Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Guard-based Partial-Order Reduction (Extended Version)

Alfons Laarman,[1,2] Elwin Pater,[2] Jaco van de Pol,[2]Henri Hansen[3]

[1] Formal Methods in Systems Engineering, Vienna University of Technology, Austria[**]
e-mail: `alfons@laarman.com`
[2] Formal Methods and Tools, University of Twente, The Netherlands
e-mail: `vdpol@cs.utwente.nl, elwin.pater@gmail.com`
[3] Department of Mathematics, Tampere University of Technology, Finland
e-mail: `henri.hansen@tut.fi`

The date of receipt and acceptance will be inserted by the editor

Abstract. This paper aims at making partial-order reduction independent of the modeling language. To this end, we present a guard-based method which is a general-purpose implementation of the stubborn set method. We approach the implementation through so-called *necessary enabling sets* and *do-not-accord sets*, and give an algorithm suitable for an abstract model checking interface. We also introduce *necessary disabling sets* and heuristics to produce smaller stubborn sets and thus better reduction at low costs. We explore the effect of these methods using an implementation in the model checker LTSmin.

We experiment with partial-order reduction on a number of Promela models, on benchmarks from the BEEM database in the DVE language, and with several with LTL properties. The efficiency of the heuristic algorithm is established by a comparison to the subset-minimal Deletion algorithm and the simple closure algorithm. We also compare our results to the Spin model checker. While the reductions take longer, they are consistently better than Spin's ample set and often surpass the upper bound for the process-based ample sets, established empirically earlier on BEEM models.

1 Introduction

Model checking is an automated method of verifying the correctness of concurrent systems by examining all possible execution paths for incorrect behaviour. The main challenge for model checking is the *state space explosion*, which refers to the exponential growth in the number of states obtained by interleaving executions of several system components. Model checking emerged in the 1980s [5] and several advances have pushed its boundaries. Among those advances, partial-order reduction is one of the most prominent examples.

Partial-order reduction (POR) exploits the fact that in concurrent systems, not all orderings or interleavings of simultaneously enabled transitions need to be explored. It has been characterized as model checking with representatives [39], because instead of an exhaustive search, verification needs to consider only a subset of all possible successors of each state to ensure all behaviours of interest to the verified property are preserved.

The idea to exploit commutativity between concurrent transitions has been investigated by several researchers, leading to various algorithms for computing sufficient successor sets. The challenge is to compute this subset during state space generation (on-the-fly), using syntactic and static information obtained from the system description.

Already in 1981, Overman [35] suggested a method to avoid exploring all interleavings, followed by Valmari's [46,50,49] *stubborn sets* in 1988, 1991 and 1992. Also from 1988 onwards, Peled [22] developed the *ample set* [39,40], later extended by Holzmann and Peled [19,40], Godefroid and Pirottin [13,15] the *persistent set* [14], and Godefroid and Wolper [16] *sleep sets*. These foundations have been extended and applied in numerous papers over the past 15 years.

Problem and Contributions. Previous work defines partial-order reduction in terms of different formalisms: Petri-nets [56], parallel components with local program counters, called processes [19,14], a parallel composition of labeled transition systems [53], as well as the more general transition/variable systems [48,50]. While focus on a specific formalism allows the exploitation of formalism-specific properties, like *fairness* [40] and token conditions [52], it also complicates the application to other formalisms, for instance, rule-based systems [9]. Moreover, most current implementations are tightly coupled with their particular specification languages. Our approach can be applied to any of these settings as long as the necessary abstractions such as guards, transition accordance, and necessary enabling and disabling sets are identified.

The Pins interface [3,31] (see Section 4.4) is one solution that allows for separating language front-ends from verification algorithms. Through Pins (Partitioned Interface to the Next-State function), a user can use various high-performance model checking algorithms for his favourite specification language, cf. Figure 1. Providing POR as Pins2Pins wrapper once and for all benefits every combination of language and algorithm. An important question is whether and how an abstract interface like Pins can support partial-order reduction.

