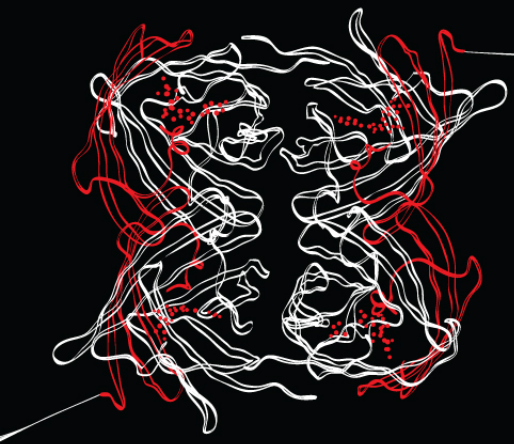
PARTIAL-ORDER REDUCTION BASED ON PROBE SETS

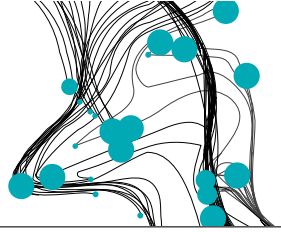
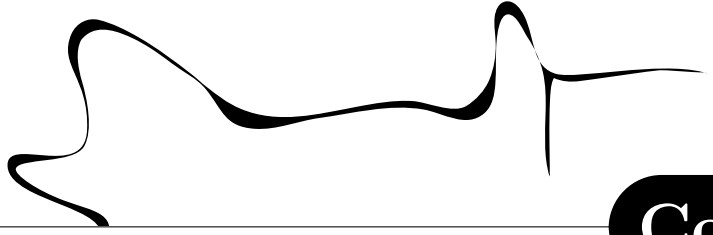
Author:
Ronald Burgman

Supervisors:
Arend Rensink
Alfons Laarman
Stefano Schivo

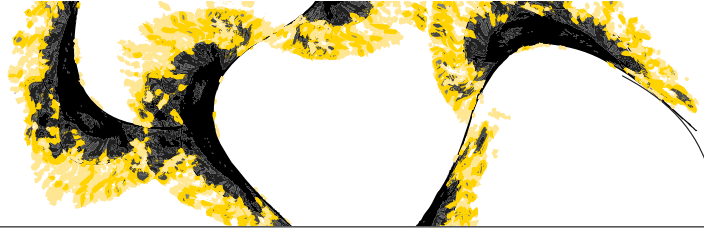
Type to enter text



UNIVERSITY OF TWENTE.


Contents



1 Introduction

Model checking is a well-known technique, which is used to verify the correctness of computer programs. In model checking, instead of verifying the computer program itself, the computer program is represented by a model called a state space, which is verified.

A major drawback of this approach is that the state space of a computer program quickly becomes very large; a phenomenon also known as the state space explosion. Because state spaces are so large, it is infeasible to verify large models.

To still be able to reason about such state spaces, without losing too much information about the original model, techniques have been developed that reduce the size of a state space that guarantee to preserve interesting information (for example deadlocks). One particular approach that proved to be successful is partial-order reduction [15, 21, 22, 26, 27].

Traditionally, Partial-order reduction uses the fact that a lot of computer programs consist of separate subprocesses that are executed in parallel. Some tasks of a single subprocess depend on other subprocesses' tasks, but some do not. These independent tasks give rise to a lot of different possible internal behaviors. However, the final result of the program is always the same, because the tasks are independent. This results in redundant information in the state space. Partial-order reduction tries to reduce state space models by removing this redundant information from the models.

To be able to apply partial-order reduction, there are two important requirements. First the parallel components need to be known in advance, and second the actions of these components need to be known in advance. In modern software systems, these two requirements are not fulfilled.

Actions within a single subprocess are always dependent on each other, because the order in which they are executed cannot be changed. When there is no clear definition of the subprocesses, every action is grouped in a single process and it will be difficult to identify independent actions. Because, in modern software, threads are dynamically created and destroyed, we do not have a clear definition of the subprocesses.

The requirement on the actions is necessary for a similar reason. When we do not know all the actions in advance, it is impossible to determine the independent actions. It is is always possible that a new action is found that is dependent with an action that was previously independent. Because modern software dynamically allocates and deallocates memory structures, new actions are dynamically created.

Our interest is in graph transformation systems. In graph transformation systems [11] the requirements of partial-order reduction are also not fulfilled. Graph transformation systems do not know processes, all actions are performed on a single shared graph, and actions are dynamically enabled and disabled because they depend on the shared graph.

To deal with these dynamic systems, new dynamic partial-order reduction algorithms have been constructed [14, 17]. In these dynamic algorithms, the dependency relation is no longer required in advance. Instead the dependency relation can be generated on the fly while exploring the state space and we no longer need to now the subprocesses and actions in advance.

1.1 Research Questions

This master thesis revolves around the dynamic partial-order reduction algorithm based on probe sets introduced in [17]. Although the theoretical concepts behind this algorithm are clear and proven to be sound, no actual implementation of the algorithm exists and nothing is known about how the algorithm performs in practice.

Therefore, the goal of our research is:

> *To discover if it is possible to develop an efficient implementation of the dynamic partial-order reduction algorithm based on probe sets.*

To reach our goal, we focus ourselves on answering the following questions:

1. Which cycle proviso should we use?
2. How do we determine the over-approximation of the missed actions?
3. How do we select probe sets?
4. What will be the input for the implementation?
5. How do we evaluate the efficiency of the algorithm?

1.2 Approach

The result of our research will be a tool: DRoP (Dynamic Reduction of Petri nets). We will use this tool to find an answer to our research questions.

1.2.1 Cycle Proviso

Like static partial-order reduction algorithms, the probe sets algorithm needs a cycle proviso [17]. To determine what a good cycle proviso is, we will develop, implement and compare multiple cycle provisos.

1.2.2 Over-approximation of the Missed Actions

The probe set algorithm determines the dependency relation by analysing the set of missed actions [17]. Static partial-order reduction needs to over-approximate the dependency relation, because determining the exact relation is too expensive. For similar reasons, the probe set algorithm does not determine the exact missed actions, but it makes an over-approximation.

In our research we will try to improve the efficiency of this over-approximation.

1.2.3 Probe Set Selection

In the dynamic partial-order reduction algorithm based on probe sets, probe sets determine which actions will be explored [17]. To be able to achieve the best possible reduction, we should not select random valid probe sets, but we should determine a strategy that suggests what 'good' probe sets are. In our research we will need to develop and implement different strategies, which we can compare.

1.2.4 Algorithm Input

Our interest is in applying the probe set algorithm to graph transformation systems, but the algorithm is designed to work on entity-based systems. Graph transformation systems and entity-based systems resemble each other, but it is not immediately clear how graph transformation systems can be represented as an entity-based system.

To be able to focus our effort on other aspects of the implementation, DRoP is designed to work with Petri nets, instead of graph transformation systems. Petri nets are still not entity-based systems, so we still need to make a transformation from Petri nets to entity-based systems. However, because Petri nets are simpler than graph transformation systems, the transformation will be easier to develop.

We choose Petri nets because they can be seen as a simpler version of graph transformation systems; it is easy to translate an arbitrary Petri net to a graph transformation system where the graphs consists only of nodes and contain no edges. And even though Petri nets can be considered as a simpler formalism than graph transformation systems, it is still a powerful modeling concept with plenty of examples in literature. So using Petri nets does not limit us when evaluating the effectiveness of the algorithm.

1.2.5 Evaluation

Dynamic partial-order reduction is designed to improve the weaknesses of static partial-order reduction. Therefore, we do not only want to create an implementation of the algorithm, but we also want to verify that it performs better than existing implementations of static partial-order reduction algorithms.

To verify that our implementation is actually able to perform better than existing algorithms, we will compare DRoP with existing state space exploration tools that are able to apply partial-order reduction.

The performance of the different tools will be compared on different properties: the size of the reduction (number of states and transitions in the state space) and the time and memory needed to explore the reduced state space.

We will use two types of test models during our experiments: models with dynamic (de)allocation of threads and objects, i.e. the type of models where dynamic partial-order reduction was designed for; and models for which static partial-order reduction is already able to perform well.

In the first case we expect dynamic partial-order reduction to perform better than static partial-order reduction. This means that it will generate a smaller state spaces and it will need less time and memory to generate these state spaces.

In the second case, we expect dynamic partial-order reduction to be able to achieve the same reduction as static partial-order reduction. However, since dynamic partial-order reduction is more complicated than static partial order-reduction, we expect it will need more time and memory to generate the state spaces.

1.3 Overview

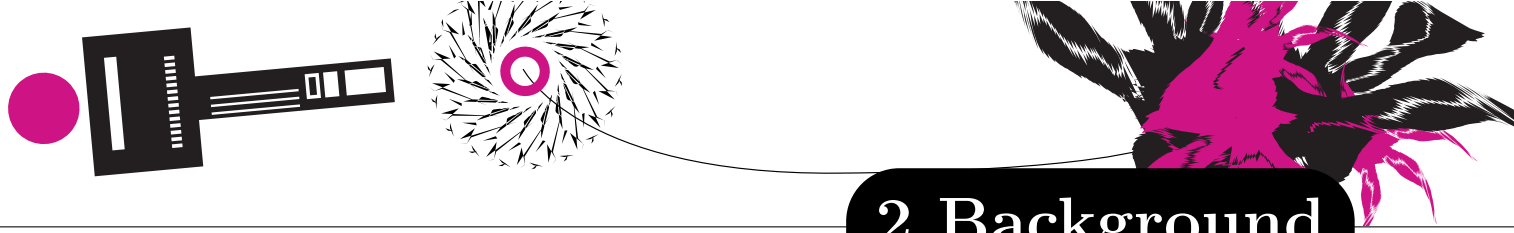
First, this report will provide some background information in Chapter 2. We discuss static partial-order reduction in Section 2.2, dynamic partial-order reduction based on probe sets in Section 2.3 and finally Petri nets in Section 2.4.

In Chapter 3 we will discuss the cycle provisos and over-approximation that we will develop and that are introduced in [17]. Next we will discuss how we select probe sets and the transformation from Petri net to entity-based system in Chapter 4.

In the Sections 5.1, 5.2 and 5.3 we discuss the tools and the test models we use for our experiments and the test setup.

Then we will discuss the results of the comparison of the different probe set selection strategies and cycle provisos in Section 5.4 and we will compare DRoP with the other tools in Section 5.5.

Finally, we will present our conclusions in the last chapter.



2 Background

This chapter provides the background information that is needed to understand the research in this thesis.

First some basic definitions are provided. Then the static partial-order reduction algorithm and the dynamic partial-order reduction algorithm are discussed. Finally we introduce Petri nets, the models on which the partial-order reduction algorithms are applied.

2.1 Definitions

In this chapter we discuss how to apply partial-order reduction on a labeled transition system (LTS). A LTS is defined as follows.

Definition 1 (Labeled transition system)**.** *A* transition system *T is a tuple $(S, A, \rightarrow, s_0, P, L)$ where*

- *S is a finite set of states.*
- *$A \subseteq Act$ is a finite set of actions available for this transition system.*
- *$\rightarrow \subseteq S \times A \times S$ is the transition relation, which labels every transition with an action.*
- *$s_0 \in S$ is the initial state.*
- *P is a set of propositions.*
- *$L : S \rightarrow 2^P$ is the labeling function that assigns a set of propositions to each state.*

For $(s, a, s') \in \rightarrow$ we also write $s \xrightarrow{a} s'$.

For each state in the transition system we can refer to the set of enabled actions.

Definition 2 (Enabled actions)**.** *The set of enabled actions in state s of transition system T is defined as:*

$$\forall s \in S : Enabled(s) = \{a \in Act \mid \exists s' \in S : (s, a, s') \in \rightarrow\}$$

In this report, we are only concerned with a subset of all transition systems: the deterministic transition systems.

Definition 3 (Deterministic transition system)**.** *A transition system is deterministic when for each state $s \in S$ and action $a \in A$, the following holds:*

$$\forall (s, a, s'), (s, a, s'') \in \rightarrow : s' = s''$$

2.2 Partial-Order Reduction

2.2.1 Independence

The core concept in partial-order reduction is the independence of actions in a state space. Independent actions are defined by a dependence relation.

Definition 4 (Dependency relation)**.** *A dependency relation is a relation $D \subseteq Act \times Act$ for a deterministic transition system T if it is reflexive and symmetric and for all $a, b \in Act, (a, b) \notin D$ the following two properties hold for all states $s \in S$:*

$$(a \text{ enabled in } s \text{ and } s \xrightarrow{a} s') \implies (b \text{ enabled in } s \Leftrightarrow b \text{ enabled in } s') \tag{2.1}$$

$$(a, b \text{ enabled in } s) \implies (\exists s_0, s_1, s' \in S : s \xrightarrow{a} s_0 \xrightarrow{b} s' \wedge s \xrightarrow{b} s_1 \xrightarrow{a} s') \tag{2.2}$$

Equation 2.1 states that when a and b are independent, and b is enabled after performing a, then b should already have been enabled before a was executed, ie. independent actions do not enable each other.

Equation 2.2 states that independent actions are commutative. This means that if a and b are both enabled and independent, it is possible to execute both $a \cdot b$ and $b \cdot a$, ie. independent actions do not disable each other. Moreover, both executions should end up in the same state.

When actions are independent, it is possible to execute them in any order. Also the final state of the system will be the same, whatever ordering of independent actions is chosen.

Because the final results of any ordering of independent actions are equal, we might choose not to explore all possible orderings of independent actions. Instead only one ordering of independent actions may be explored that represents all other orderings. This is what partial-order reduction does.

2.2.2 Defining the Dependency Relation

Defining a dependency relation is not easy, because (in)dependence is a global property. Take for example the following assignments: $x := z + y$ and $x := z$. These assignments seem to be dependent, because they both assign the variable x a different value. However, if y would be 0 throughout the entire system, the actions are actually equivalent and no longer dependent.

Because independence is a global property, it is not enough to look only at the actions themselves when determining the dependency relation: the context in which the actions are applied also needs to be taken into account. However, this is not feasible, because this entails exploring the entire state space to determine the dependency relation and this negates the gains partial-order reduction would provide.

To counter this problem, partial-order reduction uses an over-approximation of the dependency relation. If two actions are not in the over-approximation, they are certainly independent. However, some independent actions might not be detected, because they are in the over-approximated dependency relation. This guarantees that the partial-order reduction remains valid, but the reduction may be smaller than what is theoretically possible. To over-approximate the dependency relation it is possible to use only local criteria. Therefore the state space no longer needs to be completely explored to determine the dependency relation (but only the actions themselves).

The most basic approximation of the dependency relation is made by assuming that actions are dependent when either they are part of the same process or they use the same (global) variables. A more accurate dependency relation can be gained by tweaking the second condition. For example, the second condition could be changed so that actions that only read but not write the same variables are independent. This requires extra resources to calculate the dependency relation, but since the over-approximation will be smaller, the reduced state space will be smaller.

2.2.3 The Ample Set Approach

Three variants of partial-order reduction were more or less simultaneously introduced at the beginning of the nineties: stubborn sets by Valmari [27] [26], persistent sets by Godefroid [15] and ample sets by Peled [21] [22].

The three methods are very similar. They select a subset of the enabled actions in every state of a deterministic transition system and include only those selected actions in the reduction. By varying the constraints on the sets of selected actions, different properties are preserved by the reduction.

Because the three methods are very similar, we describe only one in this section, namely the ample set approach of Peled, which is also nicely described in [18].

As said before, partial-order reduction selects a subset of the enabled actions for each state. In the approach of Peled, this subset of actions is called the *ample set*. When exploring the state space, instead of the enabled actions, only the actions in the ample set are explored. By posing more constraints on the ample set, more properties of the original system are preserved in the reduced system.

In the following it is assumed that a dependency relation is given. This relation is not calculated by the ample set algorithm.

Definition 5 (Ample set)**.** *ample*(s) *is the ample set for a given state* s*, for wich the following conditions must hold:*

$$ample(s) \subseteq Enabled(s) \tag{2.3a}$$

$$ample(s) = \emptyset \implies Enabled(s) = \emptyset \tag{2.3b}$$

According to Definition 5, the ample set for a given state must be a subset of the enabled actions in that state, so no new behavior is introduced. The ample set is empty if and only if the set of enabled actions is empty, so no new deadlocks are introduced.

With these ample sets, a reduced state space can be constructed.

Definition 6 (Reduced state space)**.** *A reduced state space is a deterministic transition system* T_r*, constructed from a given original deterministic transition system* T_o *and a collection of ample sets for* T_o*. The set of states* (S)*, the set of actions* (A)*, the initial state* (s_0)*, the set of propositions* (P) *and the labeling function* (L) *are unchanged in the reduction. The reduction does have a new transition function, which is defined as follows:*

$$\rightarrow_r = \{(s, a, s') \in \rightarrow_o \mid a \in ample(s)\}$$

Equation 2.3b guarantees that an ample set does not introduce a new deadlock. However, to *preserve* all deadlocks in the reduction, an extra constraint is necessary.

Constraint 1 (Independence of future actions)**.**

$$\begin{array}{c} \textit{For all finite traces in the LTS } s_0 \xrightarrow{b_1} s_1 \xrightarrow{b_2} \cdots \xrightarrow{b_n} s_n \xrightarrow{a} t : \\ \textit{If } a \textit{ depends on an action in } ample(s_0) \textit{ then } \exists i \in 1..n : b_i \in ample(s_0) \end{array} \tag{2.4}$$

Equation 2.4 tells us that if an action a is dependent with the ample set in state s_0, it cannot be executed before an action in the ample set is executed. Another way of formulating this is that every finite execution starting in s_0 that does not contain any action from the ample set, cannot contain any action that is dependent with an action in the ample set.

The intuition of Equation 2.4 is that all dependent actions are in the ample set. It is not necessary to explore independent actions, because according to Equation 2.2 they will still be available in the next state. Dependent actions will not be available in the next state, so they will have to be considered now.

Together Definition 5 and Equation 2.4 guarantee that any deadlock that is reachable in the original transition system, is still reachable in the reduced transition system. Formally preservation of deadlocks is defined as follows:

Definition 7 (Reachability)**.** *A state* $s \in S$ *is reachable in a given transition system* T *when:*

$$\exists s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_n} s_{n+1} : s_i \in S \wedge a_i \in A \wedge (s_i, a_i, s_{i+1}) \in \rightarrow \wedge\, s_{n+1} = s$$

Theorem 1 (Preservation of deadlocks)**.** *Given a deterministic transition system* T *and its reduction* T' *according to Definition 5 and Equation 2.4, the following holds:*

$$\forall s \in S : s \textit{ reachable and deadlock in } T_o \implies s \textit{ reachable and deadlock in } T_r$$

The following example demonstrates the necessity of Equation 2.4. Figure 2.1 contains the state space of the dining philosophers problem with two philosophers. Equation 2.5 provides some ample sets for this state space, which do not take Equation 2.4 into account. Therefore deadlock state s_4 is not reachable in the reduction.

$$\begin{array}{l} ample(s_0) = \{getLeftFork_1\}, ample(s_1) = \{getRightFork_1\}, \\ ample(s_3) = \{putLeftFork_1\}, ample(s_6) = \{putRightFork_1\} \end{array} \tag{2.5}$$

These ample sets violate Equation 2.4, because $getLeftFork_2 \cdot getRightFork_2$ is a trace starting in s_0 where $getRightFork_2$ is dependent with an action in $ample(s_0)$ (namely $getLeftFork_1$), but no action from $ample(s_0)$ is performed before $getRightFork_2$.

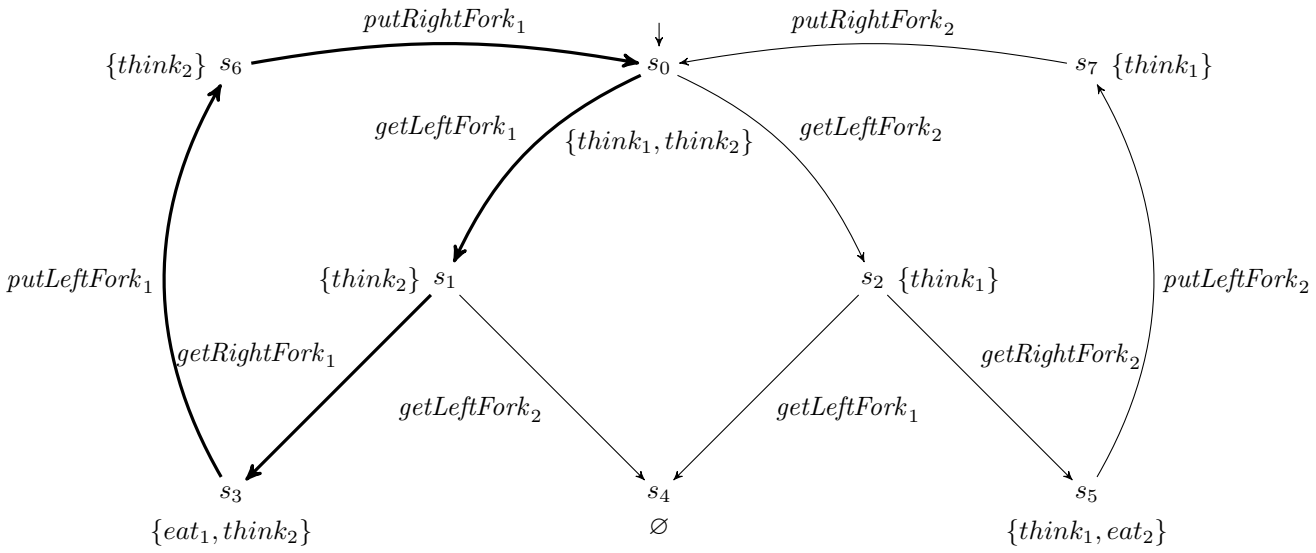
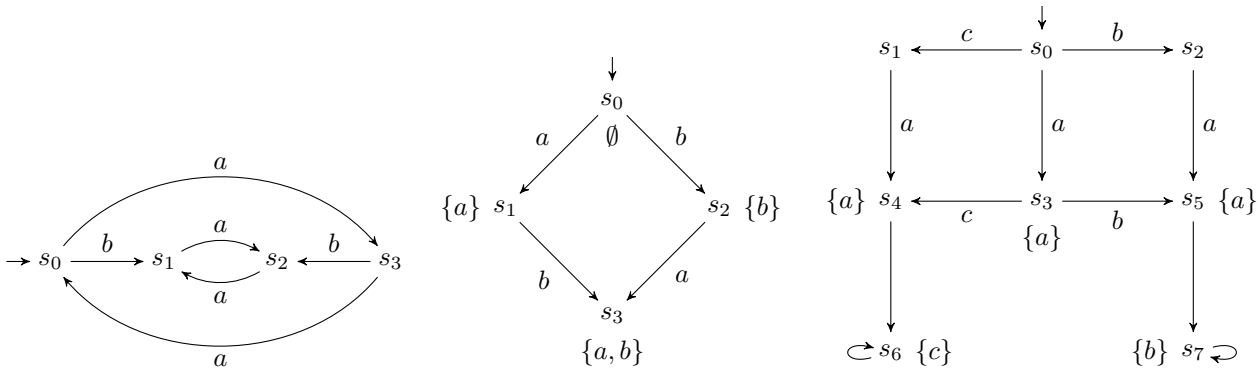


Figure 2.1: Dining philosophers, reduced (thick lines) according to Equation 2.5



(a) Cycle example (b) LTL example (c) CTL example

Figure 2.2: Motivating examples for the additional partial-order reduction constraints.

