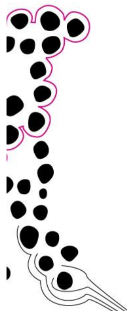
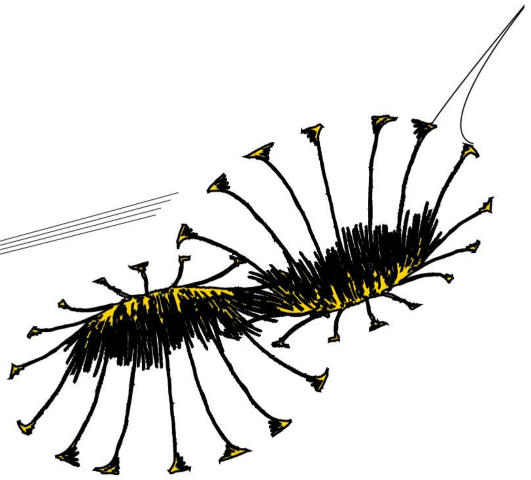
Partial Order Reduction for PINS



©*Elwin Pater*
Michael Weber, Jaco v.d. Pol, Dragan Bosnacki

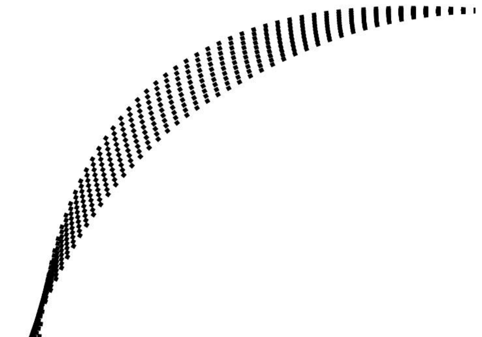


University of Twente

Master of Software Engineering

Department of Formal Methods and Tools

March 16, 2011



UNIVERSITEIT TWENTE.

Enschede

Abstract

There are a lot of different tools used for formal verification of hard- and software. Both for the scientific purpose of comparing methods and tools, as well as the industrial use cases which may require combining various methods, a uniform interface for state space generation would improve the applicability of model checkers. One such interface is the partitioned next state interface (PINS) [5] which is an abstraction layer between the model and the algorithms used for state space generation. In this thesis we develop a partial order algorithm that works using a minimal extension of the PINS interface, and is developed in a modular fashion.

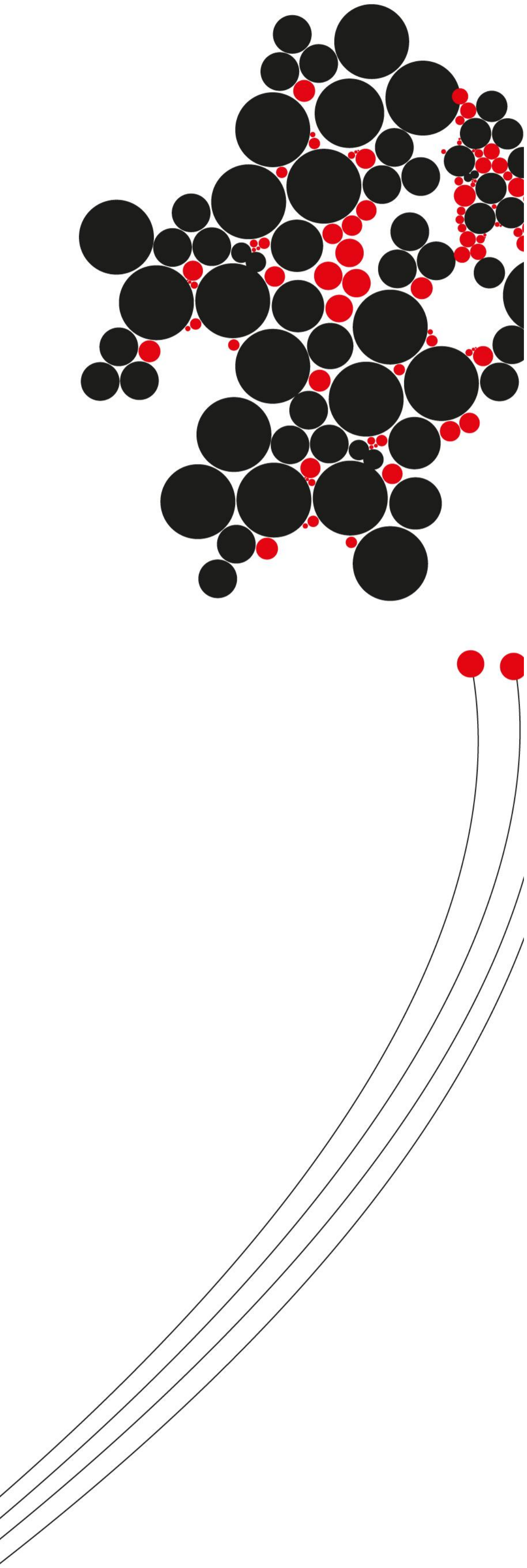
Contents

List of Tables

List of Figures

Chapter 1

Introduction



1.1 Background

Model checking is an important technique when it comes to verification of hardware and software. In short, model checking is the automatic verification that some property p holds in some model M. Typically, these properties specify a requirement such as the absence of deadlocks, a safety property ("bad things do not happen") or a liveness property ("eventually good things will happen"). The term model checking was introduced by Clarke and Emerson [15], and was meant as an efficient, practical, algorithmic method to verify a model specification. Since then, various model checkers (SPIN [28], mCRL2 [24], nuSMV [8, 9], LTSmin [5]) have been implemented. Companies apply the technique to increase their confidence in the correctness of software. Safety-critical control software used in avionics nowadays is required to be verified using model checkers. Other industries report success stories using model checking as well. Nevertheless, the use of formal methods like model checking in the industry can still be improved. Two key ingredients that may contribute to this are improving the accessibility/usability of model checkers together with the development of new methods to verify larger, more complex systems. The number of states in complex systems can quickly explode when multiple subcomponents are combined into one global system. This phenomenon is referred to as the *state space explosion*, where the exponential blowup in the size of the state space can make full exploration infeasible. The best known methods to counter the state space explosion can be summarized in three categories: partial order reduction [22, 15], symbolic representation [9], special purpose techniques [15]. The special purpose techniques include abstraction [11], compositional reasoning [15], symmetry reduction. Abstraction focuses on proving a simplification of a system such that the properties that hold in the simplified system hold in the original system too. Sometimes incremental refinement of the abstraction is used (CEGAR [10]). Compositional reasoning decomposes the system into small parts on which local properties must hold for the global property to be true. Symmetry reduction defines an equivalence relation between identical subcomponents in a global system. These techniques are valuable alternatives when both partial order reductions and symbolic representation do not significantly reduce time nor memory.

Symbolic techniques as introduced by Ken McMillan make use of binary decision diagrams (BDD) to implicitly store the state space, represented as boolean function. The benefit of using BDD based structures is that an exponential blowup can be avoided in a proper BDD encoding. Besides the BDD based approach it is possible to unroll a program to a certain bound, encode it as a binary problem and use a SAT solver [8] to verify it. The technique called partial order reduction [22, 12, 37] exploits the fact that independent transitions of the system components do not need to be ordered, and therefore do not all need to be explored. The SPIN [28] model checker is well known for the use of this technique.

The above mentioned methods all have been implemented in different research projects and model checkers, but lack uniformity in specification language making it difficult to combine, compare and use the techniques together. Uniform tool support would improve the applicability of model checkers for both scientific and industrial purposes, but requires a more modular approach to model checking. In this view, the PINS interface [5] has been designed. The interface splits the responsibilities of state space generation and exploration. The back-end algorithms take care of state space exploration, exploiting the information provided through the PINS interface. This directly enables symbolic-, distributed- and multi-core state space exploration. The goal of this research is to make a generalized partial order reduction algorithm able to work with the abstracted data provided by (an extension of) the PINS interface.

1.2 Related work

Partial Order Reduction is a very well known technique, introduced around 1980 by Overman [34], Valmari [41], Godefroid and Wolper [23] and Holzmann and Peled [28]. In explicit state space exploration, partial order reduction usually refers to the specific technique of expanding a representative subset of all enabled transitions of a particular state. This technique has also been described as model checking using representatives [37]. Valmari uses *stubborn sets* for partial order reduction. Godefroid [22] presented a similar approach using *persistent sets*, and later defined a dynamic approach where dependencies are tracked on the fly [17]. The most well known implementation uses the *ample set* approach, and is available in SPIN

[28]. The *ample set* has many of the same features of the persistent set, plus extra conditions needed for validation of temporal logic. Twophase, an algorithm of Nalumasu and Gopalakrishnan [33] embeds this proviso in the algorithm. Kurshan et al [30] apply the reduction into the model before the actual exploration (static partial order reduction). Recently, the POR techniques have been used in symbolic, sat based exploration strategies too [29, 42]. This is done by forcing processes to execute in a particular order.

1.3 Research questions

All partial order reduction algorithms mentioned in the related work heavily depend on the notion of processes. The PINS interface is an abstraction without processes. The process algebraic front-ends μCRL and mCRL2 do not rely on the notion of processes, thus adding processes to the PINS interface will limit its power. Therefore two main research questions must be answered. First

> *"How to design a partial order reduction algorithm without the notion of processes?"*

Together with the question

> *"What is a minimal extension of the PINS interface to support this algorithm?"*

Besides that the partial order reduction algorithms rely heavily on processes, they usually are intermingled with the search algorithms. From a tool designer's point of view, this makes it hard to reason about and maintain the back-end algorithms. Furthermore, all back-end algorithms need to be aware of partial order reduction, which makes it difficult to make changes in the interface or the algorithms. A more modular approach could potentially solve this.

> *"Is it possible to design this partial order reduction algorithm in a modular fashion, such that it can be implemented on top of the normal search algorithms?"*

Besides modularity, ideally the partial order reduction algorithm is accessible with the least amount of work for any front-end using it. It should be both easy to extract, and pass the extra information through the PINS interface.

> *"What is the least amount of information needed to achieve a reasonable state space reduction?"*

Finally, since the PINS back-end algorithms already support symbolic model checking using BDDs, the addition of a partial order reduction algorithm makes it possible to compare the two techniques. Currently this question is difficult to answer because there exists no tool that supports both techniques. Comparing the techniques using different tools is not fair with respect to time nor memory, since this requires different specification languages, different compilers, different optimizations and so forth.

> *"How does partial order reduction compare to symbolic model checking?"*

1.4 Summary of contributions

1. A modular partial order reduction layer for an extension of PINS (i.e. language independent). (Section 3.2 to 3.7).
2. Improvement on stubborn set algorithm. (Section 3.6).
3. A modular LTL layer for PINS. (Section 4.4).
4. Couvreur's algorithm combined with General State Expanding Algorithms. (Section 4.6).
5. A modular proviso implementation. (Section 4.5 and 4.6.3 to 4.6.7).

Chapter 2

The PINS architecture

In this chapter the PINS architecture is introduced, as is described in the original tool paper by Stefan Blom, Michael Weber and Jaco van de Pol. The PINS architecture is an abstract interface through which state space generation is enabled. This chapter introduces the concept of the interface as well as the implementation in the LTSmin toolset and the front-ends connecting to it.

2.1 The Partitioned Next State Interface

The PINS interface [5] is an Interface based on a Partitioned Next-State function. It separates the responsibility for state generation from the state space exploration tools by providing a layered architecture (see Figure 2.1) with front-ends, wrappers and back-ends. Front-ends provide the next state information while encapsulating language details of the specific tool through the PINS interface. Using PINS2PINS wrappers, various optimizations can be made at the level of the PINS interface. The back-end algorithms are based purely on the information provided by the PINS interface, and are completely unaware of the wrappers and details of the languages. Interestingly the interface often allows existing *enumerative* tools to connect easily, instantly providing explicit, symbolic and distributed state space generation. An implementation of the PINS interface exists in the LTSmin toolset [5].

Currently the following front-ends are connected to the LTSmin toolset through the PINS interface:

1. ETF [5], the Extended Table Format is an in- and output language for PINS.
2. DiVinE [2], a tool for LTL model checking and reachability analysis of discrete distributed systems.
3. CADP [20], a toolbox designed to analyze protocols and distributed systems. LTSmin also implements the CÆSAR/OPEN [19] interface to CADP's on-the-fly model checking and bisimulation reduction algorithms.
4. μCRL [3], is a process algebraic language that was especially developed to take account of data in the study of communicating processes.
5. mCRL2 [24], the successor of μCRL.
6. NIPS [43], a virtual machine able to handle a large subset of Promela [28].
7. SpinJa [13], a Java implementation of Spin supporting a large subset of Promela.

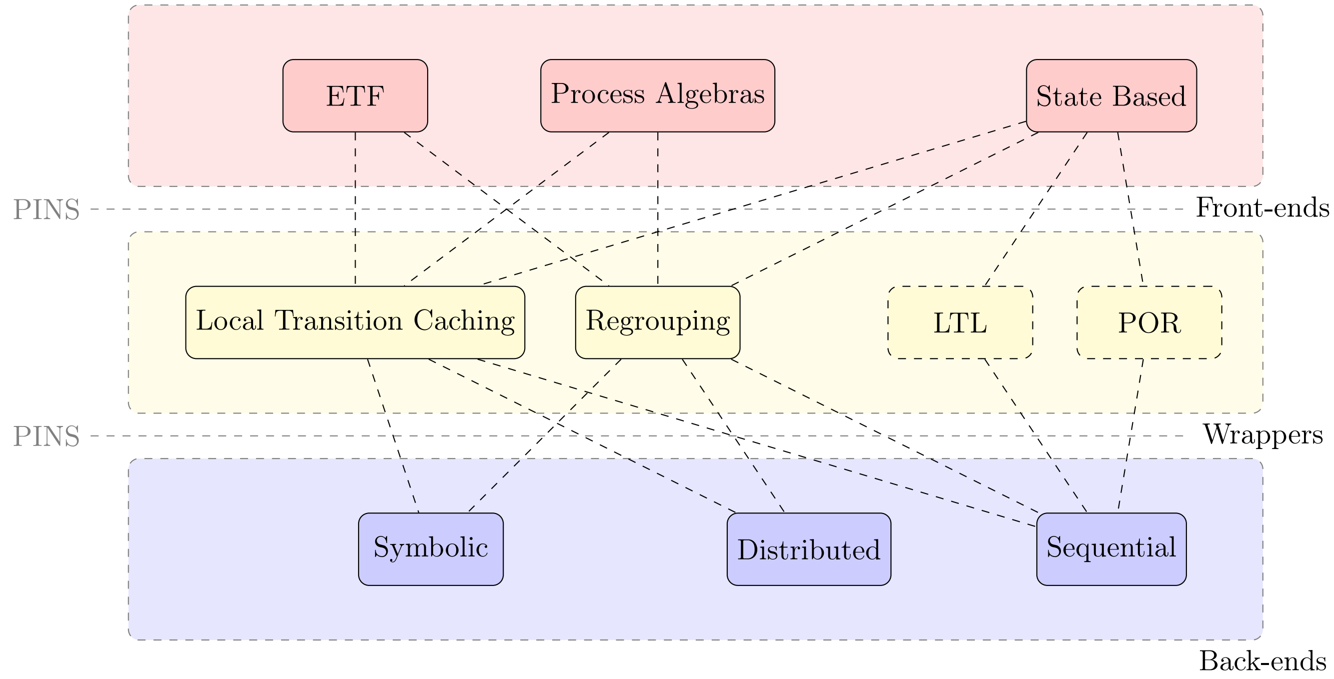


Figure 2.1: The PINS architecture

2.1.1 The PINS front-end interface

A PINS front-end or language module is responsible for the translation of the model specification (usually some textual description of the model) to the PINS interface. The underlying semantic model of the PINS interface is a *transition system* with edge- and state labels. We elide from the edge labels as these are not used for our purposes.

Definition 1. A transition system (TS) is a structure $\langle S, \rightarrow, \hat{s}, L\rangle$, where S is a set of states, $\rightarrow \subseteq S \times S$ is a transition relation, $\hat{s}$ is the initial state and L is an indexed vector of M state labeling functions $\lambda_i : S \rightarrow \mathbb{N}$ for $(1 \leq i \leq M)$. We refer to the ith state labeling function as *state label i*.

The PINS interface exports functionality describing the TS; *initial_ state()* returns $\hat{s}$ and *next_ all(s)* returns all successor states of s, *state_ label_ all(s)* returns all evaluations of the M state labeling functions. The heart of the PINS interface however lies in a more fine grained view of the transition system, which is used for exploiting *event locality*: a set of states is a Cartesian product of elements; transitions usually just depend on few of these. For example, the transition system may be a (asynchronous) product of multiple smaller transition systems each describing a process. These processes usually have their own local variables and communicate using global variables or channels. The behavior of each process depends solely on just these parts of the state, not on the local state of other processes. This is formalized by partitioning the transition system as follows:

Definition 2. A partitioned transition system (PTS) is a structure $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{e}_1, \ldots, \hat{e}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$. The sets of elements $E_1, \ldots, E_N$ define the set of states $S_P = E_1 \times \ldots \times E_N$. The transition groups $\rightarrow_i \subseteq S_P \times S_P$, $(1 \leq i \leq K)$ define the transition relation $\rightarrow = \bigcup_{i=1}^{K} \rightarrow_i$. The initial state is $\hat{s} = \langle\hat{e}_1, \ldots, \hat{e}_N\rangle \in S_P$ and $L_P = \langle\lambda_1, \ldots, \lambda_M\rangle$. The defined TS of P is $\langle S_P, \rightarrow, \hat{s}, L_P\rangle$.

The front-end must translate the model specification to a fixed length *state vector* of N slots describing the global state of a system, and split the next state function into K groups (*transition groups*) over sub-vectors of the state vector. The PINS interface provides the $next\_long(s, i) = \{s' \mid s \rightarrow_i s'\}$ function for accessing the transition relation of each group.

Transitions are grouped based on the slots in the state vector they read or write to. For each transition group, its sub-vector (slots used by the transition group) is stored in the *pins dependency matrix* $(K \times N)$.

The pins dependency matrix can be obtained from analyzing the complete state space. This information usually is not available beforehand, but it is possible to provide an overestimation based on syntactic analysis of the model specification.

Definition 3. Given a TS $T = \langle S, \rightarrow, \hat{s}, L\rangle$, the *reachable states* in T denoted $Reach(TS) = S_{reach}$ are defined as the set $S_{reach} \subseteq S$ such that $\hat{s} \in S_{reach}$ and for each state $s \in S_{reach}$, if $(s, s') \in \rightarrow$ then $s' \in S_{reach}$.

The reachable states of a partitioned transition system, $Reach(PTS)$ is given by the reach of its transition system $Reach(TS)$.

Definition 4. Given a PTS $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{e}_1, \ldots, \hat{e}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$. Transition group i is independent of state slot j if for all $\langle e_1, \ldots, e_N\rangle$ and $\langle e'_1, \ldots, e'_N\rangle \in Reach(P)$, whenever $(\langle e_1, \ldots, e_j, \ldots, e_N\rangle \times \langle e'_1, \ldots, e'_j, \ldots, e'_N\rangle) \in \rightarrow_i$, then

1. $e_j = e'_j$
   (i.e. state slot j is not modified in transition i)
2. for all $r_j \in E_j$, we also have $(\langle e_1, \ldots, r_j, \ldots, e_N\rangle \times \langle e'_1, \ldots, r_j, \ldots, e'_N\rangle) \in \rightarrow_i$,
   (i.e. the value of state slot j is not relevant in transition i)

Using this independence relation the *pins dependency matrix* is described as follows:

Definition 5. A pins dependency matrix $D_{K \times N} = DM(P)$ for PTS P is a matrix with K rows and N columns containing $\{0, 1\}$ such that if $D_{i,j} = 0$ then group i is independent of element j.
For any transition group $1 \leq i \leq K$, we define π_i as the projection $\pi_i : S \rightarrow \Pi_{\{1 \leq j \leq N \mid D_{i,j}=1\}} S_j$.

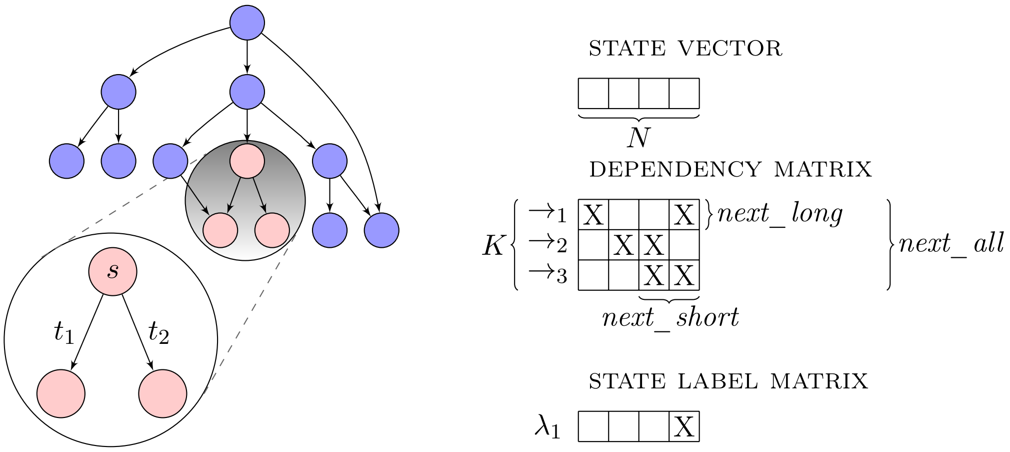


Figure 2.2: The PINS front-end interface

The PINS interface function *next_short* uses only the sub-vectors $\pi_i S$ to compute sub-vector of the next state for a particular transition group. By providing implementations for either *next_short* or *next_long* the PINS interface can supply the default implementation for the other next state functions.

In a similar fashion, the *state labeling* functions are split into $state\_label\_all(s)$, $state\_label\_long(s,i)$ and $state\_label\_short(s,i)$. The state labeling functions can be used to associate a particular label to a state, such as an atomic proposition. Similar to the pins dependency matrix a pins state label matrix is defined. Note that state labels are always read only.

Definition 6. Given a $PTS\ P = \langle\langle E_1,\ldots,E_N\rangle, \langle\rightarrow_1,\ldots,\rightarrow_K\rangle, \langle\hat{e}_1,\ldots,\hat{e}_N\rangle, \langle\lambda_1,\ldots,\lambda_M\rangle\rangle$. State label i is independent of state slot j if $\langle\lambda_1,\ldots,\lambda_i,\ldots,\lambda_M\rangle$ for all $\langle e_1,\ldots,e_N\rangle \in Reach(P)$, whenever $\lambda_i(\langle e_1,\ldots,e_j,\ldots,e_N\rangle) = n \in \mathbb{N}$ then

1. for all $r_j \in E_j$, we also have $\lambda_i(\langle e_1,\ldots,r_j,\ldots,e_N\rangle) = n$, (i.e. the value of state slot j is not relevant for evaluation of state label i)

And the *pins state label matrix* now follows from this. An overestimation for the state label matrix can also be used.