[**] Sup. by Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005.

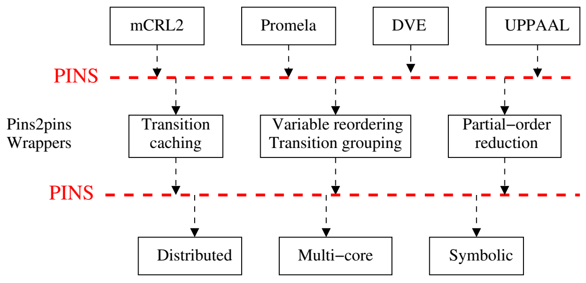

Figure 1. Modular PINS architecture of LTSMIN

We propose a solution that is based on stubborn sets. This theory shows how to choose a subset of transitions, enabled and disabled, based on a careful analysis of their independence and commutativity relations. These relations have been described on the abstract level of transition systems before [50]. Additionally, within the context of Petri-nets, the relations were refined to include multiple enabling conditions, a natural distinction in this formalism [52].

In Section 3, we define the theory of stubborn sets, and provide a general purpose version of it that is suitable for implementation in a complete language independent setting. Our approach assumes only that transitions have guard conditions that can be enabled and disabled by other transitions.

In Section 4, we extend PINS with the necessary information: a do-not-accord matrix and optional; necessary enabling matrix on guards. In addition, we introduce novel *necessary disabling sets* and a new heuristic-based selection criterion. As optimal stubborn sets are expensive to compute precisely [52], our heuristic finds reasonably effective stubborn sets fast, hopefully leading to smaller state spaces. In Section 5, we show how LTL can be supported.

Our implementation resides in the LTSMIN toolset [3], based on PINS. Any language module that connects to PINS now obtains POR without having to bother about its implementation details, it merely needs to export transition guards and their dependencies via PINS. We demonstrate this by extending the front-end in LTSMIN for DVE and PROMELA [2]. This allows a direct comparison to SPIN [18] (Section 7), which shows that the new algorithm generally provides more reduction using less memory, but takes more time to do so. We demonstrate that the method yields more reduction than the best reduction using process-based ample sets that rely on dynamic enabling and disabling relations, reported in the empirical work by Geldenhuys et al. [12] on the DVE BEEM benchmarks [38].

Summarising, these are the main contributions presented in this work:
1. *Guard-based partial-order reduction*, which is a language-independent generalization of the stubborn set method;
2. Some improvements to efficiently compute smaller stubborn sets:
   (a) A refinement based on *necessary disabling sets*;
   (b) A *heuristic selection criterion* for necessary enabling sets;
3. Two language module *implementations* exporting guards with dependencies for a the model checker LTSMIN;
4. An *empirical evaluation* of guard-based partial-order reduction in LTSMIN:
   (a) A comparison of resource consumption and effectiveness of POR between LTSMIN [3], and SPIN [18] on 18 PROMELA models/5 LTL problems.
   (b) A comparison with the best reductions achieved with the *ample-set method*, as reported by Geldenhuys et al. [12], on DVE BEEM models.

Compared to the current paper's prequel [29], we now also extend guard-based partial-order reduction with:
1. The *weak stubborn set theory* which is a theoretically more powerful yet complicated version of stubborn sets;
2. A new formulation of the *deletion algorithm*, which guarantees *subset minimal* stubborn sets, and therefore provides a good baseline to compare our heuristics approach against;
3. A discussion on the implementation of the algorithms and their complexity;
4. Experiments that better illustrate the benefits and costs of the necessary disabling sets and the heuristic stubborn set calculation;
5. And experiments that demonstrate some benefits of the weak stubborn set theory for PROMELA models.

2 A Computational Model of Guarded Transitions

In the current section, we provide a model of computation comparable to [12], leaving out the notion of processes on purpose. It has three main components: states, guards and transitions. A state represents the global status of a system, guards are predicates over states, and a transition represents a guarded state change.