To preserve more properties than deadlocks, more constraints are necessary:

Theorem 2 (Preservation of safety properties [18])**.** *To preserve safety properties, the following equation must hold in addition to Definition 5 and Equation 2.4.*

$$\begin{aligned} &\forall\ \mathit{Cycle}\ s_0 \xrightarrow{b_0} s_1 \xrightarrow{b_1} \cdots \xrightarrow{b_{n-1}} s_n \xrightarrow{b_n} s_0 \in \mathit{LTS} : \\ &\forall a \in \mathit{Enabled}(s_0) : \exists j \in \{0..n\} : a \in \mathit{ample}(s_j) \end{aligned} \tag{2.6}$$

Equation 2.6 is called the *cycle breaking condition* or *cycle proviso*. By adding this constraint, all finite traces of a state space are preserved in the reduced state space. The properties that can be verified with the finite traces are exactly the safety properties.

Without Equation 2.6, it is possible that the actions in the ample sets describe a loop. States that are outside of this loop will then be cut off and are unreachable in the reduced state-space. By adding this constraint, the ample sets need to provide the option to leave the loop, so that states outside the loop can be examined.

For example, take Figure 2.2a. Here a and b are independent actions. Therefore, without Equation 2.6, valid ample sets could be: $\mathit{ample}(s_0) = \{a\}$ and $\mathit{ample}(s_3) = \{a\}$ and the inner cycle of s_1 and s_2 is not explored.

It is possible to extend the ample set approach to preserve even more properties such as $\mathrm{LTL}_{\setminus\bigcirc}$ and $\mathrm{CTL}_{\setminus\bigcirc}$ properties. To preserve more properties than deadlocks, more constraints are necessary:

Definition 8 (Stutter actions)**.** *Action a is a stutter action when:*

$$\forall (s, a, s') \in \rightarrow : L(s) = L(s')$$

So for a stutter action, the label of the source state is the same as the label in the target state.

Theorem 3 (Preservation of $\mathrm{LTL}_{\setminus\bigcirc}$ [18])**.** *To preserve $\mathit{LTL}_{\setminus\bigcirc}$, the following must hold in addition to Definition 5, Equation 2.4 and the cycle proviso in Equation 2.6.*

$$\mathit{ample}(s) \neq \mathit{Enabled}(s) \implies \forall a \in \mathit{ample}(s) : a \textit{ is stuttering} \tag{2.7}$$

By adding Equation 2.7 either all actions of a state are in the ample set of that state, or all actions in the ample set are stutter actions. Because the actions in the ample set are stutter actions, the actions in the ample set can be given priority without altering the stutter-trace and it can be proven that the reduced state-space preserves $\mathrm{LTL}_{\setminus\bigcirc}$ [18, Theorem 8.13].

For example, take Figure 2.2b. Without Equation 2.7, a valid ample set for s_0 would be $\mathit{ample}(s_0) = \{a\}$ or $\mathit{ample}(s_0) = \{b\}$. With these ample sets, the reduced system will either include the trace with labels $\emptyset\{a\}\{a,b\}$ or the trace with labels $\emptyset\{b\}\{a,b\}$, while the original system contains both. So the LTL formulas $(\neg b)U\{a,b\}$ and $(\neg a)U\{a,b\}$ do not hold for the original system, but do hold for one of the reductions.

By including Equation 2.7, all actions in s_0 need to be explored, and the reduction satisfies the same $\text{LTL}_{\setminus\bigcirc}$ formulas as the original system.

Finally, to preserve $\text{CTL}_{\setminus\bigcirc}$ one extra constraint is necessary.

Theorem 4 (Preservation of $\text{CTL}_{\setminus\bigcirc}$ [18])**.** *To preserve $CTL_{\setminus\bigcirc}$, the following equation should hold in addition to Definition 5 and Equations 2.4, 2.6 and 2.7.*

$$ample(s) \neq Act(s) \implies |ample(s)| = 1 \tag{2.8}$$

Take, for example, Figure 2.2c. Because c and b are both independent of a, a valid ample set for state s_0 would be $ample(s_0) = \{c, b\}$. In this case, partial-order reduction removes state s_3 from the state space. However, now the CTL formula $AG(a \implies (AFb \vee AFc))$ does hold in the reduction, but it does not hold in the complete state space.

By adding Equation 2.8 as a constraint on the ample set, states like s_3, which basically model postponing a decision, are preserved in the reduction.

2.3 Dynamic Partial-Order Reduction

For static partial-order reduction as described in the previous section to work, two assumptions are made:

1. All actions are known in advance and there are finitely many.
2. A system consists of a fixed set of parallel processes.

Assumption 1 is necessary to define the dependency relation as described in Section 2.2.1. The dependency relation is defined over all actions, and it needs to be known in advance. Assumption 2 is necessary to determine the over-approximation as described in section 2.2.2. Not only the most simple over-approximation, but most over-approximations depend on a clear definition of sub-processes.

Although classically these assumptions would hold, they do not always hold for modern systems. In systems that support dynamic creation and deletion of objects, assumption 1 fails. A new object enables new actions. Because the objects are created and deleted dynamically, the actions are also created and deleted dynamically and it is impossible to determine the dependency relation in advance.

For systems that also allow dynamic creation and deletion of threads, assumption 2 also fails. Therefore, it is also hard to determine the dependency relation with the known over-approximations.

Take, for example, Petri nets as discussed in Section 2.4. The available actions in a Petri net depend on the tokens that are present. Since it is, in general, unknown in advance which tokens will be present during the execution, the actions of the Petri net are unknown before execution.

In [17] a dynamic partial-order reduction algorithm based on probe sets is introduced, that is able to deal with dynamic systems in which the above assumptions do not hold.

The two core concepts of the algorithm are *missed actions* and *probe sets*, which are combined with a new definition of dependency. This section introduces these concepts together with the algorithm itself.

2.3.1 Dependency Relation and Definitions

In the probe set algorithm, dependency is defined using two sub-relations: a stimulation relation and a disabling relation. In this section, these relations are defined in a abstract way. In the next section, they are instantiated for a formalism called entity-based systems.

Definition 9 (Stimulation)**.** *$a \triangleright b$ is an irreflexive relation that indicates that action a stimulates action b. The intuition of this relation is that action a fulfills part of the precondition of action b. Therefore b cannot occur directly before a.*

Definition 10 (Disabling)**.** *$a \blacktriangleleft b$ is a reflexive relation that indicates that action a disables action b. The intuition of this relation is that action a violates a part of the precondition of action b. Therefore b cannot occur directly after a.*

Definition 11 (Influence)**.** *a influences b, $a \rightsquigarrow b$, is defined as $a \triangleright b$ or $b \blacktriangleleft a$.*

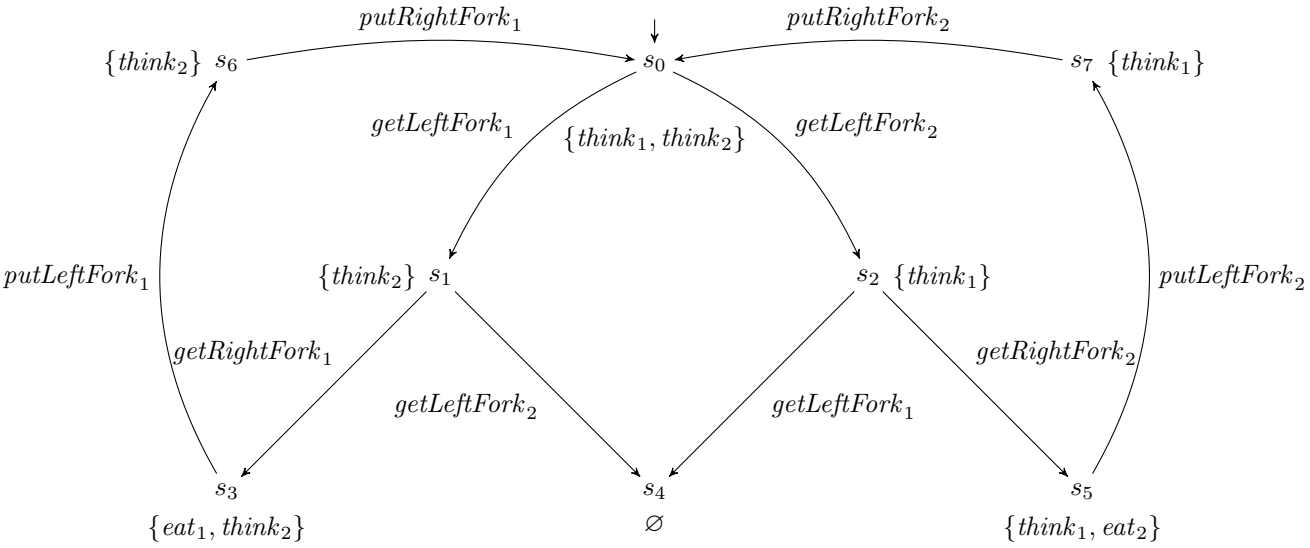


Figure 2.3: Dining philosophers state space

The influence relation is the probe set variant of the dependency relation. The main difference with Definition 4 is that the influence relation is not symmetric.

The intuition of the influence relations is as follows. If action b is independent of action a, it should be possible to execute action b before action a, so a should not stimulate b. Also, when b is independent of a, it should still be possible to execute a after b occurred, so b should not disable a. Therefore if $a \rightsquigarrow b$, a and b are dependent.

For example, take the actions of the example in Figure 2.3. This example shows a state space of the dining philosophers problem with two philosophers.

A philosopher can only pick up its right fork after it has picked up his left fork. Therefore $\mathit{getLeftFork}_1 \triangleright \mathit{getRightFork}_1$ and $\mathit{getLeftFork}_2 \triangleright \mathit{getRightFork}_2$. It is impossible for both philosophers to pick up the same fork, so $\mathit{getLeftFork}_1 \blacktriangleleft \mathit{getRightFork}_2$ and $\mathit{getRightFork}_2 \blacktriangleleft \mathit{getLeftFork}_1$.

With the influence relation it is possible to define an equivalence relation modulo permutation of the independent actions over traces. As with static partial-order reduction, if two traces are equivalent, only one has to be preserved by the reduction algorithm.

Similarly to the equivalence relation, a prefix relation can be defined.

Notation. *In the following definitions w, u, and v are traces, i.e. sequences of actions.*

Definition 12 (Equivalence modulo permutation of independent actions)**.** *$\simeq \subseteq \mathit{Act}^* \times \mathit{Act}^*$ is the smallest transitive relation such that:*

$$a \not\rightsquigarrow b \implies v \cdot a \cdot b \cdot w \simeq v \cdot b \cdot a \cdot w$$

Definition 13 (Weak prefix)**.** *The weak prefix, $\precsim$, is defined as follows:*

$$v \precsim w \Leftrightarrow \exists u : v \cdot u \simeq w$$

In the case of the dining philosophers example in Figure 2.3, $\mathit{getLeftFork}_1 \cdot \mathit{getLeftFork}_2 \simeq \mathit{getLeftFork}_2 \cdot \mathit{getLeftFork}_1$, because $\mathit{getLeftFork}_1 \not\rightsquigarrow \mathit{getLeftFork}_2$.

Similarly, $\mathit{getLeftFork}_1 \precsim \mathit{getLeftFork}_2 \cdot \mathit{getLeftFork}_1$, because $\mathit{getLeftFork}_2 \not\rightsquigarrow \mathit{getLeftFork}_1$.

With the above relations on traces, the following operations are defined.

Definition 14 (Enabled trace)**.** *Trace w is enabled in state s, $s \vdash w$ when the actions in w can be executed starting in s:*

$$s \vdash w \Leftrightarrow \exists s' : s \xrightarrow{w} s'$$

Definition 15 (s after w)**.**

$$s \uparrow w := s', \text{ such that } s \xrightarrow{w} s'$$

When $s \vdash w$ holds, trace w can be executed in state s. For example, in Figure 2.3 $s_0 \vdash \mathit{getLeftFork}_1$ holds but $s_1 \vdash \mathit{getLeftFork}_1$ does not. When $s \vdash w$ holds, $s \uparrow w$ is the state that is reached after executing w. So $s_0 \uparrow \mathit{getLeftFork}_1 = s_1$. When $s \vdash w$ does not hold, $s \uparrow w$ is not defined.

Definition 16 (Difference)**.**

$$w - v := u, \text{ such that } v \cdot u \simeq w$$

With the difference operation, it is possible to remove a weak prefix v from a trace w.

Definition 17 (Prime cause)**.** *The prime cause of action a in trace w, $\downarrow_a w$ is defined as follows:*

$$\downarrow_a w := v, \text{ such that } (w - v) \nrightarrow a \wedge \forall v' \precsim w : ((w - v') \nrightarrow a \implies v \precsim v')$$

Intuitively the prime cause of action a in trace w is a weak prefix of w, such that the prefix contains all actions of w that influence a (in)directly. Moreover, the prime cause is the shortest possible prefix for which this holds.

For example, take Figure 2.3. The prime cause of $\mathit{getRightFork}_1$ in $\mathit{getLeftFork}_1$ is $\mathit{getLeftFork}_1$.

The prime cause of $\mathit{putLeftFork}_1$ in $\mathit{getLeftFork}_1 \cdot \mathit{getRightFork}_1$ is $\mathit{getLeftFork}_1 \cdot \mathit{getRightFork}_1$. Here, $\mathit{getLeftFork}_1$ is included in the prime cause not because it directly influences $\mathit{getLeftFork}_1$, but because it influences another action in the prime cause, namely $\mathit{getRightFork}_1$. The $\mathit{getLeftFork}_1$ action is necessary in the prime cause, because the prime cause should be a weak prefix and $\mathit{getRightFork}_1 \not\precsim \mathit{getLeftFork}_1 \cdot \mathit{getRightFork}_1$.

2.3.2 Entity-Based Systems

The definitions in the previous section are abstract. In this section, some of these definitions are instantiated in the context of entity-based systems to clarify them. Entity-based systems are used because they are simple. However, other systems, such as graph transformation systems and Petri nets, can be expressed as entity-based systems.

Notation. *In the following definitions, Ent refers to the global set of all available entities. We keep using the standard notation for a LTS, so the set of states and actions are denoted as S and Act etc.*

Definition 18 (Entity-based actions)**.** *An action a is said to be entity-based when there exists finite disjoint sets:*

$R_a \subseteq \mathit{Ent}$, *the set of entities read by a.*
$N_a \subseteq \mathit{Ent}$, *the set of entities forbidden by a.*
$D_a \subseteq \mathit{Ent}$, *the set of entities deleted by a.*
$C_a \subseteq \mathit{Ent}$, *the set of entities created by a.*

With these actions and entities an entity-based transition system can be defined.

Definition 19 (Entity-based transition system)**.** *A deterministic transition system is entity-based when all actions in the transition system are entity-based actions. Moreover, for all $s \in S$ the following should hold:*

- *There is a finite set $E_s \subseteq \mathit{Ent}$, that is unique for each state. So $E_s = E_{s'} \implies s = s'$.*
- $\forall a \in \mathit{Act} : s \vdash a \overset{def}{\iff} (R_a \cup D_a) \subseteq E_s \wedge (N_a \cup C_a) \cap E_s = \emptyset$.
- $\forall a \in \mathit{Enabled}(s) : s \uparrow a := E_{s \uparrow a} := (E_s \backslash D_a) \cup C_a$.

Now entity-based systems are defined, it is possible to give a concrete definition of the stimulation and disabling relations.

Definition 20 (Stimulation in entity-based systems)**.**

$$a \rhd b \Leftrightarrow C_a \cap (R_b \cup D_b) \neq \emptyset \vee D_a \cap (C_b \cup N_b) \neq \emptyset$$

Definition 21 (Disabling in entity-based systems)**.**

$$a \blacktriangleleft b \Leftrightarrow D_a \cap (R_b \cup D_b) \neq \emptyset \vee C_a \cap (C_b \cup N_b) \neq \emptyset$$

2.3.3 Probe sets

With the entity-based systems and the new stimulating and disabling relation defined, we now introduce the actual partial-order reduction algorithm.

The probe set algorithm consists of two major parts. The first part is a state space exploration, which is done by using probe sets to select the new actions that are to be explored. This exploration can however miss actions that do need to be explored. The exploration of these missed actions is handled by the second part of the algorithm.

The probe set algorithm does not work directly with states, but it uses vectors.

Definition 22 (Vector)**.** *A vector (s, w) in a transition system is a tuple, where $s \in S$ is a state and $w \in Act^*$ is a trace. Moreover, $s \vdash w$ must hold.*

A vector (s, w) actually represents the state $s \uparrow w$. The vector notation is used because it is necessary for the missed action phase to know how a state was reached. The vectors that need to be explored are placed upon a stack, which is processed by the algorithm.

The task of a probe set is, similar to the ample set, to select new vectors that should be placed on the stack and explored. The general concept of a probe set, and how new vectors are selected is defined below. However, for the probe set to generate correct explorations, more constraints are necessary. These constraints are defined later in Definition 24.

Definition 23 (Probe set)**.** *A probe set $p_{s,w}$ for vector (s, w) is a partial function:*

$$p_{s,w} : Enabled(s \uparrow w) \rightharpoonup Act^*$$

The domain of a probe set is a subset of the enabled actions in the vector for which the probe set is generated. Analogous to the ample set approach, this subset represents the actions that should be explored. The image of the probe set is a trace, and represents the history that is no longer relevant when the associated action is explored. So it represents the history that may be forgotten.

Concretely, the probe set $p_{s,w}$ is used to generate new vectors from vector (s, w) as follows:

$$newvectors := \{(s \uparrow p_{s,w}(a), w \cdot a - p_{s,w}(a)) \mid a \in dom(p_{s,w})\} \tag{2.9}$$

The algorithm will not work if arbitrary actions are explored or arbitrary parts of the history are forgotten. Therefore the following requirements are imposed on a probe set to guarantee a correct exploration.

Definition 24 (Valid probe set)**.** *A probe set $p_{s,w}$ is a valid probe set for vector (s, w) when the following conditions hold for all $a \in dom(p_{s,w})$, $b \in Enabled(s \uparrow w)$:*

$$a \blacktriangleleft b \vee b \blacktriangleleft a \implies b \in dom(p_{s,w}) \tag{2.10a}$$

$$p_{s,w}(a) \not\precsim\!\downarrow_b w \implies b \in dom(p_{s,w}) \tag{2.10b}$$

$$p_{s,w}(a) \precsim \downarrow_a w \tag{2.10c}$$

$$Enabled(s \uparrow w) \neq \emptyset \implies dom(p_{s,w}) \neq \emptyset \tag{2.10d}$$

The idea behind Constraint 2.10a is simple. When an enabled action disables another enabled action, one of the actions will not be available in the next state and therefore the exploration of that action cannot be postponed.

Constraint 2.10b and Constraint 2.10c are a bit more complicated, because the constraints are not needed for the probe set phase, but for the missed actions phase. The missed actions phase is discussed in Section 2.3.4.

Because actions are dynamically created, the dependency relation is generated on the fly. Therefore the vector contains two types of actions: dependent actions, and actions of which the dependencies are unknown. The actions with unknown dependencies are analyzed during the missed actions phase, so they need to be remembered. Dependent actions, which are the actions in the prime cause, can be removed (Constraint 2.10c).

When an action is not explored, the missed actions phase is also postponed. Therefore it is not allowed to remove actions from the vector that have unknown dependencies with actions that are explored right now; these actions are needed when the missed actions analysis is performed later on. This is expressed in Constraint 2.10b.

	glf_1	grf_1	glf_2	grf_2	plf_1	prf_1	plf_2	prf_2
glf_1	◀	▷		◀				
grf_1		◀	◀		▷			
glf_2		◀	◀	▷				
grf_2	◀			◀			▷	
plf_1					◀	▷		
prf_1	▷					◀		
plf_2							◀	▷
prf_2			▷					◀

Table 2.1: The stimulation and disabling relations for the dining philosophers problem.

Example Exploration

To give a better understanding of the probe set algorithm, an example exploration is shown on the dining philosophers example in Figure 2.3. Usually, the algorithm will need to calculate the stimulation and disabling relations on-the-fly. But because we already have the complete state-space, it is possible to provide this relation in advance. The stimulation and disabling relation for Figure 2.3 are given in Table 2.1. In this table, $getLeftFork_1$ is abbreviated to glf_1 etc.

The necessary steps for an exploration with probe sets are explained in the previous section. Algorithm 1 combines these steps into a pseudo code. The algorithm keeps a stack of vectors left to be explored $V \backslash V_p$ and pushes the initial state with the empty trace on the stack as the initial vector. New vectors are produced according to Equation 2.9.

Algorithm 1 Probe set based state space exploration

1: **let** $V_p \leftarrow \emptyset$; *//The set of already explored vectors*
2: **let** $V \leftarrow \{(s_0, \epsilon)\}$; *//The stack with vectors to be explored*
3: **while** $V \backslash V_p \neq \emptyset$ **do**
4: **choose** $(s, w) \in V \backslash V_p$;
5: **let** $V_p \leftarrow V_p \cup \{(s, w)\}$;
6: **choose** probe set $p_{s,w}$; *//According to Equation 2.10*
7: **let** $V \leftarrow V \cup \{(s \uparrow p_{s,w}(a), w \cdot a - p_{s,w}(a)) | a \in dom(p_{s,w})\}$; *//According to Equation 2.9*
8: **end while**

The execution of the exploration algorithm can be found in Table 2.2. For every iteration of the loop, the following aspects are given: the set of unprocessed vectors ($V \backslash V_p$), the currently considered vector (s, w), the probe set generated in this iteration ($p_{s,w}$) and the new vectors that are generated in this iteration (ΔV). Figure 2.4 shows the explored state space. The thick edges are considered by the algorithm, the thin edges are not.

Several interesting things can be observed from this execution. The most obvious and important observation is that the right half of the state space is not considered, even though the probe sets are valid according to Equation 2.10. In the reduction, every action should be explored at least once. In this example, the action $getRightFork_2$ is never explored. This is bad, because possible bad behavior after this action *cannot* be detected. This is the exact situation that is dealt with by the second half of the probe set algorithm: the missed actions phase, which is explained in the next section.

Besides the problem that certain parts of the state space are not explored, two more interesting properties can be noted about the exploration itself. First, some states can be visited multiple times and second the set of reached states depends on the chosen probe sets.

During the exploration, some states are explored multiple times. For example, the initial state q_0 is visited during the first iteration, but also during the sixth iteration. It is important to do so. The difference between the visits is that the state is reached with a different vector. Because the vector is different, the missed actions phase of the algorithm can produce different results.