Definition 7. A pins state label matrix $SL_{M\times N} = SLM(P)$ for $PTS\ P$ is a matrix with M rows and N columns containing $\{0,1\}$ such that if $SL_{i,j} = 0$ then state label i is independent of element j.
For each state label $1 \leq i \leq M$, we also define π_i as the projection $\pi_i : S \rightarrow \Pi_{\{1\leq j\leq N \mid SL_{i,j}=1\}} S_j$.

Example. A model specification is translated to a transition system (see left side of Figure 2.2). A particular transition t_1 might be in relation $\rightarrow_1$, while another transition t_2 might be from another process (which uses different state slots) and is in $\rightarrow_2$. A call to $next\_long(s,1)$ for the source state s of both t_1 and t_2 highlighted in Figure 2.2 will result in the successor state of t_1 while the call $next\_long(s,2)$ will result in just the successor state produced by t_2. A call to $next\_all(s)$ will produce both successor states, and calls to $next\_short(\pi_1 s,1)$ and $next\_short(\pi_2 s,2)$ will produce the sub-vectors of the successor states. Similarly, calls to $state\_label\_all(s)$, $state\_label\_long(s,i)$ and $state\_label\_short(s,i)$ produce the evaluation of state label i using state labeling function λ_i on s.

2.1.2 PINS back-ends and algorithms

Currently, LTSmin has various back-end algorithms that can directly be used through the PINS interface. Explicit enumerative methods such as depth first search and breadth first search are implemented using different storage mechanisms (BDD based [9], hash table, tree compression [4]). Besides this, distributed and multi-core-, as well as symbolic state space exploration strategies are available. The symbolic tools learn the transition relations on the fly and use a BDD/MDD to store the state space and transition relations in memory.

2.1.3 PINS wrappers

A PINS interface wrapper is an intermediate layer between the front-end and the back-end algorithms. The layer is modular and wraps the PINS interface such that neither the back-end, nor the front-end has knowledge of the extra layer.

Local Transition Caching

If the language module is fairly slow, the back-ends might benefit from caching the transition vectors. The cache layer uses the pins dependency matrix to cache only the sub-vectors used by each transition group. Therefore the layer usually won't have a drastic memory overhead.

Regrouping

Through smart reordering of the transition groups and slots in state vector, BDD-based state representations greatly benefit from the resulting variable ordering. In combination with the symbolic search strategies some models with 10^{50} states can be generated in a matter of seconds. Reordering the transition groups itself may have a positive effect on the symbolic chaining algorithm and combining groups limits the overhead needed to store transition relations both for the symbolic algorithms and the transition caching layer.

LTL and POR

The dashed (see Figure 2.1) PINS wrappers are are contributed by this work. The LTL (linear temporal logic) wrapper is described in chapter 4 and is used to produce the synchronous product automata of the states provided by the front-end and a büchi automaton generated from a given LTL formula.

The POR (partial order reduction) wrapper is the topic of both chapter 3 and 4, and is used to explore a representative subset of the state space when finding deadlocks or verifying LTL formulas.

Chapter 3

Partial Order Reduction for Deadlocks

Partial order reduction is a well known technique in the field of model checking. In this chapter we explain the technique and show how to extend the PINS interface such that a modular partial order reduction algorithm can be implemented as PINS wrapper. The algorithm is based on the stubborn set algorithm or Valmari and is adapted to work without the notion of processes.

3.1 Theory of partial order reduction

In the field of model checking, it is quite common that a system is described as a composition of multiple subsystems or components. As the number of components grow, the number of states in the combined global system grows exponentially with it because all different interleavings of the components lead to new states. Thus, even with very small components, the number of states in combined systems quickly blows up. This is the so called *state space explosion.*

For some safety critical systems, an important property is the *absence of deadlocks.*

Definition 8. Given a *TS* $T = \langle S, \rightarrow, \hat{s}, L\rangle$, a state $s \in S$ is in a *deadlock* iff there is no transition starting from s, $\exists s \in S : (\forall s' \in S : (s, s') \notin \rightarrow)$.

A deadlock is an unwanted situation where the global system waits indefinitely, and no progress can occur. To verify that a transition system T contains no deadlocks a straightforward method is to generate all possible global states and check that each state has at least one outgoing transition. This however becomes quickly infeasible due to the state space explosion. Now consider the asynchronous composition of any transition system T with a process P with precisely one state s and one transition from s to s (self-loop). For this composition, we can readily say that the global system contains no deadlock, because for every global state, process P can execute its self loop. This example shows that if one or multiple components together can never lead to a deadlock state and are independent of the rest of the system, then there is no deadlock in the global system, and hence, it is enough to verify only this part of the state space. More formally, partial order reduction exploits independence between concurrently executed transitions. Two independent transitions will result in the same global state when executed in any order. Therefore, these events really are only partially ordered. Furthermore, the order of occurrence is irrelevant to the property being checked (in this case the absence of deadlocks). For example, the execution of two independent transitions α and β, (see Figure 3.1) from state s will, in any order, lead to the same state s', and hence, no deadlock can occur on the intermediate states. This fact is exploited by a technique know as *partial order reduction.* The independence and commutativity of system components is used to compute a representative subset of traces (sequences of transitions) through the complete state space, preserving all deadlock states.

3.1.1 Semantic and structural transitions

As shown in Figure 3.1, two independent transitions α and β can lead to a transition system with 4 states. Note that this transition system has four elements in its transition relation, which we refer to as *semantic transitions*, i.e. transitions t that model actual changes in the transition system $t = (s, s') \in\rightarrow$. For partial order reduction we need to refer to transitions on another level, namely the level of system components. This reveals extra information on the internal structure to the transition system. We refer to α and β in Figure 3.1 as structural transitions. Hence, a semantic transition is an occurrence of a structural transition, and a structural transition α can have multiple semantic transitions $\alpha = \{(s, s') \mid (s, s') \in\rightarrow\}$. A structural transition can be viewed as a transition in a system component that can only occur if some guard condition evaluates to true. For a transition system we can define such a guard condition:

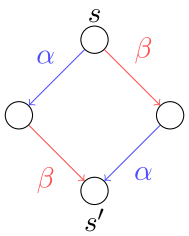


Figure 3.1: Two independent transitions α and β

Definition 9. Given a *TS* $\langle S, \rightarrow, \hat{s}, L\rangle$, a guard $g : S \rightarrow \{\mathit{true}, \mathit{false}\}$ is a total function that maps each state to a boolean value. We write $g(s)$ to denote the evaluation of the guard g in state $s \in S$.

A transition group can be seen as guarded by a conjunction of multiple guards:

Definition 10. Given a *PTS* $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{e}_1, \ldots, \hat{e}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$. A transition group $\rightarrow_i$ $(1 \leq i \leq K)$ is *guarded* by a non-empty set of guards $\mathcal{G}$ if

1. $\forall s \in \mathit{Reach}(P) : (\exists g \in \mathcal{G} : g(s) = \mathit{false}) \implies \neg\exists s' \in \mathit{Reach}(P) : (s, s') \in\rightarrow_i$
   (i.e. existence of a false guard implies there is no successor state)

2. $\forall s \in \mathit{Reach}(P) : (\forall g \in \mathcal{G} : g(s) = \mathit{true}) \implies \exists s' \in \mathit{Reach}(P) : (s, s') \in\rightarrow_i$
   (i.e. all guards true implies at least one successor)

Note that for each transition group we can define a trivial guard that is true whenever there is a successor and false otherwise, thus each transition group is always guarded by its trivial guard.

From now on, when we refer to a transition we refer to a *structural transition.* Furthermore we assume that a structural transition is a deterministic transition, i.e. it has a unique successor state.

For a partitioned transition system, a quite natural way to model structural transitions of a system component is to use one transition group for each structural transition. Since we need the structural transitions for partial order reduction it would be natural to use transition groups that represent structural transitions. Transition groups in PINS however do not need to conform to this assumption, thus may not model a deterministic transition.

Definition 11. Given a *PTS* $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{e}_1, \ldots, \hat{e}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$. A transition group $\rightarrow_i$ $(1 \leq i \leq K)$ is *singleton* if for each state $s \in \mathit{Reach}(P)$ there is at most one successor state, i.e. $\neg\exists s, s', s'' \in \mathit{Reach}(P) : s' \neq s'' \wedge (s, s') \in\rightarrow_i \wedge (s, s'') \in\rightarrow_i$.

We will now proof by construction that a singleton transition group can always be seen as a structural transition without knowing to which subcomponent of the system it belongs to.

Theorem 1. *A singleton transition group in a PTS P represents a deterministic structural transition in a (strong) bisimulation equivalent PTS P'.*

Proof. Given a *PTS* $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{e}_1, \ldots, \hat{e}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$. From this, define a new *PTS* $P' = \langle\langle E_1, \ldots, E_N, E_{N+1}, \ldots, E_{N+K}\rangle, \langle\rightarrow'_1, \ldots, \rightarrow'_K\rangle, \langle\hat{e}_1, \ldots, \hat{e}_N, \hat{e}_{N+1}, \ldots, \hat{e}_{N+K}\rangle, \langle\lambda'_1, \ldots, \lambda'_M\rangle\rangle$ with K components and N global variables. Each component $(1 \leq i \leq K)$ has precisely one local state $\hat{e}_{n+i} \in E_{n+i}$, and one structural transition α that models a self-loop on $\hat{e}_{n+i}$ and is guarded by guard g. Define $g(s) = \exists s' \in S_P : (s, s') \in\rightarrow_i$, for $s \in S_P$; i.e. the guard is true if the ith transition group of P produces a successor state. We define the transition groups of P' as $\rightarrow'_i = \langle E_1, \ldots, E_N, E_{N+1}, \ldots, e_i, \ldots, E_{N+K}\rangle \times \langle E'_1, \ldots, E'_N, E_{N+1}, \ldots, e_i, \ldots, E_{N+K}\rangle$ such that $\langle E_1, \ldots, E_N\rangle \times \langle E'_1, \ldots, E'_N\rangle \in\rightarrow_i$ of P. Since none of the components i $(1 \leq i \leq K)$ of P' changes its local state, $E_{N+1}, \ldots, E_{N+K}$ is constant. Therefore, there is a (strong) bisimulation equivalence relation $P \equiv P'$ by the one-to-one correspondence between the prefix $E_1, \ldots, E_N$. If a transition group in P is a singleton transition group, it is a singleton transition group in P' too. Therefore it must be a deterministic structural transition in P'. □

A non-singleton transition group can be seen as a non-deterministic structural transition in P'.

Since a singleton transition group in a *PTS* represents a deterministic structural transition in an equivalent *PTS*, we proved that if partial order reduction is correct for structural transitions in the *PTS* P', its equivalent original *PTS* can be reduced in a similar fashion.

To reference the transition groups or structural transitions, we introduce the following notation which we will use from now on. We assume that each *TS* has a known underlying *PTS* which we can always reference to, and that all transition groups in the *PTS* are singleton.

Definition 12. Given a *TS* $T = \langle S, \rightarrow, \hat{s}, L\rangle$ and its underlying *PTS* $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle,$ $\langle\hat{e}_1, \ldots, \hat{e}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$, such that each transition group $\rightarrow_i$ is singleton and guarded by $\mathcal{G}_i$, with $(1 \leq i \leq K)$. Let $s, s' \in \mathit{Reach}(T)$.

1. We write α, β or γ to denote a structural transition of the form $\alpha = \{(s, s') \mid (s, s') \in\rightarrow_i\}$.

2. We write $\alpha \in\rightarrow$ when we mean that α is a structural transition represented by a singleton transition group in the underlying partitioned transition system.

3. The *enabled* transitions of s are $\mathit{enabled}(s) = \{\alpha \mid \exists s' \in \mathit{Reach}(T) : (s, s') \in \alpha\}$ with $\alpha \in\rightarrow$.

4. The *disabled* transitions of s are $\mathit{disabled}(s) = \{\alpha \mid \exists\alpha \notin \mathit{enabled}(s)\}$ with $\alpha \in\rightarrow$.

5. We write $s \xrightarrow{\alpha}$ when $\exists s' \in \mathit{Reach}(T) : (s, s') \in \alpha$ with $\alpha \in \rightarrow$.
6. We write $s \xrightarrow{\alpha} s'$ when $(s, s') \in \alpha$ with $\alpha \in \rightarrow$.
7. We write $s \xrightarrow{\alpha_1,\alpha_2,\alpha_3,\ldots}$ when $\exists s_1, s_2, \ldots \in \mathit{Reach}(T) : s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$.
8. We refer to s' in $s \xrightarrow{\alpha} s'$ by $\alpha(s)$.
9. We refer to the set of guards $\mathit{guards}(\alpha) = \mathcal{G}_i$ when α represents the structural transition of transition group $\rightarrow_i$.

Besides the need for the structural transitions, more fine grained information about what parts of the state are read and written is needed. We define the following sets:

Definition 13. Given a *PTS* $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{e}_1, \ldots, \hat{e}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$ such that each transition group in the *PTS* is singleton. Let the set of reachable states be $S_r = \mathit{Reach}(P)$, and let S_P be the states from the *TS* of P. Given two states $s \in S_r, s' \in S_P$, for $1 \leq i \leq N$:

1. e_i, e_i' denotes the value of E_i in s, and
2. $\delta(s, s') = \{i \mid e_i \neq e_i'\}$ is the set of indices on which s and s' disagree.

Let $\alpha = \{(s, s') \mid (s, s') \in \rightarrow_k\}$ be the structural transition of some transition group $(1 \leq k \leq K)$.

1. The *test set* of a guard $g \in \mathit{guards}(\alpha)$ is
   $\mathit{Ts}(\alpha, g) = \{i \mid \exists s \in S_r, s' \in S_P : \delta(s, s') = \{i\} \wedge g(s) \neq g(s')\}$.
2. The *write set* of α is
   $\mathit{Ws}(\alpha) = \bigcup_{s \in S_r} \delta(s, \alpha(s))$.
3. The *read set* of α is
   $\mathit{Rs}(\alpha) = \{i \mid \exists s \in S_r, s' \in S_P : \delta(s, s') = \{i\} \wedge \exists j \in \mathit{Ws}(\alpha) : j \in \delta(\alpha(s), \alpha(s'))\}$.
4. The *variable set* of α is
   $\mathit{Vs}(\alpha) = \bigcup_{g \in \mathcal{G}_k} \mathit{Ts}(\alpha, g) \cup \mathit{Rs}(\alpha) \cup \mathit{Ws}(\alpha)$.

The *test set* is the set of variables used for the guard evaluation. The *write set* is the set of variables changed by the application of a transition, and the *read set* is the set of variables that influence the effect of the application of the transition, but not necessarily the evaluation of the guards. For example, a global variable might be in the *read set* of a transition but not in the *test set*, i.e. if $pc(s)$ is the only guard of the transition, and the effect reads and writes a global variable.

The above sets are computed in the context of the complete state space, thus after state space generation. An over-approximation can usually be provided beforehand, by syntactical analysis of the model. The *pins dependency matrix* in the PINS architecture is an over-approximation of the *variable set.*

3.1.2 Independence

As noted at the start of this chapter, interleavings of independent transitions can be pruned from the state space when searching for deadlocks. Clearly, two transitions α and β with disjoint variable sets are independent because execution of one transition can never change the evaluation of the guards, nor effect the state slots used by the other transition. An example of two independent transitions α, β is shown in Figure 3.1. Note that no deadlock can occur on the intermediate states. A formal definition of the independence relation is given by:

Definition 14. Let T be a *TS* $\langle S, \rightarrow, s^0, L\rangle$. An *independence* relation $I \subset (\rightarrow \times \rightarrow)$ is symmetric, reflexive relation, satisfying the following two conditions for each state $s \in S$ and for each pair of transitions $(\alpha, \beta) \in I$.

Enabledness If $\alpha \in \mathit{enabled}(s)$ then $\beta \in \mathit{enabled}(s) \leftrightarrow \beta \in \mathit{enabled}(\alpha(s))$.
Commutativity If $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

The *dependency* relation D is the complement of I, $D = (\rightarrow \times \rightarrow) \backslash I$.

Note that the dependency relation is closely related to the pins dependency matrix, but is different in that the pins dependency matrix refers to the dependency of a transition to the state vector whereas the dependency relation is used to describe a dependency between two transitions. The pins dependency matrix can however be used to extract the independence/dependency relation between two transitions. Two transitions are independent if they have disjoint variable sets, i.e. use disjoint parts of the state vector. This is the (in)dependence relation used in the algorithms described from now on.

Definition 15. Given a pins dependency matrix $D_{K\times N} = DM(P)$ for *PTS* P. Two transitions $\alpha, \beta \in\rightarrow$ are in the independence relation if their variable set is disjoint: $Vs(\alpha) \cap Vs(\beta) = \emptyset \iff (\alpha, \beta) \in I$.

Besides transitions with disjoint variable sets, more fine grained syntactic properties can be used to extract the independence relation, for example two transitions that only read from a shared variable are independent too. A larger independence relation (more independence) will usually result in better reduction of the state space. The maximal independence relation for the the matrix in Figure 2.2 would only contain the tuple $(\rightarrow_1, \rightarrow_2) \in I$.

3.1.3 Equivalence classes

The independence relation can be used to define equivalent transition sequences in the state space, known as Mazurkiewicz's traces [22, 31].

> "Two sequences of transitions are equivalent if they can be obtained from each other by successively permuting adjacent independent transitions"

Example. Consider the set of transitions $\{\alpha, \beta, \gamma\}$, and assume that α is dependent with both β and γ, while β and γ are independent. The sequence of transitions $\omega = \alpha\beta\gamma$ is in the equivalence class $\{\langle\alpha\beta\gamma\rangle, \langle\alpha\gamma\beta\rangle\}$. The second sequence can be obtained from the first sequence by permuting the two adjacent transitions β and γ. It is however not equivalent to the sequence $\beta\alpha\gamma$. This sequence may not even be possible; since α and β are dependent execution of β may disable α.

To preserve deadlocks, it is enough to explore precisely one trace in each distinct equivalence class.

3.1.4 Reduction function

Partial order reduction is most useful when applied on-the-fly, during the generation of the model. Therefore, at each encountered state s a decision must be made on which transitions to explore, and which to ignore. Obviously, this becomes a subset of all transitions that can possibly be explored from s, thus from *enabled*(s).

Partial order reduction restricts the exploration of all possible transition sequences, to a subset of all transition sequences such that at least one transition sequence in each distinct equivalence class is explored. This reduction must be achieved on-the-fly. To do this one can delay execution of some transition to a future state. The decision to do this must however be made based on information provided at each state $s \in S_P$ because it must be computed on-the-fly. This is formalized through a reduction function as follows:

Definition 16. A *reduction function* for a *PTS* $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{s}_1, \ldots, \hat{s}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$ where each transition group $\rightarrow_i$, $(1 \leq i \leq K)$ is singleton, is a mapping r from a state $s \in S$ to a set of transition groups (2^K) such that for all $s \in S_P : r(s) \subseteq \mathit{enabled}(s)$.

I.e., selectively using transition groups (the structural transitions) for each state $s \in S_P$.

By applying such a reduction function, one can construct a reduced graph.

Definition 17. Let $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{s}_1, \ldots, \hat{s}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$ be a partitioned transition system and let r be a reduction function on P. A reduced *PTS* $P_r = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow'_1, \ldots, \rightarrow'_K\rangle$, $\langle\hat{s}_1, \ldots, \hat{s}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$ of P by r is defined such that:

1. $s \in S_{P_r}$ iff there is a finite execution sequence $s_0 \overset{\alpha_0}{\rightarrow} \ldots \overset{\alpha_{n-1}}{\rightarrow} s_n$ such that $s = s_n$ and $\alpha_i \in r(s_i)$, $\forall s_i \in \{s_0, \ldots, s_{n-1}\}$. Hence $\mathit{Reach}(P_r) \subseteq \mathit{Reach}(P)$.
2. $(s, s') \in\rightarrow_i$ of P_r iff $(s, s') \in\rightarrow_i$ of P and $(s, s') \in r(s)$.

3.1.5 Persistent set

In order for the reduction function to precisely preserve the properties of interest (in this case the absence of deadlocks), there are some conditions that the subset must adhere to. Godefroid [22] defined the *persistent-set selective search* through the state space, in which each state s that is reached by the search explores only a set $\mathcal{T}$ of the enabled transitions. This set $\mathcal{T}$ must be persistent (defined below), and must be non-empty if there exist transitions enabled in s. Using these definitions, the reduced state space is trace equivalent to the unreduced state space, i.e. each equivalence class in the original state space is also present in the reduced state space. A similar notion of an *ample set* was defined by Clarke, Grumberg and Peled [12]. The main difference between the ample set and the persistent set is that the ample set is a set that has to be used with a particular algorithm in order to work. A slightly less restricting notion of independence is used (where transitions can disable each other) which has to be "corrected" by the algorithm. Furthermore the ample set usually refers to a set that also preserves temporal properties (LTL). The persistent set is a more general definition of the restriction the reduction function r must adhere to in order to preserve deadlocks in the reduced state space.

Definition 18. Given a *TS* $\langle S, \rightarrow, s^0, L\rangle$. A set $\mathcal{T}$ of transitions enabled in a state $s \in S$ is *persistent in* s iff for all non-empty sequences of transitions $s \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} s_2 ... \overset{\alpha_{n-1}}{\rightarrow} s_n \overset{\alpha_n}{\rightarrow} s_{n+1}$ with $s, s_1, \ldots s_n, s_{n+1} \in S$ from s and including only transitions $\alpha_i \in \rightarrow$ such that $\alpha_i \notin \mathcal{T}, 0 \leq i \leq n$, α_n is independent in s_n with all transitions in $\mathcal{T}$.

Note that this definition precisely resembles condition C0 and C1 of the ample set (Clarke, Grumberg and Peled [12]) when using the stronger definition of independence:

C0 $\quad$ *ample*$(s) = \emptyset \leftrightarrow$ *enabled*$(s) = \emptyset$
C1 $\quad$ Along every path in the full state graph that starts at s, the following condition holds:
a transition that is dependent on a transition in *ample*(s) cannot be executed
without a transition in *ample*(s) occurring first.

Theorem 2. *Using definition 14 of independence, ample set condition C1 is equal to a persistent set.*

Proof. Note that the set *enabled*(s) trivially is both a persistent set and an ample set. In case of a deadlock in state s, *enabled*$(s) = \emptyset$. Since $r(s) \subseteq \emptyset$, $r(s)$ is a persistent set since there are no non-empty sequences from s, i.e. no path starting from s. If there is no deadlock, *enabled*$(s) \neq \emptyset$, and $r(s) \subset$ *enabled*(s), then all paths from s start with a transition either from $r(s)$ or from *enabled*$(s) \backslash r(s)$.