Definition 1 (state). Let $S = E_1 \times \ldots \times E_n$ be a set of vectors of elements with some finite domain. A state $s = \langle e_1, \ldots, e_n \rangle \in S$ associates a value $e_i \in E_i$ to each element. We denote a projection to a single element in the state as $s[i] = e_i$.

Definition 2 (guard). A guard $g : S \to \mathbb{B}$ is a total function that maps each state to a boolean value, $\mathbb{B} = \{\textbf{true}, \textbf{false}\}$. We write $g(s)$ or $\neg g(s)$ to denote that guard g is **true** or **false** in state s. We also say that g is enabled/disabled.

Definition 3 (structural transition). A *structural transition* $t \in \mathcal{T}$ is a tuple $(\mathcal{G}, a)$ such that a is an assignment $a : S \to S$ and $\mathcal{G}$ is a set of guards, also denoted as $\mathcal{G}_t$. We denote the set of enabled transitions by $en(s) := \{t \in \mathcal{T} \mid \bigwedge_{g \in \mathcal{G}_t} g(s)\}$. We write $s \xrightarrow{t}$ when $t \in en(s)$, $s \xrightarrow{t} s'$ when $s \xrightarrow{t}$ and $s' = a(s)$, and we write $s \xrightarrow{t_1 t_2 \ldots t_k} s_k$, when $\exists s_1, \ldots, s_k \in S : s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_k} s_k$.

Definition 4 (state space). Let $s_0 \in S$ and let $\mathcal{T}$ be the set of transitions. The state space from s_0 induced by $\mathcal{T}$ is $M_{\mathcal{T}} = (S_{\mathcal{T}}, s_0, \Delta)$, where $s_0 \in S$ is the *initial state*, and $S_{\mathcal{T}} \subseteq S$ is the set of *reachable states*, and $\Delta \subseteq S_{\mathcal{T}} \times \mathcal{T} \times S_{\mathcal{T}}$ is the set of *semantic transitions*. These are defined to be the smallest sets such that $s_0 \in S_{\mathcal{T}}$, and if $t \in \mathcal{T}$, $s \in S_{\mathcal{T}}$ and $s \xrightarrow{t} s'$, then $s' \in S_{\mathcal{T}}$ and $(s, t, s') \in \Delta$.

Guards or conditions are used also in [52, Def. 6], where they take the role of enabling conditions for disabled transitions. We explore their role on disabling of transitions as well for our necessary disabling sets in Section 4.2.

In the rest of the paper, we fix an arbitrary set of vectors $S = E_1 \times \ldots \times E_n$, initial state $s_0 \in S$, and set of transitions $\mathcal{T}$, with induced reachable state space $M_{\mathcal{T}} = (S_{\mathcal{T}}, s_0, \Delta)$. We often just write "transition" for elements of $\mathcal{T}$.

It is easy to see that our model is general enough to express processes as in [12]; by considering the program counter of each process as a normal state variable, a separate guard can check its current value, and a transition can update its value. Moreover, the definition can also be applied to models without any natural notion of a fixed set of processes, for instance, rule-based systems such as the linear process equations in mCRL [9].

Besides guarded transitions, structural information is required on the exact involvement of state variables in a transition.

Definition 5 (disagree sets). Given states $s, s' \in S$, for $1 \leq i \leq n$, we define the set of indices on which s and s' disagree as $\delta(s, s') := \{i \mid s[i] \neq s'[i]\}$.

Definition 6 (affect sets). For $t = (\mathcal{G}, a) \in \mathcal{T}$ and $g \in \mathcal{G}$, we define

1. the test set of g is $Ts(g) \supseteq \{i \mid \exists s, s' \in S : \delta(s, s') = \{i\} \wedge g(s) \neq g(s')\}$,
2. the test set of t is $Ts(t) := \bigcup_{g \in \mathcal{G}} Ts(g)$,
3. the write set of t is $Ws(t) \supseteq \bigcup_{s, s' \in S_{\mathcal{T}}} \delta(s, s')$ with $s \xrightarrow{t} s'$,
4. the read set of t is $Rs(t) \supseteq \{i \mid \exists s, s' \in S : \delta(s, s') = \{i\} \wedge s \xrightarrow{t} \wedge s' \xrightarrow{t} \quad \wedge \quad Ws(t) \cap \delta(a(s), a(s')) \neq \emptyset\}$ (notice the difference between S and $S_{\mathcal{T}}$), and
5. the variable set of t is $Vs(t) := Ts(t) \cup Rs(t) \cup Ws(t)$.