Instead of the current probe set in iteration six, the algorithm could have chosen $p_{s,w}(glf_2) = \epsilon$ as a valid alternative probe set. With the alternative probe set, action $getRightFork_2$ would have been explored, so the choice of the probe set influences the reached states.

iteration	$V \backslash V_p$	(s, w)	$p_{s,w}$	ΔV
1	$\{(s_0, \epsilon)\}$	$\{(s_0, \epsilon)\}$	$(glf_1 \mapsto \epsilon)$	(s_0, glf_1)
2	$\{(s_0, glf_1)\}$	$\{(s_0, glf_1)\}$	$(grf_1 \mapsto glf_1)$	(s_1, grf_1)
			$(glf_2 \mapsto \epsilon)$	$(s_0, glf_1 \cdot glf_2)$
3	$\{(s_1, grf_1),$	$(s_0, glf_1 \cdot glf_2)$		
	$(s_0, glf_1 \cdot glf_2)\}$			
4	$\{(s_1, grf_1)\}$	(s_1, grf_1)	$(plf_1 \mapsto grf_1)$	(s_3, plf_1)
5	$\{(s_3, plf_1)\}$	(s_3, plf_1)	$(prf_1 \mapsto plf_1)$	(s_6, prf_1)
6	$\{(s_6, prf_1)\}$	(s_6, prf_1)	$(glf_1 \mapsto \epsilon)$	$(s_6, prf_1 \cdot glf_1)$
7	$\{(s_6, prf_1 \cdot glf_1)\}$	$(q_s, prf_1 \cdot glf_1)$	$(grf_1 \mapsto prf_1 \cdot glf_1)$	(s_1, grf_1)
			$(glf_2 \mapsto \epsilon)$	$(s_6, prf \cdot glf_1 \cdot glf_2)$
8	$\{(s_6, prf \cdot glf_1 \cdot glf_2)\}$	$(s_6, prf \cdot glf_1 \cdot glf_2)$		

Table 2.2: The execution of the exploration algorithm.

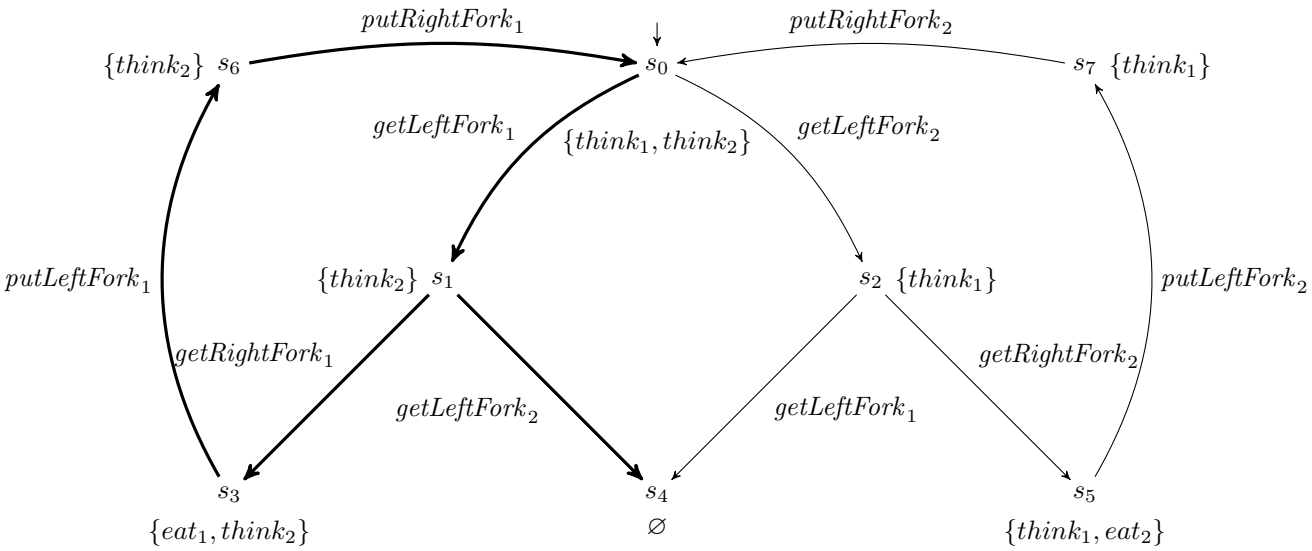


Figure 2.4: State space for the dining philosophers, explored with the probe set algorithm

2.3.4 Missed actions

As can be seen in the example in the previous section, the exploration of the state space using only probe sets is not sufficient. Some actions in the exploration of the state space are missed. Formally a missed action is defined as follows.

Definition 25 (Missed actions)**.** *For a vector (s, w) an action a is a missed action when*

$$s \nvdash w \cdot a \wedge \exists v \precsim w : s \vdash v \cdot a$$

A missed action a in (s, w) is not only characterized by the action itself, but also by its prime cause in the enabling trace v. Therefore a missed action a is given as: $(\downarrow_a v) \cdot a$

So a is a missed action when: a is not enabled right now, but a would have been enabled if the independent actions in the trace w were executed in a different order. Take, for example, vector $(s_0, getLeftFork_1 \cdot getLeftFork_2)$. This vector has $getRightFork_2$ as a missed action, because $getLeftFork_2 \precsim getLeftFork_1 \cdot getLeftFork_2$ and $s_0 \nvdash getLeftFork_1 \cdot getLeftFork_2 \cdot getRightFork_2$, but $s_0 \vdash getLeftFork_2 \cdot getRightFork_2$.

This missed action is characterized by $getLeftFork_2 \cdot getRightFork_2$. The prime cause is included, so we know the shortest weak prefix in which the action would have been enabled.

In the probe set algorithm a subset of the missed actions, the set of freshly missed actions is used.

Definition 26 (Fresh missed actions)**.** *An action a is freshly missed in vector $(s, w \cdot b)$, when a is missed in $(s, w \cdot b)$ but not missed in (s, w). The set of fresh missed actions of a vector is denoted with $fma(s, w)$ and, like a missed action, a fresh missed action is prefixed with the prime cause of its enabling trace.*

Because the algorithm explores one action at a time, every missed action is freshly missed exactly once. By only considering fresh missed actions, a single missed action does not have to be processed multiple times.

When an action a is freshly missed, the last action b must have some dependency with a and either $b \rhd a$ or $b \blacktriangleleft a$ must hold. This fact is used to decide on an over-approximation of the freshly missed actions in the next section.

Besides the probe sets, the probe set algorithm will use the fresh missed actions to generate new vectors to be explored. When some fresh missed actions are found during the exploration of vector (s, w), the following new vectors will be generated besides those generated by the probe set.

$$newvectors := \{(s \uparrow v \cdot a, \epsilon) \mid \forall v \cdot a \in fma(s, w)\} \tag{2.11}$$

For example, in Figure 2.4 $getLeftFork_2 \cdot getRightFork_2$ is a fresh missed action for vector $(s_0, getLeftFork_1 \cdot getLeftFork_2)$. Therefore, according to Equation 2.11, vector (s_5, ϵ) will be added to the stack of vectors to be explored. The normal exploration using probe sets, will now also include the remaining part of the state space that was not included before.

It can be shown that the probe set and missed actions phases together preserve deadlocks in the reduction of an acyclic system.

2.3.5 Determining the Fresh Missed Actions

In each iteration of the algorithm, it is necessary to calculate the fresh missed actions for the vector that is currently being considered.

As defined in Definition 25, the missed actions depend on the weak prefix of a trace. Determining all possible weak prefixes effectively results in the entire state space being explored. This of course negates any effects that might be gained by partial-order reduction. Therefore extra care is necessary when determining the (fresh) missed actions.

In general, it is not easy to determine the missed actions in a efficient way. However, in the case of entity-based systems, [17] proposes to use an over-approximation to calculate the missed actions: the potentially missed actions.

To determine the potentially missed actions, two new definitions are necessary, which are similar to Definition 14 and Definition 15, but which are only defined in the context of entity-based systems. In these definitions only the presence of entities is considered, and not their absence.

Definition 27 (Weakly enabled action)**.** *An action a is weakly enabled in s, denoted as $s \Vdash a$, when:*

$$(R_a \cup D_a) \subseteq E_s$$

So an action is weakly enabled in a state when the reader and deleter entities are present in the set of entities associated with that state.

Definition 28 (s weakly after w)**.** *s weakly after w is denoted $s \Uparrow w$ and is defined as follows:*

$$s \Uparrow w \overset{def}{\Longleftrightarrow} E_s \cup \bigcup_{a \in A_w} C_a$$

With $s \Uparrow w$, the target state of the trace w starting in s is calculated, like with $s \uparrow w$. However, in the case of $s \Uparrow w$ entities are only created and not deleted.

With the above two definitions, the set of potentially missed actions is defined.

Definition 29 (Potentially missed actions)**.** *When $(s, w \cdot b)$ is a vector, then action a is potentially missed when:*

$$\underbrace{s \Uparrow w \Vdash a}_{2.12a} \quad \wedge \quad \underbrace{\exists c \in A_w : c \blacktriangleleft a}_{2.12b} \quad \wedge \quad \underbrace{b \rhd a}_{2.12c} \tag{2.12}$$

holds. The set of all potentially missed actions of vector (s, w) is denoted as $pma(s, w)$.

As discussed in the previous section, for all missed actions a in $(s, w \cdot b)$ either $b \blacktriangleleft a$ or $b \triangleright a$ holds. Therefore a simple over-approximation would be all actions for which $b \blacktriangleleft a$ or $b \triangleright a$ holds. However, in the case of $b \blacktriangleleft a$, action a is already explored according Equation 2.10a (or a is not a missed action) and in the case of $b \triangleright a$ it is possible to pose some extra constraints to make the over-approximation more accurate.

While $b \triangleright a$, b must not enable a. Because if a is enabled, it is not a missed action. Therefore we check that w contains an action that disables a before b is executed, Equation 2.12b.

Equation 2.12a states that a should be weakly enabled in $s \Uparrow w$. If this would not be the case, a requires the presence of an entity that is never present along w. So whatever permutation of action of w is chosen, that entity will never be present. This last means that there cannot exist a weak prefix in which a is enabled. So a is not a missed action.

2.3.6 Algorithm

Now, the fresh missed actions phase can be added to the algorithm. The new algorithm can be found in Algorithm 2.

When the example in Table 2.2 is run again, a fresh missed action is detected in iteration 3. The missed action is $getRightFork_2$ with prime cause $getLeftFork_2$. According to the new algorithm, the vector (s_5, ϵ) now also has to be explored. Eventually this will also result in the exploration of vectors $(s_5, putLeftFork_2)$ and $(s_7, putRightFork_2)$, after which the algorithm returns to the initial state and is done.

So in the case of this new algorithm all desired states are explored.

Algorithm 2 Probe set based state space exploration 2

1: **let** $S \leftarrow \emptyset$; *//The set of states explored*
2: **let** $V_p \leftarrow \emptyset$; *//The set of already explored vectors*
3: **let** $V \leftarrow \{(s_0, \epsilon)\}$; *//The stack with vectors to be explored*
4: **while** $V \backslash V_p \neq \emptyset$ **do**
5: **choose** $(s, w) \in V \backslash V_p$;
6: **let** $V_p \leftarrow V_p \cup \{(s, w)\}$;
7: **for all** $v \cdot a \in pma(s, w)$ **do** *//Approximated according to Equation 2.12*
8: **let** $V \leftarrow V \cup \{(s \uparrow v \cdot a, \epsilon)\}$;
9: **end for**
10: **choose** probe set $p_{s,w}$; *//According to Equation 2.10*
11: **let** $V \leftarrow V \cup \{(s \uparrow p_{s,w}(a), w \cdot a - p_{s,w}(a)) | a \in dom(p_{s,w})\}$; *//According to Equation 2.9*
12: **end while**

2.3.7 Cycles and Fairness

In contrast to the ample set approach on partial-order reduction, the probe set approach is not immediately able to preserve deadlocks in cyclic state spaces. To be able to deal with cycles, a proviso, similar to Equation 2.6 for the ample set approach, is necessary.

See for example Figure 2.5. Without a cycle proviso it would be valid to explore only action a_1 or a_2 at every state. This does not only mean that the algorithm continuously cycles between states s_0 and s_3, it also means that deadlock state s_4 will not be found.

Static partial-order reduction does not suffer from this problem. Because actions a_1 and a_2 are dependent with action c, Equation 2.4 will force the algorithm to explore action b in state s_0.

To counter this problem in the probe set approach, only fair explorations of the state space are allowed.

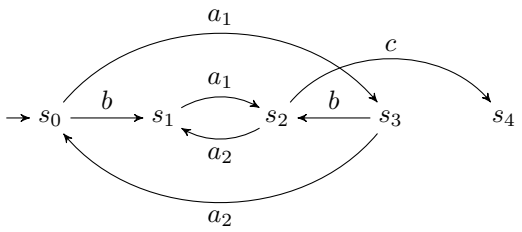


Figure 2.5: Probe sets and cycle proviso

Definition 30 (Fairness)**.** *Let V be the set of al vectors explored by the algorithm, and $p_{s,w}$ is the probe set for a given vector $(s, w) \in V$.*

An exploration of the state space is called fair if there exists a valuation function $n_{s,w} : Enabled(s \uparrow w) \to \mathbb{N}$, that assigns a natural number to every enabled action in the target of vector $(s, w) \in V$.

Moreover, one of the following should hold for every $a \in Enabled(s, w)$:

- *Either $a \in dom(p_{s,w})$, or*
- $\exists b \in dom(p_{s,w}) : n_{(s\uparrow p_{s,w}(b), w \cdot b - p_{s,w}(b))}(a) < n_{s,w}(a)$

The second constraint on the valuation function might seem complicated. However, $(s \uparrow p_{s,w}(b), w \cdot b - p_{s,w}(b))$ is defined in Equation 2.9 as the new vector that should be explored when action b is in the probe set $p_{s,w}$. Therefore $n_{(s\uparrow p_{s,w}(b), w \cdot b - p_{s,w}(b))}$ denotes the valuation function of one of the explored successors of vector (s, w).

The valuation function states that if an action a is enabled but not explored, the valuation of a in one of the successor vectors should be strictly smaller than the valuation in the current vector. Because the valuation is in $\mathbb{N}$, the valuation cannot decrease indefinitely and the second condition guarantees that the exploration of an enabled action cannot be postponed indefinitely.

Ensuring Fairness

[17] Shows one way to adapt Algorithm 2 to ensure fairness by means of an age function, which will be discussed here. There are more ways to ensure fairness, which are discussed in Section 3.1.

The age function is used to track the age of actions. When an action is enabled but not explored, its age increases. By requiring that the probe set contains at least one action whose age is maximal, fairness can be ensured.

The aging function and its operations are formally defined as follows:

Definition 31 (Aging function)**.** *An aging function α is a partial function: $\alpha : Act \rightharpoonup \mathbb{N}$.*

To manipulate the age function, the following operations are defined.

$$\begin{aligned} \alpha \oplus A &:= \{(a, \alpha(a)) + 1 \mid a \in dom(\alpha)\} \cup \{(a, 0) \mid a \in A \backslash dom(\alpha)\} \\ \alpha \ominus A &:= \{(a, \alpha(a)) \mid a \in dom(\alpha) \backslash A\} \\ max\ (\alpha) &:= \{a \in dom(\alpha) \mid b \in dom(\alpha) : \alpha(a) \geq \alpha(b)\} \\ A\ satisfies\ \alpha &:= \alpha = \emptyset \vee A \cap\ max\ (\alpha) \neq \emptyset \end{aligned}$$

The algorithm is changed to use the aging function, as shown in Algorithm 3. In this new algorithm, the vectors are extended to include an age function. On line 11 the aging function is updated; the age of actions already in the age function is increased, the enabled actions are added and the actions that are explored in the probe set are removed.

Finally, the newly generated probe sets must contain at least one element from max(α), as can be seen on line 10.

Algorithm 3 Probe set based state space exploration 3

1: **let** $S \leftarrow \emptyset$; *//The set of states explored*
2: **let** $V_p \leftarrow \emptyset$; *//The set of already explored vectors*
3: **let** $V \leftarrow \{(s_0, \epsilon, \emptyset)\}$; *//Start with an empty age function*
4: **while** $V \backslash V_p \neq \emptyset$ **do**
5: **choose** $(s, w, \alpha) \in V \backslash V_p$;
6: **let** $V_p \leftarrow V_p \cup \{(s, w, \alpha)\}$;
7: **for all** $v \cdot a \in pma(s, w)$ **do** *//Approximated according to Equation 2.12*
8: **let** $V \leftarrow V \cup \{(s \uparrow v \cdot a, \epsilon, \emptyset)\}$;
9: **end for**
10: **choose** probe set $p_{s,w}$, such that $dom(p_{s,w})$ satisfies α; *//According to Equation 2.10*
11: **let** $\alpha \leftarrow \alpha \oplus Enabled(s \uparrow w) \ominus dom(p_{s,w})$;
12: **let** $V \leftarrow V \cup \{(s \uparrow p_{s,w}(a), w \cdot a - p_{s,w}(a), \alpha) | a \in dom(p_{s,w},)\}$; *//According to Equation 2.9*
13: **end while**

2.3.8 Preserved properties

When considering the properties preserved by the probe set algorithm introduced in [17], it compares to a basic partial-order reduction algorithm with a cycle proviso; the probe set algorithm preserves deadlock states in cyclic state spaces. Note that in the probe set approach, the cycle proviso is not only necessary to preserve safety properties, but also to preserve deadlocks in cyclic state spaces.

Theorem 5 (Preservation of deadlocks)**.** *Given a deterministic (and possibly cyclic) transition system T and its reduction T' according to the probe set partial-order reduction algorithm in [17], the following holds:*

$$\forall s \in S : s \text{ reachable and deadlock in } T \implies s \text{ reachable and deadlock in } T'$$

2.4 Petri Nets

The partial-order reduction algorithms described in the previous sections are applied on Petri nets [23], which are introduced in this section. Petri nets are not entity-based systems, so they cannot be used directly. Section 4.2 shows how we transform Petri nets to entity-based systems.

2.4.1 Definition

The definition of Petri nets as used in this report is as follows:

Definition 32 (Petri net)**.** *A Petri net is a tuple: $PN = \langle P, T, A, A_i \rangle$, where*

- *P is the set places,*
- *T is the set of transitions,*
- *$A \subseteq (P \times T) \cup (T \times P)$ is the set of (normal) arcs,*
- *$A_i \subseteq (T \times P)$ is the set of inhibitor arcs.*

The places of a Petri net can contain zero or more tokens. These tokens can move through the arcs and transitions to new places. A 'snapshot' of the number of tokens on each location is called a *marking*:

Definition 33. *A marking $M : P \rightarrow \mathbb{N}$ is a function from the places to the natural numbers, that represents the number of tokens present at a given location.*

Usually, a Petri net PN is accompanied by an initial marking M_i, to denote the initial location of all the tokens.

Tokens move from one place to another when a transition connecting those places fires. A transition is able to fire when: all places that have an arc to the transition contain at least one token, and all places that have an inhibitor arc from the transition to itself have zero tokens.

When a transition fires, one token is consumed from every place that has an arc to the transition, and a token is produced on every place that has a normal arc from the transition to itself.

Definition 34 (Enabled transition)**.** *A transition t in a Petri net is allowed to fire when:*

- $\forall(p,t) \in A : M(p) > 0$
- $\forall(t,p) \in A_i : M(p) = 0$

Definition 35 (Firing a transition)**.** *When a transition t fires, a new marking M_{new} is created from the old marking M_{old} as follows:*

$$M_{new} = \begin{cases} M_{old}(p) - 1 & \text{if } (p,t) \in A, \\ M_{old}(p) + 1 & \text{if } (t,p) \in A, \\ M_{old}(p) & \text{otherwise.} \end{cases}$$

Petri nets have no mechanism in place to decide which specific transition is allowed to fire, so when multiple transitions are enabled for a single marking, one is chosen nondeterministically that is allowed to fire.

It is possible to generate a state space in the form of a transition system for a Petri net. In such a transition system each state represents a possible marking of the Petri net. The initial state of the transition system represents the initial marking of the Petri net.

There exists a transition in the transition system when there exists a transition in the Petri net, which when fired produces the marking represented by the target state from the marking represented by the source state. The label of the transition transition system is the transition that needs to be fired in the Petri net.

2.4.2 Example Petri Net

An example Petri net can be found in Figure 2.6. This is a Petri net describing the dining philosophers example from from Figure 2.3.

The circles in the Petri net are its places and the rectangles are the transitions. The upper part of the Petri net represents the first philosopher, and the lower part represents the second philosopher. The two places in the middle represent the forks.

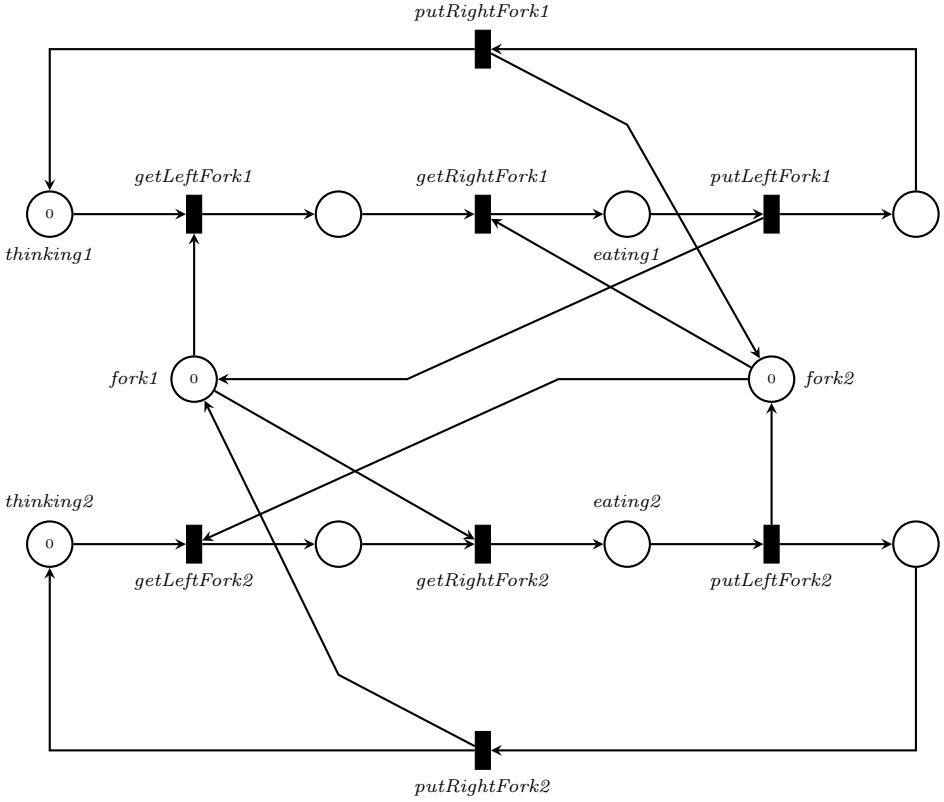


Figure 2.6: A Petri net of the dinging philosophers example

3 Algorithm Improvements

To improve the performance of the dynamic partial-order reduction algorithm based on probe sets, we made a few changes to the algorithm. First we introduce a new cycle proviso, and we prove that the new cycle proviso has the desired properties. Second, we describe a small improvement to the over-approximation of the potentially missed actions.