Suppose $r(s)$ is a persistent set. All paths starting from *enabled*$(s) \backslash r(s)$ contain only transitions independent with $r(s)$ or in $r(s)$. So along all these paths no transitions are dependent on $r(s)$, or they are in $r(s)$, hence $r(s)$ is an ample set. Suppose $r(s)$ is not a persistent set. Then there path starting from *enabled*$(s) \backslash r(s)$ leading to a transition that is not in $r(s)$ and that is dependent on a transition in $r(s)$. This violates the ample set condition C1, so $r(s)$ is not an ample set either. Suppose $r(s)$ is an ample set, then all paths starting from *enabled*$(s) \backslash r(s)$ are independent with transitions in $r(s)$, or they are dependent with transitions in $r(s)$ but a transition from $r(s)$ occurred first. Hence, all transitions of the non-empty sequence of transitions not in $r(s)$ are independent with transitions of $r(s)$, thus $r(s)$ it is a persistent set too. Suppose $r(s)$ is not an ample set, then there is a path starting with a transition from *enabled*$(s) \backslash r(s)$ containing only transitions not in $r(s)$ and leading to a transition dependent on a transition in $r(s)$. By definition of a persistent set, $r(s)$ is not a persistent set. □

By restricting the reduction function $r(s)$ to persistent sets that are non-empty if there are transitions enabled in s, we can explore only a subset of the full state space while preserving a trace in each equivalence class (and therefore all deadlocks, for a formal proof, see [22]).

3.2 A modular PINS POR wrapper

In general, a partial order reduction algorithm selects a subset of all enabled transitions (i.e. application of the reduction function $r(s)$), such that the resulting subset forms a persistent set. Because the persistent set preserves a trace through the state space such that there is an equivalent trace (with respect to

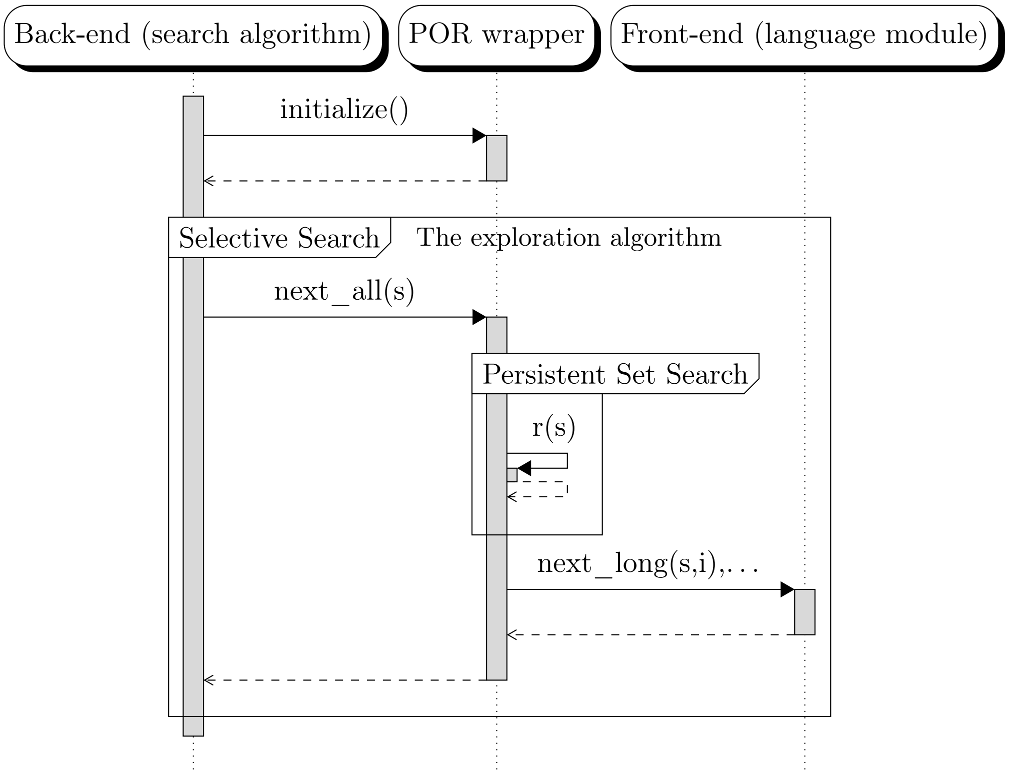


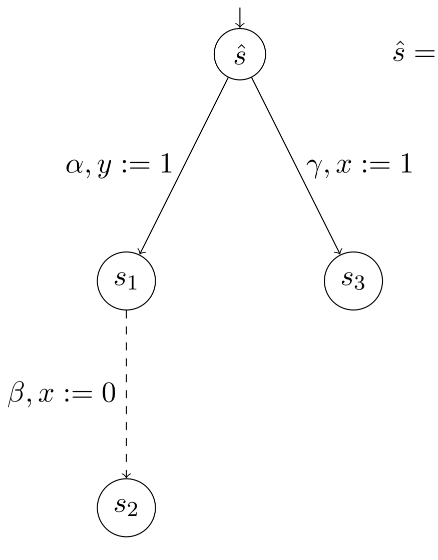
Figure 3.2: A POR wrapper for PINS

deadlocks) in the unreduced state space, deadlocks are preserved. Since we use a transition group as a structural transition, the partial order reduction algorithm should select a subset of the transition groups for each state s encountered. Therefore, the idea is use a PINS wrapper which can be activated without the knowledge of the underlying search algorithm. In principle, the PINS wrapper can select a subset of the transitions generated by the front-end, and pass this through to the back-end algorithm. Figure 3.2 shows how a PINS POR wrapper would work. The back-end search algorithm requests the states reachable from some state s by calling *next_all*. Then, the PINS POR wrapper intercepts this call and selects a subset of the transition groups. This subset of transition groups must be carefully selected. The PINS POR wrapper then forwards the request for the next states to the front-end which provides new states to the back-end search algorithm by calling a callback. This way, a subset of the enabled transitions is selected. The PINS POR wrapper thus serves as a filter.

Note that the POR wrapper requires the front-end to have a $next\_long(s, i)$ or $next\_short(s, i)$ function. Furthermore, the POR wrapper is called using $next\_all(s)$ in the search algorithm, however some search algorithm might require $next\_long(s, i)$ or $next\_short(s, i)$ calls. Since the POR wrapper needs a subset of the enabled transition for this it either needs to recompute this subset of each call to $next\_long(s, i)$ or $next\_short(s, i)$, or needs to cache the generated persistent set. Furthermore, since computing the persistent set requires the set $enabled(s)$, the POR wrapper minimally needs a pins dependency matrix in which each transition group reads the guards of all other transition groups, i.e. each transition group $(1 \leq i \leq K)$ is guarded by a set of guards $\mathcal{G}_i$ in the front-end, and all transition groups must be guarded by the set $\bigcup_{i \in \{1 \ldots K\}} G_i$ after application of the POR wrapper. Adding these guards to transition groups affects the pins dependency matrix, which in turn limits the effect of POR in combination with the symbolic tool.

3.3 Conflicting transitions

To have a baseline for the partial order reduction algorithm, the conflicting transitions algorithm has been implemented. The conflicting transitions algorithm is a very simple partial order reduction algorithm. It was first described by Godefroid and Wolper in 1993 [23]. The algorithm used here is from Godefroid [22].



$$\hat{s} = \langle a_1, a_2, 0, 0\rangle \in S, S = E_{P1} \times E_{P2} \times E_x \times E_y$$

$$E_{P1} = \{a_1, b_1, c_1\}, E_{P2} = \{a_2, b_2\}$$

$$E_x, E_y \in \{0, 1\}$$

$$\alpha = (\langle a_1, *, *, *\rangle \times \langle b_1, *, *, 1\rangle) \in \rightarrow_1$$

$$\beta = (\langle b_1, *, *, *\rangle \times \langle c_1, *, 0, *\rangle) \in \rightarrow_2$$

$$\gamma = (\langle *, a_2, *, *\rangle \times \langle *, b_2, 1, *\rangle) \in \rightarrow_3$$

Figure 3.3: Example conflicting transitions

3.3.1 Theory and algorithm

To implement the conflicting transitions algorithm, the notion of processes and process local states is needed. Each process has a special variable (state slot) called *process counter* used to encode the *local state* the process is in.

Definition 19. Given a *TS* $T = \langle S, \rightarrow, \hat{s}, L\rangle$, let $\mathcal{C} = \{1, \ldots, C\}$ the set of C processes (components) and let $\alpha \in \rightarrow$ be a transition. We define a mapping $proc(\alpha) = \mathcal{C}_\alpha$ such that $\emptyset \neq \mathcal{C}_\alpha \subseteq \mathcal{C}$.

Note that a transition can be synchronized and belong to multiple processes. In our benchmark sets, a transition belongs to at least one, and at most two processes.

Definition 20. Given a singleton *PTS* $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{s}_1, \ldots, \hat{s}_N\rangle, \langle l_1, \ldots, l_M\rangle\rangle$, and the set of processes $\mathcal{C} = \{1, \ldots, C\}$, each process has a *program counter* in some state slot E_i $(1 \leq i \leq N)$ of the *PTS*, containing *local states* $e \in E_i$. Each transition α is guarded by $\mathcal{G}$ and contains for each process $j \in proc(\alpha)$ a guard of the form $(E_i = e)$. We denote this guard by $pc_j \in \mathcal{G}$.

The conflicting transitions algorithm will try to find a non-trivial subset of all enabled transitions satisfying the constraints imposed by a persistent set.

Definition 21. A transition α is *conflicting* with transition β either if it starts from the same local state $(\exists pc \in guards(\alpha) \cap guards(\beta))$ or if it belongs to a parallel process $(proc(\alpha) \cap proc(\beta) = \emptyset)$ and is dependent on α (thus $(\beta, \alpha) \in D$).

The algorithm starts with a set $\mathcal{T}$ containing a single transition enabled in state s, and continues by adding to $\mathcal{T}$ all transitions that conflict with transitions in $\mathcal{T}$ until either no more conflicting transitions exist or a disabled transition is added to $\mathcal{T}$. In the latter case the algorithm fails to find a non-trivial persistent set and returns all enabled transitions, while in the case no disabled transition is encountered, the set $\mathcal{T}$ is returned. A proof of the algorithm is given in [22]. The intuition behind the algorithm is that, if a transition shares a local state, obviously executing either α or β changes the local state and disables the other. For transitions within a single process (not parallel), all transitions disabled because their local state guard evaluates to false, can in all possible traces from the current state s only become enabled if some other transition from the same process executes and changes the local state. Thus, disabled transitions within a process already selected are handled by the algorithm. When a disabled parallel transitions is added however, the algorithm doesn't know what traces lead to enabling it because it doesn't use the internal process structure.

Theorem 3. *Any set of transitions returned by the conflicting transitions algorithm is a persistent set.*

For the proof we refer to the thesis of Godefroid [22].

Proof sketch. If a disabled transition is encountered *enabled*(s) is returned, which is trivially persistent. The set $\mathcal{T}$ thus contains only enabled transitions. Suppose $\mathcal{T}$ is not a persistent set. Then there must exist a shortest sequence $\omega = s_1 \overset{\alpha_1}{\to} s_2 \dots s_n \overset{\alpha_n}{\to} s_{n+1}$ of transitions $\alpha_1, \dots, \alpha_n \notin \mathcal{T}$, such that $\alpha_1, \dots, \alpha_{n-1}$ are all independent with all transitions in $\mathcal{T}$, and α_n is dependent with at least one transition α in $\mathcal{T}$. Given is that α_n is not added to $\mathcal{T}$ by the algorithm, thus it doesn't conflict: it is not parallel and dependent, nor shares a local state. Thus α is enabled in s, and α_n shares a process, but not a local state with α. Since the local state of this process must change before α_n can become enabled after execution of $\alpha_1, \dots, \alpha_{n-1}$. Therefore, one of these transitions must change the local state of this process, and hence, is dependent with α. This contradicts that $\alpha_1, \dots, \alpha_{n-1}$ is independent with all transitions in $\mathcal{T}$, thus sequence ω can't exist. □

Example. Figure 3.3 illustrates a system configuration with two processes $P1$ and $P2$ and two shared variables x and y. In this case α and γ are enabled in $\hat{s}$ while β is disabled. Selecting α as initial transition will result in a singleton persistent set $\{\alpha\}$. If γ is selected on the other hand, it is dependent with β and there may exist a trace such that γ is executed before β, and one where γ is executed after β. But since β is disabled the algorithm can't simply try both orders. It must know which transition starting from $\hat{s}$ leads to a state where β is enabled, i.e. it must select α. This information is not available to the algorithm, therefore it returns *enabled*$(\hat{s})$.

3.3.2 Modifications to PINS

As described in the theory, the conflicting transition algorithm relies on knowledge about structural enabled/disabled transitions of a single process, the process id and the local state of a process. Therefore, this information must be passed through the PINS interface. In order for the conflicting transitions algorithm to work with the PINS interface, the following restrictions and additions are made:

1. All transition groups are singletons (return at most one next state).
2. All transition groups have a guard for enabledness/disabledness.
3. All transition groups have a set of processes associated to it.
4. Each process has a process counter in the state vector, which records the local state.

Note that these restrictions apply only for the input of the POR wrapper, not to the PINS interface in general.

The first restriction makes it possible to see a transition group as structural transition, which makes it possible to apply it in the context of a partial order reduction algorithm. The guard to determine whether a transition is enabled or disabled is needed to allow the algorithm select the enabled transition to start with. This (trivial) guard doesn't need to be explicitly implemented. The POR wrapper can execute a *next_ long* call on the front-end and see whether this results in a new state. We chose to implement this guard explicitly instead. The advantage of this is that we don't need to cache or regenerate the next states after evaluation of the guards. The process ids and local states are needed to determine the conflicting transition groups.

3.3.3 Evaluation

The implementation of the conflicting transitions algorithm successfully shows that it is possible to build a PINS wrapper for partial order reduction preserving deadlocks. From a tool designers point of view this is a very modular approach, since the POR wrapper is completely independent from the back-end search algorithms. The downside however is that it requires information about processes and local states that is not available in all possible front-ends (for example not in mCRL2 [24], and μCRL [3]). These front-ends thus require a more general solution.

Table 3.1: Conflicting transitions algorithm

model	algorithm	states		transitions		walltime [sec]
		count	percent	count	percent	
firewire_link.5		18553032	100.0	59782059	100.0	180.91
	conflict	17020279	91.7	53925483	90.2	733.47
iprotocol.4		3290916	100.0	11071177	100.0	23.52
	conflict	773823	23.5	1788848	16.2	5.31
iprotocol.5		31071582	100.0	104572634	100.0	259.32
	conflict	7364208	23.7	17066341	16.3	59.66
lann.7		160025986	100.0	944322648	100.0	1505.77
	conflict	159956844	100.0	899143733	95.2	1939.58
peterson.4		1119560	100.0	3864896	100.0	5.20
	conflict	969649	86.6	3258303	84.3	5.80
peterson.5		131064750	100.0	565877635	100.0	829.76
	conflict	127647114	97.4	520299577	91.9	945.35

Table 3.1 shows some results of the conflicting transitions algorithm applied using the DiVinE 2.2 [2] front-end on models from the BEEM [35] database. Not shown here is that of the 277 models that completed with a 8 gigabyte memory- and a 2 hour time limit, only 42 had some reduction (average about 20 percent) in the number of states. But even with 20 percent reduction, the runtime can still be significantly larger. For 198 out of 277 models, the runtime with partial order reduction was more than the runtime without partial order reduction, while for only 16 models it was less then 95 percent of the original runtime, of which 13 had less then 25 percent runtime.

All in all, the algorithm is not very effective (average runtime went up from 167.6 seconds to 219.51 seconds) except for a small set of models. Most notably, the *iprotocol*-family benefits from the algorithm, with about 76 percent reduction. For the iprotocol-family, there is also beneficial effect on memory usage, while for most of the other models the memory usage is roughly the same.

There can be multiple reasons for the lack of reduction by the algorithm. First, the dependency relation used is rather coarse. A more refined dependency relation could improve the reduction. Secondly, as Godefroid [22] mentions in his thesis, it might be useful to run the algorithm multiple times with transitions not selected in the non-trivial persistent set. The current implementation runs the algorithm only once, with the first enabled transition.

As seen in the examples, the algorithm barely benefits to any model.

3.4 Stubborn set

A more sophisticated algorithm is the *stubborn set* algorithm of Valmari [41]. As described in the thesis of Godefroid [22], this algorithm uses more knowledge of the internal structure of processes to generate better persistent sets. Although the algorithm as described by Godefroid is still based on processes, we will show that is is possible to describe a similar algorithm on a more abstract level such that the notion of processes disappears and the algorithm can be applied in a more general context.

3.4.1 Theory

A necessary enabling set for a disabled transition α is a set of transitions such that α cannot become enabled before at least one transition in the set is executed. This set depends on the current state s and does not need to be uniquely defined.

Definition 22. Given a *TS* $T = \langle S, \rightarrow, \hat{s}, L \rangle$. Let α be a transition that is disabled in state $s \in S$. A *necessary enabling set* for α in s, denoted $\mathit{NES}(\alpha, s)$, is a set of transitions such that, for all states $s' \in S$ in all sequences $s \xrightarrow{\omega} s'$ in *TS* such that α is enabled in s', ω contains at least one transition of $\mathit{NES}(\alpha, s)$.

The necessary enabling set is used to find a stubborn set, from which a persistent set can be derived.

To define a stubborn set we first define a *may be co-enabled* relation. Suppose two transitions α and β are dependent, but transition β can't be simultaneously enabled with α and α is enabled in state s. In this case, there is no trace from s to some s' where both α and β are enabled, thus β won't influence the execution of α. The may be co-enabled relation is defined as:

Definition 23. Given a *TS* $T = \langle S, \rightarrow, \hat{s}, L \rangle$. The *may be co-enabled* relation $MC \subseteq (\rightarrow \times \rightarrow)$ is a symmetric, reflexive relation, such that two transitions $\alpha, \beta \in \rightarrow$ may be co-enabled iff there is a state in the state space where they are both enabled: $\exists s \in \mathit{Reach}(T) : \alpha, \beta \in \mathit{enabled}(s) \implies (\alpha, \beta) \in MC$.

Note that it is possible to over-approximate the relation. A stubborn set is now defined as follows:

Definition 24. A set $\mathcal{T}_s$ of transitions is a *stubborn set* in a state s if $\mathcal{T}_s$ contains at least one enabled transition, and if for all transitions $\alpha \in \mathcal{T}_s$, the following conditions hold:

1. If α is disabled in s, then all transition in one necessary enabling set $\mathit{NES}(\alpha, s)$ are also in $\mathcal{T}_s$,

2. If α is enabled in s, then all transitions β that are dependent and may be co-enabled with α are also in $\mathcal{T}_s$.

Using the stubborn set, a persistent set can be found using the following theorem:

Theorem 4. *Let $\mathcal{T}$ be the set of all transitions in a stubborn set $\mathcal{T}_s$ in state s that are enabled in s. Then $\mathcal{T}$ is a persistent set in s.*

For the proof we refer to the thesis of Godefroid [22].
Proof sketch. All enabled transitions in $\mathcal{T}_s$ will be in the persistent set $\mathcal{T}$. For all disabled transitions in $\mathcal{T}_s$, all transition of one *NES* are also in $\mathcal{T}_s$, thus by definition any trace leading to a state in where the transition is enabled, will contain a transition from this necessary enabling set. For all transitions in the *NES*, again either the transitions are enabled, and thus in the persistent set, or disabled, and by induction on the length of the path have a *NES* that is also in $\mathcal{T}_s$. □

3.4.2 The algorithm

Godefroid describes in his thesis [22] an algorithm to use the stubborn set theory with simple static analysis. The stubborn set algorithm works like the conflicting transitions algorithm, but instead of building a persistent set, it builds a stubborn set. This means that it can handle disabled transitions. For disabled transitions, instead of returning the trivial persistent set $\mathit{enabled}(s)$, a $\mathit{NES}(\alpha, s)$ is added to the set and the search is continued. In the end, all enabled transitions in the stubborn set are returned. To find a necessary enabling set for a disabled transition α, Godefroid uses heuristics similar to the following two, either:

1. If α has a process that isn't in the local state in state s, ($\exists pc \in \mathit{guards}(\alpha) : pc(s) = \mathit{false}$) , then use as *NES* all transitions that change the local state this process to the state where α might be enabled, $\mathit{NES}(\alpha, s) = \{\beta \mid \exists s \in S : pc(s) = \mathit{false} \wedge pc(\beta(s)) = \mathit{true}\}$.

2. Choose a guard g of transition α that evaluates to false in s. Then, use all transitions that write to state slots the guard reads. Remember $\mathit{Ws}(\alpha)$ denotes the *write set* and $\mathit{Ts}(\alpha, g)$ the *test set* : $\mathit{NES}(\alpha, s) = \{\beta \mid \mathit{Ts}(\alpha, g) \cap \mathit{Ws}(\beta) \neq \emptyset\}$ with $\alpha, \beta \in \rightarrow$.

The first heuristic basically uses knowledge about the process structure to find all transitions β that occur before α within the process. All transitions β lead to a local state where α can be enabled. This heuristic can however only be applied when a local state guard of α is false. The other heuristic can be applied in all cases where some guard, including the local state guard, evaluates to false. In this case, all transitions β that may change the evaluation of this guard to true because transition β writes to variables used in the guard, are added to $\mathcal{T}_s$. Note that in fact Godefroid makes a distinction between guards and local states, while we see the local state simply as guard. Therefore, we can always apply the second heuristic without knowing details about the process structure, which makes the algorithm applicable in a more general setting.

Guards for PINS

To implement the stubborn set algorithm we must have access to a set of guards for each transition. Besides this we must be able to access a *may be co-enabled* relation for each pair of transitions. A guard maps a state to a boolean function. The closest functionality already available in the PINS interface is a state label, which maps a state to a natural number. Using a mapping from a natural number to boolean (for instance zero is *false*, all non-zero numbers represent *true*), a state label can be used for guard evaluation. Besides that this doesn't require changing the PINS interface, it also provides an immediate mapping from a guard to its test set by using the pins state label matrix. Since the guard state label's test set can be obtained from the pins state label matrix (thus without knowledge of the associated transitions) we introduce a new notation for the test set of a guard:

Definition 25. Given a $PTS\ P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{e}_1, \ldots, \hat{e}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$ where each transition group ($1 \leq k \leq K$) is singleton and represents a transition α_k with a set of guards $guards(\alpha_k) = \mathcal{G}_k$.

