lemma 3.2. This means by definition that $\alpha \in WI_s^\Pi(w)$, indeed. So one of the weak initials of w is in P, again.

□

5 Experiments

In this section, I evaluate the efficiency of my algorithmic contributions presented in Section 4. First, I introduce the plans of the experiment in Section 5.1, along with the research questions I aim to answer in Section 5.1.1, and the technical configuration details in Section 5.1.2. Then, I present and discuss the results of the experiment in Section 5.2 with respect to the research questions. Finally, I outline the conclusions of my experiments in Section 5.3, and discuss the potential threats to their validity in Section 5.4.

5.1 Experiment Design

The goal of my experiment is to evaluate the performance of the *static abstraction-aware partial order reduction* algorithm presented in Section 4. To facilitate the experimentation, I implemented both a traditional static partial order reduction approach (later *SPOR*), as well as the *abstraction-aware* static partial order reduction technique introduced in this paper (later *AASPOR*). In pursuit of a fair comparison among the algorithms, both implementations are extensions of the Theta [26] verification framework, which had prior support for multi-threaded C programs (later *NOPOR*) [8]. In my experiments, I executed different *configurations* of Theta over a set of *input programs* written in C.

5.1.1 Research Questions

To evaluate the presented approach, I aim to answer the following research questions.

RQ1 How does the performance of *AASPOR* compare to *SPOR* over the *explicit* abstract domain?
RQ2 How does the performance of *AASPOR* compare to *SPOR* over the *predicate* abstract domain?
RQ3 How does the practical performance compare to the theoretically exponential gain over the program family introduced in Figure 1.2?

5.1.2 Experimental Configuration

I used the subset of the concurrency safety benchmark suite[5] of SV-COMP [9] that is parsable by Theta for RQ1-RQ2, and a direct implementation of Figure 1.2 for RQ3 with $N := 2^{0 \leq i \leq 8}$. The configurations were executed on virtual machines with Intel Core (Haswell) processors, 3 dedicated CPU cores were allocated to each task. Experiments regarding RQ3 were executed with 1800 seconds of timeout, while all others used 900 seconds as their time limit. I used a *sequence interpolation*-based refinement strategy with depth-first search and thread-safe large-block encoding [17]. For the predicate domain I used *atoms* as the basis of predicate splitting. For the explicit domain I used a maximum number of enumerated successor states (*maxenum*) of 1 to preserve action-determinism, which is required from the abstract state space.

5.2 Experimental Results

I executed 4 different configurations of Theta on the SV-COMP benchmark suite seen in Table 5.1. Out of the 605 tasks successfully parsed by Theta and supported by the analysis, a common subset of 334 tasks were successfully solved by the *EXPL* configurations and 357 tasks by the *PRED* configurations within the time limit. No configuration reported any wrong results. Table 5.1 shows the number of successfully solved tasks and the sum of CPU time over the commonly solved tasks per abstract domain, as well as the average explored transition count over the common subsets.

Based on the results, utilization of *AASPOR* as opposed to *SPOR* reduced both the verification time and the number of expanded transitions, and managed to solve

[5] gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/commit/2fa025c8

domain	por	explored actions	CPU time	solved tasks
EXPL	**SPOR**	15854	5916s	338
	AASPOR	12963	5516s	344
PRED	**SPOR**	11625	34982s	365
	AASPOR	10453	33850s	369

Table 5.1: Different metrics of the evaluation

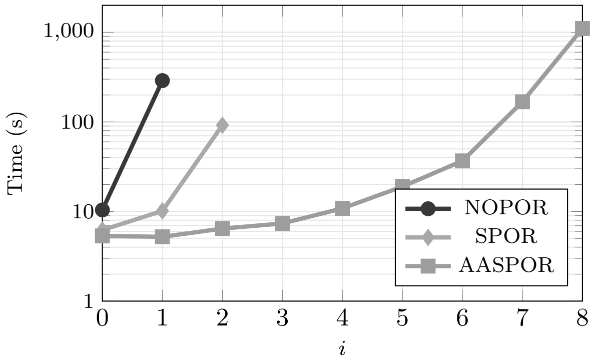

Fig. 5.1: Execution time given i for $N := 2^i$ in Figure 1.2.

more tasks. In both configuration-pairs consisting of an *AASPOR* and a respective *SPOR* configuration, *AASPOR* outperformed the traditional *SPOR* algorithm. While the number of explored transition is significantly reduced, the overall CPU time and the number of solved tasks are only marginally affected.