3.1 Cycle Proviso

Section 2.3.7 introduces fairness and shows how fairness can be enforced by the means of an age function.

When an age function is used, the probe set should always include one of the oldest actions. This means that an action should be explored as soon as possible, and postponing the action is almost impossible, while this may be desired to achieve a larger reduction.

We introduce a second cycle proviso as an alternative to the aging function, which does allow actions to be postponed.

Definition 36 (No duplicates cycle proviso)**.** *Given vector* (s, w) *and probe set* $p_{s,w}$*:*

- *Either:* $\forall a \in dom(p_{s,w}) : a \notin w$.
- *Or :* $dom(p_{s,w}) = Enabled(s \uparrow w) \wedge \forall a \in dom(p_{s,w}) : p_{s,w}(a)$ *contains* a.

With this cycle proviso, we are only allowed to explore actions that are not already in the trace of the current vector. If this is not possible, we force the algorithm to explore every enabled action and we make sure the new traces will not contain any duplicate actions by first removing the action we later append. Since $a \blacktriangleleft a$, we are allowed to add action a to $p_{s,w}$ without changing any of the existing constraints.

With the no duplicates cycle proviso, the exploration is guaranteed to terminate.

Theorem 6 (Termination)**.** *The no duplicates cycle proviso in Definition 36 guarantees termination of the probe set algorithm for finite state spaces.*

Proof. Because no trace contains duplicate actions, the maximum number of traces we can generate is equal to sum of the permutations for every combination of the actions, or:

$$\sum_{w \in \mathcal{P}(Act)} |w|!$$

Because the state space is finite, Act is finite, $|\mathcal{P}(Act)|$ is finite and the sum itself is finite. Because Act is finite, $|w|$ and thus $|w|!$ are finite for every w. So the result of the sum is finite.

Because the maximum number of traces we can generate is finite and the number of states is finite, the number of vectors we can generate is also finite and the exploration eventually terminates.

□

Besides guaranteeing termination, the no duplicates cycle proviso, like the cycle proviso for static partial-order reduction, also ensures that no actions are ignored.

Theorem 7 (No ignoring)**.** *The no duplicates cycle proviso in Definition 36 guarantees that no actions are ignored when a finite state space is explored.*

Proof.
By Contradiction.

- Assume we have a vector (s, w) with action b enabled in $s_n = s \uparrow w$. Assume the remainder of the exploration is $s_n \overset{a_{n+1}}{\rightarrow} s_{n+1} \overset{a_{n+2}}{\rightarrow} \cdots$ with $a_i \neq b$ for $i > n$, ie. action b is never explored in the remainder of the exploration.
- Because action b is not explored, it must be independent with with a_i for all $i > n$. Therefore action b is never disabled, and it is enabled in every state s_i with $i > n$. Because action b is always enabled, the sequence $s_n \overset{a_{n+1}}{\rightarrow} s_{n+1} \overset{a_{n+2}}{\rightarrow} \cdots$ does not contain a deadlock and it is infinitely long.

- Because the state space is finite, the sequence $s_n \overset{a_{n+1}}{\rightarrow} s_{n+1} \overset{a_{n+2}}{\rightarrow} \cdots$ must contain a loop. Now we have 2 cases:

1. **No actions on the cycle are removed from the trace of the vector before the cycle proviso detects the cycle.**
   - At some point the no duplicates cycle proviso will detect the cycle and explore every enabled action. Since action b is always enabled, this includes action b. ↯
2. **Actions on the cycle are removed from the trace of the vector before the cycle proviso detects the cycle.**
   - Assume no actions are removed until we reach state s_k with $k > n$.
   - Then we have a vector (s, w') with $w \precsim w'$ and $w' - w$ consists of a_i for $n < i < k$.
   - Since b is independent with and enabled before a_i for $n < i < k$, $\downarrow_b w'$ does not contain a_i with $n < i < k$.
   - If $a_{k+1} \in dom(p_{s,w'})$ contains an action a_i with $n < i < k$, then $p_{s,w'}(a_{k+1}) \not\precsim \downarrow_b w'$.
   - Therefore, according to Equation 2.10c, we cannot remove any action a_i with $n < i < k$ from the vector, without exploring action b. Therefore, action b is not ignored when we remove an action from the vector that is on the cycle. ↯

□

3.2 Potentially Missed Actions

In [17] the potentially missed actions are the union of the actions defined by Equation 2.12 and the actions in the following definition:

Definition 37 (Potentially missed actions 2)**.** *When $(s, w \cdot b)$ is a vector, then a is a potentially missed action when*

$$b \blacktriangleleft a$$

If an action is a potentially missed action according to the above definition, it is only a fresh missed action if it was enabled in the previous state $s \uparrow w$ and is disabled by the last executed action b. It makes much more sense to explore these type of actions during the probe set phase. We therefore removed these actions from the potentially missed actions and we changed the first constraint on valid probe sets (Equation 2.10a) from

$$a \blacktriangleleft b \implies b \in dom(p_{s,w})$$

to

$$a \blacktriangleleft b \vee b \blacktriangleleft a \implies b \in dom(p_{s,w})$$

4 Algorithm Implementation

The DRoP (Dynamic Reduction of Petri nets) tool is an implementation of the dynamic partial-order reduction algorithm introduced in Section 2.3. Although the original paper [17] provides some suggestions, the algorithm does not strictly define how a valid probe set should be selected. Therefore we present an alternative strategy to select probe sets to the strategy introduced in [17]. We compare the new strategy with the method suggested in [17] in Section 5.4.

The dynamic partial-order reduction algorithm takes entity-based transition systems as its input, while DRoP will work with Petri nets. Therefore, DRoP contains a transformation layer that transforms the Petri nets to entity-based systems. In Section 4.2 we introduce this transformation.

4.1 Probe Set Selection

In Section 2.3.3, probe sets are introduced and Definitions 23 and 24 define what a valid probe set is. The requirements in these definitions are loose, and allow us to select different valid probe sets for a single vector. To examine different selection strategies for probe sets, two strategies are implemented and compared during the experiments: the *reversing free* selection strategy and the *independent action* selection strategy.

4.1.1 Reversing Free Probe Set Selection

Definition 24 specifies when a probe set is valid. The probe set algorithm [17] also provides a way to actually select a valid probe set within these bounds: the reversing free probe set selection strategy. This strategy provides us with a more efficient way to calculate the missed actions and it is based on the reversing actions.

Definition 38 (Reversing actions)**.** *Two entity-based actions a and b are reversing when either $C_a \cap D_b \neq \emptyset$ or $D_a \cap C_b \neq \emptyset$, ie. two actions are reversing when they (partially) undo each others effect.*

A trace w is said to be reversing free when no two actions in w are reversing. Also, we use $rev_a(w)$ to denote the set of actions from trace w that are reversing with respect to action a.

With the definition of reversing free actions, we add an extra constraint to Definition 24:

$$rev_a(w) \subseteq p_{s,w}(a) \tag{4.1}$$

When an action is added to the probe set, the image at least needs to contain all the actions that are reversing with this action (but it is allowed to contain more actions). It is clear that all actions in $rev_a(w)$ are also part of $\downarrow_a w$. Therefore, this constraint does not conflict with the other constraints already present in Definition 24.

Equation 4.1 states that all reversing actions need to be part of the image of the probe set. According to Equation 2.9, these actions are removed from the vector, when new vectors are constructed. Together this means that we will only explore vectors that have reversing free traces.

When a vector (s, w) has a reversing free trace w, two interesting propositions can be proven:

Proposition 1. *For any action a, $q \vdash w' \cdot a$ with $w' \precsim w$ implies $w' = \downarrow_a w$.*

Proposition 2. *$fma(s, w) = \{a \in pma(s, w) \mid s \vdash \downarrow_a \cdot w\}$.*

The above propositions provide us with a way to extract the fresh missed actions from the potentially missed actions in an efficient way. Therefore, the reversing free probe set selection strategy allows us to explore the fresh missed actions directly instead of the over-approximated potentially missed actions, increasing the efficiency of the missed actions phase.

However, with this strategy Equation 2.10b in Definition 24 might cause more actions to be explored in the probe set phase of the algorithm.

4.1.2 Independent Action Probe Set Selection

To try and obtain a better reduction we introduce the independent action selection strategy, which selects a probe set according to the following constraints.

Definition 39 (Independent action probe set selection strategy)**.** *Given vector* (s, w)
if $\exists a \in Enabled(s \uparrow w)$*, such that* $\forall b \in Enabled(s \uparrow w), a \neq b \implies a \not\blacktriangleleft b \wedge b \not\blacktriangleleft a$ ***then***
$p_{s,w} = \{a \mapsto \epsilon\}$
else
$p_{s,w} = \{c \mapsto \downarrow_c w \mid \forall c \in Enabled(s \uparrow w)\}$
end if

In the independent action strategy, we explore only a single action that is independent of all other currently enabled actions according to Equation 2.10a. This action has an empty image in the probe set, so we do not need to check Equation 2.10b and Equation 2.10c because they trivially hold.

If such an independent actions does not exist, we explore every action, and we remove as many actions from the vector as allowed by Equation 2.10c. Because we already explore every action, we do not need to check Equation 2.10b.

The independent action strategy, is designed to circumvent Equation 2.10b in Definition 24. This equation is relatively expensive, because it includes a prime cause and a weak prefix operation.

It also causes more actions to be explored, even if actions are independent. For example, when two actions a and b are independent according to Equation 2.10a, we ideally only explore one of them. From Equation 2.10a and Equation 21 we can derive that a and b do not share creators and erasers ($C_a \cap C_b = \emptyset$ and $D_a \cap D_b = \emptyset$) and therefore it is unlikely that a and b share reversing actions, because these reversing actions would depend on the creators and erasers. When using the reversing free strategy to select actions for the probe set, Equation 2.10b will force both action a and b to be explored, unless they have exactly the same reversing actions in the current vector.

With the independent selection strategy, often only a single action is added to the probe set. Therefore the probe set phase of the algorithm becomes more efficient. However, we also do not remove many actions from the vectors, so the traces in those vectors can grow long, making the fresh missed actions phase less efficient.

4.2 Petri Net Transformation

As discussed in Section 2.3, the dynamic partial-order reduction algorithm is applied on entity-based systems, while DRoP works on Petri nets. Therefore DRoP contains a transformation layer that transforms Petri nets to entity-based systems.

Two transformations, a symbolic and a concrete transformation, were considered to be included in DRoP. The symbolic transformation preserves the symbolic nature of the tokens in the Petri net, but some changes to the partial-order reduction algorithm are necessary to accommodate this. In the concrete transformation multiple tokens in a single place are considered as different entities. This is different from the standard symbolic interpretation of the tokens, however no changes to the partial-order reduction algorithm are necessary to support this transformation.

The final version of DRoP contains only the concrete transformation, as discussed in Section 4.2.3.

4.2.1 A Symbolic Transformation

The symbolic transformation is aimed at preserving the symbolic nature of Petri nets as much as possible. To do this, we define a symbolic entity-based system. In this system, a state is not a set of entities, but a multiset of entities. The global set of entities, *Ent*, is defined to be equal to the set of places in the Petri net, P.

Definition 40 (Entity-based state)**.** *A symbolic state s is defined as a function $s : P \to \mathbb{N}$. $s(p)$ defines how many tokens are present in place p for symbolic state s.*

The symbolic states are accompanied by symbolic actions that are defined as follows:

Definition 41 (Symbolic Action)**.** *A symbolic action is a tuple $\langle R, D, C, N\rangle$, where:*

$$\begin{array}{ll} R : P \to \mathbb{N} & \textit{is the reader set} \\ D : P \to \mathbb{N} & \textit{is the eraser set} \\ C : P \to \mathbb{N} & \textit{is the creator set} \\ N : P \to \mathbb{N} & \textit{is the forbidden set} \end{array}$$

$dom(F)$ must be disjoint with $dom(R)$, $dom(D)$ and $dom(C)$. $dom(C)$ must be disjoint with $dom(D)$.

Definition 42 (Enabled symbolic action)**.** *A symbolic action $a = \langle R, D, C, N\rangle$ is enabled in symbolic state s, when:*

$$\begin{aligned} \forall p \in R \cup D : s(p) &\geq R(p) + D(p) \\ \forall p \in R \backslash D : s(p) &\geq R(p) \\ \forall p \in D \backslash R : s(p) &\geq D(p) \\ \forall p \in N : s(p) &= 0 \end{aligned}$$

Definition 43 (Executing a symbolic action)**.** *When executing the symbolic action $a = \langle R, D, C, N\rangle$ in symbolic state s, the result is a new symbolic state s':*

$$\forall p \in P : s'(p) = s(p) + C(p) - D(p)$$

Because the symbolic entity-based actions are different from the entity-based actions defined in [17], it is necessary to redefine the stimulating and disabling relations for the symbolic entity-based transition system to continue to be able to apply partial-order reduction.

There are two possible ways to redefine these relations: 1) a global definition and 2) a local definition.

The global definition for the dependency relations are similar to the original definitions in [17]. For the global definition we only take the actions themselves into account. Therefore the global relation can be defined in advance and it can be reused for different Petri nets that share the same actions. The global definitions are as follows:

Definition 44 (Global symbolic stimulating relation)**.** *Given actions $a = \langle R_a, D_a, C_a, N_a\rangle$ and $b = \langle R_b, D_b, C_b, N_b\rangle$, the simulating relation is defined as follows:*

$$a \rhd b \overset{def}{\iff} dom(C_a) \cap (dom(R_b) \cup dom(D_b)) \neq \emptyset \vee dom(D_a) \cap dom(N_b) \neq \emptyset$$

Note that it is always allowed to create more entities (tokens), so we do not need to check the intersection of $dom(D_a)$ and $dom(C_b)$, as is the case in the original stimulating relation.

Definition 45 (Global symbolic disabling relation)**.** *Given actions $a = \langle R_a, D_a, C_a, N_a\rangle$ and $b = \langle R_b, D_b, C_b, N_b\rangle$, the disabling relation is defined as follows:*

$$a \blacktriangleleft b \overset{def}{\iff} dom(D_a) \cap (dom(R_b) \cup dom(D_b)) \neq \emptyset \vee dom(C_a) \cap dom(N_b) \neq \emptyset$$

Note that it is always allowed to create more entities (tokens), so we do not need to check the intersection of $dom(C_a)$ and $dom(C_b)$, as is the case in the original disabling relation.

The advantage of the global dependency relation is that it only depends on the actions. It can be calculated in advance, and it will be valid for the entire exploration.

The disadvantage is that actions that are globally dependent are not always locally dependent. See for example Figure 4.1. Here action T_0 and action T_1 disable each other because they both consume tokens from the same place. However, the actual disabling occurs only when the last token in the place is consumed. Therefore actions are marked as dependent more often than is necessary.

To counter the disadvantage of the global relation, a local relation can be defined that takes into account the number of tokens a place currently contains. Therefore the actions are less often marked as dependent. Formally the local dependency relations are defined as follows:

Definition 46 (Local symbolic stimulating relation)**.** *Action a stimulates action b in state s, denoted $a \rhd_s b$, when:*

$$a \rhd b \wedge b \notin Enabled(s)$$

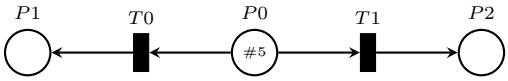


Figure 4.1: Example Petri net

Definition 47 (Local symbolic disabling relation)**.** *Action a disables action b in state s, denoted $a \blacktriangleleft_s b$, when:*

$$a \blacktriangleleft b \wedge \forall p \in P : s(p) - D_a(p) < R_b(p) + D_b(p)$$

Because the dependency relations do now depend on the states of the transition system, the operations that depend on the dependency relations also have to be redefined. For example, the influence and weak equivalence relation are redefined as follows:

$$a \rightsquigarrow_s b \overset{def}{\Longleftrightarrow} a \rhd_s b \vee b \blacktriangleleft_s a$$

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} s_4 \simeq s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_2} s_2' \xrightarrow{a_1} s_3 \xrightarrow{a_3} s_4 \Leftrightarrow a_1 \not\rightsquigarrow_{s_1} a_2$$

A major drawback of the symbolic approach is that calculations involving weak equivalent traces become much harder. For weak equivalent traces it is defined that their target is the same. However, the states we pass before reaching the target are different for each weak equivalent trace. Because the dependency relations rely on these states, weak equivalent traces can no longer just be substituted, but we also have to perform extra calculations to discover the new intermediate states.

4.2.2 An Explicit Transformation

The symbolic transformation preserves the semantics of Petri nets, but it requires unwanted changes to the algorithm. Therefore we have also defined an explicit transformation. With this transformation we require much fewer changes to the original algorithm, but the semantics of Petri nets are not preserved as well as with the symbolic transformation.

With the explicit transformation, we do not simply count the number of tokens at a specific place, but we also add a unique identifier to each token. With this identifier it is possible to explicitly reference a single token. Therefore, tokens can now be used as concrete entities.

Since we do not know in advance how many tokens a Petri net will contain before executing, it is not possible to determine the token identifiers in advance. Therefore it is also not possible to determine all the concrete actions in advance. Our solution is to represent the transitions in the Petri net as symbolic actions and to provide a *concretization* function that transforms the set of symbolic actions to a set of enabled concrete actions for any given state.

Below we use Ent_p to denote the set of all concrete entities in place p.

Concretization

The *concretization* function transforms a set of symbolic actions to a set of enabled concrete actions for any given state. Only the enabled concrete actions are created; since the concretization is state specific, disabled concrete actions would never be used.

The concretization consists of five phases. In the first four phases the concrete readers, the concrete erasers, the concrete creator and concrete forbidden entities are determined. In the last phase, the previous phases are combined to form the concrete actions.

Reader Phase Producing the possible concrete reader set occurs in two steps, as can be seen in Algorithm 4.

First, concrete reader sets are created for each individual place (Line 5). This amounts to creating all permutations of length $R(p)$ of the tokens currently in place p.

Next, the different places are combined in the final result (Line 6). The result of this phase is a set of sets CR. Each set in CR is a valid concrete reader set.

Algorithm 4 Select concrete reader sets

given state s and transition t
let $\langle R, D, C, N\rangle \leftarrow \mathit{AbstractAction}(t)$; *//Get the abstract action representing transition t.*
let $CR \leftarrow \{\emptyset\}$;
for all $p \in P$ **do**
    $Z_p = \{E \subseteq s \cap \mathit{Ent}_p \mid |E| = |R(p)|\}$;
    $CR = \{cr \cup z \mid cr \in CR, z \in Z_p\}$;
end for

Eraser Phase The concretization of the eraser entities happens in exactly the same way as the concretization of the reader entities. The result is a set of sets CE. Each set in CE is a valid concrete eraser set.

Note that no restrictions are posed on the concretization of the eraser entities. This means that a valid set of concrete eraser entities may contain an entity that is also present in a valid set of concrete reader entities. Since the set of erasers and readers of a concrete action must be disjoint, we have to check for this in the final phase of the concretization, where the erasers and readers are combined.

Creators Phase In contrast to the readers and eraser entities, only a single set of concrete creator entities is necessary. Because we only create entities that are not already present, every set of concrete creator entities we can create, will be equivalent.

Algorithm 5 shows how the creator entities are selected. The result is the set CC, which is a single set containing the creator entities.

Algorithm 5 Select concrete creator entities

let $\langle R, D, C, N\rangle \leftarrow aa(t)$;
for all $p \in \mathit{dom}(C)$ **do**
    let $E \subseteq \mathit{Ent}_p$, such that $|E| = C(p)$ and $E \cap s = \emptyset$;
    $CC = CC \cup E$;
end for

Forbidden Phase The concretization function should only return enabled concretizations. Therefore if any forbidden entity is present in the current state, this action does not return any concretizations.

If the current state does not contain any forbidden entities, this action can return concretizations. However, we will not create any concrete forbidden entities. Tokens, and thus entities, are dynamically created during the exploration of the state space. Therefore it is currently impossible to determine the exact concrete forbidden entities.

Even if we do not have any concrete forbidden entities, the execution of actions will remain the same. Only the enabledness of actions can change, but we already check if actions are enabled during the concretization, so this does not matter.

Combining the Results In the final phase of the concretization, the results of the previous four phases are combined. Algorithm 6 shows the pseudo-code for this phase.

The concrete readers CR and the concrete erasers CE, contain multiple valid concrete readers sets and concrete erasers sets. We create a concrete action for every combination of a reader set and eraser set, where both sets are disjoint.

We have only one concrete creator set, which is already disjoint from the others. Therefore, it can directly be added to every concrete action we generate.

The set CA contains the final concrete enabled actions.

Algorithm 6 Combining the results

$CA = \{\langle R, D, CC\rangle \mid R \in CR, D \in CD, R \cap D = \emptyset\}$;

4.2.3 Comparison

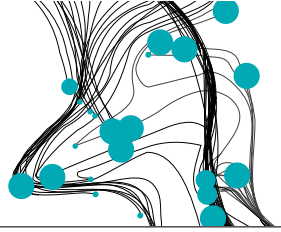
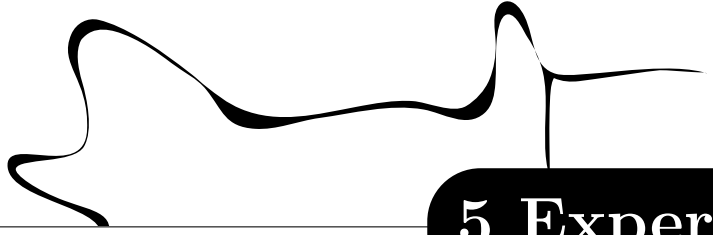
Between the symbolic and explicit transformation, it is clear that the symbolic transformation better preserves the semantics of Petri nets. In the explicit representation each token is unique, which leads to multiple concrete actions that represent the same transition in the Petri net. In the symbolic representation every Petri net transition has exactly one action in the entity-based transition system.

By having multiple concrete actions per Petri net transition, the explicit representation does not only change the semantics of the Petri net, it also blows up the state space of the transition system exponentially. This happens because we need to explore every possible permutation of the tokens that we can consume, while in the symbolic representation only one permutation suffices.

In contrast to Petri nets, the semantics of graph transformation systems are explicit. So while the explicit representation is not well-suited for Petri nets, it is suited as a representation for graph transformation systems. Since the partial-order reduction algorithm is designed to be applied on graph transformation systems, the explicit representation better suits the purpose of the reduction algorithm.

Also, even though the symbolic representation itself is simpler, the reduction algorithm becomes less efficient. The dependency relations either become a local relation, leading to much more calculations. Or the dependency relations become a global inefficient over-approximation, that leads to less reduction. So even though we do not have the explosion in actions as with the explicit transformation, exploring the state space will still become more complex.

Because the symbolic representation essentially does not remove but only moves complexity, and because the explicit representation fits the purpose of the partial-order reduction algorithm better, we decided to implement only the explicit transformation in DRoP.


5 Experiments

In this chapter, we discuss the tools, the models and the experiments that we performed to verify the efficiency of DRoP.