1. Define $\mathcal{G}_P = \bigcup_{k\in\{1\ldots K\}} G_k$, $G = |\mathcal{G}_P|$.

2. Each guard $g \in \mathcal{G}_P$ is represented in PINS by state labels λ_i, $(1 \leq i \leq M)$, $G \leq M$ such that a mapping function $gsl : \mathcal{G}_P \times L$ maps the guard to a state label, $\forall g \in \mathcal{G}_P : \exists i \in \{1, \ldots, M\} : gsl(g) = \lambda_i$.

3. The *test set* of a guard g and its state label $gsl(g) = \lambda_i$ is defined as:
   $Ts(*, g) = \{j \mid \exists j \in \{1, \ldots, N\} : SL_{i,j} = 1\}$ where $SL = SLM(P)$.

From now on we assume that the mapping $guards(\alpha)$ represents the mapping to state labels and is also supplied by the PINS interface.

The may be co-enabled relation MC between transitions α and β can be constructed from a may be co-enabled relation between guards. Therefore, instead of adding a may be co-enabled matrix for the transition groups, we chose to add this for the guards.

Definition 26. Given a $TS\ T = \langle S, \rightarrow, \hat{s}, L\rangle$. Two guards $g, g' \in \bigcup_{\alpha\in\rightarrow} guards(\alpha)$ of the transitions in T may be co-enabled iff there exist a state where they both evaluate to true: $\exists s \in Reach(T) : g(s) = true \wedge g'(s) = true \implies (g, g') \in MC_g$.

Note that the MC_g relation may be an over-approximation. The relation between the may be co-enabled relation of transitions and guards is now as follows:

Definition 27. $(\alpha, \beta) \in MC \leftrightarrow \forall g \in guards(\alpha), g' \in guards(\beta) : (g, g') \in MC_g$ where MC_g is the may be co-enabled relation of the guards, and MC is the may be co-enabled relation of transitions.

Finally, the write set of a transition is needed to find a *NES*. To do this we implemented more fine grained versions of the pins dependency matrix, separating it into a pins read dependency matrix and a pins write dependency matrix.

Definition 28. Let P be a *PTS* where each transition group i ($1 \leq i \leq K$) is singleton and represented by transition α_i. The refinement of the pins dependency matrix is defined such that:

1. The pins read dependency matrix $D^r_{K\times N} = DM^r(P)$ is a matrix with K rows and N columns containing $\{0, 1\}$ such that if $D^r_{i,j} = 0$ then $j \notin Rs(\alpha_i)$.

2. The pins write dependency matrix $D^w_{K\times N} = DM^w(P)$ is a matrix with K rows and N columns containing $\{0, 1\}$ such that if $D^w_{i,j} = 0$ then $j \notin Ws(\alpha_i)$.

3. The pins dependency matrix $D_{K\times N} = DM(P)$ is related to $D^r_{K\times N}$ and $D^w_{K\times N}$ such that
   $D_{i,j} = 0 \iff D^r_{i,j} = 0 \wedge D^w_{i,j} = 0$.

Stubborn sets for PINS

With these additions to PINS, it is possible to compute a *NES* for each guard beforehand. For any transition system T, there is a finite set of transitions $\alpha \in \rightarrow$ which have predefined read and write sets exported through the pins read- and write dependency matrices. Furthermore each guard has some associated state label for which the test set can be obtained through the state label matrix. Hence, if some guard fails, all information to form a *NES* is statically available, thus this *NES* can be precomputed before the exploration starts. We first define what a necessary enabling set for a guard is:

Definition 29. Given a *TS* $T = \langle S, \rightarrow, \hat{s}, L\rangle$. Let g be a guard of transition $\alpha \in \rightarrow$ that evaluates to false in state s. A *necessary enabling set* for g in s, denoted $\mathit{NES}(g, s)$, is a set of transitions such that, for all sequences $s \overset{\omega}{\rightarrow} s'$ in *TS* where g evaluates to true in s', ω contains at least one transition of $\mathit{NES}(g, s)$.

In our implementation, we calculate beforehand the following sufficient (unique) *NES* for a guard in any state:

$$\forall s \in S : \mathit{NES}(g, s) = \{\beta \mid \exists \beta \in \rightarrow : \mathit{Ts}(*, g) \cap \mathit{Ws}(\beta) \neq \emptyset\}$$

Note that the information of the test set and write set is extracted from the pins write dependency matrix and the pins state label matrix. There are now several necessary enabling sets for any transition α, namely one for each disabled guard. In our implementation we collect these into a set. This set does depend on the current state s and can only be assembled during the search.

Definition 30. Given a *TS* $T = \langle S, \rightarrow, \hat{s}, L\rangle$. Let $\alpha \in \rightarrow$ be a transition and let $\mathit{guards}(\alpha)$ be the set of guards of transition α. Now $\mathit{GNES}(\alpha, s)$, is a set of necessary enabling sets for each guard of transition α that evaluates to false in state s. $\mathit{GNES}(\alpha, s) = \{\mathit{NES}(g, s) \mid \exists g \in \mathit{guards}(\alpha) : g(s) = \mathit{false}\}$

The final algorithm looks like this.

```
function stubborn_set($s$)
    $T_{work} = \{\alpha\}$ such that $\alpha \in enabled(s)$
    $T_s = \emptyset$
    while $T_{work} \neq \emptyset$ do
        $\beta \in T_{work}$, $T_{work} = T_{work} - \beta$, $T_s = T_s \cup \{\beta\}$
        if $\beta \in enabled(s)$ then
            $T_{work} = T_{work} \cup \{\alpha \in \rightarrow \mid (\alpha, \beta) \in D \cap MC\} \setminus T_s$
        else
            $T_{work} = T_{work} \cup (nes \in GNES(\beta, s)) \setminus T_s$
    return $T_s \cap enabled(s)$
```

Algorithm 1: A stubborn set algorithm for PINS

3.4.3 Modifications to PINS

The stubborn set algorithm for PINS is now a generalized version of the stubborn set algorithm as proposed by Godefroid [22]. It does not require knowledge of processes, but uses guards instead.

For the implementation, the following restrictions and additions have been made to the interface:

1. All transition groups are singletons (return at most one next state).
2. All transition groups are associated to a set of guard state labels.
3. A may be co-enabled matrix for guards MC_g is added.

The singleton restriction again is needed to see the transition group as structural transition. Enabledness of a transition can be deduced from the conjunction of the guards. The guard state labels have the same requirements as normal state labels with respect to the pins state label matrix. In addition the may be co-enabled matrix provides an abstract insight on the composition of the transition system.

Table 3.2: Stubborn set algorithm

model	algorithm	states		transitions		walltime [sec]	memory [vsize]
		count	percent	count	percent		
anderson.6		18206917	100.0	86996322	100.0	119.78	485468KB
	conflict	18206917	100.0	86996322	100.0	150.10	485524KB
	stubborn-set	13510727	74.2	58399022	67.1	115.66	327672KB
at.6		160589600	100.0	697096560	100.0	808.42	3202196KB
	conflict	160589600	100.0	697096560	100.0	1278.33	3202360KB
	stubborn-set	160232958	99.8	691058033	99.1	1890.70	3202308KB
brp2.6		5742313	100.0	9058624	100.0	15.81	221128KB
	conflict	5742313	100.0	9058624	100.0	23.41	221180KB
	stubborn-set	5204363	90.6	7970060	88.0	26.21	200688KB
exit.3		2356294	100.0	7047332	100.0	27.42	99492KB
	conflict	2356294	100.0	7047332	100.0	37.42	99692KB
	stubborn-set	2356235	100.0	7047186	100.0	37.33	99560KB
extinction.4		2001372	100.0	7116790	100.0	19.81	81368KB
	conflict	2001372	100.0	6444463	90.6	31.92	81760KB
	stubborn-set	1382570	69.1	2647195	37.2	15.61	73508KB
firewire_link.5		18553032	100.0	59782059	100.0	180.91	468188KB
	conflict	17020279	91.7	53925483	90.2	733.47	473356KB
	stubborn-set	18538624	99.9	58933708	98.6	1977.71	470092KB
iprotocol.7		59794192	100.0	200828479	100.0	544.95	1695048KB
	conflict	13924014	23.3	32341127	16.1	116.98	349480KB
	stubborn-set	41078037	68.7	101990547	50.8	387.95	1046772KB
lann.4		966855	100.0	3189852	100.0	6.21	80116KB
	conflict	954288	98.7	2981886	93.5	8.71	80216KB
	stubborn-set	391131	40.5	846273	26.5	2.40	59052KB
leader_election.6		35777100	100.0	233195212	100.0	3850.76	2289068KB
	conflict	35777100	100.0	233195212	100.0	4183.44	2290088KB
	stubborn-set	32400204	90.6	44693743	19.2	944.46	2081964KB
leader_filters.7		26302351	100.0	91692858	100.0	181.43	839208KB
	conflict	26074330	99.1	89458207	97.6	242.85	839284KB
	stubborn-set	26074330	99.1	89458207	97.6	307.38	839196KB
peg_solitaire.2		187636299	100.0	1487175000	100.0	4613.12	5309800KB
	conflict	187636299	100.0	1487175000	100.0	5710.90	5310832KB
	stubborn-set						
peterson.6		174495861	100.0	747072150	100.0	1091.93	3218568KB
	conflict	154606103	88.6	631631896	84.5	1150.26	3218544KB
	stubborn-set	153103069	87.7	603909316	80.8	1226.22	3218564KB
phils.8		43046720	100.0	459165008	100.0	960.12	1817408KB
	conflict	43046720	100.0	459165008	100.0	1080.59	1817528KB
	stubborn-set	14733803	34.2	56023811	12.2	154.52	548644KB

3.4.4 Evaluation

Table 3.2 contains the results of the stubborn set algorithm and of the conflicting transitions algorithm. In general, the stubborn set is able to reduce the state space more then the conflicting transitions algorithm (101 of the 277 models have some reduction, average reduction of these models is about 20 percent too). The more complicated stubborn set algorithm thus achieves more reduction, at the cost of spending more time to compute a persistent set. As shown in Table 3.2, the *at.6* model with about 0.2 percent reduction, takes more then 2 times as long to complete. Some models even fail to run within the given time and memory constraints (*peg_solitaire.2*). This factor is not constant (see *exit.3*) but seems to depend mostly on the number of transition groups. Furthermore, even with about 30 percent reduction (*anderson.6*), the runtime doesn't improve using the implemented stubborn set algorithm. Interesting to see in Table 3.2 is that a) the algorithm never uses more memory b) a reduction in the number of states is more or less proportional to the memory used c) the runtime benefits most from a reduction in the number of transitions (*leader_election.6* v.s. *brp2.6*). Therefore, we might conclude that the stubborn set algorithm always helps to reduce the memory needed. The benefit in runtime can be explained because tree compression [4] is used. Packing and unpacking a compressed state takes relatively long, therefore when the number of transitions is less, less states are unpacked resulting in better runtimes. This effect likely becomes less prominent when states are stored in a hash table. The stubborn set algorithm is not always better then the conflicting transitions algorithm (see *iprotocol.7*). This may happen because the algorithm uses heuristics and will not necessarily find the best solution.

3.5 Better necessary enabling sets

The stubborn set algorithm doesn't yet give satisfactory reduction. The algorithm of Godefroid [22] uses fine grained information about the structure of a process and operations used in transition effects to find a small *NES*, while the adapted algorithm simply marks all transitions that write to some part of the state vector. This is a crude overestimation which makes the necessary enabling sets too large for a good reduction.

Example. Suppose a process has 10 sequential transitions $\alpha_1, \ldots, \alpha_{10}$. A *NES* for a transition α_n currently not in its local state would, with exception of the first, be $\{\alpha_{n-1}\}$. But since all transitions write to the same local state (same part of the state vector). The *NES* selected by the adapted algorithm would be $\{\alpha_1, \ldots, \alpha_{10}\}$.

A simple solution to both these problems is to let the front-end provide a more detailed *NES* for each guard. Front-ends that use processes can now statically precompute a *NES* and use all underlying information of process structure and operations available. Precomputation of a *NES*, especially the local state, is still rather straightforward, and not nearly as involved as writing a partial order reduction tool. With this optimization, the stubborn set algorithm for PINS should behave the same as the stubborn set algorithm of Godefroid [22], and give a decent reduction. In case the front-end doesn't provide the *NES* for a guard, it is possible to fall back on the *NES* as computed in the previous section. Hence, the more information the front-end provides, the better the reduction will become.

3.5.1 Modifications to the algorithm

The PINS stubborn set algorithm is not changed, only the preprocessing is. In the previous section, a *NES* was precomputed for each guard before the exploration took place. The new algorithm will still do this, but intersect it with the *NES* provided by the front-end. We denote the provided *NES* by $NES_{fe}(g)$.

$$\forall s \in S : NES(g, s) = \{\beta \mid \exists \beta \in \rightarrow : Ts(*, g) \cap Ws(\beta) \neq \emptyset\} \cap NES_{fe}(g)$$

The *NES* that is provided by the front-end must be a valid necessary enabling set.

Definition 31. Given a *TS* $T = \langle S, \rightarrow, \hat{s}, L\rangle$, the front-end provided *NES* for each of the G guards in the *TS* is a $G \times K$ matrix $NES_{i,j}$ $(1 \leq i \leq G), (1 \leq j \leq K)$ such that $NES_{i,j} = 1$ iff transition j is in the necessary enabling set of guard i, computed by the front-end.

Besides the addition of the provided *NES*, a further change has been made to the algorithm. Instead of taking the first enabled transition and computing the stubborn set once, multiple stubborn sets are computed, one for each enabled transition. Then the stubborn set with the smallest persistent set is emitted. Note that there is some non-determinism involved in computing the stubborn set. Starting the search twice from the same enabled transition may result in a different set because there is no selection criteria for choosing the *NES* to add. Starting the search from a different enabled transition the implementation likely uses a different *NES* for a transition and therefore might find a different stubborn set. There is however no guarantee that this is the case. The algorithm does not compute all possible stubborn sets, nor does it exclude doing possibly redundant computations.

3.5.2 Modifications to PINS

Since the only change compared to the stubborn set algorithm is that the front-end now provides the *NES*, the changes to the PINS interface are the following.

1. All transition groups are singletons (return at most one next state).
2. All transition groups are associated to a set of guard state labels.
3. A may be co-enabled matrix for guards MC_g is added.
4. A NES_{fe} matrix for each guard state label is added.

The NES_{fe} matrix is a matrix that assigns to each guard the transition groups that can enable it.

3.5.3 Evaluation

Because two changes have been made to the algorithm, they must be evaluated separately. Table 3.3 contains the results of the stubborn set algorithm, the stubborn set algorithm with the *NES* provided by the front-end (nes1), the stubborn set algorithm executed for each enabled transition (stubborn-all) and the stubborn set algorithm with the *NES* provided by the front-end executed for each enabled transitions (nes). For most of the models in the BEEM database, the additional information provided by the front-end (nes1) results in a reduction more or less the same as without it (stubborn-set). For some models however the extra information gives a significantly more reduction (*iprotocol, production_ cell*).

The second modification to the algorithm is running the stubborn set algorithm for each enabled transition (stubborn-all). This results in a far better reduction. Comparing this to the algorithm that has the extra information provided by the front-end (nes), we see that in most cases the results are similar in terms of reduction. Hence, the extra information provided by the front-end only helps in some cases (*iprotocol, phils*). For some models the extra work pays off (*extinction.4*) while for others there is just more overhead (*at.6, needham.4, lamport_ nonatomic.5*). Interestingly, the stubborn-all and nes algorithm are executing exactly the same code. In some cases, they both give the same result, but the nes algorithm is faster (*lann.7*). The extra information provided by the front-end thus appears to improve the performance of the algorithm instead of the reduction. This is likely because the necessary enabling sets are smaller, thus require less work.

Based on Table 3.3, we can conclude that most of the difference in reduction between algorithm stubborn-set and algorithm nes comes from executing the stubborn set algorithm multiple times, not from extra information provided by the front-end.

3.6 Necessary disabling sets

In the stubborn set algorithm, we observe that whenever a disabled transition is added to the stubborn set, a *NES* must also be added and chances are that this set too contains disabled transitions. This cascading effect might cause a lot of transition groups to be selected, which in turn gives rise to a lots of dependencies to other transition groups. Ideally, we would limit this cascade. Imagine the following scenario with processes such as they are defined for the conflicting transitions algorithm:

Table 3.3: Stubborn-set with NES algorithm

model	algorithm	states		transitions		walltime [sec]	memory [vsize]
		count	percent	count	percent		
anderson.6		18206917	100.0	86996322	100.0	119.78	485468KB
	stubborn-set	13510727	74.2	58399022	67.1	115.66	327672KB
	nes1	13510727	74.2	58399022	67.1	109.66	327672KB
	stubborn-all	4858411	26.7	16898226	19.4	71.94	171448KB
	nes	4858411	26.7	16898226	19.4	70.75	171448KB
at.6		160589600	100.0	697096560	100.0	808.42	3202196KB
	stubborn-set	160232958	99.8	691058033	99.1	1890.70	3202308KB
	nes1	157642038	98.2	663556258	95.2	1539.60	3202308KB
	stubborn-all	148884155	92.7	616793155	88.5	4819.76	3202312KB
	nes	148826634	92.7	604298885	86.7	4265.48	3202308KB
brp.6		42728113	100.0	89187437	100.0	152.90	946328KB
	stubborn-set	42683473	99.9	89032341	99.8	310.88	946188KB
	nes1	42090177	98.5	86473543	97.0	265.95	946188KB
	stubborn-all	40551877	94.9	80120333	89.8	842.29	946532KB
	nes	33682079	78.8	54392322	61.0	335.10	737516KB
extinction.4		2001372	100.0	7116790	100.0	19.81	81368KB
	stubborn-set	1382570	69.1	2647195	37.2	15.61	73508KB
	nes1	1509720	75.4	3026478	42.5	18.92	81628KB
	stubborn-all	327647	16.4	479654	6.7	10.21	55168KB
	nes	284469	14.2	409965	5.8	6.41	54624KB
iprotocol.7		59794192	100.0	200828479	100.0	544.95	1695048KB
	stubborn-set	41078037	68.7	101990547	50.8	387.95	1046772KB
	nes1	9720456	16.3	19201940	9.6	79.08	267676KB
	stubborn-all	12996668	21.7	29115750	14.5	213.93	349452KB
	nes	9667540	16.2	19122279	9.5	124.09	267736KB
krebs.4		1047405	100.0	5246321	100.0	6.61	76436KB
	stubborn-set	1047245	100.0	5153420	98.2	8.71	76268KB
	nes1	1046921	100.0	4998014	95.3	8.21	76268KB
	stubborn-all	897482	85.7	2922675	55.7	10.81	65348KB
	nes	897653	85.7	2922770	55.7	10.71	65348KB
lamport_nonatomic.5		95118524	100.0	505734961	100.0	1034.97	2855088KB
	stubborn-set	95118524	100.0	505734961	100.0	3602.42	2855484KB
	nes1	95118524	100.0	499299259	98.7	4533.82	2855484KB
	stubborn-all						
	nes						
lann.7		160025986	100.0	944322648	100.0	1505.77	3206496KB
	stubborn-set	149890075	93.7	509618588	54.0	1159.35	3206484KB
	nes1	149910575	93.7	510345108	54.0	1238.79	3206592KB
	stubborn-all	89028594	55.6	214370864	22.7	1711.13	1982388KB
	nes	89028594	55.6	214370864	22.7	1523.40	1982388KB
leader_election.6		35777100	100.0	233195212	100.0	3850.76	2289068KB
	stubborn-set	32400204	90.6	44693743	19.2	944.46	2081964KB
	nes1	32691072	91.4	45096417	19.3	1030.36	2082372KB
	stubborn-all	280446	0.8	356341	0.2	15.01	62324KB
	nes	280446	0.8	356341	0.2	13.31	62316KB
needham.4		6525019	100.0	22203081	100.0	41.13	163544KB
	stubborn-set	6288225	96.4	20217909	91.1	98.47	163756KB
	nes1	6328616	97.0	20396912	91.9	143.41	163692KB
	stubborn-all	6525019	100.0	22202721	100.0	720.93	163700KB
	nes	5918126	90.7	18399892	82.9	450.00	163692KB
phils.8		43046720	100.0	459165008	100.0	960.12	1817408KB
	stubborn-set	14733803	34.2	56023811	12.2	154.52	548644KB
	nes1	14448128	33.6	27504194	6.0	91.56	548724KB
	stubborn-all	504639	1.2	1086478	0.2	11.01	70544KB
	nes	55575	0.1	92945	0.0	1.10	48424KB
production_cell.6		14520700	100.0	45593810	100.0	78.86	326336KB
	stubborn-set	14520700	100.0	45557720	99.9	149.39	326064KB
	nes1	8160831	56.2	19069040	41.8	63.84	253060KB
	stubborn-all	6218940	42.8	9615480	21.1	95.48	173800KB
	nes	1716280	11.8	2273470	5.0	26.82	84896KB
synapse.5		83263	100.0	189639	100.0	0.50	48464KB
	stubborn-set	83263	100.0	189639	100.0	1.60	48952KB
	nes1	83071	99.8	188439	99.4	2.30	48952KB
	stubborn-all	83263	100.0	189639	100.0	8.21	48964KB
	nes	80723	96.9	174123	91.8	5.10	48912KB
train-gate.7		50199556	100.0	106056460	100.0	257.21	962644KB
	stubborn-set	50199556	100.0	106056460	100.0	484.72	962496KB
	nes1	50199556	100.0	105837268	99.8	527.05	962496KB
	stubborn-all	26785155	53.4	55720696	52.5	801.49	751700KB
	nes	16921595	33.7	37089426	35.0	390.21	477928KB

Example. Suppose a process i has a disabled transition α with two guards $guards(\alpha) = \{pc_i, g\}$. The process counter guard pc and another guard checking a shared variable v, $g =$ "$(v = 1)$". Moreover, let $pc(s) = true$ and $g(s) = false$, i.e. the transition is in its local state but has a guard that evaluates to false because the value of v is not 1 in s. Suppose α is the only transition which has the local state guard pc. The best *NES* that can be selected for this transition would be all transitions that can assign 1 to v. It could be the case that process i has a transition β which has an effect that assigns 1 to v, and thus would be selected as *NES*, and hence this transition is disabled because it belongs to process i and process i is in another local state ($pc(s) = true$). If β now has a guard with a *NES* that contains transition groups of other processes, this could cascade to even more transition groups later on in the selection, while in fact, it could also be argued that α is a *NES* for β because β can't become enabled without a local state change, and α is the only transition that can do this in s.