RQ1 In the *predicate* abstract domain, by using *AASPOR*, the sum of CPU time utilization decreased by 6.77%, and the number of expanded transitions by 18.23%. The number of solved tasks increased by 6.

RQ2 In the *explicit* abstract domain, by using *AASPOR*, the sum of CPU time utilization decreased by 3.23%, and the number of expanded transitions by 10.08%.

In addition, I executed three configurations of Theta on the program family in Figure 1.2, as seen in Figure 5.1. I used the *predicate* abstract domain with initial predicates extracted from the program (*allassumes*).

RQ3 While *NOPOR* only solved 2 tasks ($N = 1, N = 2$) within the time limit, and *SPOR* solved 1 additional task $N = 4$, the *AASPOR* configuration solved all 9 configurations, up to $N = 256$. Indeed, my abstraction-based POR algorithm scales much better on this task.

5.3 Evaluation Summary

As demonstrated by the data in Figure 5.1, there are certain scenarios where the utilization of *AASPOR* may lead to vastly improved verification performance (see RQ3).

In contrast, in the subset of the benchmark suite of SV-COMP parsable by Theta, this advantage is less pronounced, presumably due to the lack of patterns where an abstraction-based verification algorithm could show its strengths. However, the significantly fewer explored transitions, and the decreased execution time of solved tasks shows that the presented approach is confidently outperforming its baseline (RQ1-RQ2).

5.4 Threats to Validity

The following factors may influence the validity of the experiments.

Internal Validity. Consistency and accuracy of the experiments were insured by using the BenchExec framework [11]. Memory consumption statistics may deviate between executions due to the managed nature of the Java virtual machine, therefore such metrics are not used. CPU time and therefore solved tasks may be influenced by external factors such as other processes or environmental temperature fluctuations, therefore minute differences are disregarded.

External Validity. The results of the experiments are at risk of not being generalizable due to the relatively low number of input tasks. As mentioned in Section 5.3, the lack of certain patterns in the benchmark suite leads to the diminished advantage of the *AASPOR* algorithm. However, the SV-COMP benchmark suite is the *de-facto* standard for academic verification tool development, and immense work would be necessary to extend the suite with further real-life examples.

Construct Validity. In order to corroborate that the right metrics are used in the evaluation of the experiments, I considered both the user-facing and the backend-related interactions of the verification tool. The number of solved tasks, and the CPU time necessary for the solution both directly affect the user's ability to verify programs at hand, and the decrease in expanded transitions will influence the number of solver invocations, reducing the load on the entire system. Therefore, these metrics accurately represent the expected outcomes of the executions.

6 Related Work

Partial order reduction has been a field of active research in the last decades [1, 7, 14, 15, 23, 27]. Early POR methods [7, 15, 27] approximated the conflicts between actions statically. Later, a depth-first search manner dynamic partial order reduction (DPOR) algorithm was introduced [14], where dependence between actions is decided dynamically during the state space exploration looking at the exploration stack. Source DPOR is a dynamic POR algorithm [1] whose extension, Optimal DPOR [1] is proven to be optimal in the number of explored interleavings. Many optimizations exist for determining dependency where information is retrieved from the state space search context: in these works, actions are considered dependent only in certain states under certain conditions [4, 5, 29], although these conditions are typically not related to information about the applied abstraction.