First we compare the different probe set selection strategies introduced in [17] and developed in Section 4.1 and the different cycle provisos introduced in [17] and developed in Section 3.1. The results of these experiments can be found in Section 5.4.

Once we have decided which probe set selection strategy and cycle proviso yield the best result, we verify that our implementation is actually able to perform better than existing algorithms. To do this, we compare DRoP with existing state space exploration tools that are able to apply partial-order reduction. The performance of the different tools are compared on different properties: the size of the reduction (number of states and transitions in the state space) and on the time and memory needed to explore the reduced state space. The results of this experiment can be found in Section 5.5 and the description of the tools we use to compare DRoP with can be found in Section 5.1.

A description of the models that we use in our test can be found in Section 5.2. We use two types of test models: the first type of models are models with dynamic (de)allocation of threads and objects, ie. the type of models where dynamic partial-order reduction was designed to be used for; the second type of models are models for which static partial-order reduction is already able to perform well.

In the first case, we show that DRoP performs better than existing tools in both the size of the reduction and the amount of time and memory that is needed to explore the reduction. In the second case we show that DRoP is able to achieve the same amount of reduction. However since dynamic partial-order reduction is more complicated than static partial-order reduction DRoP has a small overhead in the time and memory that is needed to explore the reduction.

Figure 5.1 below summarizes the models and the tools used in the experiments. It also shows how we obtained the specific models for each tool.

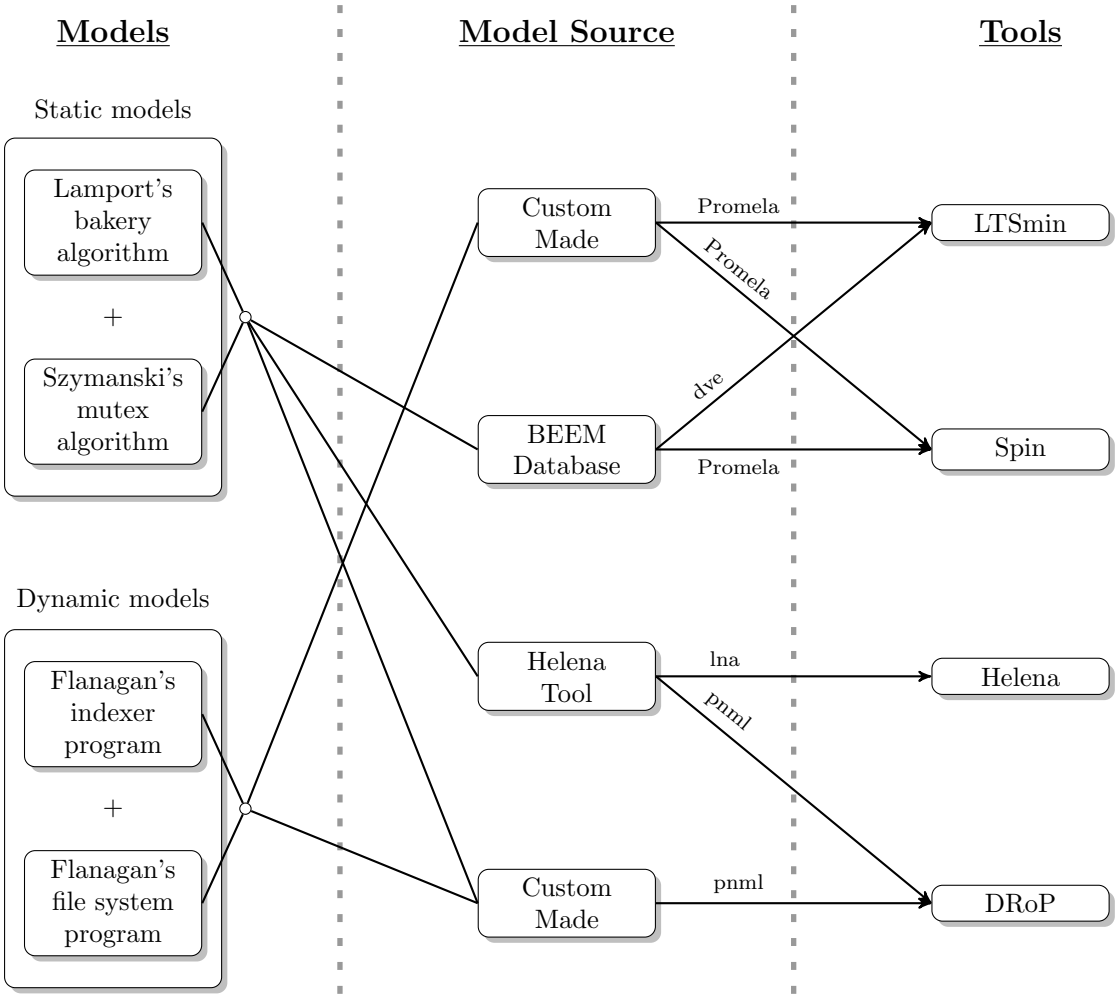


Figure 5.1: Overview of the experiments performed in this chapter.

5.1 Reference Tools

To verify the efficiency of DRoP, we explore our test models with different tools and compare the results. The tools that we use as a reference are: LTSmin (version 2.0) [6], Spin (version 6.2.2) [8], and Helena (version 2.1) [13].

DRoP

DRoP is our own implementation and its code can be found in the FMT git repository [4]. DRoP accepts input models in the Petri Net Markup Language (PNML) [1, 2, 7]. To obtain the PNML models, we used two methods: the first is to create these models ourselves, and the second way is to export the models provided by Helena to PNML.

Not all the test models we use are available in Helena. When the models are available in Helena, we perform our tests with both our own PNML model, and the PNML model exported from Helena. This way, we can verify the quality of our own models. When the model is not available in Helena, we use only our own model.

Spin

Spin [8], which we use because of its popularity, is a well known state space exploration tool that has been under development since the 1980's. Spin takes its input as Process Meta Language (Promela) models [16]. It compiles these models to C code, which in turn can be compiled to a model checker for the original model.

One of the reasons that Spin became a popular model checker is that it is continuously kept up to date by implementing new features as soon as a new breakthrough is reached in model checking research. One such feature, the one that interests us, is support for partial-order reduction.

Because of the support of partial-order reduction together with the popularity of the model checker, we decided to use Spin as a reference for DRoP. In our experiments, version 6.2.2 of Spin was used.

To compile the Promela models and run the exploration with Spin, the following commands are used:

```
spin -o3 -a /path/to/promela/model
gcc -o pan [-DNOREDUCE] pan.c
./pan
```

The `-DNOREDUCE` disables (static) partial-order reduction. The `-o3` disables some automatic optimizations that are performed independent of partial-order reduction. By disabling these optimizations, Spin produces the same complete state spaces as LTSmin.

LTSmin

Second we compare DRoP to the LTSmin [6] model checker which is created and maintained by our research group, the Formal Methods and Tools group [5] at the University of Twente. LTSmin is build around the PINS interface [10]. The PINS interface is an abstraction layer between state space exploration back ends and language front ends, which allows for a modular design.

For example, by implementing a small translation layer for PINS, LTSmin can easily integrate with existing tools. We will be using the Promela [16] and DiVinE [9] interfaces [28]. The PINS interface also allows for a modular approach on the side of the exploration algorithms. One example is the partial-order reduction layer, based on persistent sets, that was recently implemented [20].

For our experiments we used version 2.0 of LTSmin. We use DVE and Promela for input models, Section 5.2 states for each model whether a DVE or Promela version of the model was used.

To run the DiVinE models, the following commands are used:

```
divine compile -l /path/to/divine/model
dve2lts-seq [--por] model2C
```

To run the Promela models, the following commands are used:

```
spinjal -o3 /path/to/promela/model
prom2lts-seq [--por] model.spinja
```

In both cases the `--por` option determines if (static) partial-order reduction will be enabled. The `-o3` for the Promela models disables certain optimizations that are performed independent of partial-order reduction. By disabling these optimizations for both LTSmin and Spin, both tools will generate the exact same complete state spaces for the Promela models. This way, the comparison between the tools will be more fair.

LTSmin uses tree compression to reduce the size of the state vectors, which the other tools do not. This is to compensate for the design of LTSmin, which makes the state vectors larger when compared to other tools.

Helena

Helena [12,13] is a state space exploration tool based on high-level Petri nets. High-level Petri nets, which are not discussed in this report, are Petri nets where the tokens in the net can contain data. This makes the definition of Petri nets more flexible and more compact. Helena is also able to export the high-level Petri nets it uses to low-level Petri nets in PNML format, which is the input format of DRoP.

We use Helena as a reference for DRoP, because it is also a tool that works based on Petri nets. Because Helena is able to export PNML models, we also use it as an alternative source of test models. For our experiments, version 2.1 of Helena was used.

To compile the Helena models and explore their state spaces, the following command is used:

```
helena --static-reductions=0 [--partial-order=1] /path/to/helena/model
```

The `--partial-order=1` option is added to enable partial-order reduction. The `--static-reductions=0` option is used to disable the reductions that are performed by default.

5.2 Models

Two types of test models are considered when comparing the tools. The first type of models are models with dynamic (de)allocation of threads and objects, ie. the type of models where dynamic partial-order reduction was designed for. The second type of models are models for which static partial-order reduction is already able to perform well.

The first type of models are taken from Flanagan's paper [14] on dynamic partial-order reduction, the second type of models are taken from the BEEM database [3].

Flanagan's Examples

The first two examples are the indexer program and the file system program from Flanagan's paper [14].

Indexer Program

The pseudo code of the indexer program can be found in Algorithm 7. This program revolves around a hash table. Each thread in the program generates some messages, calculates the hash of these messages, and tries to store those messages in the hash table.

The model contains three parameters, as can be seen in the table below. The number of messages generated by each thread and the size of the hash table are fixed for every experiment, and are set to 4 and 128 respectively. This equals the values in [14], so we can compare the results. Multiple instances of the model are verified, with an increasing number of threads. This way, we can verify models of increasing size.

Name	Variable	Description
SIZE	no	The size of the hash table.
MAX	no	The number of messages generated per thread.
THREADS	yes	The number of threads.

Static partial-order reduction is unable to reduce the state space for this program, because all access to the hash table must be marked as dependent. It is impossible to statically determine which hashes will be generated in advance and therefore it is unknown which locations in the hash table are accessed. Therefore we need to assume that every access to the hash table collides, and that every action is dependent.

When using dynamic partial-order reduction, we do not need to know all actions in advance. Therefore dependencies can be resolved after the hashes have been generated, and it can be detected when collisions actually occur. When the example is run with at most 11 threads, no hash collisions will occur, and we expect that dynamic partial-order reduction is able to reduce the state space. For more than 11 threads, hash collisions will occur in this model, and more actions become dependent. Therefore, dynamic partial-order reduction will no longer be able to reduce as much after the number of threads exceeds 11.

To be able to use the model, the pseudo code in Algorithm 7 was converted to a promela model to be used for the Spin and LTSmin tools, and a Petri net to be used in DRoP. All models can be found in the examples directory of the git repository [4].

Algorithm 7 Flanagan's indexer program, taken from [14]

```
int SIZE = 128;
int MAX = 4;
int[SIZE] table;

function THREAD(tid)
    int m=0, w, h;
    while true do                                   //Repeat, getmsg() takes care of termination.
        w := getmsg();                                                    //Get a new message.
        h := hash(w);                                                     //Hash the message.
        while cas(table[h],0,w) == false do         //Loop until table[h] is empty and store the message.
            h := (h + 1) % SIZE;
        end while
    end while
end function

function GETMSG(())                          //Generate new messages until MAX messages are generated.
    if m < MAX then
        return (++m) * 11 + tid;
    else
        exit();
    end if
end function

function HASH(int w)
    return (w * 7) % SIZE;
end function
```

File System Program

The pseudo code of the second example, the file system program, can be found in Algorithm 8. The program is inspired by the synchronization mechanism of the Frangipani file system [25]. In this file system, each file is represented by an inode. This inode contains a pointer to a block on disk that contains the actual data of the file. Each block has a busy bit to indicate that is has been assigned to an inode, and each individual inode and busy bit have a lock to regulate access.

Each thread in the file system program in Algorithm 8 picks an arbitrary inode, and when this inode is node yet assigned with a block, the threads tries to find an unused block and assigns it to the inode. We are not interested in the actual assignment of blocks to inodes, but only to the locking mechanism protecting this operation. Therefore, threads only assign the busy bit.

The program contains three parameters, as can be seen in the table below. The number of inodes and the number of blocks are fixed for every experiment, and are set to 32 and 26 respectively. This equals the values in [14], so we can compare the results. Multiple instances of the model are verified, with an increased number of threads. This way, we can verify models of increasing size.

Name	Variable	Description
NUMBLOCKS	no	The number of blocks in the file system.
NUMINODES	no	The number of inodes in the file system.
THREADS	yes	The number of threads.

Static partial-order reduction is unable to reduce the state space for this program. Because we are unable to statically determine which inodes and blocks will be accessed (and thus we are unable to determine if there are collisions), we have to assume every action collides. Therefore all actions become dependent and nothing can be reduced.

When using dynamic partial-order reduction, we do not need to know all actions in advance. Therefore dependencies can be resolved after we know which inodes and blocks are accessed, and it can be detected when collisions actually occur. When the number of threads is at most 13, there are no collisions and

we expect that dynamic partial-order reduction is able to reduce the state space. When there are more than 13 threads, collissions will occur, and more actions become dependent. Therefore we expect that dynamic partial-order reduction is not able to reduce as much when the number of threads exceeds 13.

To be able to use the model, the pseudo code in Algorithm 8 was converted to a Promela model to be used for the Spin and LTSmin tools, and a Petri net to be used in DRoP. All models can be found in the examples directory of the git repository [4].

Algorithm 8 Flanagan's File System Program, taken from [14]

```
int NUMBLOCKS = 26;
int NUMINODE = 32;
boolean[NUMINODE] locki;
int[NUMINODE] inode;
boolean[NUMIBLOCK] lockb;
boolean[NUMIBLOCK] busy;

function THREAD(tid)
    int i, b;
    i := tid % NUMINODE;                          //Select a random inode.
    acquire(locki[i]);                            //Acquire the lock on the inode.
    if inode[i] == 0 then                         //If the inode has no block, assign one.
        b := (i*2) % NUMBLOCKS;
        while true do                             //Loop until an unused block is found.
            acquire(lockb[b])                     //Lock the block.
            if !busy[b] then                      //If the block is unused, assign it to the inode.
                busy[b] := true;
                inode[i] := b;
                release(lockb[b]);
                break;                            //An empty block was found, so quit seraching.
            end if
            release(lockb[b]);
            b := (b+1) % NUMBLOCKS;               //Go to the next block.
        end while
    end if
    release(locki[i]);
    exit();
end function
```

BEEM Models

Besides the two models from Flanagan's paper [14], two models are chosen from the BEEM database [3]: Lamport's bakery algorithm [19] and Szymanski's mutual exclusion algorithm [24]. The BEEM database is essentially a list of models and statistics that is meant to be used for benchmarking model checkers. The models of the BEEM database are divided into sections, according to the purpose of the algorithms. Both models we selected are models from the mutual exclusion section of the database, because mutual exclusion algorithms lend themselves well to be represented as a Petri net.

Bakery Algorithm

The pseudo code of the Lamport's bakery algorithm [19] can be found in Algorithm 9. The bakery algorithm is a mutual exclusion algorithm that is based on a waiting line at the bakery. When a thread wants to enter the critical section, it picks a ticket with a number that is one higher than all other currently assigned tickets. The thread that has the ticket with the lowest number, is allowed in the critical section first. When two threads get a ticket at the same time, they can acquire the same number. In this case, the thread with the lowest `tid` is allowed in the critical section first.

The bakery algorithm has two parameters, as can be seen in the table below. By increasing the number of threads, the model gets larger and it is possible to verify the model with different sizes. The maximum ticket number is an artificial upper bound on the the ticket numbers the threads can acquire. The bakery algorithm itself does not have this limit. But then the ticket numbers could increase infinitely, thus creating an infinite state space. Since we are unable to explore infinite state spaces, we impose a maximum ticket number to make the state space finite. Using this approach introduces extra deadlock states to the state space, namely when a thread cannot acquire a ticket because the maximum ticket number is already reached. Since we add this limit to the model for every tool, all models will still contain the same number of deadlock states. In our experiments we also increase the maximum ticket number to get larger models. All models can be found in the examples directory of the git repository [4].

Name	Variable	Description
MAX (m)	yes	Artificial maximum ticket number.
NUMTHREADS (p)	yes	The number of threads.

In our experiments we use different versions of the bakery model. For the Spin and LTSmin tools we use the Promela and DVE versions from the BEEM database respectively and Helena has its own model lna included with the Helena binaries. For DRoP we have created our own model, but we also exported the Helena version of the model. Therefore we can run DRoP with different input models. The parameters in the models from the BEEM database are not easily changed. Therefore we do not have a BEEM counterpart (and thus Spin and LTSmin model) for every model we created ourselves and exported from Helena.

Algorithm 9 Lamport's bakery algorithm

```
boolean[NUMTHREADS] entering = false;
int[NUMTHREADS] number = 0;

function THREAD(tid)
    entering[tid] = true;                                   //Signal that we are picking a ticket number.
    number[tid] = 1 + max (number[0], number[1], ..., number[NUMTHREADS-1]);   //Pick a number.
    entering[tid] = false;                                  //Signal that we have picked a ticket number.
    for all i ∈ 1..NUMTHREADS-1 do                          //Check the ticket numbers of all threads.
        while entering[i] do          //If thread i is busy picking a ticket number, wait until it is done.
        end while
        while number[i] != 0 & (number[i] < number[tid] | (number[i] == number[tid] & i < tid)) do
        end while       //If thread i has a lower ticket number, wait until it exits the critical section.
    end for
    Critical section
    number[tid] = 0;                                        //Signal we have left the critical section.
end function
```

Szymanski's Algorithm

Algorithm 10 shows the pseudo code of Szymanski's mutual exclusion algorithm [24]. This algorithm guarantees mutual exclusion of the critical section for any number of threads, with only a fixed amount of memory per thread. The algorithm is modeled after a waiting room. First, threads have to enter a waiting room. Then the door of the waiting room is locked, and the threads in the waiting room are allowed in the critical section one by one. When the waiting room is empty again, the next group of threads may enter.

In more detail, when a thread wants to enter the critical section it first has to pass through the waiting room and it sets it `flag` to 1 to signal other processes it wants to enter the waiting room. When the waiting room is empty (all `flags` are 0 or 1), all processes that currently waiting, enter the waiting room. To signal this, all threads change their `flag` from 1 to 2 or 3. When all threads are in the waiting room, they set their `flag` to 4, and no other threads can enter the waiting room until it is empty again. When the waiting room is closed (ie. not empty), all threads in the room are allowed in the critical section in the order of their `tid`.

Szymanski's algorithm has a single parameter: the number of threads. In our experiments we increase the number of threads, to explore increasingly larger models.

Name	Variable	Description
NUMTHREADS	yes	The number of threads.

In our experiments we use different versions of Szymanski's algorithm. For the Spin and LTSmin tools we use the Promela and DVE versions from the BEEM database respectively and Helena uses its own model included with the Helena binaries. For DRoP we have created our own model, but we also exported the Helena version of the model. Therefore we can run DRoP with different input models. The parameters in the models from the BEEM database are not easily changed. Therefore we do not have a BEEM counterpart (and thus Spin and LTSmin model) for every model we create ourselves and exported from Helena.

Algorithm 10 Szymanski's algorithm

```
int[NUMTHREADS] flag = 0;

function THREAD(tid)
    flag[tid] = 1;                                   //Signal we want to enter the waiting room.
    while ∃ i ∈ 0..NUMTHREADS-1 : flag[i] ∉ {0,1,2} do   //Wait until the waiting room is empty.
    end while
    flag[tid] = 3;                                   //Enter the waiting room.
    if ∃ i ∈ 0..NUMTHREADS-1 : flag[i] == 1 then
        flag[tid] = 2;                               //If more threads want to enter the waiting room ...
        while ∄ i ∈ 0..NUMTHREADS-1 : flag[i] == 4 do
        end while                                    //... wait until they have done so.
    end if
    flag[tid] = 4;                                   //Close the waiting room, and signal that we are to enter the cs.
    while ∃ i ∈ 0..tid-1 : flag[i] ∉ {0,1} do //Wait until all threads with a lower tid have passed the cs.
    end while
    Critical section
    while ∃ i ∈ tid+1..NUMTHREADS-1 : flag[i] ∉ {0,1,4} do
    end while   //Wait until all threads with a higher tid are aware they can leave the waiting room.
    flag[tid] = 0;                                   //Signal that we have left the critical section.
end function
```

5.3 Test Setup

Before continuing to the results of the experiments, this section first discusses how the results are gathered.

Each exploration is run on a machine with an Intel E5520 CPU. If the exploration is not finished within one hour, or if the exploration requires more than 20GB of memory, the exploration is cancelled and we remove intermediate results. These time and memory limits are monitored and enforced by the `memtime` program.

For the LTSmin, Spin and Helena tools each experiment is run 5 times. The final result is the average of these results.

DRoP is written in Java. The initial run of Java is always worse than successive runs, because at the start the JVM still has to initialize its optimizations. Because the result of the first run is always bad, experiments with DRoP are run 6 times. The final result for the experiments with DRoP is the average of the second through the sixth run.

As discussed in Section 5.1, DRoP randomly selects a probe set, when multiple valid probe sets are available. The choice of probe sets is very important to the amount of reduction we can obtain. Therefore we use a different random seed for every test run, so different random probe sets are selected every run. This way we avoid that our results are extremely good or extremely bad because of a (un)lucky random selection of probe sets.

The results of the experiments are collected from the following sources.

	#Nodes	Time	Memory
DRoP	Internal	`System.currentTimeMillis()`	`Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()`
LTSmin	Tool output	`memtime` output	`memtime` output
Spin	Tool output	`memtime` output	`memtime` output
Helena	Tool output	`memtime` output	`memtime` output

For DRoP, the data is collected directly inside the source code. The time is measured by the difference in the result of `System.currentTimeMillis()` at the start and end of the exploration. The memory usage is measured by a call to `Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()` after a single call to garbage collector at the end of the exploration.

For the other tools, the number of states is read from the output of the tool itself. For the amount of time and memory that was used, the output of `memtime` is used. By relying on `memtime`, the results are not completely accurate, especially for the memory usage. However, each tool uses different (compression) techniques to store the state space they are exploring, so by relying on the output from the tools, we will still have an inaccurate result. Therefore we have chosen to use `memtime` to at least have a consistent method in measuring memory usage.

5.4 Reduction for Different DRoP Configurations

In the first experiment we try to determine which probe set selection strategy and which cycle proviso give the best performance of DRoP. To do this, we explore the state spaces of Flanagan's examples [14] described in Section 5.2, with the following four combinations of strategies.

- Reversing free probe set selection (Section 4.1.1) with the aging proviso (Definition 31).
- Reversing free probe set selection (Section 4.1.1) with the no duplicates cycle proviso (Definition 36).
- Independent action probe set selection (Section 4.1.2), with the aging proviso (Definition 31).
- Independent action probe set selection (Section 4.1.2), with the no duplicates cycle proviso (Definition 36).