In the above example, a disabled transition α that is already selected for the stubborn set can serve as a *NES* for other transitions β, since the local state of process i must change and α is the only transition able to do this. An "optimal" scenario for the stubborn set algorithm in the above example would be that the *NES* for β would give only disabled transitions of process i, and for these disabled transitions again a *NES* with only transitions of process i would be selected. This would go on until all disabled transitions of process i that can occur before β are selected, including transition α. The only way that this cascade could stop before all these transitions are added is when an intermediate transition selects a guard that is not guarding the local state, and for this guard the *NES* has already been selected. It is however more likely, that such a guard would trigger transitions in other processes causing the *NES* cascade to spread over more transition groups initiating more cascades. Therefore it would be nice to shortcut this and detect that in case there is a single disabled transition for which the local state guard evaluates to true, it can serve as a *NES* for other transitions in this process. Even if multiple transitions have g_i as guard, as soon as they are all selected, they together form a *NES*. Typically, it is the case that for any local state the number of outgoing transitions is small. Since a small *NES* is empirically a good heuristic for a small persistent set, which in turn is a good heuristic [22] for a good reduction, using this will likely pay off.

The reason that α can be used as *NES* for β in the same process, is that β can't become enabled without α executing first. This is because the local state guards of α and β are different and disjoint, thus can't be enabled at the same time. Since α is the only transition of process i that can execute, it is the only transition that can disable the local state guard of α. Therefore, it must execute before the local state guard of β can possibly become enabled. We can extend this notion to guards in general, without the need for processes.

Take all transitions β that use some guard g' that can't be co-enabled with a guard g that currently is enabled in state s. We can now conclude that all transitions β must be disabled, and can't become enabled before the enabled guard g becomes disabled. Hence, the set of transitions needed to disable guard g is a necessary enabling set for all transitions β.

We define this set similarly to the necessary enabling set:

Definition 32. Given a *TS* $T = \langle S, \rightarrow, \hat{s}, L\rangle$. Let g be a guard of transition α that evaluates to true in state s. A *necessary disabling set* for g in s, denoted $NDS(g, s)$, is a set of transitions such that, for all sequences $s \xrightarrow{\omega} s'$ in *TS* where g evaluates to false in s', ω contains at least one transitions of $NDS(g, s)$.

In our implementation, we calculate beforehand the following sufficient (unique) *NDS* for a guard in any state:

$$\forall s \in S : NDS(g, s) = \{\beta \mid \exists \beta \in \rightarrow : Ts(*, g) \cap Ws(\beta) \neq \emptyset\} \cap NDS_{fe}$$

Note that the *NDS* is equivalent to the default *NES* if the front-end doesn't provide a more detailed version.

As with the necessary enabling sets, we group the necessary disabling sets per transitions:

Definition 33. Given a *TS* $T = \langle S, \rightarrow, \hat{s}, L\rangle$. Let $\alpha \in \rightarrow$ be a transition and let $guards(\alpha)$ be the set of guards of transition α. Now $GNDS(\alpha, s)$, is a set of necessary disabling sets for each guard of transition α that evaluates to true in state s. $GNDS(\alpha, s) = \{NDS(g, s) \mid \exists g \in guards(\alpha) : g(s) = true\}$.

3.6.1 The algorithm

To define the algorithm, we group the necessary enabling sets, i.e. the union of the necessary enabling sets and the necessary disabling sets that can't be co-enabled:

Definition 34. A *necessary set* (NS) for a transition α is defined as $\mathit{GNS}(\alpha, s) = \mathit{GNES}(\alpha, s) \cup \{\mathit{nds} \mid \exists \beta \in \rightarrow : (\alpha, \beta) \notin \mathit{MC} \wedge \mathit{nds} \in \mathit{GNDS}(\beta, s)\}$.

The necessary set is a simply a set of necessary enabling sets, just using the disabled guards as well as the enabled guards. Because of this addition, there are more necessary enabling sets to choose from and therefore it is more likely that it is possible to select a small set. To select a small set, another simple improvement over the previous approach is to predict the effect of using a certain *NES* on the resulting stubborn set.

We will define a selection heuristic to find the "best" one. The best heuristic for selecting a persistent set is to select a small persistent set [22]. Therefore we want a stubborn set with the least possible amount of enabled transitions. Furthermore, if no enabled transitions are found, we try to select the smallest set because this will add the least amount of work (i.e. other necessary enabling sets). We achieve this by penalizing the transitions in the necessary enabling sets.

The heuristic h used to select a *NES* works as a cost function:

1. For each disabled transition in the *NES* and not in $T_{work} \cup T_s$ add 1 to the costs

2. For each enabled transition in the *NES* and not in $T_{work} \cup T_s$ add n to the costs

The proposed algorithm will select the *NES* with the lowest cost on the heuristic function. Because enabled transitions weight more then disabled transitions, the algorithm is less likely to select a *NES* with enabled transitions, thus more likely to find a small persistent set.

To select a good necessary enabling set, we define a set of the "best" available necessary enabling sets:

$$\mathit{best\_nes} = \{\mathit{nes} \mid \exists \mathit{nes} : \forall \mathit{nes}' \in \mathit{GNS}(\beta, s) \mid h(\mathit{nes}) <= h(\mathit{nes}')\}$$

The algorithm is now defined as follows:

```
function $stubborn\_set\_ns(s)$
    $T_{work} = \{\alpha\}$ such that $\alpha \in enabled(s)$
    $T_s = \emptyset$
    while $T_{work} \neq \emptyset$ do
        $\beta \in T_{work}$, $T_{work} = T_{work} - \beta$, $T_s = T_s \cup \{\beta\}$
        if $\beta \in enabled(s)$ then
            $T_{work} = T_{work} \cup \{\beta \in\rightarrow \mid (\alpha, \beta) \in D \cap MC\} \setminus T_s$
        else
            $T_{work} = T_{work} \cup (nes) \setminus T_s$ where $nes \in best\_nes$
    return $T_s \cap enabled(s)$
```

Algorithm 2: A stubborn set nes/nds algorithm for PINS

The actual implementation of the algorithm knows all *NES* before it starts the search, and can associate precomputed initial values for the heuristic function before the exploration starts since T_{work} is empty. These values are stored for each *NES*. In total, there are at most the number of guards times two (enabled/disabled) different *NES*.

Then, the algorithm will evaluate all the guard state labels and for each enabled transition set up a special *search context*. This context is initialized by copying the initial values of the heuristic functions. During the search, each time the algorithm adds a transition to the set T_{work}, the associated heuristic values are updated. The update will cost at most linear time (two times the number of guards). The lookup of $h(\mathit{nes})$ can occur in constant time. The algorithm is a best-first search that will search in the search context with the least amount of enabled transitions in it. If it finds a stubborn set with one enabled transition it will stop the search directly. Otherwise it will continue until no smaller persistent set can be found.

3.6.2 Modifications to PINS

The only extra modification on top of the previous algorithm is the addition of the extra NDS_{fe} matrix.

1. All transition groups are singletons (return at most one next state).

2. All transition groups are associated to a set of guard state labels.

3. A may be co-enabled matrix for guards MC_g is added.

4. A NES_{fe} matrix for each guard state label is added.

5. A NDS_{fe} matrix for each guard state label is added.

3.6.3 Evaluation

The effect of the additional NDS_{fe} matrix is shown in Table 3.4. The table contains the results of the previous algorithm (nes), the nes/nds algorithm where the NES/NDS information is not provided by the front-end (nes/nds*), and the nes/nds algorithm where the front-end provides the NES/NDS matrix. As can be seen from the table, the nes/nds algorithm can help to reduce the state space (*needham.4, public_subscribe.4*).

There are however also cases where it performs similar to the nes algorithm (*anderson.6*). As with the last algorithm, the extra information provided by the front-end doesn't seem to have the biggest impact on the reduction. Most of the models run with the algorithm nes/nds*, where the front-end didn't provide the extra information, result in similar reductions as the nes/nds algorithm with the extra information. The small variations are likely caused by a minor variation in the choice of a *NES* somewhere during the search. Again, the extra information does seem to help for the runtime. This might be because it needs to update less values for the heuristic function h or finds a persistent set of size one faster.

Geldenhuys, Hansen and Valmari [21] recently presented empirical upper bounds for partial order reduction using Ample sets [28, 26], based on the best possible (conditional) dependency relation derived after computation of the full state space. Since these reductions are computed *after* generation of the full state space, it is likely that they never occur in practice. The experiments they performed were done using models from the BEEM [35] database as well, and are directly comparable to the reductions found by our partial order reduction algorithm. The ample set implementation in [21] is roughly comparable to the partial order reduction implemented in SPIN [28]. The maximal reduction based on the full state space, as computed in [21] (AMPLE2, D_f/R_f) is shown next to reduction achieved based on static data by the nes/nds algorithm.

Note that the stubborn set nes/nds algorithm uses only static, precomputed data and in 5 of the cases shown in Table 3.5, still outperforms the maximum reduction achievable using the ample set algorithm. Furthermore note that the results in Table 3.5 are ordered using the maximum possible reduction of the models in the BEEM database using the ample set algorithm, not by the performance of the stubborn set nes/nds algorithm. The reason that the nes/nds algorithm is able to achieve better reduction is that the ample set algorithm selects as persistent set the transitions of either one or all processes in a system. It will never find a subset some containing some of the processes. The stubborn set nes/nds algorithm on the other hand is able to select such a subset and therefore can outperform the ample set algorithm in terms of reduction.

3.6.4 Future optimizations

Some remarks must be made regarding the extra overhead.

1. The models use the *next_long* functions instead of *next_all*, which is slower. This is because the *next_long* callback needs to check guard conditions on each call while *next_all* can combine these checks and perform them only once for a set of transition groups.

2. The guard evaluation is performed twice. The *next_long* call performs the guard evaluation but this has already been performed by the exported guards.

Table 3.4: Stubborn set with NES/NDS algorithm

model	algorithm	states		transitions		walltime [sec]	memory [vsize]
		count	percent	count	percent		
anderson.6		18206917	100.0	86996322	100.0	119.78	485468KB
	nes	4858411	26.7	16898226	19.4	70.75	171448KB
	nes/nds*	4858411	26.7	16898226	19.4	108.67	171480KB
	nes/nds	4858411	26.7	16898226	19.4	94.26	171476KB
at.6		160589600	100.0	697096560	100.0	808.42	3202196KB
	nes	148826634	92.7	604298885	86.7	4265.48	3202308KB
	nes/nds*						
	nes/nds	148690835	92.6	595369635	85.4	6299.36	3202360KB
brp.6		42728113	100.0	89187437	100.0	152.90	946328KB
	nes	33682079	78.8	54392322	61.0	335.10	737516KB
	nes/nds*	22735728	53.2	30084951	33.7	285.73	468404KB
	nes/nds	21985368	51.5	29001591	32.5	213.33	466304KB
collision.4		41465543	100.0	113148818	100.0	172.11	930996KB
	nes	35080093	84.6	64504078	57.0	476.90	723744KB
	nes/nds*	10712473	25.8	16455778	14.5	167.19	295924KB
	nes/nds	10712473	25.8	16455778	14.5	139.29	295920KB
cyclic_scheduler.4		473414	100.0	1736712	100.0	3.60	58364KB
	nes	395834	83.6	1066338	61.4	7.21	53952KB
	nes/nds*	29971	6.3	32276	1.9	0.50	47924KB
	nes/nds	29971	6.3	32276	1.9	0.50	47920KB
driving_phils.4		265262511	100.0	695149567	100.0	1671.19	5303484KB
	nes	265262511	100.0	413840931	59.5	2898.90	5303548KB
	nes/nds*	265262511	100.0	413840931	59.5	7026.39	5303752KB
	nes/nds	182293923	68.7	302267279	43.5	4529.43	5303584KB
exit.3		2356294	100.0	7047332	100.0	27.42	99492KB
	nes	2356235	100.0	7047186	100.0	111.07	99560KB
	nes/nds*	2356207	100.0	7047158	100.0	132.61	99712KB
	nes/nds	2356207	100.0	7047158	100.0	105.67	99692KB
extinction.4		2001372	100.0	7116790	100.0	19.81	81368KB
	nes	284469	14.2	409965	5.8	6.41	54624KB
	nes/nds*	96470	4.8	125532	1.8	4.70	49588KB
	nes/nds	82265	4.1	104273	1.5	2.70	49600KB
firewire_tree.5		3807023	100.0	18225703	100.0	580.23	258920KB
	nes	3240623	85.1	15383113	84.4	2582.06	246172KB
	nes/nds*	348633	9.2	495815	2.7	674.94	137564KB
	nes/nds	348709	9.2	495932	2.7	542.74	137080KB
iprotocol.7		59794192	100.0	200828479	100.0	544.95	1695048KB
	nes	9667540	16.2	19122279	9.5	124.09	267736KB
	nes/nds*	9640730	16.1	13114368	6.5	141.59	267412KB
	nes/nds	9640730	16.1	13114368	6.5	112.31	267408KB
krebs.4		1047405	100.0	5246321	100.0	6.61	76436KB
	nes	897653	85.7	2922770	55.7	10.71	65348KB
	nes/nds*	870660	83.1	2801870	53.4	13.31	65416KB
	nes/nds	874192	83.5	2807131	53.5	11.01	65412KB
lamport_nonatomic.5		95118524	100.0	505734961	100.0	1034.97	2855088KB
	nes						
	nes/nds*						
	nes/nds	40472911	42.5	105095256	20.8	7033.05	1340124KB
leader_election.6		35777100	100.0	233195212	100.0	3850.76	2289068KB
	nes	280446	0.8	356341	0.2	13.31	62316KB
	nes/nds*	232398	0.6	301559	0.1	18.11	61296KB
	nes/nds	232398	0.6	301559	0.1	13.51	61252KB
needham.4		6525019	100.0	22203081	100.0	41.13	163544KB
	nes	5918126	90.7	18399892	82.9	450.00	163692KB
	nes/nds*	2040051	31.3	3367460	15.2	142.00	81312KB
	nes/nds	2050399	31.4	3377808	15.2	95.57	81284KB
pgm_protocol.8		3069390	100.0	7125121	100.0	62.55	173060KB
	nes	1923190	62.7	2722351	38.2	63.06	132084KB
	nes/nds*	1334438	43.5	1634677	22.9	74.05	114780KB
	nes/nds	1310300	42.7	1543991	21.7	48.54	114628KB
phils.8		43046720	100.0	459165008	100.0	960.12	1817408KB
	nes	55575	0.1	92945	0.0	1.10	48424KB
	nes/nds*	69056	0.2	117936	0.0	1.20	49692KB
	nes/nds	722	0.0	976	0.0	0.10	3688KB
public_subscribe.4		1846603	100.0	6087556	100.0	13.01	84776KB
	nes	344089	18.6	637134	10.5	9.01	55888KB
	nes/nds*	206013	11.2	334536	5.5	23.32	53848KB
	nes/nds	206013	11.2	334536	5.5	9.31	53704KB

Table 3.5: Ample set v.s. stubborn set NES/NDS

model	ample	stubborn-set nes/nds
cyclic_scheduler.1	1.19	1.2
mcs.4	4.13	16.4
firewire_tree.1	6.25	99.3
phils.3	10.84	10.8
mcs.1	18.35	88.6
anderson.4	22.78	46.5
iprotocol.2	25.97	15.7
mcs.2	34.45	64.6
phils.1	47.50	47.5
fwlink.2	50.66	18.8
krebs.1	51.09	93.4
leader_election.3	54.26	5.5
telephony.2	59.55	99.9
leader_election.1	60.52	10.4
szymanski.1	62.98	71.5
production_cell.2	63.12	22.1

The remarks show that there is some opportunity to optimize the runtimes for our partial order reduction algorithms. We however take this penalty for granted and try to optimize the reduction instead.

All the partial order reduction strategies described here were optimized to get more reduction in the number of states and transitions. As can be seen from experiments shown, this is not always beneficial for the runtime of the algorithm. There is however still some room to improve this, some suggested optimizations are:

1. Move evaluation of guards entirely to the back-end algorithms, such that the *next_long* call doesn't need to perform them anymore. Alternatively one might extend the PINS interface with a call similar to *next_long* with doesn't perform the guard evaluation.

2. When large persistent sets are found, the partial order reduction layer might call *next_all* instead of *next_long*, and filter the resulting successor states. This would require the front-end to provide the transition group of each generated successor state.

3. The precomputed *NES* of several guards might be exactly the same. By preprocessing one can merge multiple necessary enabling sets into one. Doing this leads to the same reduction however requires less work when updating the heuristic function. If updating the heuristic function appears to costs too much time, alternatively several necessary enabling sets can be merged together when some *NES* subsumes another, or just by taking the union of the sets. Doing this will likely result in less reduction but might speed up the search algorithm.

4. At the moment the search algorithm will directly stop whenever it finds a persistent set with just one transition in it. Adjusting this threshold to two or more will likely be beneficial for the runtime, while trading in some of the reduction. An alternative could be to dynamically adjust this, and maybe other search parameters based on the effectiveness of the algorithm in other states (i.e. on the fly).

5. Some guards are dependent on each other, for example guard $i = 1$ and $a[i] = 3$. The second guard can not be evaluated without the first guard. Therefore, these two guards must be combined into a single guard $i = 1 \wedge a[i] = 3$. It may be beneficial to the reduction to, besides this guard, also include the $i = 1$ guard. This results in more necessary enabling sets, making it more likely that a small *NES* is found.

6. Currently the algorithms is run once for each enabled transition in some state s. When the state vector is long, there are likely a lot of enabled transitions. Geldenhuys, Hansen and Valmari [21] suggest that running the ample set on a random subset of the enabled transitions will result in similar reductions. It might be worthwhile to investigate this and see whether taking a semi-random (based

on the current state) subset of the enabled transitions results in similar reduction yet improves the runtime. The number of semi-random enabled transitions can also be determined on the fly.

3.7 Partial order reduction with non-deterministic transitions

Until now, we assumed that transition groups must be singleton (return at most one successor state) to represent deterministic structural transitions. This restriction is a real limitation to the PINS interface, and it would be nice if it could be removed.

Theorem 5. *The proposed algorithms are also correct without the singleton constraint*

Proof. Given a *PTS* $P = \langle\langle E_1, \ldots, E_N\rangle, \langle\rightarrow_1, \ldots, \rightarrow_K\rangle, \langle\hat{e}_1, \ldots, \hat{e}_N\rangle, \langle\lambda_1, \ldots, \lambda_M\rangle\rangle$. Let each transition group $\rightarrow_i$ $(1 \leq i \leq K)$ be *guarded* by a non-empty set of guards $\mathcal{G}_i$. Let a transition group j return two successor states $s \overset{\alpha}{\rightarrow} s'$ and $s \overset{\beta}{\rightarrow} s''$. Since both $\alpha \in\rightarrow_j$ and $\beta \in\rightarrow_j$, and transition group j is guarded by $\mathcal{G}_j$, both α and β are guarded by the same set of guards. Therefore, for all guards $g \in \mathcal{G}_j$ such that $g(s) = \mathit{false}$ in some state $s \in S_P$, the necessary enabling set $\mathit{NES}(g, s)$ is a necessary enabling set of both α and β. Similarly, for all guards $g \in \mathcal{G}_j$ such that $g(s) = \mathit{true}$ in some state $s \in S_P$, the necessary disabling set $\mathit{NDS}(g, s)$ is a necessary disabling set of both transitions. Since this holds for any two transitions $\alpha, \beta \in\rightarrow_j$, the singleton constraint is irrelevant. □

3.8 Partial order reduction v.s. symbolic reachability

The availability of the partial order reduction PINS layer enables us to compare it against the symbolic reachability tools. Table 3.6 shows the effect of using partial order reduction versus the effect of using symbolic reachability when the goal is deadlock checking. The symbolic reachability was run with regrouping (optimizing the variable ordering of the BDD/MDD structures together with the chaining heuristic). Interestingly enough, both techniques seem to work well on different classes of models.

For example, *exit.3* doesn't seem to benefit from partial order reduction, while the symbolic reachability seems to work well on it. On the other hand, partial order reduction reduces the runtime for *anderson.6* while symbolic reachability is much slower. Symbolic reachability does however use less memory. Note, that the comparison isn't entirely fair. The partial order reduction algorithms have more information available for the reduction than the symbolic reachability. The symbolic reachability may be improved by adding the guard information. Therefore there might be cases where partial order reduction wins it from symbolic reachability in the comparison. On the other hand, the cases where symbolic reachability wins it from partial order reduction can be coincidental. It might be that some other run of the algorithm, or some other partial order reduction algorithm is able to achieve more reduction on the same model. Nevertheless, since usually there is no optimal partial order reduction algorithm, it is useful to have both techniques available.

3.9 Discussion and future work

To improve confidence in the correctness of the implementations, the implementations have been tested against the full BEEM [35] database. All implementations preserved *all* deadlocks.

The additional information that is added to the PINS interface has some down sides too. The addition of guards to each transition group would require changes in the way regrouping currently works. Currently regrouping can combine transition groups into a new transition group. This would require combining the guards of the transition group. The guards must be conjunctive, while combining the guards would require a disjunction. Happily, the symbolic optimizations on BDD level come from reordering the state vector. The PINS caching layer and the symbolic tools benefit from combining the transition groups too, but this is because less transition relations over sub-vectors need to be stored. Combining the transition groups for this purpose can also be implemented in a different manner, however it would be less elegant.

Last, the matrix is heavily affected by the POR layer. Since each state now uses all the guards of all transition groups to determine a subset of the enabled transitions, each transition group should mark all

Table 3.6: Partial order reduction v.s. symbolic reachability

model	algorithm	states		transitions		walltime [sec]	memory [vsize]
		count	percent	count	percent		
anderson.6		18206917	100.0	86996322	100.0	119.78	485468KB
	nes/nds	4858411	26.7	16898226	19.4	94.26	171476KB
	reach	18206917	100.0	0	0.0	241.41	124012KB
at.6		160589600	100.0	697096560	100.0	808.42	3202196KB
	nes/nds	148690835	92.6	595369635	85.4	6299.36	3202360KB
	reach	160589600	100.0	0	0.0	476.50	229400KB
exit.3		2356294	100.0	7047332	100.0	27.42	99492KB
	nes/nds	2356207	100.0	7047158	100.0	105.67	99692KB
	reach	2356294	100.0	0	0.0	5.91	64160KB
lamport.5		1066800	100.0	3630664	100.0	3.40	70300KB
	nes/nds	998746	93.6	3173546	87.4	24.82	70400KB
	reach	1066800	100.0	0	0.0	4.71	58220KB
leader_election.6		35777100	100.0	233195212	100.0	3850.76	2289068KB
	nes/nds	232398	0.6	301559	0.1	13.51	61252KB
	reach	35777100	100.0	0	0.0	28.02	61360KB
leader_filters.7		26302351	100.0	91692858	100.0	181.43	839208KB
	nes/nds	25130516	95.5	79009993	86.2	825.97	839284KB
	reach	26302351	100.0	0	0.0	176.61	316568KB
peterson.7		142471098	100.0	615983127	100.0	1096.91	3313560KB
	nes/nds	93246567	65.4	388201901	63.0	2106.05	2089468KB
	reach	142471098	100.0	0	0.0	397.00	150064KB
phils.8		43046720	100.0	459165008	100.0	960.12	1817408KB
	nes/nds	722	0.0	976	0.0	0.10	3688KB
	reach	43046720	100.0	0	0.0	0.60	57248KB
schedule_world.3		166649331	100.0	2020849194	100.0	4377.84	3205884KB
	nes/nds						
	reach	166649331	100.0	0	0.0	13.72	92856KB
synapse.4		2292286	100.0	4921830	100.0	7.61	104152KB
	nes/nds	2188566	95.5	4138700	84.1	133.29	104892KB
	reach	2292286	100.0	0	0.0	68.46	456208KB

slots used by a guard as read. This has serious implications on using POR combined with the symbolic tool, or with the incremental hashing that depends on the pins dependency matrix. The latter can be solved by splitting the dependency matrix into two matrices, read and write. This can also improve the dependency relation D as defined in section 3.1.2, since two transitions that only read variables can now be considered independent too. An implementation of this has been made and tested on the BEEM database. There appeared to be little effect for most of the examples as most of the transitions in the BEEM database both read and write to elements of the state vector. It might however still be beneficial to other front-ends.