Several works use POR in the context of an abstraction-based verification algorithm [10, 16, 19, 20, 24, 25, 28]. However, most of them do not take advantage of using information about the current abstraction in order to increase the reducing effect of POR, whereas this is the key concept of my proposed approach.

CPAchecker is a program verification framework that supports several analysis techniques, including Counterexample-Guided Abstraction Refinement (CEGAR) and POR [10]. However, the POR algorithm implemented in CPAchecker is relatively simple: only thread-local operations of different threads are considered independent (where an operation is global if it accesses a global memory location and thread-local otherwise). That is, the application of POR is orthogonal to CEGAR in CPAchecker.

In the works of *Su et al.* [25], *Kroening et al.* [20] and *Wachter et al.* [28], the specific abstraction-based verification algorithm Impact is combined with a POR algorithm. Although some of them use conditional dependency, their conditions are similar to the guarded independence relation described by *Wang et al.* [29], and they do not exploit information about the applied abstraction to reduce dependency. *Kroening et al.* [20] also discuss the necessary extensions for DPOR algorithms when the abstraction-based algorithm uses *covering*. Covering is applied in abstract state space exploration when an abstract state is reached that is over-approximated by another abstract state reached earlier during the exploration. The exploration stops at these states as all possible behavior is already explored from the other (more general) state. Combining this technique with the depth-first style DPOR algorithms needs extra consideration [20]. *Hansen et al.* [19] use POR for the abstraction-based verification of timed automata. Though they use information about the current abstraction, they define relations such that one order of execution simulates the other one. My case is more general: any execution order can be selected for exploration based on my approach and none of the executions have to be the over-approximation of the other.

Several works realize that even the traditional dependency relation is not a valid dependency relation [12, 24]. The combination of POR and abstraction in the work of *Cimatti et al.* [12] is specific to the Explicit Scheduler and Symbolic Threads (ESST) algorithm, while my approach is general. In my general setting, two syntactically independent actions may disable each other in the abstract state space (see Figure 3.2 in Example 3.3) whereas they implicitly assume in their proof that such situation cannot happen. *Hansen* [18] also considers the application of partial order reduction in nondeterministic abstract state spaces. However, the work focuses on the challenges posed by non-determinism, and not the generalization of the commutativity relation.

A central element of my work is the idea of abstract commutativity relations investigated by *Farzan et al.* [13]. They realize that the commutativity relation can also be relaxed in an abstraction-based setting. However, their theoretical analysis focuses on the properties of abstract commutativity relations and their combinations, and they do not embed the commutativity checking in a verification algorithm. On the other hand, my work assumes an abstract state space exploration and a partial order reduction algorithm (this assumption is still a general partial order reduction in a general abstraction-based verification algorithm). Thus, my work faces the challenges and proposes a solution to the application of abstract commutativity in verification algorithms, constituting the main contribution compared to the work of *Farzan et al.* [13].

References

1. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. pp. 373–384. ACM (2014). https://doi.org/10.1145/2535838.2535845
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Comparing Source Sets and Persistent Sets for Partial Order Reduction. Lecture Notes in Computer Science, vol. 10460, pp. 516–536. Springer (2017). https://doi.org/10.1007/978-3-319-63121-9_26

3. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. J. ACM **64**(4), 25:1–25:49 (2017). https://doi.org/10.1145/3073408
4. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-Sensitive Dynamic Partial Order Reduction. Lecture Notes in Computer Science, vol. 10426, pp. 526–543. Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_26
5. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal Dynamic Partial Order Reduction with Observers. Lecture Notes in Computer Science, vol. 10806, pp. 229–248. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_14
6. Baelde, D., Delaune, S., Hirschi, L.: Partial Order Reduction for Security Protocols. CoRR **abs/1504.04768** (2015)
7. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
8. Bajczi, L., Telbisz, C., Somorjai, M., Ádám, Z., Dobos-Kovács, M., Szekeres, D., Mondok, M., Molnár, V.: Theta: Abstraction based techniques for verifying concurrency (competition contribution). Lecture Notes in Computer Science, vol. 14572, pp. 412–417. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_30
9. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. Lecture Notes in Computer Science, vol. 13994, pp. 495–522. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
10. Beyer, D., Friedberger, K.: A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker. EPTCS, vol. 233, pp. 61–71 (2016). https://doi.org/10.4204/EPTCS.233.6
11. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
12. Cimatti, A., Narasamdya, I., Roveri, M.: Software Model Checking with Explicit Scheduler and Symbolic Threads. Log. Methods Comput. Sci. **8**(2) (2012). https://doi.org/10.2168/LMCS-8(2:18)2012
13. Farzan, A., Klumpp, D., Podelski, A.: Stratified Commutativity in Verification Algorithms for Concurrent Programs. Proc. ACM Program. Lang. **7**(POPL), 1426–1453 (2023). https://doi.org/10.1145/3571242
14. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. pp. 110–121. ACM (2005). https://doi.org/10.1145/1040305.1040315
15. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, Lecture Notes in Computer Science, vol. 1032. Springer (1996). https://doi.org/10.1007/3-540-60761-7
16. Govind, R., Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Abstractions for the local-time semantics of timed automata: a foundation for partial-order methods. pp. 24:1–24:14. ACM (2022). https://doi.org/10.1145/3531130.3533343
17. Hajdu, Á., Micskei, Z.: Efficient Strategies for CEGAR-based Model Checking. Journal of Automated Reasoning **64**(6), 1051–1091 (2020). https://doi.org/10.1007/s10817-019-09535-x
18. Hansen, H.: Abstractions for transition systems with applications to stubborn sets. Lecture Notes in Computer Science, vol. 10160, pp. 104–123. Springer (2017). https://doi.org/10.1007/978-3-319-51046-0_6
19. Hansen, H., Lin, S., Liu, Y., Nguyen, T.K., Sun, J.: Diamonds Are a Girl's Best Friend: Partial Order Reduction for Timed Automata with Abstractions. Lecture Notes in Computer Science, vol. 8559, pp. 391–406. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_26
20. Kroening, D., Sharma, S., Wachter, B.: AbPress: Flexing Partial-Order Reduction and Abstraction. CoRR **abs/1410.6044** (2014)
21. Mazurkiewicz, A.W.: Trace Theory. Lecture Notes in Computer Science, vol. 255, pp. 279–324. Springer (1986). https://doi.org/10.1007/3-540-17906-2_30
22. Overman, W.T., Crocker, S.D.: Verification of Concurrent Systems: Function and Timing. pp. 401–409. North-Holland (1982)
23. Peled, D.A.: Ten Years of Partial Order Reduction. Lecture Notes in Computer Science, vol. 1427, pp. 17–28. Springer (1998). https://doi.org/10.1007/BFb0028727
24. Sousa, M.: Abstractions and independence. Ph.D. thesis, University of Oxford, UK (2018), https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.780457
25. Su, J., Tian, C., Yang, Z., Yang, J., Yu, B., Duan, Z.: Prioritized Constraint-Aided Dynamic Partial-Order Reduction. pp. 78:1–78:13. ACM (2022). https://doi.org/10.1145/3551349.3561159

26. Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., Majzik, I.: Theta: a Framework for Abstraction Refinement-Based Model Checking. pp. 176–179 (2017). https://doi.org/10.23919/FMCAD.2017.8102257
27. Valmari, A.: Stubborn sets for reduced state space generation. Lecture Notes in Computer Science, vol. 483, pp. 491–515. Springer (1989). https://doi.org/10.1007/3-540-53863-1_36
28. Wachter, B., Kroening, D., Ouaknine, J.: Verifying multi-threaded software with impact. pp. 210–217. IEEE (2013)
29. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole Partial Order Reduction. Lecture Notes in Computer Science, vol. 4963, pp. 382–396. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_29