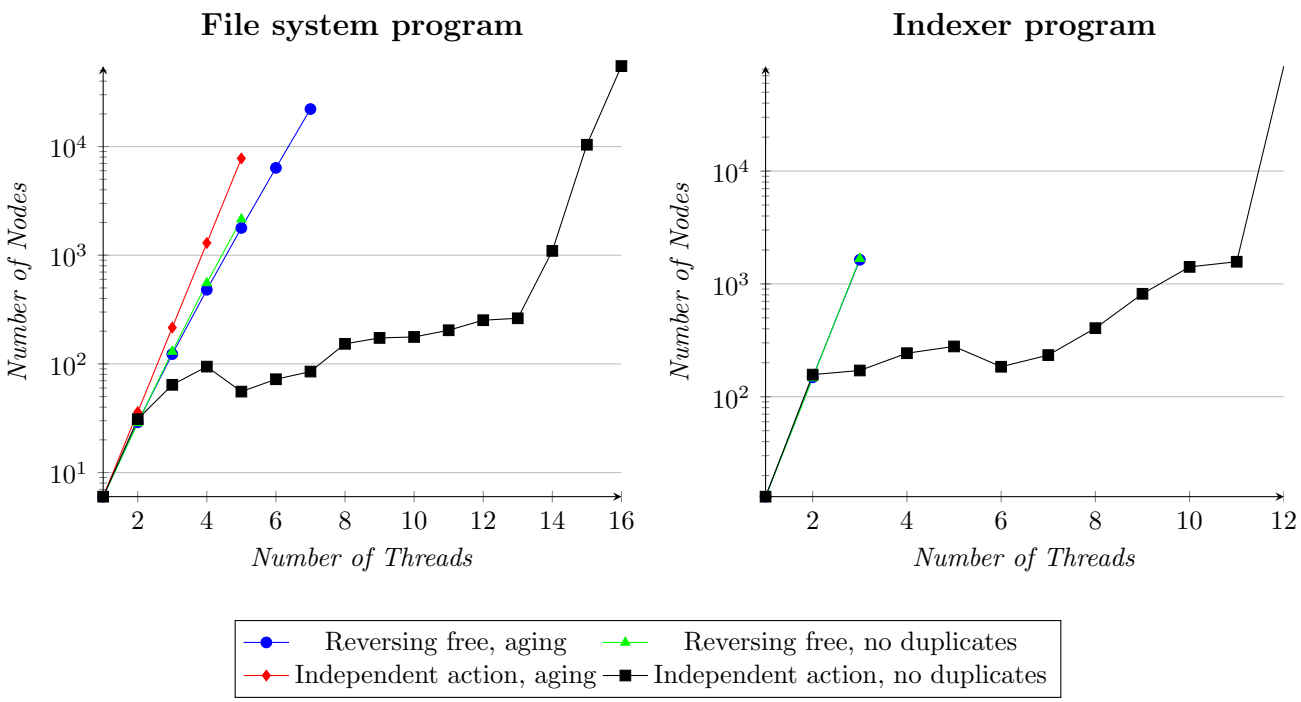


Figure 5.2: The number of nodes in Flanagan's examples [14] obtained by DRoP for different combinations probe set selection strategies and cycle provisos.

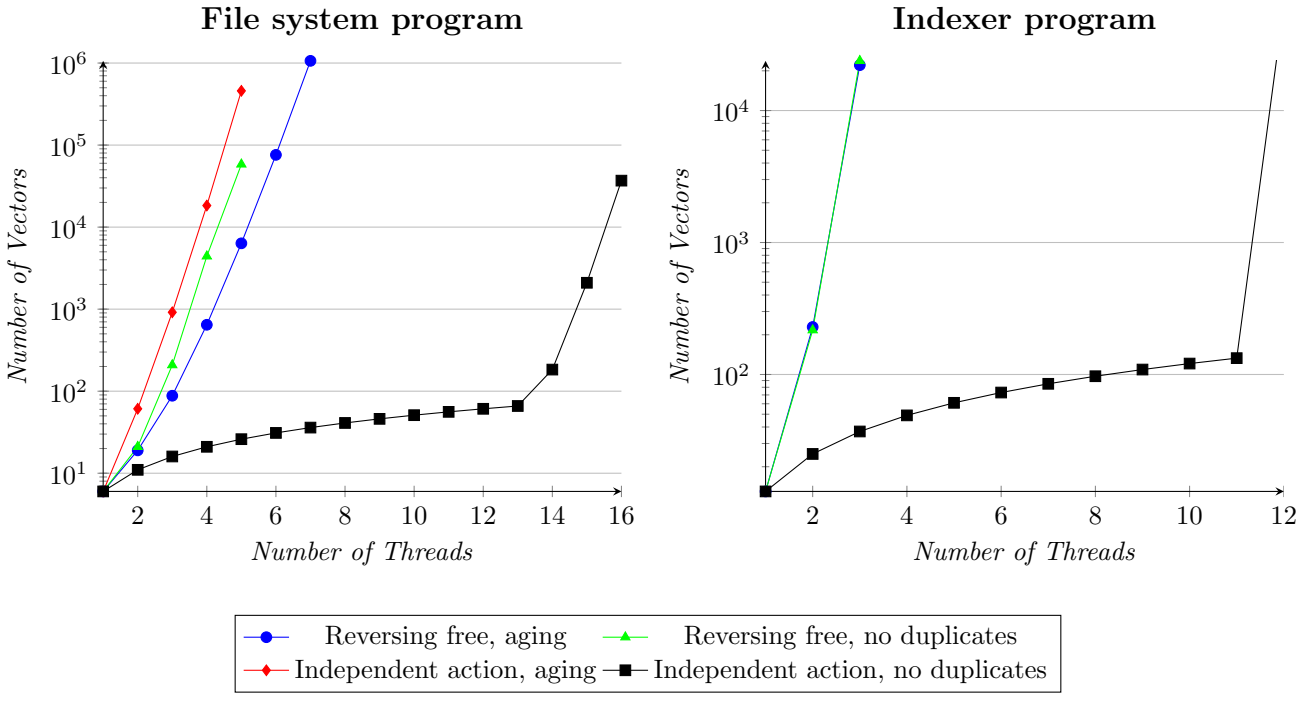


Figure 5.3: The number of vectors in Flanagan's examples [14] obtained by DRoP for different combinations probe set selection strategies and cycle provisos.

Threads	All	Rev. + age		Rev. + DRoP		Indep. + age		Indep. +DRoP	
	Nodes	Nodes	Red.	Nodes	Red.	Nodes	Red.	Nodes	Red.
1	6	6	0%	6	0%	6	0%	6	0%
2	36	29	19%	29	19%	36	0%	31	14%
3	216	123	43%	130	40%	216	0%	64	70%
4	1296	481	63%	555	57%	1296	0%	94	93%
5	7776	1783	77%	2146	72%	7776	0%	56	99%
6	46656	6369	86%	-	-	-	-	72	100%
7	279936	22151	92%	-	-	-	-	85	100%
8	-	-	-	-	-	-	-	153	-
9	-	-	-	-	-	-	-	173	-
10	-	-	-	-	-	-	-	177	-
11	-	-	-	-	-	-	-	204	-
12	-	-	-	-	-	-	-	252	-
13	-	-	-	-	-	-	-	263	-
14	-	-	-	-	-	-	-	1097	-
15	-	-	-	-	-	-	-	10395	-
16	-	-	-	-	-	-	-	55060	-

Table 5.1: Comparison of the four different exploration strategies for Flanagan's file system program.

Threads	All	Rev. + age		Rev. + DRoP		Indep. + age		Indep. +DRoP	
	Nodes	Nodes	Red.	Nodes	Red.	Nodes	Red.	Nodes	Red.
1	13	13	0%	13	0%	13	0%	13	0%
2	169	148	12%	146	14%	169	0%	157	7%
3	2197	1636	26%	1674	24%	2197	0%	171	92%
4	28561	-	-	-	-	28561	0%	243	99%
5	371293	-	-	-	-	-	-	279	100%
6	-	-	-	-	-	-	-	184	-
7	-	-	-	-	-	-	-	234	-
8	-	-	-	-	-	-	-	405	-
9	-	-	-	-	-	-	-	817	-
10	-	-	-	-	-	-	-	1416	-
11	-	-	-	-	-	-	-	1572	-

Table 5.2: Comparison of the four different exploration strategies for Flanagan's indexer program.

Figure 5.2, Table 5.1 and Table 5.2 show the number of nodes in the reduced state spaces for each strategy. Figure 5.3 shows the number of vectors needed to calculate the reduced state space for each strategy.

Note that the indexer program did not finish exploration within the set limits. Because we expect an exponential growth we added a dummy result to show this expectation in our graphs.

From both figures it can easily be concluded that the independent action selection strategy in combination with the no duplicates cycle proviso performs the best; it is the only combination for which the state space does not increase exponentially.

It can also be seen that the number of nodes is greater than the number of vectors and that the number of nodes does not necessarily increase as the model size increases and can, in fact, decrease.

Ideally, the number of vectors is an upper bound on the number of nodes in the state space. Each vector represents exactly one node, its target, and multiple vectors can represent a single node. The fact that the number of nodes is larger than the number of vectors, is because vectors do not only represent their target, but also a path, and intermediate calculations are performed on this path.

Take for example the simple state space below in Figure 5.4 with vector $(s_0, a_1 \cdot a_2 \cdot b_1)$. The probe set of the vector could be $p = b_2 \mapsto b_1$. According to Equation 2.9 the new vector with this probe set will be $(s_2, a_1 \cdot a_2 \cdot b_2)$. This new vector does not only contain a new target state, but also the new states s_2 and s_4. DRoP does not always generates these new states, but it cannot avoid generating some of them.

New states can also be calculated by the missed actions analysis. Take again vector $(s_2, a_1 \cdot a_2 \cdot b_2)$ in the same state space. The missed actions analysis requires us to consider the weak prefix b_2, and we need to explore state s_5 to determine its enabled actions. However, state s_5 does not have any missed actions, so no new vectors will be produced, even though new states were found.

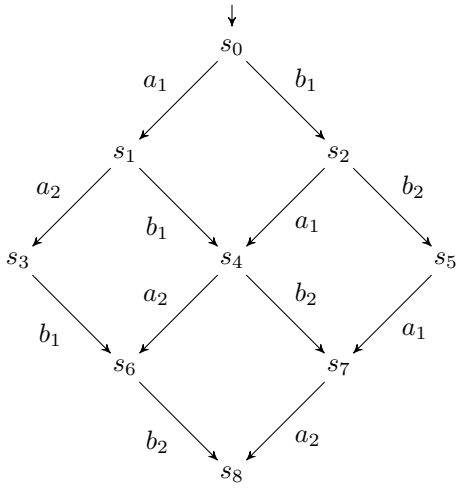


Figure 5.4: Example state space.

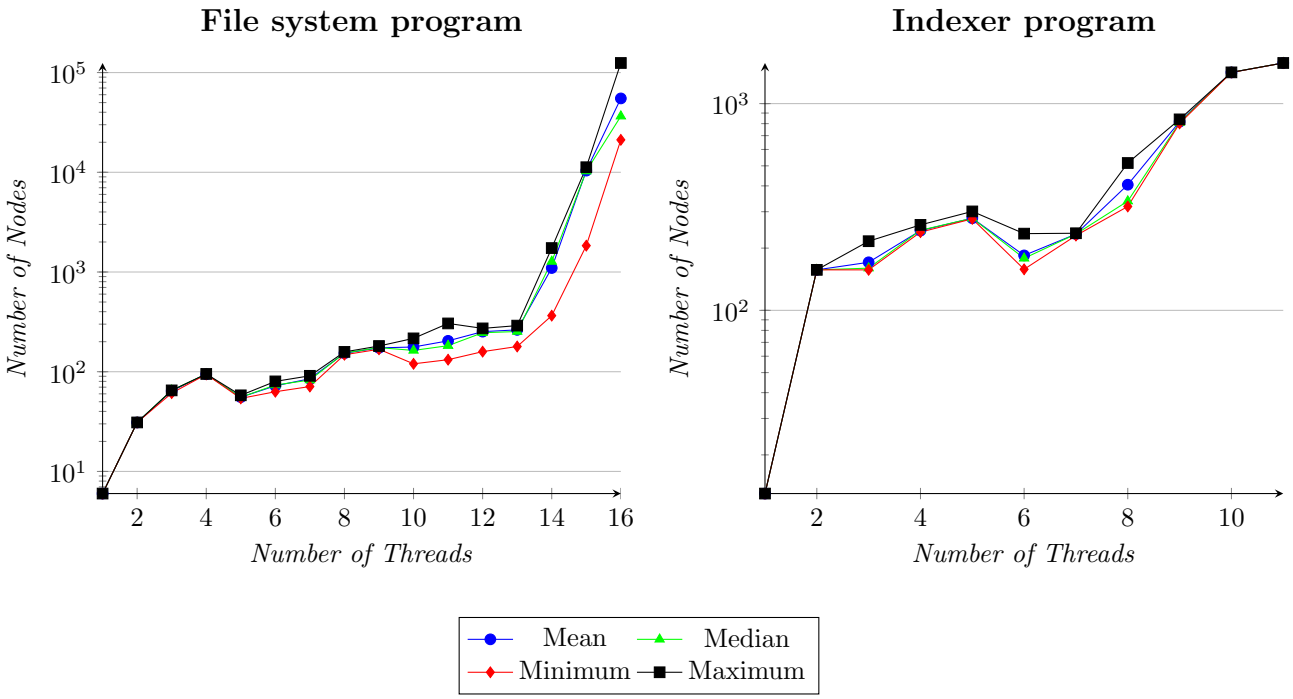


Figure 5.5: The minumum, maximum and average number of nodes in Flanagan's file system program and indexer program [14] for the independent action probe set selection strategy and the no duplicates cycle proviso

Both in the file system program and the indexer program, the number of nodes decreases sometimes, while size of the model increases. One possible explanation is selecting a 'bad' probe set, when there are multiple valid probe sets to choose from. To rule out this possibility, a random valid probe set is chosen when there are multiple options. Each test run is performed with a different random seed, so the change the number of nodes increases because bad probe sets are chosen decreases.

Figure 5.5 show some detailed results for these experiments with the independent action strategy in combination with the DRoP cycle proviso. Besides the average number of nodes, we also show the median number of nodes, and the worst and the best run.

The figure shows a difference between the best and the worst run, but it is very small. Moreover, the decrease in nodes as we saw in the average, returns in the median, maximum and minimum amount of nodes. Therefore it is unlikely that the number of nodes increases because the unlucky selection of probe sets.

It is unknown what the exact cause is for the decrease in nodes. However, since the number of vectors does not show the same decrease, the problem is likely unrelated to the exploration strategy.

5.5 Comparison between DRoP and other Tools

In the previous section we compared different options for the implementation of DRoP. In this section we will compare the best options for DRoP, the independent action probe set selection strategy with the cycle proviso in Equation 36, with already existing tools capable of partial-order reduction, namely Spin and LTSmin. We will first discuss the examples from Flanagan [14], then the examples from the BEEM database [3].

5.5.1 Flanagan's Examples

Threads	DRoP			LTSmin			Spin		
	Complete	Reduced		Complete	Reduced		Complete	Reduced	
1	6	6	0%	12	12	0%	12	12	0%
2	36	31	14%	133	111	17%	133	111	17%
3	216	64	70%	1464	1066	27%	1464	1066	27%
4	1296	94	93%	16105	10301	36%	16105	10301	36%
5	7776	56	99%	177156	99240	44%	177156	99240	44%
6	46656	72	100%	1948717	952171	51%	1948717	952171	51%
7	279936	85	100%	21435888	9100934	58%	21435888	9100934	58%
8	-	153	-	-	-	-	-	-	-
9	-	173	-	-	-	-	-	-	-
10	-	177	-	-	-	-	-	-	-
11	-	204	-	-	-	-	-	-	-
12	-	252	-	-	-	-	-	-	-
13	-	263	-	-	-	-	-	-	-
14	-	1097	-	-	-	-	-	-	-
15	-	10395	-	-	-	-	-	-	-
16	-	55060	-	-	-	-	-	-	-

Table 5.3: The number of nodes in the (reduced) state space for Flanagan's file system program for different tools.

Threads	DRoP			LTSmin			Spin		
	Complete	Reduced		Complete	Reduced		Complete	Reduced	
1	13	13	0%	28	28	0%	28	28	0%
2	169	157	7%	757	203	73%	757	211	72%
3	2197	171	92%	29080	2003	93%	29080	1972	93%
4	28561	243	99%	1309288	21653	98%	1309288	21478	98%
5	371293	279	100%	107233101	540564	99%	-	538633	-
6	-	184	-	-	3566907	-	-	3561666	-
7	-	234	-	-	54332225	-	-	-	-
8	-	405	-	-	-	-	-	-	-
9	-	817	-	-	-	-	-	-	-
10	-	1416	-	-	-	-	-	-	-
11	-	1572	-	-	-	-	-	-	-

Table 5.4: The number of nodes in the (reduced) state space for Flanagan's indexer program for different tools.

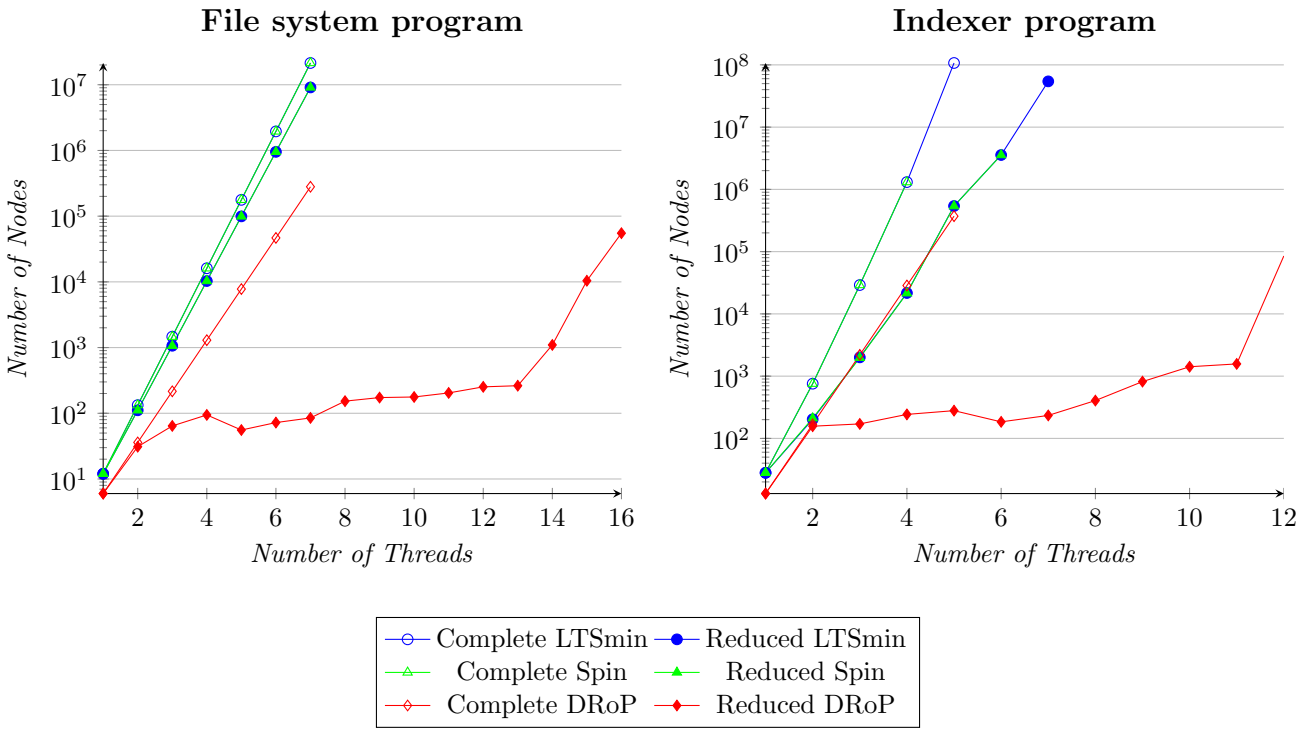


Figure 5.6: The size of the state space for Flanagan's examples for different tools.

Table 5.3 shows the size of state space of Flanagan's file system program for the different tools. Table 5.4 shows the same for the indexer program. Figure 5.6 presents the size of the state spaces for both examples in graphical form.

Note that the indexer program did not finish exploration within the set limits. Because we expect an exponential growth we added a dummy result to show this expectation in our graphs.

For both examples, DRoP performs better than LTSmin and Spin in size of the reduced state space. This is expected, because Flanagan's examples are what dynamic partial-order reduction is designed to be used for.

For the indexer program, all tools are able to achieve a large reduction. However, the size of the state space still grows exponentially for the LTSmin and Spin tools. The most plausible explanation is that because we created the models ourselves, the model for DRoP combines several dependent actions into one. As a result, LTSmin and Spin have to explore all interleavings of these dependent actions, while DRoP only has to explore a single action.

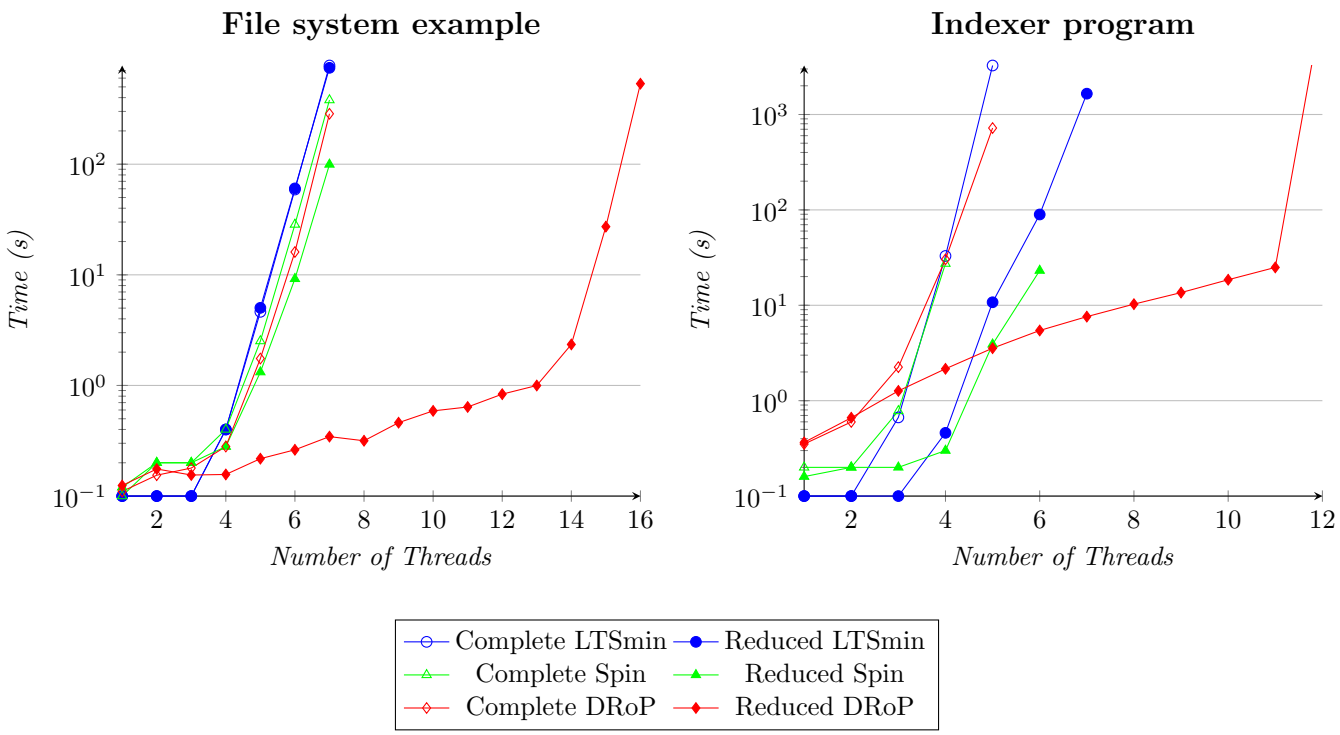


Figure 5.7: The time needed to explore the state space of Flanagan's examples for different tools.