3.10 Conclusion

The proposed stubborn set algorithm using necessary sets successfully manages to reduce the state space. It is implemented as a PINS layer and therefore can work for any front-end connecting to PINS, and supplying guards labels and the extra matrices. The static analysis used in the front-end to supply the matrices needed for the partial order reduction will determine the amount of reduction one will get.

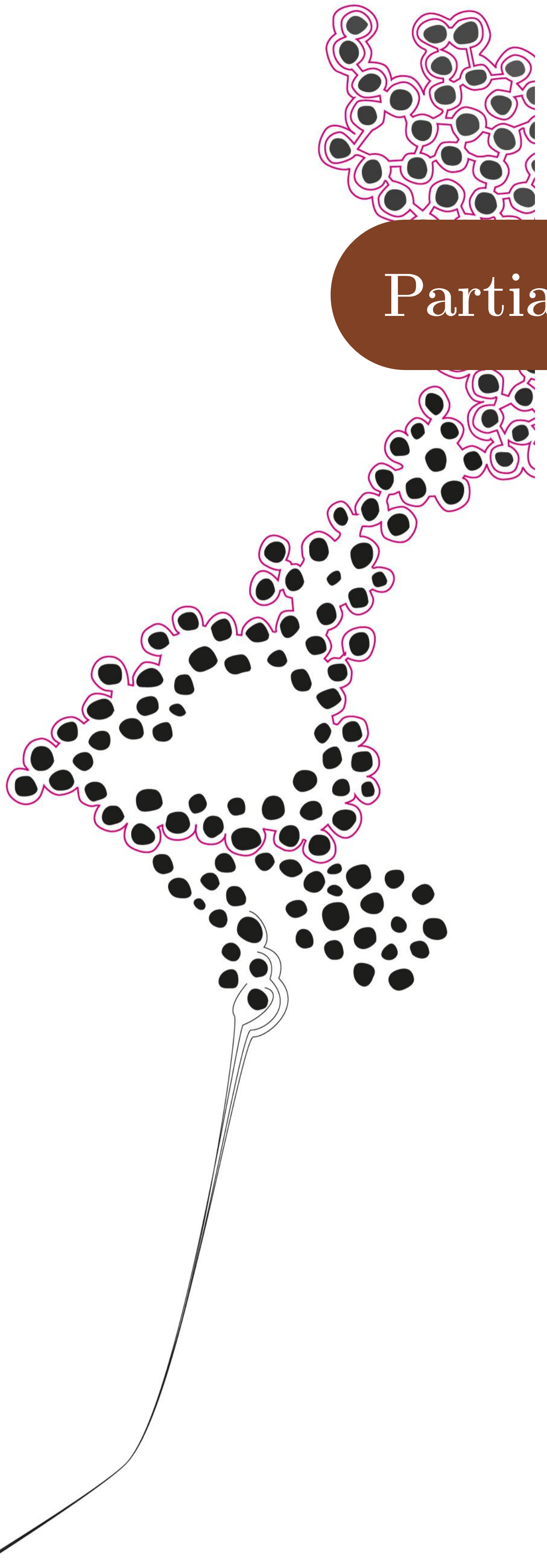
Since the partial order reduction algorithm is implemented modularly, the algorithm will also work on the multi-core and distributed implementations. The wrapper however results in a very dense pins dependency matrix and will therefore not combine well with the symbolic tools, with the exception of the cases where there is a lot of reduction or where the matrix was dense already.

The most important changes for the PINS interface needed for the partial order reduction to work is a) guards b) the may be co-enabled relation of guards.

Adding more fine grained guards will provide more and smaller necessary enabling sets to choose from and will likely result in better reduction. The *NES* and *NDS* matrix can automatically be computed from the provided guards but can manually be improved for better reduction. The results of last chapter suggest that the reduction gained by improving these matrices will not be as significant for most of the models, although there might be large improvements for some.

Chapter 4

Partial Order Reduction for LTL



This chapter the focus lies on model checking linear temporal logic. We develop a PINS wrapper to compute the synchronous product automata of the model as given by some front-end with a büchi automaton generated from an linear temporal time formula. We show how this wrapper can be used independently with the back-end search algorithms, and give a modular approach to implement an accepting-cycle search algorithm. Furthermore it is shown how the partial order reduction wrapper developed in the previous chapter can be used in combination with the LTL wrapper and the back-end algorithms, such that the reduced state space preserves the LTL properties.

4.1 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) is a logic for expressing properties over paths in computation trees. A computation tree can be seen as a tree that has an initial state as its root, and then an infinite unwinding of all transitions that can be taken from there as branches. In LTL it is possible to express properties with respect to the path using temporal operators. It is a formalism that is used to express the safety and liveness properties, and was first proposed for the verification of computer programs by Amir Pnueli in 1977 [39].

4.1.1 LTL semantics

LTL properties can be used to reason about labels associated with each state $s \in S$ from a *TS* $T = \langle S, \rightarrow, \hat{s}, L \rangle$. To do this a set of *atomic propositions* Π is needed, together with a labeling function $\lambda : S \rightarrow 2^{\Pi}$ that assigns a truth value for each proposition in Π to each state $s \in S$. LTL [32] is defined from atomic propositions $p_1, \ldots, p_n$ by means of boolean connectives, X ("next"), $\Box$ ("always"), $\Diamond$ ("eventually") and U ("until") operators. If $\omega = s_0, s_1, \ldots$, where $s_0, s_1, \ldots \in S$ is an infinite sequence of states for which the truth values $p_1, \ldots, p_n$ are assigned by the labeling function λ, then let $\omega, i \models \phi$ denote that the LTL formula ϕ holds at index i for sequence ω, and let $\omega(i)$ denote the state s_i in ω.

Table 4.1: LTL

$\omega, i \models p$	where $p \in \Pi$, iff $p \in \lambda(\omega(i))$
$\omega, i \models \overline{\phi}$	iff $\omega, i \not\models \phi$
$\omega, i \models \phi \vee \varphi$	iff $\omega, i \models \phi$ or $\omega, i \models \varphi$
$\omega, i \models \mathrm{X}\phi$	iff $\omega, i+1 \models \phi$
$\omega, i \models \Box\phi$	iff for all $k \geq i$ holds that $\omega, k \models \phi$
$\omega, i \models \Diamond\phi$	iff there exists $k \geq i$ such that $\omega, k \models \phi$
$\omega, i \models \phi \mathrm{~U~} \varphi$	iff there exists $k \geq i$ such that $\omega, k \models \varphi$ and for all $i \leq j < k$ $\omega, j \models \phi$

For a sequence of states ω, the LTL formula ϕ holds when $\omega, 0 \models \phi$. For a transition system T, we denote that $T \models \phi$ is true iff for each sequence ω generated by T from the initial state $\hat{s}$, $\omega, 0 \models \phi$.

4.1.2 On the fly LTL model checking

The DFS or BFS search algorithm can be used to generate all states s in the state space. After this, the property ϕ can be verified. For small models this is no problem, for larger models however, creating the entire state space might not be possible. To still get some meaningful results, it would be better to verify the truth value of the property during the generation of the state graph. This is known as *on-the-fly* model checking. If the truth value of the property is violated before the entire state graph has been built, a trace to this error can be reported before the generation of the state space is finished.

An important way to check properties is using LTL without next-time operator (X), LTL$_x$. For all LTL$_x$ formula there exists a corresponding *Büchi automaton*, which can be seen as the automata-theoretic version of the LTL formula. It is an extension of a finite state automaton to infinite words. Most importantly Büchi automata have an acceptance condition such that it accepts infinite sequences that go infinitely often through an accepting state.

Definition 35. A Büchi automaton is a tuple $B = \langle Q, q_0, \Sigma, \Delta, F \rangle$ where Q is the set of states, q_0 is the initial state, Σ is the alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is a nondeterministic transition relation, and $F \subseteq Q$ is the set of final (accepting) states. A *run* of B on an ω-word $\alpha = \alpha(0)\alpha(1)\ldots$ from Σ^{ω} (the set of infinite words over Σ) is an infinite sequence of states $\sigma = \sigma(0)\sigma(1)\ldots$ such that $\sigma(0) = q_0$ and $(\sigma(i), \alpha(i), \sigma(i+1)) \in \Delta$ for all $i \geq 1$. A run of B is *accepting* if some $s \in F$ appears infinitely many times in the sequence of states

σ, i.e. there exists a cycle though an accepting state. The infinite word ω is accepted by B if there is an accepting run of B over ω. The *language* of infinite words accepted by B is denoted $L(B)$.

The model checking problem, $T \models \phi$, may be viewed as an *automata-theoretic verification* problem, $L(B_T) \subseteq L(B_\phi)$ (the language accepting the transition system should be included in the language accepting all words for which ϕ holds). The question $L(B_T) \subseteq L(B_\phi)$ can be answered by constructing the state graph of the synchronous product of B_T and $B_{\neg\phi}$, $S = B_T \otimes B_{\neg\phi}$. (There should not be a word in $L(B_T)$ that is not in $L(B_\phi)$, thus that is in $L(B_{\neg\phi})$). If any strongly connected components (cycles) of the graph represented by S satisfies the acceptance condition of $B_{\neg\phi}$ then and only then ϕ is violated in T.

Several algorithms exist to find infinite paths satisfying a Büchi acceptance condition (NDFS, SCC) [40, 18].

4.2 The ignoring problem for partial order reduction

Besides checking the absence of deadlocks in a model, there are two important classes of properties, that both can be expressed using LTL:

1. Safety properties ("bad things do not happen")
2. Liveness properties ("eventually good things will happen")

Application of partial order reduction while preserving these properties is more complicated than preserving just deadlocks. Using only the persistent set technique the state space is reduced too aggressive.

4.2.1 Preserving safety properties

A safety property asserts that nothing bad happens on a finite trace.

> *"if something bad happens on an infinite run, then it happens on a finite prefix of this run too"*

The persistent set (section 3.1) preserves enough information to keep all deadlocks in the state space. It may however prune too much of the state space to preserve safety or liveness properties. For example, when there are no deadlocks in the state space, due a cycle in one or multiple subcomponents that together are independent of the rest of the system. Consider again the asynchronous composition of a transition system T and process P with a self loop from s to s. As concluded in 3.1, the combined system has no deadlock, independent of the transition system T. A typical safety property however concerns reachability of some local state of a process (for instance, an assertion on a local variable). Note that in T, the reachability of local states in the components of T can be irrelevant to the deadlock property. Transitions in T can be delayed, and a global cycle can be found in P before any other local state in T is reached. Thus, for safety properties the persistent set alone is not enough. This problem was first identified by Valmari [41] and is named the "ignoring-problem" (by the fact that some processes/transitions can be ignored).

Transitions outside a persistent set can be seen as delayed. As soon as a global cycle is found, some delayed transitions can become delayed indefinitely. To solve the ignoring problem one needs to make sure that in the reduced state space S_r, no transition is delayed indefinitely:

C3^S For any state $s \in S_r$, if transition $\alpha \in enabled(s)$ there
is a state $s' \in S_r$ reachable from s such that $\alpha \in r(s')$ [16].

Now, all possible finite traces will be included in the reduced state space. With this extra condition C3^S, called *proviso*, added as a condition that must hold for each state in the reduced state space, safety properties concerning reachability of a local state of a process are preserved.

Theorem 6. *Let T be a TS and let r be a reduction function on T such that T' is the transition system of T by r. If for T' conditions C0, C1 and C3^S hold, then for all processes p for all local states l reachable from the initial state s_0 in T iff l is reachable from the initial state s_0 in T'.*

Proof. For the proof we refer to Godefroid [22], Evangelista [16], Peled [37]. □

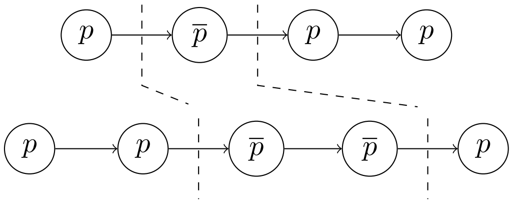


Figure 4.1: Stuttering equivalent paths

4.2.2 Preserving liveness and LTL properties

Both safety and liveness properties work under fairness assumptions on the environment. For example, one might want to assume that the scheduler never ignores a process forever (infinite times). The actual implementation might take more then a lifetime to schedule it, however *eventually* it will do so. Hence, whatever finite prefix is used, there may be an extension in which the liveness property does hold under the fairness assumption. Therefore, contrary to safety properties liveness properties need infinite traces.

> *"no matter what happens along a finite run, something good can still happen later"*

Both for safety and liveness properties, fairness plays a role. However, it has been shown that all fair executions of a program satisfy a safety property iff all executions, fair and unfair satisfy the property [37]. This means that for safety properties we do not need to be concerned about fairness. For liveness properties however fairness is essential. Combining partial order reduction and fairness for liveness properties is a bit more complex.

$C3^L$ A cycle is not allowed in the reduced state space if it contains a state in which some transition α is enabled, but never included in $r(s)$ for any state s on the cycle. [16]

Peled uses a stronger proviso that ensures that all fair runs in the original state space have at least one (stuttering) equivalent fair run in the reduced state space: For each cycle in the reduced state space, there exists one state s that is fully explored (all enabled transitions are explored in s) [22, 41, 16].

LTL can express safety and liveness properties, thus when applying partial order reduction for LTL we need at least the proviso for liveness properties. The liveness proviso alone however is not enough. The checked property might be sensitive to the choice of successor states in the persistent set. The property might use multiple propositions, for instance $p_1 \wedge p_2$, while a single transition might just alter the evaluation of either p_1 or of p_2, and might be independent of other transitions that have an effect on the property. What is actually important for LTL is that the checked property remains the same along paths that are pruned. For this we need some definitions.

A transition $\alpha \in T$ is *invisible* [36, 12] in respect to a set of propositions Π if for each pair of states $s, s' \in S$ such that $s \overset{\alpha}{\to} s'$, $\lambda(s) \cap \Pi = \lambda(s') \cap \Pi$. In other words, a transition is invisible when its execution from any state does not change the value of the propositional variables. A transition is *visible* if it is not invisible.

Closely related to the concept of (in)visibility is that of *stuttering*. Stuttering refers to a sequence of identically labeled states along a path. A finite sequence of identically labeled states is called a *block*. Intuitively, two paths are *stuttering equivalent* when they can be partitioned into (in)finitely many blocks such that the states in the *k*th block of each path have the same label. For example, Figure 4.1 has two stuttering equivalent paths, divided into three blocks $(p, \overline{p}, p)$ [36, 12].

It can be proven that any LTL_x formula is invariant under stuttering (i.e. formulas are not affected by repeating similar states) [12]. Conversely, every property that is stuttering closed can be expressed in LTL_x [38, 12].

Besides the liveness proviso, to preserve LTL_x the persistent set needs to prune only stuttering equivalent traces. The following condition does this:

C2 All transitions in $r(s)$ are invisible, except when $r(s) = \mathit{enabled}(s)$ [36, 16, 37, 1].

This extra condition allows on-the-fly LTL model checking in combination with partial order reduction. Applying the partial order reduction algorithm with these constraints preserves the LTL$_x$ properties in the reduced state space. The reduced state space contains traces all stuttering-equivalent to a trace from the original state space. Therefore the normal accepting cycle search algorithms can now be applied without knowledge of the partial order reduction [36].

Theorem 7. *Let T be a transition system and let r be a reduction function. The reduced transition system T' produced from T by r is stuttering equivalent to T if conditions C0, C1, C2, C3^L hold.*

Proof. C0, C1 are equivalent to the persistent set generated by the stubborn algorithms of chapter 3 by theorem 2. See Katoen [1] theorem 8.13 for the proof. □

4.3 Safety and liveness proviso

Several strengthened versions of the safety- and liveness proviso exist, trading implementation efficiency against state space reduction. Because safety and liveness proviso both require knowledge on cycles in the state space, the search order is very important. In general, depth first search is better to detect cycles as a trace of the cycle is on the stack. Moreover, the acceptance cycle detection algorithms are based on depth first search. Therefore the focus lies on proviso for depth first search.

The safety and liveness provisos are defined to hold globally in the reduced state space. The reduction function however requires the proviso to be locally checkable for individual states. The provisos presented here are all locally checkable.

4.3.1 The stack/queue proviso

The best known proviso is the stack proviso [22, 28], as implemented in the SPIN model checker [28], and the BFS queue variant [6]. Let S_r denote the reduced state space obtained by applying a reduction function r.

Safety

(stack proviso) For any $s \in S_r$, there exists at least one transitions $\alpha \in \mathcal{T}(s)$ and state $s' \in S_r$ such that $s \xrightarrow{\alpha} s'$ and s' is not on the DFS stack, i.e. $s' \notin \mathit{Stack}$. Otherwise $\mathcal{T}(s) = \mathit{enabled}(s)$.

(queue proviso) There exists at least one transition $\alpha \in \mathcal{T}(s)$ and a state $s' \in S_r$ such that $s \xrightarrow{\alpha} s'$, s' is in the BFS queue, i.e. $s' \in \mathit{Queue}$. Otherwise, $\mathcal{T}(s) = \mathit{enabled}(s)$.

Liveness

(stack proviso) For all transitions $\alpha \in \mathcal{T}(s)$ and states $s' \in S_r$ such that $s \xrightarrow{\alpha} s'$, s' is not on the DFS stack, i.e. $s' \notin \mathit{Stack}$. Otherwise, $\mathcal{T}(s) = \mathit{enabled}(s)$.

(queue proviso) For all transitions $\alpha \in \mathcal{T}(s)$ and states $s' \in S_r$ such that $s \xrightarrow{\alpha} s'$, s' is in the BFS queue, i.e. $s' \in \mathit{Queue}$. Otherwise, $\mathcal{T}(s) = \mathit{enabled}(s)$.

4.3.2 The closed set proviso

Bosnacki, Leue and Lluch-Lafuente developed General State Expanding algorithms [7] (see section 4.6.1). The algorithms captures both the DFS and BFS search order by changing just the underlying data structure. It uses the notion of open- and closed sets of states. An additional advantage of the algorithm is that it can be used for directed searches (best first, A*) as well. Because the data structure describes the search order the notion of stack and queue is replaced by the open- and closed set. Therefore expressing a proviso in these terms is useful as well. The main idea is that a transition can be delayed as long as the next states are new (thus not yet visited, visited set proviso [14]). A slightly improved version called the closed set proviso [7] states that states can be delayed as long as they are not handled by the algorithm

(It is better because states that are visited might still be in the open set). Since the stack proviso may include states that are closed and off the stack, it is expected that the stack proviso results in a better reduction for most models.

Safety

(closed set proviso) For any $s \in S_r$, there exists at least one transition $\alpha \in \mathcal{T}(s)$ and state $s' \in S_r$ such that $s \overset{\alpha}{\to} s'$ and $s' \notin \mathit{Closed}$. Otherwise $\mathcal{T}(s) = \mathit{enabled}(s)$.

Liveness

(closed set proviso) For all transitions $\alpha \in \mathcal{T}(s)$ and states $s' \in S_r$ such that $s \overset{\alpha}{\to} s'$, $s' \notin \mathit{Closed}$. Otherwise, $\mathcal{T}(s) = \mathit{enabled}(s)$.

4.3.3 The color proviso

The color proviso by Evangelista and Pajault [16] can be used only for liveness properties in a depth first search. The proviso exploits the fact that cycles that lead back to the stack may contain a fully expanded state already, i.e. the state that closes the cycle doesn't need to be fully expanded. This is done by coloring the states red, green or orange. Green states are safe states, and may be reached by any other state without risking of closing an invalid cycle. Orange states are potentially dangerous states still on the stack. Red states are dangerous states that have left the stack, that may close a "bad" cycle. When encountering a red successor a state must be fully expanded. For an orange state, it must be checked whether the cycle contains a fully expanded state. This is done by counting the number of fully expanded states between the orange state and the state that closes the cycle. If the count is the same, no fully explored state is on the cycle and the orange state is dangerous. If a fully explored state is on the cycle however, it is save to go back to the stack. A large difference between this proviso and the stack proviso is that using the stack proviso, states off the stack are all save to reach, while for the color proviso, red states may exist off the stack. Nevertheless, the color proviso seems to result in better reductions for various examples [16].

4.4 A PINS LTL wrapper

The two main ingredients for LTL model checking are a) the synchronous cross product of the model with the negated büchi automata b) the search algorithm for accepting cycles. Usually model checkers have intermingled implementations of both these components. Using the PINS interface it is possible to separate these two. The search algorithm can be implemented as back-end PINS algorithm like the dfs- and bfs searches, while the cross product with the büchi automaton can be hidden inside a PINS wrapper. The wrapper can intercept the next state call and combine the next state generated by the front-end with the büchi automaton. The PINS wrapper adds a state label for accepting/non-accepting büchi states. This is used by the back-end algorithm to find the accepting cycles. The additional advantage of this setup is that it is still possible to let the front-end compute the cross product and provide the accepting state label. In other words, if the front-end provides an interface for LTL or büchi automata this can still be used. To combine the LTL wrapper with the POR wrapper, partial order reduction must occur before the cross product with the büchi automaton [36]. As shown in Figure 4.2, the POR wrapper executes between the LTL wrapper and the front-end. Thus the POR wrapper produces a reduced state space preserving LTL, then the LTL wrapper combines the state with the büchi automaton and finally the back-end searches for accepting cycles.

Theorem 8. *Let T be a transition system, and let T' be the reduced transition system of T by some reduction function r, satisfying conditions C0, C1, C2 and C3. The intersection of the buchi automaton G of T with the buchi automaton of a property $\neg\phi$ is empty iff the intersection of the büchi automaton G' of T' with $\neg\phi$ is empty, i.e. $L_T \cap L_{\neg\phi} = \emptyset$ iff $L_{T'} \cap L_{\neg\phi} = \emptyset$.*

Proof. For the proof, see [36]. □

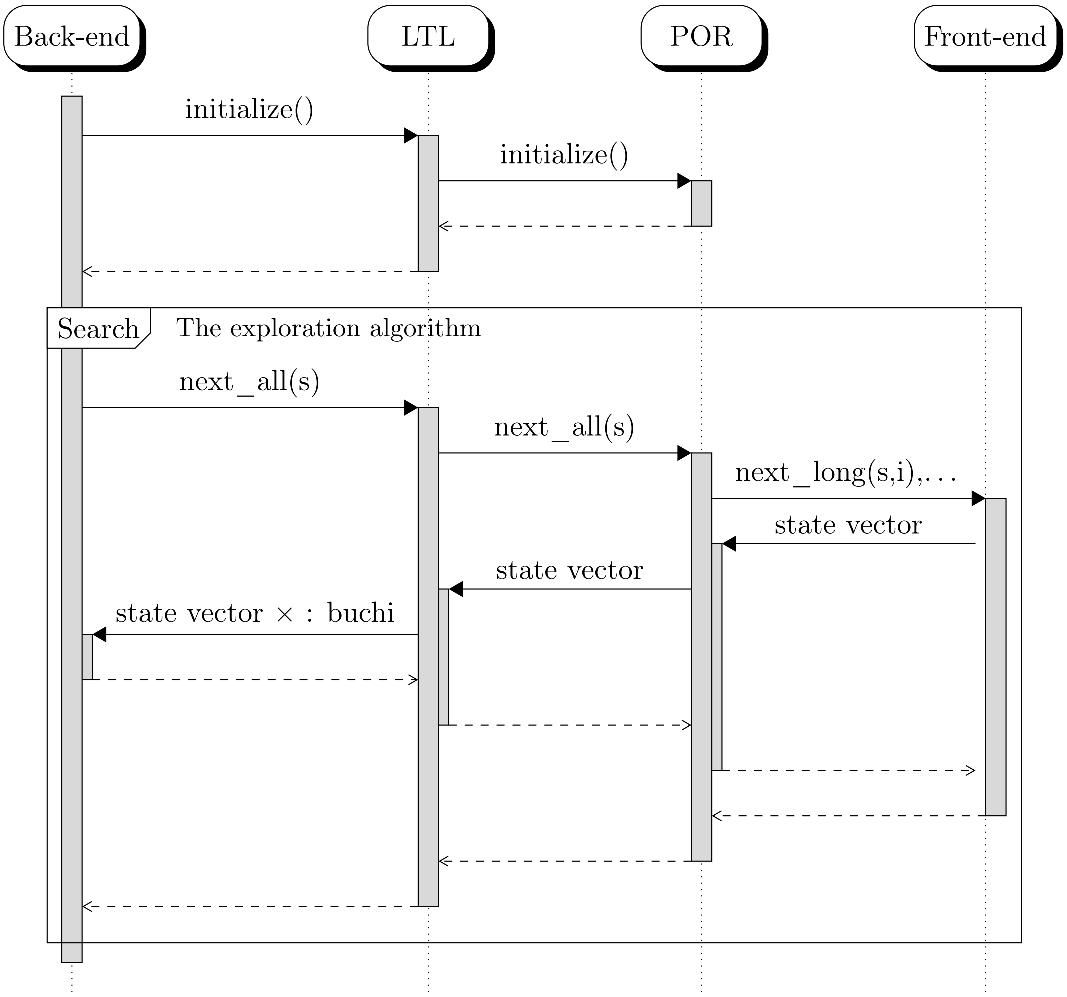


Figure 4.2: The LTL and POR wrapper for PINS

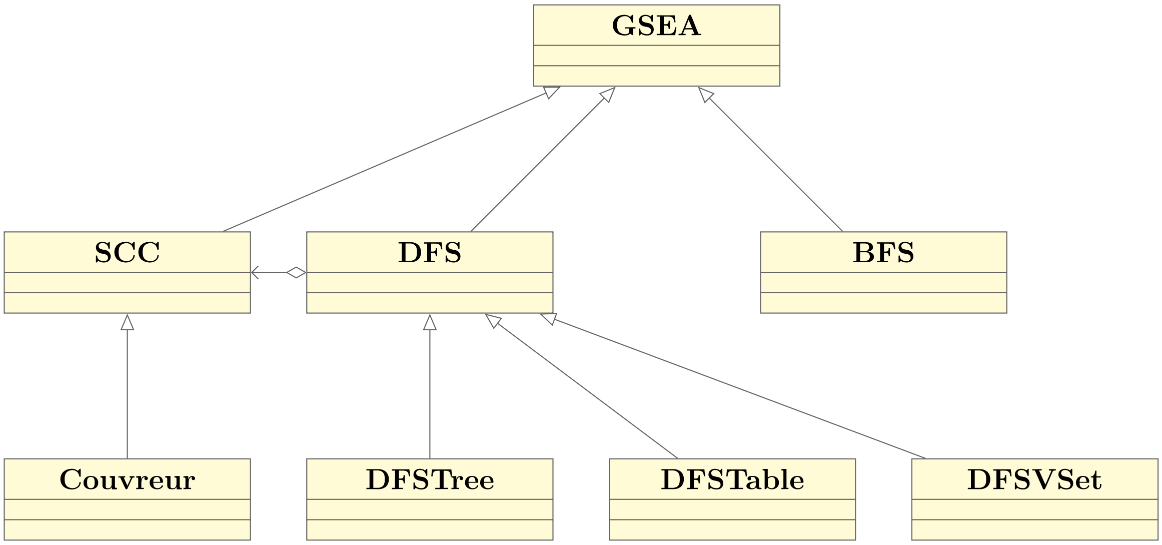


Figure 4.3: LTSmin framework class diagram

4.5 Observations

As with partial order reduction for deadlocks, the reduction for LTL should be modular too. Information on (in)visibility can be statically obtained and passed from the LTL wrapper to the POR wrapper.

The safety- and liveness proviso however requires knowledge about cycles in the state space. This knowledge can be made available in the search algorithm, but is not available in the POR wrapper. Therefore, a clean separation where the search algorithm is unaware of the PINS wrapper is very hard.

As written in the previous section, we know that the stuttering requirement (invisible condition), as well as the liveness proviso are based on just the current and the next states. Shown in Figure 4.2 is how the PINS wrapper passes along the next state to the search algorithm. Because this happens in a callback, there is an information flow in both ways: from the POR wrapper to the back-end algorithm, and from the back-end algorithm to the POR wrapper. This information flow covers precisely the interesting states (current and next) for which information is needed in the POR wrapper. To make a modular POR wrapper for LTL, the idea is that the search algorithm can pass along information on cycles using the callback dataflow.

Another observation is that both for safety and liveness properties, the same information needs to be passed to the POR wrapper for each transition. The difference between safety and liveness is that for safety properties there must be at least one transition that doesn't close a cycle on which no state is fully expanded, while for liveness properties this must hold for all transitions. Hence that the extra information needed for each transition is a boolean flag (from here on called *por_proviso*) that says:

Definition 36. Given a *TS* T and a *TS* T' reduced by the reduction function r. The $por\_proviso(s \rightarrow s')$ holds for a transition currently being explored if there exist no cycle $s_0, s_1, \ldots, s_n \in S_{T'}$ with $s_0, s_n = s'$ and $s_{n-1} = s$ where on each state of the cycle $r(s_i) \neq enabled(s_i)$ for $0 \leq i \leq n$.

A clean separation can now be defined between the POR wrapper and the back-end search algorithm, by annotating the successor state s' with the value of the *por_proviso* function. The search algorithm is in charge of setting the *por_proviso* flag, while the POR wrapper can check that for safety properties there exist a successor state for which the flag is set, and for liveness properties the flag is set for all successors. Note that *false* is chosen as default value for this flag. Algorithms that do not support partial order reduction do not touch the flag, and therefore there will not be reduction.

4.6 Implementation

As observed, the detection of cycles is needed for partial order reduction to work with LTL, and there is a direct connection to the search algorithm used. However, the search algorithm used is less tightly coupled to the acceptance cycle search. From the perspective of a tool designer, it would be nice to have a framework in which this is exploited such that code can be reused in a modular fashion.

4.6.1 General State Expanding Algorithms

As a starting point for such a framework we use General State Expanding Algorithms (GSEA) [7]. In short, this algorithm (algorithm 3) can do both the dfs- and bfs exploration strategy depending on the data structure chosen to represent the *Open* set. If the open set is represented as a queue, the search order is bfs, while it is dfs if the open set is represented as a stack.

```
1  function $GSEA(s)$
2      $Closed = \{\}$;
3      $Open = \{s\}$;
4      foreach $s \in Open$ do
5          $Open = Open - \{s\}$;
6          $Closed = Closed \cup \{s\}$;
           // generate next states
7          foreach $s' \in s \to s'$ do
8              $state\_process(s')$;
9              if $s' \notin Closed \wedge s' \notin Open$ then
10                 $Open = Open \cup \{s'\}$;
```

Algorithm 3: General State Expanding Algorithm

The LTSmin implementation

It is possible to use an object-oriented approach to implement the General State Expanding Algorithm. The functionality of each line of code of algorithm 3 can be mapped to a function that belongs to a GSEA object. Then a more specialized objects for DFS- and BFS searches can inherit from this and reuse/share code. From these objects again more specialized versions can be used with different storage mechanisms to represent a state. Figure 4.3 shows this setup. Each more specialized class can be used to overwrite behavior and add extra work before or after the original function call. The foreach statements in Algorithm 3 can be replaced by a function call and a callback representing the underlying code block. An example of such an implementation is shown in Algorithm 4 where the foreach (line 4 Algorithm 3) is replaced by a while statement (line 4,5 Algorithm 4). Line 6 to 12 are put in a callback function such that sets more suitable for a different foreach implementation can reuse the same callback code. Similarly, the foreach used to generate the next states is a combination of a wrapped PINS *next_ all* call and a callback (see Figure 4.2).

```
1  function $GSEA(s)$
2      $init()$;
3      $open\_insert(s)$;
4      while $has\_open()$ do
5          $s = open\_extract()$;
6          $closed\_insert(s)$;
           // generate next states (next_all + callback)
7          foreach $s' \in s \to s'$ do
8              $state\_process(s')$;
9              if $open\_insert\_condition(s')$ then
10                 $open\_insert(s')$;
11             else
12                 $state\_matched(s')$;
```

Algorithm 4: LTSmin GSEA Framework

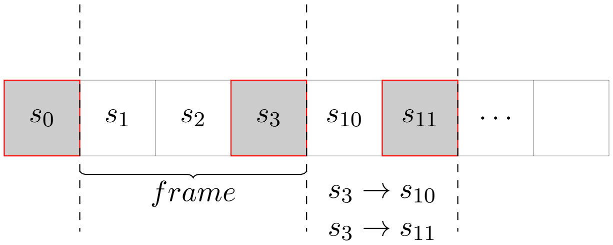


Figure 4.4: A stack with frames

The DFS implementation

The framework described in Algorithm 4, Figure 4.3 can be used to share code at different levels. As an example, a generalized implementation for a DFS search is given below. It can be used to share a stack but still represent the states using different storage mechanisms.

```
function $closed\_insert(s)$
    return $Closed = Closed \cup \{s\}$;
function $open\_insert\_condition(s)$
    return $s \notin Closed$;
function $open\_insert(s)$
    return $push(Stack, s)$;
function $has\_open()$
    while $peek(Stack) \in Closed$ do
        $pop(Stack)$
    return $!empty(Stack)$;
function $open\_extract()$
    return $pop(Stack)$;
```

Algorithm 5: DFS configuration

Note that the real implementation in LTSmin uses a framed stack, where each of the generated successors s' of a state s are put on a new stack frame (see Figure 4.4). Using the frame information it is possible to generate traces, i.e. the last state on the frame is a state currently being explored. The downside of this approach is that it is possible that some state occurs multiple times on the stack. The *has_ open* function will take care of this by popping it off the stack when it is closed already. There can thus be two events leading to a *pop* operation in *has_ open*. Either a state is popped directly after closing a frame, which means that all successor states are closed and the algorithm is backtracking, or a state is popped in between a frame (the last action was a *pop*, not closing a frame) which means that the state must have been closed in some future execution of the DFS search. Both these events provide valuable information for some algorithms. Therefore, two extra calls (*state_ backtrack, state_ backtrack_ drop*) are added to be able add some extra code on these events.

4.6.2 Couvreur

For LTL model checking, efficient acceptance cycle detection algorithms exists. Most notably the Nested DFS and SCC variants [40, 18]. Both algorithms rely on the DFS post-order to detect accepting cycles. Ideally, these algorithms are implemented on top of the DFS search implemented in the GSEA framework described in the previous section, such that the acceptance cycle detection works whenever a new DFS algorithm is implemented. There are however some known problems with LTL in combination with partial order reduction. NDFS and some variants of SCC require a state s to be explored more then once. Since with LTL the result depends on the search order this might select different subsets of $enabled(s)$ each call, and give invalid results [25]. There are solutions [40] to overcome these problems, but the easiest solution is to use an algorithm that requires just a single exploration of each state. This makes the SCC based algorithms better candidates. The Amendment of Couvreur's algorithm as described by Gaiser and Schwoon [18] is very well suited. It explores each state only once, and needs some extra work only in well defined places of the DFS search: when a search is started for a state, when a state is popped off the stack, and when a state is matched. Using the GSEA framework (Figure 4.3), it is possible to add an SCC component that uses a generalized DFS search and overwrites precisely these functions to add some extra behavior needed to turn the DFS search into an SCC algorithm.

```
function $open\_extract()$
    $s = dfs.open\_extract();$
    $count = count + 1;$
    $s.dfsnum = count;$
    $s.current = true;$
    $push(Roots, (s, is\_accepting(s)));$
    $push(Active, s);$
    return $s;$
function $state\_backtrack(s)$
    if $top(Roots) == (s, ?)$ then
        $pop(Roots);$
        repeat
            $u = pop(Active);$
            $u.current = false;$
        until $u = s;$
    return $dfs.state\_backtrack(s);$
function $state\_matched(s)$
    if $s.current$ then
        repeat
            $(u, accept) = pop(Roots);$
            if $accept$ then
                report cycle
        until $u.dfsnum \leq s.dfsnum;$
        $push(Roots, (u, false));$
    return $dfs.state\_matched(s);$
```

Algorithm 6: Couvreur's algorithm for GSEA framework

4.6.3 Combining POR with LTL

Using the partial order reduction wrapper works well for deadlocks. For LTL extra constraints on the persistent sets are needed (section 4.2.2). Safety and liveness properties depend on the search order and need extra information for each generated successor. Shown in Figure 4.2, there exists a path through which the data can flow from the POR wrapper to the back-end algorithms and back, that can be used to provide the POR wrapper with the extra information needed. By letting the algorithm annotate each transition with a boolean flag to detect improperly closed cycles, the POR wrapper can select a persistent sets for safety and liveness properties as well. Furthermore, preserving all LTL requires that none of the transitions in the persistent set is visible.

Adding invisible constraint for LTL

The LTL wrapper can determine the set of transition groups that modify (write to) the slots of the state vector that are used for the evaluation of the predicates on the edges of the büchi automaton. These transition groups are marked visible, the others are marked invisible. On initialization, this information is passed to the POR wrapper which from then on uses the static visibility information to find a persistent set with only invisible transitions in it.

The NES/NDS algorithm of section 3.6.1 can easily be modified to return either only invisible transitions or all transitions. Because the algorithm uses a heuristic function $h(nes)$ to select the best persistent set, a minor change in the heuristic function can steer the algorithm away from selecting visible transitions.

The new heuristic h used to select a NES is computed as follows.

1. For each disabled transition in the NES and not in $T_{work} \cup T_s$ add 1 to the cost

2. For each enabled invisible transition in the NES and not in $T_{work} \cup T_s$ add n to the cost

3. For each enabled visible transition in the NES and not in $T_{work} \cup T_s$ add n^2 to the cost

Now, after selecting the persistent set the algorithm can simply confirm that no visible enabled transition has been selected. If this is the case, it will return $\mathcal{T}(s)$, otherwise it must return *enabled*(s). Hence, the extra condition on visibility can easily be incorporated in the original algorithm (for deadlocks, all transitions are invisible). Information regarding the visibility of transition groups can be passed from the LTL wrapper to the POR wrapper on initialization.

Communicating the safety and liveness proviso

To communicate the safety and liveness proviso with the back-end algorithm the algorithm needs to provide the *por_proviso* flag for each successor state. How this is done will be discussed in the next section. Enforcing the constraint is now a matter of verifying a simple condition on the persistent set or returning *enabled*(s).

Preserving safety properties For safety properties, the following condition must hold for each transition $\alpha = s \to s'$ generated

$$\exists \alpha \in \mathcal{T}(s) : \mathit{por\_proviso}(\alpha) \vee \mathcal{T}(s) = \mathit{enabled}(s)$$

This is a simple as verifying that one of the generated successor states has the *por_proviso* flag set to true. If this is not the case, one might consider adding an unselected transition to the emitted persistent set and recomputing a smallest set.

Preserving liveness properties For liveness properties, the extra condition is that each transition $\alpha = s \to s'$ needs to have the *por_proviso* flag set:

$$\forall \alpha \in \mathcal{T}(s) : \mathit{por\_proviso}(\alpha) \vee \mathcal{T}(s) = \mathit{enabled}(s)$$

4.6.4 Implementing the proviso

The algorithm is responsible for setting the *por_proviso* flag for each of the successor states. The value of the flag can depend on the chosen version of the proviso: the closed set proviso, the stack proviso or the color proviso.

To set the proviso flag for the successor state, the LTSmin framework is adapted as follows:

```
function $GSEA(s)$
    $init()$;
    $open\_insert(s)$;
    while $has\_open()$ do
        $s = open\_extract()$;
        $closed\_insert(s)$;
        // generate next states (next_all + callback)
        foreach $s' \in s \to s'$ do
            $state\_process(s')$;
            $s'.por\_proviso = por\_proviso(s')$;
            if $open\_insert\_condition(s')$ then
                $open\_insert(s')$;
            else
                $state\_matched(s')$;
```

Algorithm 7: General State Expanding Algorithm + Proviso

Depending on the implementation of the *por_proviso* function, the proviso flag can now be set. Note that a different proviso can be chosen without changing the search algorithm, while still being compatible with the SCC algorithm.

4.6.5 The closed set proviso

This is the default proviso

```
function $por\_proviso(s)$
    return $s \notin Closed$;
```

Algorithm 8: The closed set proviso

This proviso is the most straightforward proviso and works well with other search strategies for GSEA, like best-first and A* [7]. It is however not necessary the best proviso for a good reduction when using a DFS search.

4.6.6 The stack proviso

The stack proviso is the best known proviso [23, 22, 27]. It is likely to result in a better reduction than the closed set proviso because more states in the state space will lead to a por_proviso flag that is set to true. Only successor states that are on the current search stack set the proviso flag to false. Contrary to the closed set proviso, closed states that are not on the stack will set the proviso flag to true too. In order to implement the stack proviso we keep the set of closed states that are not on the stack anymore. These are precisely the states that have been backtracked. By overwriting the state_backtrack function we create a set *Offstack* of all the states that are not on the stack anymore. These states either do not form a cycle of are in a cycle with a fully expanded state.

```
function $state\_backtrack(s)$
    $Offstack = Offstack \cup \{s\}$;
function $por\_proviso(s)$
    return $s \notin Closed \vee s \in Offstack$;
```

Algorithm 9: General State Expanding Algorithm

Due to the framed stack implementation where more then one successor state is pushed on the stack at once, it might be the case that a successor state is generated and put on the stack more then once. This is popped off (*state_ backtrack_ drop*) again by the implementation of *has_open* (see section 4.6.1), but might have consequences for the validity of the persistent set that was generated. In the following section it is proven that the generated persistent set is always correct:

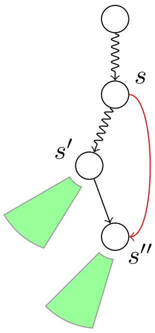


Figure 4.5: The off-stack proviso

Proof of correctness

The following assumption holds:

1. For each state, the *state_ backtrack* function is called precisely once, when it is popped off the stack

We use an *Offstack* set that marks states as closed and off the stack. Each time a successor state is generated, one of the following possibilities hold.

1. The successor state is new ($s \notin Closed$)
2. The successor state is on the stack, in the current trace (being explored) ($s \in Closed, s \notin Offstack$)
3. The successor state is off the stack and backtracked ($s \in Closed, s \in Offstack$)

In the last two cases (2 and 3), no future execution can change the result of the por off-stack proviso. In both cases the successor state is added to the *Closed* set, thus will never be pushed to the stack in any future execution (see 4.6.1). Only when a state is new, future executions may regenerate the same state (see Figure 4.5) which might interfere with the correctness of the persistent set.