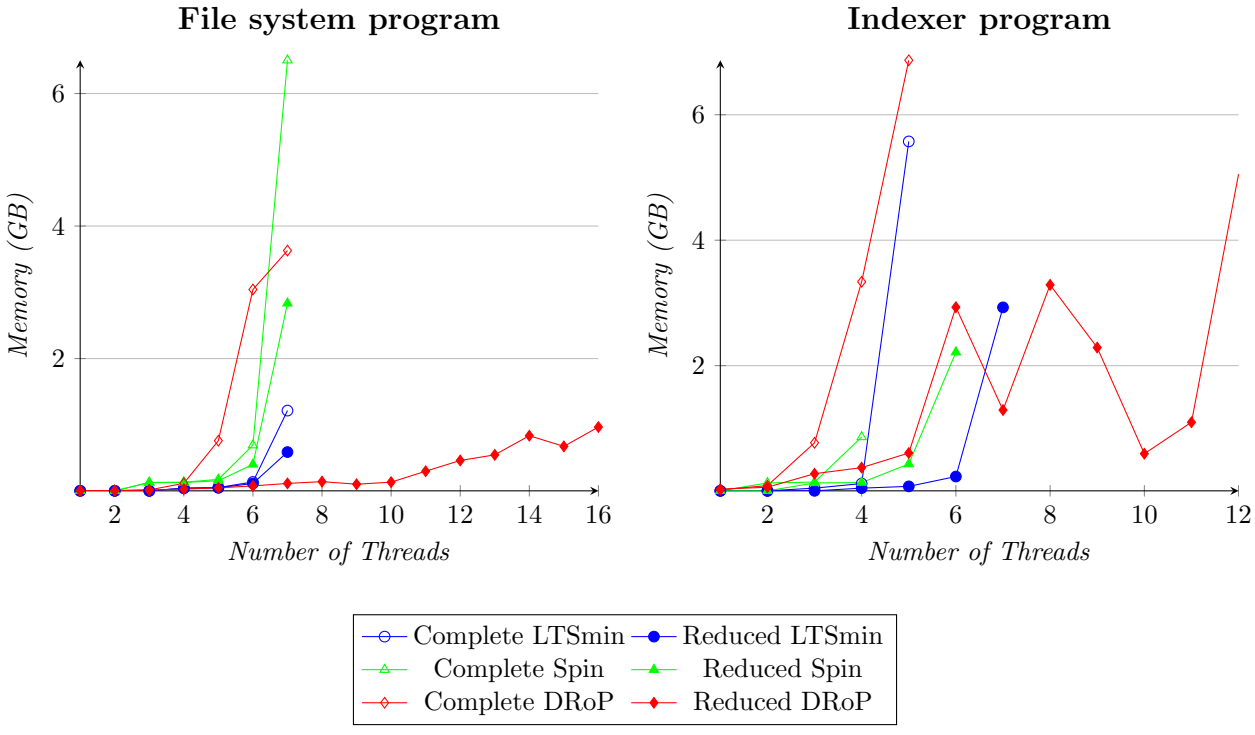


Figure 5.8: The memory needed to explore the state space of Flanagan's examples for different tools.

Figure 5.7 and Figure 5.8 show the time and memory needed to explore the state space of Flanagan's examples. The exact results can be found in the Appendix in Table A.3 and Table A.4. It should be noted that the results in these tables can be inaccurate, since the time and memory usage is measured using `memtime`.

Two small remarks can be made about these results. First, in case we are exploring the small models with at most three or four threads, DRoP takes more time and memory than existing tools. As can be seen in Table 5.3 and Table 5.4 DRoP and the other tools are only able to reduce the state space a little. This reduction is not enough to compensate the extra effort that DRoP has to spend to perform dynamic partial-order reduction.

Second, the memory usage of DRoP in the case of Flanagan's indexer program is variable. However, the memory usage remains manageable in contrast to the other tools, whose memory usage increases exponentially. Therefore we are not concerned with this for now.

5.5.2 BEEM Examples

Threads	DRoP		LTSmin	Spin	Helena
	DRoP	Helena			
p2m2	1%	10%	N/A%	N/A%	0%
p2m3	19%	18%	N/A%	N/A%	0%
p2m4	43%	20%	N/A%	N/A%	0%
p2m5	36%	18%	N/A%	N/A%	0%
p2m6	7%	15%	N/A%	N/A%	0%
p2m7	18%	28%	N/A%	N/A%	0%
p2m8	28%	19%	N/A%	N/A%	0%
p2m9	14%	N/A%	12%	0%	0%
p3m2	28%	9%	N/A%	N/A%	0%
p3m3	20%	10%	N/A%	N/A%	0%
p3m4	18%	-	4%	0%	0%
p3m5	17%	N/A%	N/A%	N/A%	0%
p3m6	17%	N/A%	N/A%	N/A%	0%
p3m7	19%	N/A%	N/A%	N/A%	0%
p4m2	42%	-	N/A%	N/A%	0%
p4m7	-	N/A%	3%	0%	-
p4m12	-	N/A%	3%	0%	-
p5m5	-	N/A%	-	0%	-

Table 5.5: Amount of reduction achieved by different tools for the bakery example. Extract from Table A.1 ("N/A" = model not available, "-" = model could not be explored).

Threads	DRoP		LTSmin	Spin	Helena
	DRoP	Helena			
1	0%	0%	35%	1%	0%
2	0%	36%	N/A%	N/A%	0%
3	8%	4%	51%	-2%	1%
4	13%	-	N/A%	N/A%	3%
5	23%	-	64%	12%	7%

Table 5.6: Amount of reduction achieved by different tools for the Szymanski's algorithm. Extract from Table A.2 ("N/A" = model not available, "-" = model could not be explored).

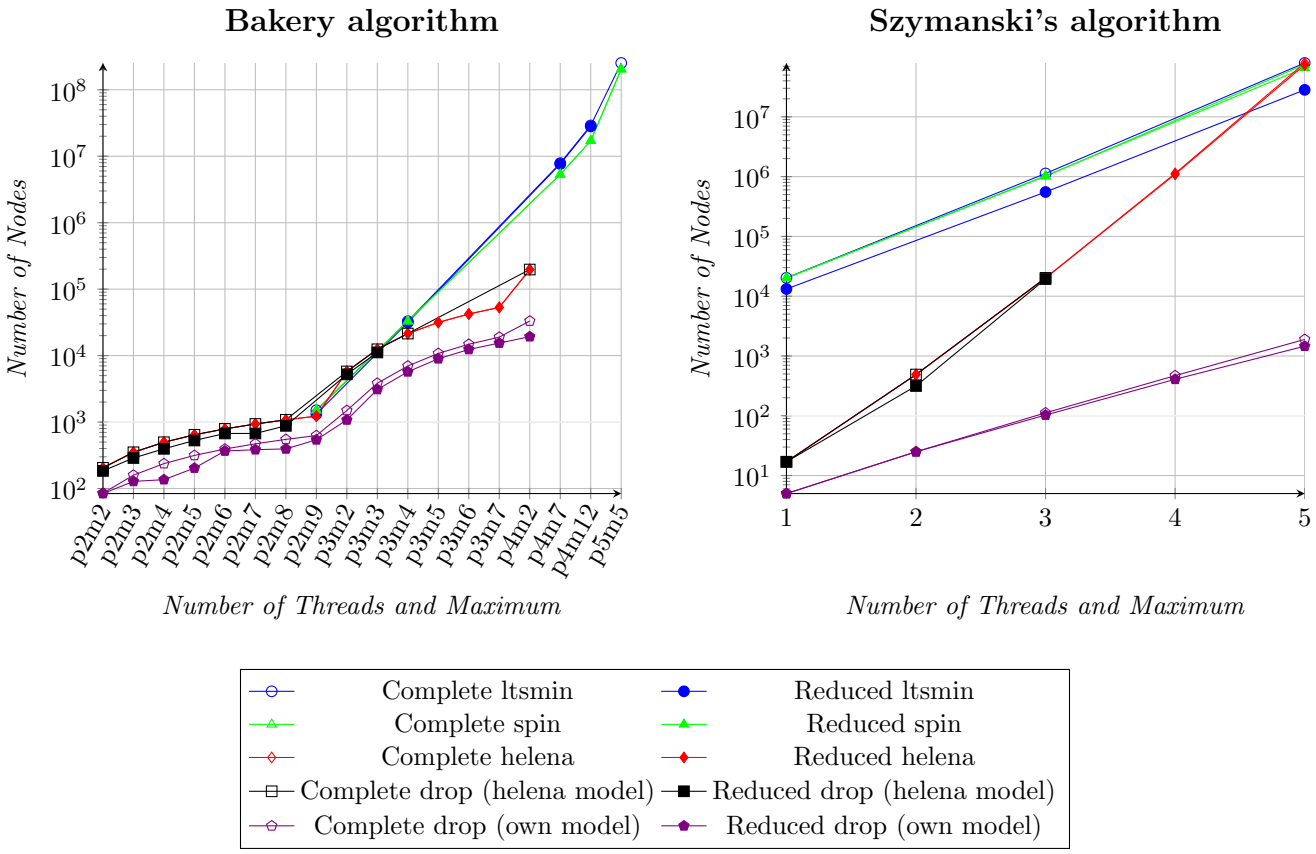


Figure 5.9: Number of nodes for the BEEM models

Table 5.5 and Table 5.6 show how much reduction each tool is able to achieve for the bakery example and Szymanski's algorithm respectively. Table A.1 and Table A.2 in de Appendix also include the number of nodes for the complete and reduced explorations. In Figure 5.9, the number of nodes in the state spaces is graphically represented.

For the bakery example, we can see that DRoP achieves the same amount of reduction as the other tools, and sometimes it achieves more reduction.

For Szymanski's algorithm, DRoP performs better than Spin and Helena. However, LTSmin is able to reduce much more than DRoP. A possible explanation is that the complete state spaces of the models used by DRoP already contain much fewer states, as can be seen in Table A.2. Therefore they probably contain fewer independent actions and it is impossible to reduce these models much further.

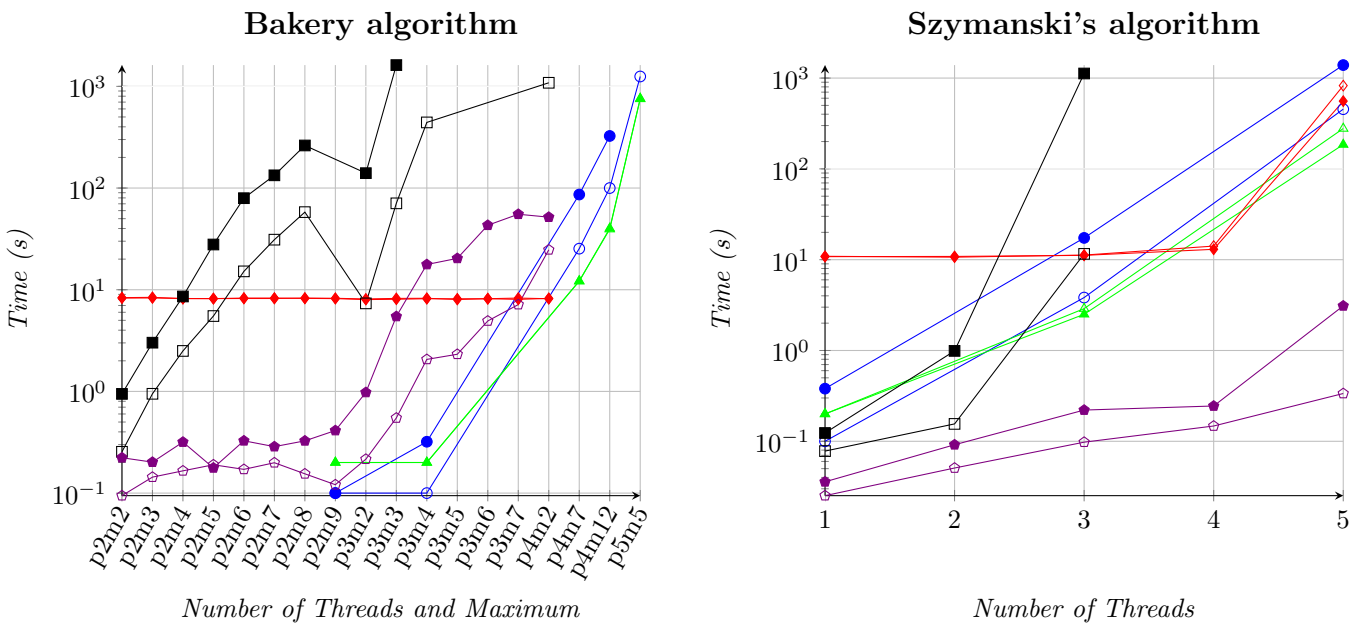
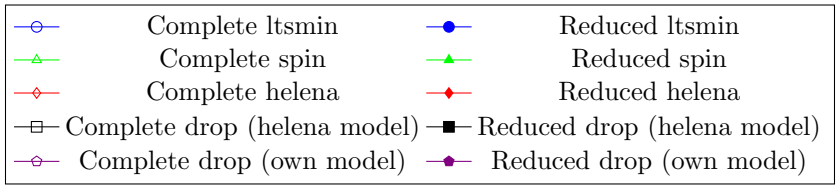


Figure 5.10: Time needed to explore the BEEM models



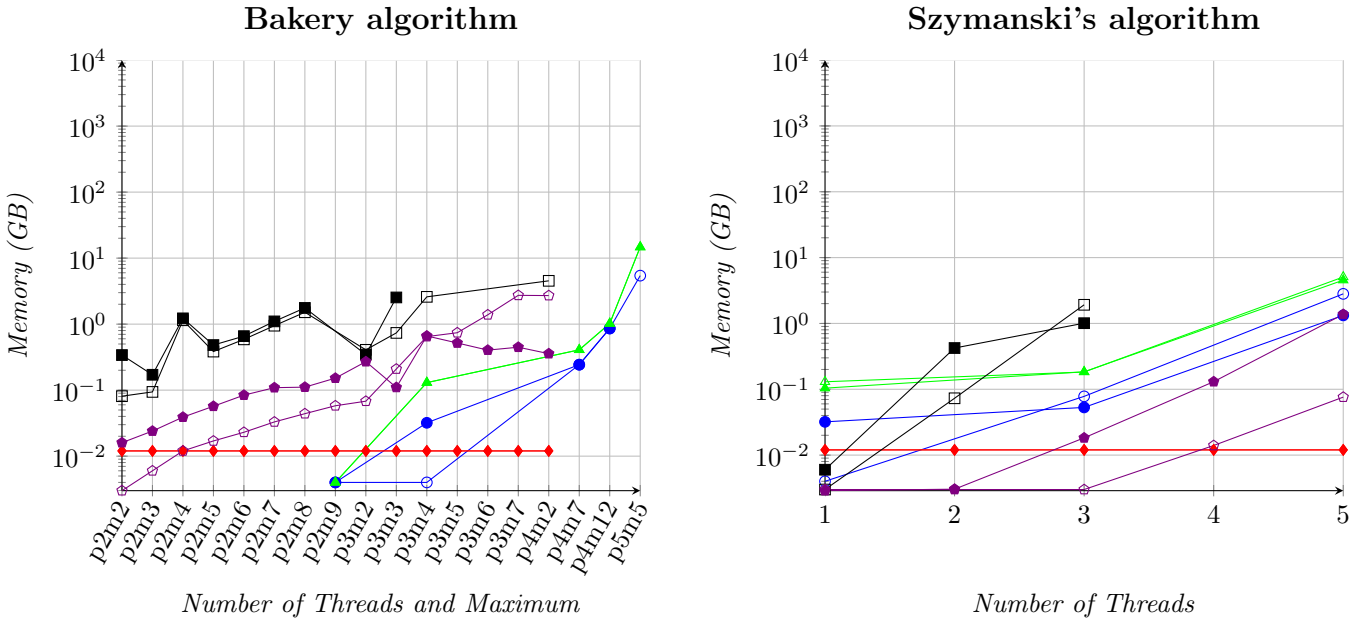


Figure 5.11: Memory needed tot explore the BEEM models

Figure 5.10 and Figure 5.11 show the amount of time and memory needed to explore the state spaces of the BEEM models. The exact results can be found in the Appendix in Table A.5 and Table A.6. It should be noted that the results in these tables can be inaccurate, since the time and memory usage is measured using `memtime`.

For the bakery example, DRoP behaves as expected. The amount of reduction in the bakery model is small, and dynamic partial-order reduction is more complicated. Therefore, we see that DRoP requires more time and memory to explore the models. Although the difference is small, it is probably still the cause that it is impossible to explore the larger models with DRoP, while this is possible with Spin and LTSmin.

When comparing the amount of time and memory needed by DRoP to explore the state space of Szymanski's algorithm, we see that it is much less when compared to the other tools. The state spaces of the models explored by DRoP are also much smaller, so this is as expected. From the graph in Figure 5.10 we can see that the slope of the time needed by DRoP is lower compared with the other tools, and in Figure 5.11 we can see that the slope for the memory usage by DRoP is comparable to the other tools. Therefore we assume that the time and memory needed by DRoP will only differ with a constant factor, when we would increase the size of the model.

For both BEEM models, DRoP is unable to explore the larger models that are exported from the Helena tool. Probably this is caused by the format of the model. For example, the model for the bakery algorithm with three threads and a maximum ticket number of four that we created is an XML document of 160kB and the same model exported from Helena is an XML document of 179MB. So while both models describe the same algorithm, the model exported from Helena is much larger. Therefore it is to be expected that the exploration of the model exported from Helena requires more time and memory.

5.6 Summary

In the previous sections we have seen several experiments. This section provides a summary of their results.

In our first experiments in Section 5.4 we compared different probe set selection strategies and cycle provisos for DRoP. During these experiments, the combination of the independent action probe set selection strategy (Section 4.1.2) and the no duplicates cycle proviso (Section 3.1) clearly provided the largest reductions.

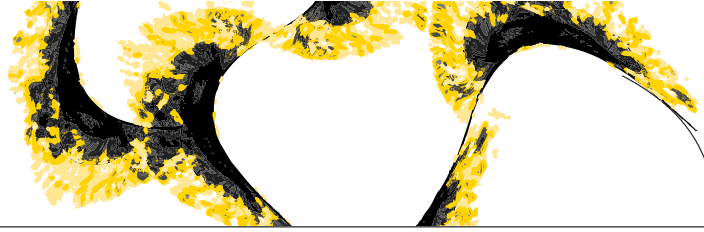
In the same experiments, we discovered that the number of states is larger than the number of vectors, while the number of vectors should be an upper bound on the number of states. Although a vector only represents a single state, its target state, it also contains a path that can be used to reach this state. Changes in these paths also result in new states being discovered.

In Section 5.5 we compared DRoP with other tools. For the models that we expected to only work with dynamic partial-order reduction, DRoP performed better than most tools. This means DRoP achieved a larger reduction in less time and memory than other tools. It should be noted that the models do not have an equal amount of states for every tool. However, for Flanagan's examples the difference in number of states alone is to small to account for the difference in time and memory.

For Flanagan's indexer program, some of the other tools were also able to achieve a large reduction, however their state spaces would still grow exponentially. This is probably caused by differences in the input models.

The last experiments showed that static and dynamic partial-order reduction perform equally, when static partial-order reduction is already able to perform well. This means that the reduction both algorithms are able to achieve is equal.

As expected DRoP needs more time and memory to explore the models, because the dynamic partial-order reduction algorithm is more complicated. However, we should take into account that the number of states in the models was not equal for every tool, especially in Szymanski's mutual exclusion algorithm. Therefore, the results the results for time and memory usage are not as reliable as in the previous experiments.



6 Conclusions

In this master thesis we introduced partial-order reduction and the limitations of static partial-order reduction when we want to apply this on modern computer programs that dynamically create and destroy threads and objects in memory. To counter these limitations we discussed dynamic partial-order reduction, which attempts to solve the limitations of static partial-order reduction by dynamically creating the important dependency function during the exploration of the state space instead of statically determining this dependency relation in advance.

We also wondered if the technically superior but more complicated dynamic partial-order reduction algorithm would be able to perform as well in practise as existing static partial-order reduction algorithm. To verify this we tried to answer the following questions:

1. Which cycle proviso should we use?
2. How do we determine the over-approximation of the missed actions?
3. How do we select probe sets?
4. What will be the input for the implementation?
5. How do we evaluate the efficiency of the algorithm?

Cycle Proviso In Section 3.1 we introduced an alternative cycle proviso to the one introduced in [17]. The cycle proviso is used to select a new probe set together with the selection strategy when the algorithm detects a cycle during the exploration. The cycle proviso introduced in the original algorithm [17] was based on exploring every action as soon as possible, and did not allow the exploration of actions to be postponed. With the new cycle proviso the exploration of actions can be postponed without weakening the guarantees of the dynamic-partial-order reduction algorithm. During our experiments we compared both cycle provisos and showed that by using the new proviso we could gain better reductions.

Over-approximation of the Missed Actions We made a small improvement to the over-approximation of the potentially missed actions as described in Section 3.2. Some missed actions in the over-approximation, we could easily detect during the generation of the probe sets. By changing the requirements on valid probe sets, we can avoid a part of the expensive missed action analysis.

Probe Set Selection To be able to efficiently apply the dynamic partial-order reduction algorithm, some parameters of the algorithm needed to be tweaked. One important thing we did was create a probe set selection strategy. The paper that introduces partial-order reduction on probe sets [17] suggests an selection strategy based on reversing actions that allows for optimizations in the missed action analysis of the algorithm at the cost of a more complicated probe set phase. However, our experiments show that the reductions we gain by using this strategy could be improved. As an alternative we suggested a selection strategy based on independent actions that allows for an efficient probe phase at the cost of a less efficient missed actions phase. We implemented both strategies in DRoP and compared them in our experiments. In our experiments, our strategy based on independent actions out-performed the selection strategy based on reversing actions.