We will show that a new state can only change to closed and off the stack. In any future trace starting from s, for each successor s'' from s that is new, either there exist some state s' reachable from s that regenerates s'' (Figure 4.5), or s'' doesn't reappear in the trace. In the latter case, s'' will still be new when backtracking. In the first case, since s'' has been regenerated in some trace via s', it must have been backtracked before s'. Without loss of generality, we assume $s \to s'$ is explored before $s \to s''$ (otherwise, s'' wouldn't be pushed on the stack since it was closed already). Since transition $s \to s'$ is searched before $s \to s''$, and s'' is backtracked precisely once, and before s', in proper dfs postorder $s \to s''$ should result in s'' being closed and backtracked ($s \in Offstack$) instead of being new ($s \notin Closed$). Therefore, the valuation of the *por_ proviso* might have been invalid while generating the successor. But since the valuation of *por_ proviso* for new (case 1) and closed, off the stack (case 3) states is the same, the generated persistent set is still correct. □

4.6.7 The color proviso

The color proviso [16] needs to keep track of the color of each state. Note that states that are not on the stack can either be red or green, and only states that are on the stack can be orange. Furthermore, for each state still on the stack the number of fully expanded states must be stored. This makes it possible to detect that a cycle back to the stack contains a fully expanded state ($s.expanded > s'.expanded$). The number of fully expanded states must be assigned to a each state in the order the states are explored by the dfs algorithm. Therefore overwriting the *state_ next* function is suitable for assigning it. There is however not easy way to mark a state as fully expanded, except by explicit communication with the POR wrapper. Alternatively, it is possible to assume that no state is fully expanded. In this case the algorithm will still be correct but will not result in better reduction than the stack proviso. A third option is to approximate the knowledge of fully expanded by evaluating the liveness condition that all successor states must have the *por_proviso* flag set to true. In case there exist a successor which has the flag set to false it is certain that the POR wrapper must return $enabled(s)$. This has been implemented to make the implementation of the proviso completely independent with the POR wrapper. Contrary to the stack proviso, the generated persistent set is not always a valid one. It can happen that a successor state s'' is generated from s and is regenerated from s' like is the case with the off stack proviso (Figure 4.5). The first time it is generated s'' is new and will have the orange color. Then, the second time, on backtracking this color can change to either green or red. Whenever it turns out green it is fine, but when it becomes red the persistent set generated from s was invalid. To overcome this problem it is possible to regenerate the successors of s using the new information available, replacing the successor states on the stack frame (this can only add successors). The implementation can overwrite the *state_backtrack_drop* to detect this. Experiments of Evangelista and Pajault [16] show that this will happen in few occasions during exploration of the state space. Note that this implementation differs from the original implementation because the detection of the red successor is delayed, which likely results in a bit less reduction. Another difference is that the information that a state is fully expanded is not available, and under-approximated, resulting in less reduction, too.

```
function $state\_next(s)$
    $global.fully\_expanded = false$;
    $parent.state\_next(s)$;
    $expanded = s.expanded$;
    if $global.fully\_expanded$ then
        $s.color = green$;
        $expanded++$;
    // mark all successors on the stack
    foreach $s' \in s \to s'$ do
        $s'.expanded = expanded$;
function $state\_backtrack(s)$
    if $\neg s.color = green$ then
        if $s.all\_successors\_green$ then
            $s.color = green$;
        else
            $s.color = red$;
function $state\_backtrack\_drop(s')$
    peek $s$ off the previous stack frame;
    if $\neg s'.color = green$ then
        if $s.color = orange$ then
            // clear stack frame
            while $stack\_frame\_size() > 0$ do
                $pop(Stack)$
            $this.state\_next(s)$;
            $s.color = green$;
            // push successor to drop it again
            $push(Stack, s')$;
function $por\_proviso(s')$
    peek $s$ off the previous stack frame;
    $proviso = s' \notin Closed \lor s'.color = green \lor (s'.color = orange \land s'.dfsnum < s.dfsnum)$;
    if $s'.color = orange \land s'.dfsnum = s.dfsnum$ then
        $s.all\_successors\_green = false$
    $global.fully\_expanded = global.fully\_expanded \lor \neg proviso$;
    return $proviso$;
```

Algorithm 10: Color Proviso

4.7 Evaluation

To evaluate the effectiveness of the partial order reduction algorithm in combination with on the fly LTL model checking, LTL formulas (Table 4.2) extracted from the BEEM database have been used in combination with the corresponding models. Table 4.3 shows the results of a) just the LTL wrapper in combination with the model b) a breadth first exploration where the closed set proviso is used c) a depth first exploration where the closed set proviso is used. As can be seen from the table, the closed set proviso works best for breadth first search exploration of the state space. For depth first search there is little to no reduction, with some exceptions where there is a lot of reduction. Since verifying LTL on the fly requires either a SCC or NDFS algorithm, and both are based on DFS, the lack of reduction for DFS makes the closed set proviso unsuitable for most models. The benefit of the proviso is that there is little to no extra overhead, and it does work for some models.

Since a DFS search is required for on the fly LTL model checking, we compare the different DFS

Table 4.2: LTL properties

name	ltl property
anderson.prop2.ltl	! ([] ((P_0 == 1 | P_0 == 2 | P_0 == 3) -> <> P_0 == 4))
anderson.prop3.ltl	! ([] ((P_0 == 0 | P_0 == 4) -> <> (P_0 == 1 | P_0 == 2 | P_0 == 3)))
at.prop2.ltl	! ([] ((P_0 == 2) -> <> (P_0 == 9)))
bakery.prop2.ltl	! ([](((P_0 == 1) | (P_0 == 2) | (P_0 == 3)) -> <> (P_0 == 4)))
bopdp.prop1.ltl	! ([] (((sleeping == 0) & (sleep_op == 0)) -> ((! (sleep_op == 0)) R ((sleeping == 0) | (! (sleep_op == 0))))))
brp.prop2.ltl	! ([]((Producer == 2) -> <> (Producer == 3)))
elevator2.prop1.ltl	! ([]((req[0] == 1)->(<>((p==1) & (cabin == 2)))))
iprotocol.prop3.ltl	! ([] (<> (Consumer == 1)))
lann.prop3.ltl	! ([] <> (P_0 == 1))
leader_election.prop2.ltl	! (<> !(nr_leaders == 0))
lifts.prop2.ltl	! (([] <> (Environment == 1)) -> ([] <> (Wheels == 2)))
pgm_protocol.prop3.ltl	! ([] (((receiver == 6) & (RXW_LEAD == 2)) -> <> ((sender == 1) & (packet == 3))))
phils.prop1.ltl	! ([] <> (phil_0 ==2))
protocols.prop2.ltl	! (<> ((Consumer == 2) | (Consumer == 3)))
train-gate.prop2.ltl	! ([] ((Train_1 == 3) -> <> (Train_1 == 2)))

proviso as discussed in this chapter. Table 4.4 shows a) the unreduced state space b) a depth first search exploration with closed set proviso c) a depth first search exploration with stack proviso d) a depth first search exploration with color proviso. Table 4.4 shows that the stack proviso in general improves the reduction a lot over the closed set proviso. This was to be expected as states that are off the stack are closed, and the closed set proviso can't distinguish these from closed states on the stack. The color proviso (with the LTSmin adaptations) in general results in a comparable result to the stack proviso. There are some models where results in more reduction (*dining philosophers, anderson*).

Although the results shown in Table 4.4 do not seem to be extraordinary, the appear to be better than the results obtained by the DiVinE-2.4 native POR algorithm (shown in Table 4.5). There is no reduction for models not in this table.

4.8 Discussion and future work

The modular algorithm described here makes it possible to easily switch between different provisos. Nevertheless, the reduction is not as convincing as it could be. One of the reasons why the reduction is not good enough might be the restricted communication with the search algorithms, effectively limiting knowledge available to compute persistent sets satisfying conditions C2 and C3. One major limitation currently is that (in)visibility is statically computed. It might be beneficial to compute (in)visibility of a transition group on the fly. This however requires constant communication with the LTL wrapper, making the wrappers more intermingled. Similarly, knowing in advance the evaluation of the *por_proviso* flag could improve the reduction, yet requires communication with the back-end algorithm.

Note that the implementation of the color proviso would be much easier if the computation of the persistent set could be delayed as long as states are on the stack. In other words, the POR wrapper should emit one successor state and then the back-end search should continue. When backtracking the POR wrapper should be asked to emit a state again. This way the POR wrapper will not emit erroneous persistent sets (i.e. sets for which condition $C3^L$ doesn't hold).

The modular approach where the proviso can be switched can pay off. Using partial order reduction, the resulting error traces are usually a lot shorter, and that errors are detected much sooner in the model checking process. A nice side effect of the ability to change the proviso is that the different proviso can lead to different error traces and the user can easily switch proviso to find the most readable trace. Having the closed set proviso as default will result in less overhead and possibly a nice reduction. If this is not the case, a better proviso such as the stack- or color proviso can be chosen. Although the cost of calculating the color proviso may be excessive for some models while it may help for others. We do not observe the same amount of improvement using the color proviso as seen by experiments by Evangelista and Pajault [16], but this might be caused by the changes needed for the modular implementation. Preliminary results seem to indicate that adding a communication channel to the POR wrapper to obtain the *fully_expanded*

Table 4.3: Closed set proviso

model	algorithm	states		transitions		walltime [sec]	memory [vsize]
		count	percent	count	percent		
anderson.6		29315027	100.0	181632960	100.0	272.52	809120KB
+ prop2	bfs+por	6396905	21.8	26105842	14.4	116.98	183944KB
	dfs+por	27716425	94.5	159337550	87.7	491.99	954932KB
anderson.6		27088324	100.0	165086026	100.0	252.41	774544KB
+ prop3	bfs+por	8317813	30.7	40803840	24.7	165.34	257684KB
	dfs+por	26096976	96.3	148189066	89.8	497.99	908320KB
at.5		58764238	100.0	234503620	100.0	317.78	1511156KB
+ prop2	bfs+por	52408658	89.2	191806454	81.8	1425.04	938092KB
	dfs+por	55742666	94.9	212550310	90.6	1556.04	1587960KB
bakery.7		51104603	100.0	248438907	100.0	379.02	1020000KB
+ prop2	bfs+por	49414239	96.7	218146564	87.8	924.75	1020056KB
	dfs+por	51104603	100.0	246481911	99.2	1001.01	1062196KB
bopdp.3		1414004	100.0	4603047	100.0	8.51	84260KB
+ prop1	bfs+por	1187988	84.0	2796778	60.8	19.42	74748KB
	dfs+por	1363961	96.5	3947860	85.8	23.71	78408KB
brp.6		84553381	100.0	262786191	100.0	394.23	1979276KB
+ prop2	bfs+por	48101417	56.9	102214733	38.9	402.72	935688KB
	dfs+por	83768893	99.1	244292444	93.0	812.30	1989396KB
elevator2.1		3292	100.0	16280	100.0	0.10	3688KB
+ prop1	bfs+por	3189	96.9	11419	70.1	0.10	3688KB
	dfs+por	3263	99.1	14030	86.2	0.10	3688KB
iprotocol.6		62195661	100.0	258517149	100.0	654.83	1535884KB
+ prop3	bfs+por	35566810	57.2	74147439	28.7	380.05	954720KB
	dfs+por	58663962	94.3	190513741	73.7	795.77	1538304KB
lann.5		1986963	100.0	10368751	100.0	23.92	82844KB
+ prop3	bfs+por	668349	33.6	1681005	16.2	8.01	63216KB
	dfs+por	1982265	99.8	9333477	90.0	33.92	84960KB
leader_election.6		35773430	100.0	233181690	100.0	4136.49	2287700KB
+ prop2	bfs+por	231005	0.6	299749	0.1	10.81	59544KB
	dfs+por	31352420	87.6	178259472	76.4	3841.73	2082432KB
lifts.7		10150585	100.0	41100242	100.0	101.67	246792KB
+ prop2	bfs+por						
	dfs+por						
pgm_protocol.3		201298	100.0	389567	100.0	3.60	59584KB
+ prop3	bfs+por	139670	69.4	174490	44.8	4.30	55240KB
	dfs+por	139670	69.4	174912	44.9	4.40	57868KB
phils.5		1062879	100.0	11849119	100.0	18.41	80384KB
+ prop1	bfs+por	1037387	97.6	8831073	74.5	19.52	81232KB
	dfs+por	1062879	100.0	10735859	90.6	24.62	100344KB
protocols.2		1464	100.0	3241	100.0	0.10	3688KB
+ prop2	bfs+por	127	8.7	202	6.2	0.10	3688KB
	dfs+por	129	8.8	208	6.4	0.10	3688KB
train-gate.5		1389185	100.0	3055280	100.0	6.20	80880KB
+ prop2	bfs+por	547794	39.4	1269138	41.5	9.81	58344KB
	dfs+por	595992	42.9	1390114	45.5	10.41	61336KB

Table 4.4: POR proviso

model	algorithm	states		transitions		walltime [sec]	memory [vsize]
		count	percent	count	percent		
anderson.6		29315027	100.0	181632960	100.0	272.52	809120KB
+ prop2	dfs+por	27716425	94.5	159337550	87.7	491.99	954932KB
	dfs+stack-por	9264497	31.6	34982375	19.3	163.23	332860KB
	dfs+color-por	6768620	23.1	28941520	15.9	140.71	256948KB
anderson.6		27088324	100.0	165086026	100.0	252.41	774544KB
+ prop3	dfs+por	26096976	96.3	148189066	89.8	497.99	908320KB
	dfs+stack-por	11048925	40.8	48885316	29.6	214.07	357416KB
	dfs+color-por	10390873	38.4	49317882	29.9	223.55	349180KB
at.5		58764238	100.0	234503620	100.0	317.78	1511156KB
+ prop2	dfs+por	55742666	94.9	212550310	90.6	1556.04	1587960KB
	dfs+stack-por	51624514	87.9	184132445	78.5	1468.27	1037352KB
	dfs+color-por	51458556	87.6	190004430	81.0	1525.87	1238048KB
bakery.7		51104603	100.0	248438907	100.0	379.02	1020000KB
+ prop2	dfs+por	51104603	100.0	246481911	99.2	1001.01	1062196KB
	dfs+stack-por	49236840	96.3	216835275	87.3	953.09	1062020KB
	dfs+color-por	49236028	96.3	218969626	88.1	977.27	1269300KB
bopdp.3		1414004	100.0	4603047	100.0	8.51	84260KB
+ prop1	dfs+por	1363961	96.5	3947860	85.8	23.71	78408KB
	dfs+stack-por	1048757	74.2	2336657	50.8	17.61	78920KB
	dfs+color-por	1048853	74.2	2338536	50.8	17.71	84052KB
brp.6		84553381	100.0	262786191	100.0	394.23	1979276KB
+ prop2	dfs+por	83768893	99.1	244292444	93.0	812.30	1989396KB
	dfs+stack-por	43219019	51.1	85372935	32.5	372.98	946744KB
	dfs+color-por	43218991	51.1	85372879	32.5	376.96	1136280KB
elevator2.1		3292	100.0	16280	100.0	0.10	3688KB
+ prop1	dfs+por	3263	99.1	14030	86.2	0.10	3688KB
	dfs+stack-por	3181	96.6	10526	64.7	0.10	3688KB
	dfs+color-por	3182	96.7	10959	67.3	0.10	3688KB
iprotocol.6		62195661	100.0	258517149	100.0	654.83	1535884KB
+ prop3	dfs+por	58663962	94.3	190513741	73.7	795.77	1538304KB
	dfs+stack-por	13054116	21.0	26620599	10.3	141.68	306776KB
	dfs+color-por	13118506	21.1	27059315	10.5	145.70	369372KB
lann.5		1986963	100.0	10368751	100.0	23.92	82844KB
+ prop3	dfs+por	1982265	99.8	9333477	90.0	33.92	84960KB
	dfs+stack-por	16628	0.8	27082	0.3	0.20	49364KB
	dfs+color-por	15909	0.8	25941	0.3	0.20	50652KB
leader_election.6		35773430	100.0	233181690	100.0	4136.49	2287700KB
+ prop2	dfs+por	31352420	87.6	178259472	76.4	3841.73	2082432KB
	dfs+stack-por	231005	0.6	299749	0.1	10.91	63876KB
	dfs+color-por	231005	0.6	299749	0.1	10.91	64908KB
lifts.7		10150585	100.0	41100242	100.0	101.67	246792KB
+ prop2	dfs+por						
	dfs+stack-por						
	dfs+color-por						
pgm_protocol.3		201298	100.0	389567	100.0	3.60	59584KB
+ prop3	dfs+por	139670	69.4	174912	44.9	4.40	57868KB
	dfs+stack-por	139670	69.4	174283	44.7	4.40	59924KB
	dfs+color-por	139670	69.4	174283	44.7	4.40	60696KB
phils.5		1062879	100.0	11849119	100.0	18.41	80384KB
+ prop1	dfs+por	1062879	100.0	10735859	90.6	24.62	100344KB
	dfs+stack-por	387921	36.5	1839551	15.5	5.21	66304KB
	dfs+color-por	167362	15.7	531127	4.5	1.90	57400KB
protocols.2		1464	100.0	3241	100.0	0.10	3688KB
+ prop2	dfs+por	129	8.8	208	6.4	0.10	3688KB
	dfs+stack-por	129	8.8	208	6.4	0.10	3688KB
	dfs+color-por	127	8.7	208	6.4	0.10	3688KB
train-gate.5		1389185	100.0	3055280	100.0	6.20	80880KB
+ prop2	dfs+por	595992	42.9	1390114	45.5	10.41	61336KB
	dfs+stack-por	552215	39.8	1277980	41.8	9.81	62832KB
	dfs+color-por	548309	39.5	1270415	41.6	9.81	67196KB

Table 4.5: DiVinE 2.4 POR

model	states		transitions	
	count	percent	count	percent
brp.6.prop.2	84553381	100	263092767	100
	52964221	62	122582742	46
iprotocol 6.prop3	62195661	100	258517149	100
	28077744	45	95705273	37
leader_election.6.prop2	35773430	100	233181690	100
	4532159	12	8093347	3
pgm_protocol.3.prop3	201298	100	388004	100
	187780	93	348306	89
protocols.2.prop3	1464	100	3241	100
	761	51	1348	41

flag does not improve the reduction a lot. An interesting alternative to set the *por_proviso* flag may be provided by the extra information obtained through the SCC computation. Another interesting question to answer in future research is whether it is possible to switch the order of the POR and LTL wrapper (Figure 4.2). This way, the POR wrapper could be used both in combination with the LTL wrapper, or directly with a front-end providing the cross product with the büchi automata. The reason that this currently is not possible is that the LTL wrapper extends the state vector with the büchi automaton, and marks this part as write for all transition groups. The partial order reduction wrapper is not able to compute a proper dependency relation. Another application of the POR wrapper would be LTL model checking in a multi-core or distributed setting.

4.9 Conclusion

The LTL model checking algorithms and PINS wrapper can successfully be combined with a PINS partial order reduction wrapper. For some models the reduction provided through this setup results in a lot of reduction, either in memory or in speed. Besides the advantage of the memory reduction, the partial order reduction wrapper may cause the traces found by the SCC algorithm to be much shorter, and errors to be found much faster.

For partial order reduction to be applicable in general, it is best to use the stack set implementation. This seems to be easy to implement, require a reasonable amount of memory and produce a good reduction where possible. The color set proviso can be used in some rare cases to get more reduction, but it requires a more involved implementation and uses more memory. The closed set proviso integrates best with the GSEA algorithm and can be used for other types of searches such as A* and best first search as well. Besides these advantages for some models it provides a reduction comparable to the stack proviso.

To improve the partial order reduction in combination with LTL it is likely needed to have more communication with the algorithm and LTL wrapper. Several things may improve the reduction: a) on the fly (in)visibility computation b) communicating the next states with the algorithm before emitting a persistent set. These modifications are likely to cause more intermingled code for the proviso or persistent set computations.

Bibliography

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[2] 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.

[3] S. Blom, J. R. Calamé, B. Lisser, S. Orzan, J. Pang, J. Van De Pol, M. T. Dashti, and A. J. Wijs. Distributed analysis with μcrl: a compendium of case studies. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 683–689, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] S. Blom, B. Lisser, J. P. van de, and M. Weber. A database approach to distributed state space generation. In *Sixth International Workshop on Parallel and Distributed Methods in verifiCation, PDMC*, pages 17–32, Enschede, July 2007. CTIT.

[5] S. Blom, J. P. van de, 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, Berlin, July 2010. Springer Verlag. See also Technical Report TR-CTIT-09-30 (http://eprints.eemcs.utwente.nl/15703/).

[6] D. Bošnački and G. J. Holzmann. Improving spin's partial-order reduction for breadth-first search. In P. Godefroid, editor, *Model Checking Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 91–105. Springer Berlin / Heidelberg, 2005.

[7] D. Bosnacki, S. Leue, and A. L. Lafuente. Partial-order reduction for general state exploring algorithms, 2006.

[8] A. Cimatti, E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: BDD-based + SAT-based symbolic model checking.

[9] A. Cimatti, E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Integrating BDD-based and SAT-based symbolic model checking. In *Proceedings of the 4th International Workshop on Frontiers of Combining Systems*, FroCoS '02, pages 49–56, London, UK, 2002. Springer-Verlag.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin / Heidelberg, 2000.

[11] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16:1512–1542, September 1994.

[12] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model checking.* MIT Press, Cambridge, MA, USA, 1999.

[13] M. de Jonge and T. C. Ruys. The spinja model checker. In *Proceedings of the 17th international SPIN conference on Model checking software*, SPIN'10, pages 124–128, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] S. Edelkamp, StefanLeue, and A. Lluch-lafuente. Partial-order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology Transfer*, 6:277–301, 2004.

[15] E. A. Emerson. 25 years of model checking. chapter The Beginning of Model Checking: A Personal Perspective, pages 27–45. Springer-Verlag, Berlin, Heidelberg, 2008.

[16] S. Evangelista and C. Pajault. Solving the ignoring problem for partial order reduction. *Int. J. Softw. Tools Technol. Transf.*, 12:155–170, May 2010.

[17] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.

[18] A. Gaiser and S. Schwoon. Comparison of algorithms for checking emptiness on büchi automata. In P. Hlinený, V. Matyáš, and T. Vojnar, editors, *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, volume 13 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[19] H. Garavel. Open/cæsar: An open software architecture for verification, simulation, and testing. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 68–84, London, UK, 1998. Springer-Verlag.

[20] H. Garavel, R. Mateescu, F. r. Lang, and W. Serwe. CADP 2006: a toolbox for the construction and analysis of distributed processes. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 158–163, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] J. Geldenhuys, H. Hansen, and A. Valmari. Exploring the scope for partial order reduction. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*, ATVA '09, pages 39–53, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] P. Godefroid. Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem, 1995.

[23] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Form. Methods Syst. Des.*, 2:149–164, April 1993.

[24] J. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. v. Weerdenburg, W. Wesselink, T. Willemse, and J. v. d. Wulp. The mCRL2 toolset. In *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, 2008.

[25] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *In The Spin Verification System*, pages 23–32. American Mathematical Society, 1996.

[26] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23:279–295, May 1997.

[27] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proceedings of the IFIP TC6/WG6.1 Twelth International Symposium on Protocol Specification, Testing and Verification XII*, pages 349–363, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[28] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 197–211, London, UK, UK, 1995. Chapman & Hall, Ltd.

[29] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 398–413, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Static partial order reduction. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 345–357, London, UK, 1998. Springer-Verlag.

[31] A. Mazurkiewicz. Trace theory. In *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[32] M. Mukund. Linear-time temporal logic and büchi automata. Tutorial talk, Winter School on Logic and Computer Science, Indian Statistical Institute, Calcutta, 1997.

[33] R. Nalumasu and G. Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Form. Methods Syst. Des.*, 20:231–247, May 2002.

[34] W. T. Overman and S. D. Crocker. Verification of concurrent systems: Function and timing. In *Proceedings of the IFIP WG6.1 Second International Workshop on Protocol Specification, Testing and Verification*, pages 401–409, Amsterdam, The Netherlands, The Netherlands, 1982. North-Holland Publishing Co.

[35] R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[36] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, pages 377–390, London, UK, 1994. Springer-Verlag.

[37] D. Peled. Partial order reduction: Model-checking using representatives. In W. Penczek and A. Szalas, editors, *Mathematical Foundations of Computer Science 1996*, volume 1113 of *Lecture Notes in Computer Science*, pages 93–112. Springer Berlin / Heidelberg, 1996.

[38] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Inf. Process. Lett.*, 63:243–246, September 1997.

[39] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[40] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In N. Halbwachs and L. D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer Berlin / Heidelberg, 2005.

[41] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pages 491–515, London, UK, 1991. Springer-Verlag.

[42] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 382–396, Berlin, Heidelberg, 2008. Springer-Verlag.

[43] M. Weber. An embeddable virtual machine for state space generation. In *Proc. 14th SPIN Workshop, LNCS 4595 (2007)*, pages 168–185.