Input As input for DRoP we decided to use Petri nets. To support this we first created a transformation from Petri nets to the entity based systems used by the dynamic partial-order reduction algorithm. In this transformation we chose a non-standard approach to Petri nets by using an explicit interpretation of the tokens, instead of a symbolic one. Entity-based systems also have a concrete interpretation, so this choice made the transformation layer simpler. It also makes it easier to extend the implementation to support another formalism that holds our interest: graph transformation systems, since graph transformation systems also have a concrete interpretation. The transformation layer only assumes that entities can be represented by a type and an identifier. Every formalism that can be expressed in this form, can easily be supported by DRoP by simply adding a new transformation class to the implementation.

Evaluation of Efficiency Our research was not only focused on improving the efficiency of the dynamic-partial-order reduction algorithm; we also wanted to show that the algorithm performs better or just as good as static-partial-order reduction. To verify this, we did not only run experiments with DRoP, but also with existing tools capable of partial-order reduction. In our experiments we differentiated between two types of experiments. Experiments with models for which dynamic partial-order reduction was designed, and where static partial-order reduction would not be able to gain (much) reduction. And experiments in which we expected dynamic and static partial-order reduction to be able to achieve the same amount of reduction.

In the experiments where we used models that work well with dynamic partial-order reduction, DRoP achieves *greater reductions* in *less time* and with *less memory* than existing partial-order reduction tools.

When we use models for which static and dynamic partial-order reduction are on equal ground, we achieve *equal reduction* and only a little extra time and memory are necessary. It is impossible to achieve equal reduction with the same amount of time and memory as a static partial-order reduction algorithm, simply because the dynamic partial-order reduction algorithm is more complicated and requires more overhead.

Alternative Dynamic Partial-order Reduction Algorithm Besides the probe set algorithm, another dynamic partial-order algorithm is introduced by Flanagan in [14]. This algorithm has been implemented and experimented with, and it has been shown that it is able to produce greater reductions than static partial-order reduction. However, the algorithm in [14] is not able to deal with cyclic state spaces. By using techniques to shorten vectors and by including the cycle proviso, the probe set based algorithm is able to deal with cyclic state spaces. Our experiments also confirm this property, so the probe set based algorithm is an improvement over the algorithm introduced in [14].

Conclusion By developing DRoP we have successfully shown that it possible to apply dynamic partial-order reduction and gain greater reductions without using much more resources. We have also shown that our dynamic partial-order reduction algorithm is able to deal with cyclic state spaces.

6.1 Future Work

In this master thesis, we wanted to show that dynamic partial-order reduction is feasible. DRoP did an excellent job on providing this proof, but this does not mean that the tool is finished. We think that more work is necessary in the following subjects:

- Vector representation.
- Probe set selection strategy and cycle proviso.
- Experimentation.
- Code optimization.

Besides the important subjects above, the following subject is also very interesting and fun to investigate, but is maybe less urgent:

- Preserve more properties in the reduced state space (e.g. LTL and CTL).

Vector Representation

Vectors consist of a start state and a trace, which represent a single target state, namely the state that we reach when we execute the trace in the start state. So, the number of vectors should be an upper bound on the number of nodes in the state space. However, as discussed at the end of Section 5.4, this is not the case.

Each vector does represent a single target state, but the vector also stores the path to this state in its trace. Vectors that represent successive target states do not necessarily have overlapping traces, and therefore more states are visited when we calculate the path that vectors represent.

DRoP already calculates targets for new vectors from the target of old vectors in an attempt to minimize the number of states visited because of the above problem. However, extra states are still needed to calculate the start states of vectors and the missed actions.

More time should be spent on how vectors are represented and calculated to minimize the amount of states that we need in our calculations but that we do not want to explore. Besides improving the handling of vectors, we could also select different probe sets to generate more efficient vectors.

Probe Set Selection Strategy and Cycle Proviso

In this thesis we introduce a new strategy to select probe sets and a new cycle proviso. The basis of this new strategy and proviso is empirical: by applying the algorithm by hand on some small examples, the new strategy and proviso seemed to work well.

While the strategy and proviso still work well in the experiments with larger models, we do not have a good mathematical analysis of why this strategy and proviso results in a good exploration. Making such an analysis may yield new insights on why they work, and these insights then may allow us to improve the strategy and proviso.

Another disadvantage of the current strategy is that is contains no heuristics on how to select a good probe set, when multiple valid probe sets can be selected. Our experiments show that the number of states in the reduced state space does depend on what probe sets are selected. Therefore it is important that some strategy is developed to determine which probe sets are the best.

Finally it should be noted that the probe set selection strategy does directly determine which vectors are created. Therefore a new smart strategy might be able to (partially) solve the problems with the vector representation described above.

Experiments

We have presented several experiments in this thesis that verify that partial-order reduction based on probe sets works. We have also been able to draw some interesting conclusions and we know where our future effort should be pointed. However, to firmly establish that DRoP is also a good implementation of the dynamic partial-order reduction algorithm, it should be subjected to more experiments with different models.

Code Optimization

DRoP has been created to be an efficient implementation of the probe set algorithm. However, we believe it is still possible to improve the code further and to decrease the time and memory DRoP needs to reduce state spaces.

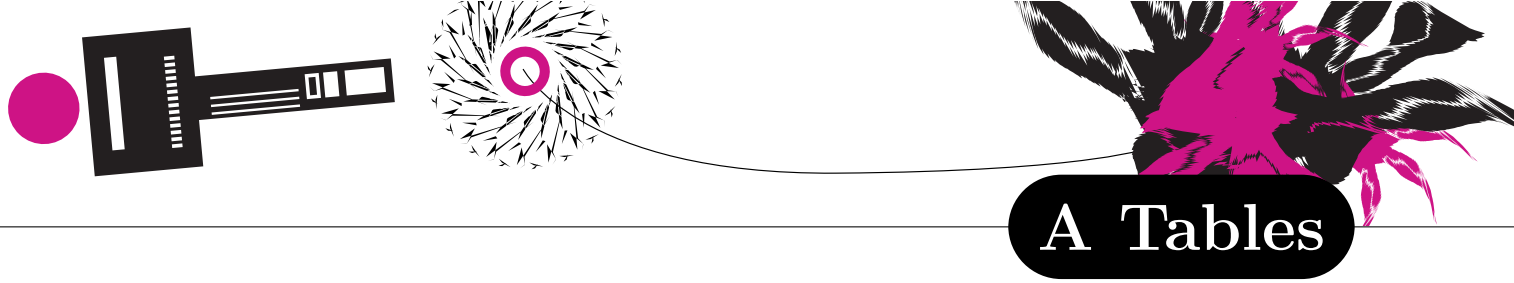
Properties

The current version of the algorithm only preserves deadlocks in the reduced state space. Static partial-order reduction algorithms, such as the persistent set approach [26,27] and the ample set approach [21,22], are also able to preserver safety properties, LTL properties and CTL* properties.

Our cycle proviso preserves the same properties as the cycle proviso for static partial-order reduction. This includes invariants, which constitute a large part of the safety properties.

It should be easy to extend the algorithm to preserve all safety, LTL and CTL* properties. To preserve LTL properties we need the notion of stuttering actions. Static partial-order reduction explores either only stuttering enabled actions, or all enabled actions in every state. This constraint does not conflict in any way with the constraints currently in place on probe sets. Therefore it should be straightforward to modify this constraint to enable dynamic partial-order reduction to preserve LTL properties.

To preserve CTL* properties, static partial-order reduction either explores a single enabled action or all enabled actions in every state. This is something we have already incorporated in our probe set selection strategy and cycle proviso, so it should be straightforward to enable dynamic partial-order reduction to preserve CTL* properties.



A Tables

This appendix contains some of the tables with the result of the experiments in Chapter 5.

A.1 State Space Sizes

Threads	DRoP (DRoP model)			DRoP (Helena model)			LTSmin		
	Comp.	Reduced		Comp.	Reduced		Comp.	Reduced	
p2m2	85	84	1%	205	184	10%	N/A	N/A	N/A%
p2m3	159	128	19%	351	288	18%	N/A	N/A	N/A%
p2m4	237	136	43%	497	396	20%	N/A	N/A	N/A%
p2m5	315	203	36%	643	528	18%	N/A	N/A	N/A%
p2m6	393	367	7%	789	674	15%	N/A	N/A	N/A%
p2m7	471	384	18%	935	674	28%	N/A	N/A	N/A%
p2m8	549	395	28%	1081	878	19%	N/A	N/A	N/A%
p2m9	627	541	14%	N/A	N/A	N/A%	1506	1320	12%
p3m2	1494	1077	28%	5773	5226	9%	N/A	N/A	N/A%
p3m3	3843	3091	20%	12454	11164	10%	N/A	N/A	N/A%
p3m4	6978	5707	18%	21466	-	-	32919	31445	4%
p3m5	10796	8975	17%	N/A	N/A	N/A%	N/A	N/A	N/A%
p3m6	14833	12365	17%	N/A	N/A	N/A%	N/A	N/A	N/A%
p3m7	18983	15460	19%	N/A	N/A	N/A%	N/A	N/A	N/A%
p4m2	33153	19279	42%	197364	-	-	N/A	N/A	N/A%
p4m7	-	-	-	N/A	N/A	N/A%	7866401	7644120	3%
p4m12	-	-	-	N/A	N/A	N/A%	29047471	28118640	3%
p5m5	-	-	-	N/A	N/A	N/A%	253131202	-	-

Threads	Spin			Helena		
	Comp.	Reduced		Comp.	Reduced	
p2m2	N/A	N/A	N/A%	205	205	0%
p2m3	N/A	N/A	N/A%	351	351	0%
p2m4	N/A	N/A	N/A%	497	497	0%
p2m5	N/A	N/A	N/A%	643	643	0%
p2m6	N/A	N/A	N/A%	789	789	0%
p2m7	N/A	N/A	N/A%	935	935	0%
p2m8	N/A	N/A	N/A%	1081	1081	0%
p2m9	1506	1506	0%	1227	1227	0%
p3m2	N/A	N/A	N/A%	5773	5773	0%
p3m3	N/A	N/A	N/A%	12454	12454	0%
p3m4	32919	32919	0%	21466	21466	0%
p3m5	N/A	N/A	N/A%	31615	31615	0%
p3m6	N/A	N/A	N/A%	42282	42282	0%
p3m7	N/A	N/A	N/A%	53141	53141	0%
p4m2	N/A	N/A	N/A%	197364	197364	0%
p4m7	5325075	5325075	0%	-	-	-
p4m12	17080914	17080914	0%	-	-	-
p5m5	203817680	203817680	0%	-	-	-

Table A.1: The size of the state space of the bakery example for different tools and model configurations ("N/A" = model not available, "-" = model could not be explored).

Threads	DRoP (DRoP model)			DRoP (Helena model)			LTSmin		
	Comp.	Reduced		Comp.	Reduced		Comp.	Reduced	
1	5	5	0%	17	17	0%	20264	13190	35%
2	25	25	0%	489	315	36%	N/A	N/A	N/A%
3	112	103	8%	20264	19379	4%	1128424	552572	51%
4	469	407	13%	-	-	-	N/A	N/A	N/A%
5	1894	1458	23%	-	-	-	79518740	28251303	64%

Threads	Spin			Helena		
	Comp.	Reduced		Comp.	Reduced	
1	20264	20098	1%	17	17	0%
2	N/A	N/A	N/A%	489	488	0%
3	1013455	1031694	-2%	20264	20079	1%
4	N/A	N/A	N/A%	1128424	1092638	3%
5	74842972	65930161	12%	79518740	73878755	7%

Table A.2: The size of the state space of Szymanski's algorithm for different tools and model configurations ("N/A" = model not available, "-" = model could not be explored).

A.2 Time and Memory Requirements

Threads	DRoP				LTSmin				Spin			
	Time		Memory (GB)		Time		Memory (GB)		Time		Memory (GB)	
	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.
1	0.0	0.0	0.005	0.005	0.1	0.1	0.004	0.004	0.1	0.1	0.004	0.004
2	0.0	0.0	0.006	0.009	0.1	0.1	0.004	0.004	0.2	0.2	0.004	0.004
3	0.1	0.1	0.018	0.016	0.1	0.1	0.004	0.004	0.2	0.2	0.130	0.130
4	0.2	0.1	0.117	0.030	0.4	0.4	0.044	0.044	0.4	0.3	0.134	0.130
5	1.6	0.1	0.761	0.049	4.6	5.0	0.053	0.050	2.5	1.3	0.177	0.156
6	15.7	0.2	3.042	0.078	59.0	60.5	0.138	0.115	28.5	9.2	0.693	0.403
7	284.0	0.2	3.630	0.116	780.5	743.8	1.215	0.589	380.1	99.5	6.502	2.834
8	-	0.2	-	0.143	-	-	-	-	-	-	-	-
9	-	0.3	-	0.104	-	-	-	-	-	-	-	-
10	-	0.5	-	0.135	-	-	-	-	-	-	-	-
11	-	0.5	-	0.300	-	-	-	-	-	-	-	-
12	-	0.7	-	0.462	-	-	-	-	-	-	-	-
13	-	0.9	-	0.547	-	-	-	-	-	-	-	-
14	-	2.2	-	0.836	-	-	-	-	-	-	-	-
15	-	27.1	-	0.675	-	-	-	-	-	-	-	-
16	-	534.0	-	0.967	-	-	-	-	-	-	-	-

Table A.3: The time and memory required by different tools to explore Flanagan's file system program.

Threads	DRoP				LTSmin				Spin			
	Time		Memory (GB)		Time		Memory (GB)		Time		Memory (GB)	
	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.
1	0.0	0.1	0.022	0.030	0.1	0.1	0.004	0.004	0.2	0.2	0.004	0.004
2	0.1	0.2	0.087	0.062	0.1	0.1	0.004	0.004	0.2	0.2	0.130	0.004
3	1.6	0.6	0.770	0.277	0.7	0.1	0.047	0.004	0.8	0.2	0.144	0.130
4	29.0	1.3	3.337	0.374	32.9	0.5	0.120	0.047	27.4	0.3	0.862	0.136
5	717.9	2.5	6.869	0.607	3263.6	10.8	5.575	0.075	-	3.9	-	0.432
6	-	4.1	-	2.932	-	89.5	-	0.232	-	23.1	-	2.214
7	-	6.1	-	1.292	-	1653.2	-	2.929	-	-	-	-
8	-	8.6	-	3.288	-	-	-	-	-	-	-	-
9	-	11.7	-	2.288	-	-	-	-	-	-	-	-
10	-	16.6	-	0.596	-	-	-	-	-	-	-	-
11	-	22.6	-	1.097	-	-	-	-	-	-	-	-

Table A.4: The time and memory required by different tools to explore Flanagan's indexer program.

Threads	DRoP (DRoP model				DRoP (Helena model)				LTSmin			
	Time		Memory (GB)		Time		Memory (GB)		Time		Memory (GB)	
	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.
p2m2	0.1	0.2	0.003	0.016	0.3	0.9	0.081	0.340	N/A	N/A	N/A	N/A
p2m3	0.1	0.2	0.006	0.024	0.9	3.0	0.094	0.171	N/A	N/A	N/A	N/A
p2m4	0.2	0.3	0.012	0.039	2.5	8.6	1.145	1.224	N/A	N/A	N/A	N/A
p2m5	0.2	0.2	0.017	0.057	5.5	27.8	0.384	0.482	N/A	N/A	N/A	N/A
p2m6	0.2	0.3	0.023	0.084	15.2	79.5	0.588	0.658	N/A	N/A	N/A	N/A
p2m7	0.2	0.3	0.033	0.109	31.1	133.4	0.947	1.101	N/A	N/A	N/A	N/A
p2m8	0.2	0.3	0.044	0.111	58.0	262.2	1.514	1.760	N/A	N/A	N/A	N/A
p2m9	0.1	0.4	0.058	0.152	N/A	N/A	N/A	N/A	0.1	0.1	0.004	0.004
p3m2	0.2	1.0	0.068	0.270	7.3	140.1	0.408	0.346	N/A	N/A	N/A	N/A
p3m3	0.5	5.5	0.209	0.110	70.7	1619.0	0.736	2.537	N/A	N/A	N/A	N/A
p3m4	2.1	17.7	0.647	0.662	442.0	-	2.591	-	0.1	0.3	0.004	0.032
p3m5	2.3	20.3	0.741	0.518	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
p3m6	4.9	43.1	1.385	0.404	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
p3m7	7.1	55.3	2.730	0.447	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
p4m2	24.7	51.7	2.707	0.357	1084.3	-	4.529	-	N/A	N/A	N/A	N/A
p4m7	-	-	-	-	N/A	N/A	N/A	N/A	25.4	86.4	0.243	0.243
p4m12	-	-	-	-	N/A	N/A	N/A	N/A	100.0	325.3	0.859	0.860
p5m5	-	-	-	-	N/A	N/A	N/A	N/A	1250.3	-	5.427	-

Threads	Spin				Helena			
	Time		Memory (GB)		Time		Memory (GB)	
	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.
p2m2	N/A	N/A	N/A	N/A	8.4	8.3	0.012	0.012
p2m3	N/A	N/A	N/A	N/A	8.4	8.3	0.012	0.012
p2m4	N/A	N/A	N/A	N/A	8.2	8.2	0.012	0.012
p2m5	N/A	N/A	N/A	N/A	8.2	8.2	0.012	0.012
p2m6	N/A	N/A	N/A	N/A	8.2	8.3	0.012	0.012
p2m7	N/A	N/A	N/A	N/A	8.2	8.3	0.012	0.012
p2m8	N/A	N/A	N/A	N/A	8.3	8.2	0.012	0.012
p2m9	0.2	0.2	0.004	0.004	8.2	8.2	0.012	0.012
p3m2	N/A	N/A	N/A	N/A	8.1	8.0	0.012	0.012
p3m3	N/A	N/A	N/A	N/A	8.2	8.1	0.012	0.012
p3m4	0.2	0.2	0.130	0.130	8.2	8.2	0.012	0.012
p3m5	N/A	N/A	N/A	N/A	8.0	8.1	0.012	0.012
p3m6	N/A	N/A	N/A	N/A	8.2	8.2	0.012	0.012
p3m7	N/A	N/A	N/A	N/A	8.3	8.0	0.012	0.012
p4m2	N/A	N/A	N/A	N/A	8.2	8.2	0.012	0.012
p4m7	12.2	12.2	0.408	0.408	-	-	-	-
p4m12	39.4	40.1	1.021	1.021	-	-	-	-
p5m5	759.5	752.2	14.703	14.703	-	-	-	-

Table A.5: The time and memory required by different tools to explore Lamport's bakery algorithm ("N/A" = model not available, "-" = model could not be explored).

Threads	DRoP (DRoP model				DRoP (Helena model)				LTSmin			
	Time		Memory (GB)		Time		Memory (GB)		Time		Memory (GB)	
	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.
1	0.0	0.0	0.003	0.003	0.1	0.1	0.003	0.006	0.1	0.4	0.004	0.032
2	0.1	0.1	0.003	0.003	0.2	1.0	0.073	0.423	N/A	N/A	N/A	N/A
3	0.1	0.2	0.003	0.018	11.6	1123.2	1.919	1.009	3.8	17.4	0.078	0.053
4	0.1	0.2	0.014	0.130	-	-	-	-	N/A	N/A	N/A	N/A
5	0.3	3.1	0.076	1.374	-	-	-	-	454.4	1390.3	2.814	1.325

Threads	Spin				Helena			
	Time		Memory (GB)		Time		Memory (GB)	
	Comp.	Red.	Comp.	Red.	Comp.	Red.	Comp.	Red.
1	0.2	0.2	0.130	0.104	10.9	10.8	0.012	0.012
2	N/A	N/A	N/A	N/A	10.7	10.9	0.012	0.012
3	2.9	2.5	0.183	0.184	11.2	11.2	0.012	0.012
4	N/A	N/A	N/A	N/A	14.1	13.0	0.012	0.012
5	278.2	185.3	5.076	4.545	825.8	557.4	0.012	0.012

Table A.6: The time and memory required by different tools to explore Szymanski's mutual exclusion algorithm ("N/A" = model not available, "-" = model could not be explored).

B Bibliography

[1] Systems and software engineering – high-level petri nets – part 1: Concepts, definitions and graphical notation, 2004.

[2] Systems and software engineering – high-level petri nets – part 2: Transfer format, 2011.

[3] Beem database. `http://anna.fi.muni.cz/models/`, March 8, 2012.

[4] Drop git repository. `http://fmt.cs.utwente.nl/tools/scm/drop.git/`, September 2012.

[5] Fmt - formal methods and tools. `http://fmt.cs.utwente.nl`, September 2012.

[6] Ltsmin. `http://fmt.cs.utwente.nl/tools/ltsmin/`, February 1, 2012.

[7] Pnml reference site. `http://www.pnml.org/`, August 2012.

[8] Spin - formal verification. `http://spinroot.com`, Februari 1, 2012.

[9] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. Divine - a tool for distributed verification. In T. Ball and R. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 278–281. Springer Berlin / Heidelberg, 2006. 10.1007/11817963_26.

[10] S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-14295-6_31.

[11] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 1 edition, Mar. 2006.

[12] S. Evangelista. High Level Petri Nets Analysis with Helena. In *ATPN'2005*, volume 3536 of *LNCS*, pages 455–464. Springer, 2005.

[13] S. Evangelista. Helena a high level net analyzer. `http://www.lipn.univ-paris13.fr/~evangelista/helena/`, August 2012.

[14] 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 '05, pages 110–121, New York, NY, USA, 2005. ACM.

[15] P. Godefroid. *Partial-order methods for the verification of concurrent systems. An Approach to the state-explosion problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1996.

[16] G. J. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2003.

[17] H. Kastenberg and A. Rensink. Dynamic partial order reduction using probe sets. In F. van Breugel and M. Chechik, editors, *CONCUR 2008 - Concurrency Theory*, volume 5201 of *Lecture Notes in Computer Science*, pages 233–247. Springer Berlin / Heidelberg, 2008.

[18] J. P. Katoen and C. Baier. *Principles of model checking*. Cambridge, Mass ; London : MIT Press, 2008.

[19] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, Aug. 1974.

[20] E. Pater. Partial order reduction for pins. Master's thesis, Univerity of Twente, 2011.

[21] D. Peled. All from one, one for all: on model checking using representatives. In C. Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-56922-7_34.

[22] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8:39–64, 1996. 10.1007/BF00121262.

[23] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*. Springer-Verlag, 1998.

[24] B. K. Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, pages 621–626, New York, NY, USA, 1988. ACM.

[25] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. *SIGOPS Oper. Syst. Rev.*, 31(5):224–237, Oct. 1997.

[26] A. Valmari. A stubborn attack on state explosion. In E. Clarke and R. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer Berlin / Heidelberg, 1991. 10.1007/BFb0023729.

[27] A. Valmari. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer Berlin / Heidelberg, 1991. 10.1007/3-540-53863-1_36.

[28] F. I. van der Berg and A. W. Laarman. Extending ltsmin with promela through spinja. In K. Heljanko and W. J. Knottenbelt, editors, *11th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2012, London, UK*, Electronic Proceedings in Theoretical Computer Science, London, September 2012. Open Publishing Association.