Although these sets are defined in the context of the complete state space, they may be statically over-approximated ($\supseteq$) by the language front-end.

Example 1. Suppose $s \in S = \mathbb{N}^3$, consider the transition: $t := \mathit{IF}\ (s[1] = 0 \wedge s[2] < 10)\ \mathit{THEN}\ s[3] := s[1] + 1$. It has two guards, $g_1 = (s[1] = 0)$ and $g_2 = (s[2] < 10)$, with test sets $Ts(g_1) = \{1\}$, $Ts(g_2) = \{2\}$, hence: $Ts(t) = \{1, 2\}$. The write set $Ws(t) = \{3\}$, so $Vs(t) = \{1, 2, 3\}$. The minimal read set $Rs(t) = \emptyset$ (since $s[1] = 0$), but simple static analysis may over-approximate it as $\{1\}$.

3 Partial-Order Reduction with Stubborn Sets

We now first give the definition of stubborn sets. We follow the definitions from [51, Section 7.4] with minor differences, and include some aspects from Godefroid's thesis [14]. Second, we explain how stubborn sets can be calculated efficiently.

3.1 Stubborn Set Theory

A stubborn set for a state s is a subset $\mathcal{T}_s \subseteq \mathcal{T}$ of all transitions (disabled and enabled) used to reduce the successors of s. We call the complement $\mathcal{T} \setminus \mathcal{T}_s$ the *non-stubborn transitions*. The non-stubborn transitions include all transitions at s that may be *omitted* from $en(s)$, possibly omitting an entire sequence of transitions. i.e. future computations that are enabled by a non-stubborn transition.

Definition 7 (Stubborn set). Given a state s, the set $\mathcal{T}_s \subseteq \mathcal{T}$ is a *stubborn set* at s, if it satisfies the following two conditions.

D1 For every $t \in \mathcal{T}_s$ and $t_1, t_2, \ldots, t_n \notin \mathcal{T}_s$, if $s \xrightarrow{t_1, \ldots, t_n t} s'_n$, then $s \xrightarrow{t t_1, \ldots, t_n} s'_n$, and

D2 Either $en(s) = \emptyset$, or there is at least one $t \in \mathcal{T}_s$ such that for every $t_1, t_2, \ldots, t_n \notin \mathcal{T}_s$, $s \xrightarrow{t_1, \ldots, t_n t}$.

It is perhaps easiest to think of the conditions as talking about the relationship between omitted transitions and stubborn transitions. D1 says that if t is stubborn, and enabled after some sequence of omitted transitions, then t is also enabled at the initial state and the omitted sequence as a whole commutes with t. This is illustrated in the following graphically; the vertical transition is stubborn while the horizontal sequence consists of non-stubborn transitions:

$$\begin{tikzcd}
s \arrow[r, "t_1"] & s_1 & \cdots & s_{n-1} \arrow[r, "t_n"] & s_n \arrow[d, "t"] \\
 & & & & s'_n
\end{tikzcd}$$

$$\Downarrow$$

$$\begin{tikzcd}
s \arrow[d, "t"] & & & & \\
s' \arrow[r, "t_1"] & s'_1 & \cdots & s'_{n-1} \arrow[r, "t_n"] & s'_n
\end{tikzcd}$$

D2 guarantees that some stubborn transition t – we call it a *key transition* – remains enabled if only non-stubborn transitions are explored. If $\mathcal{T}_s$ is a stubborn

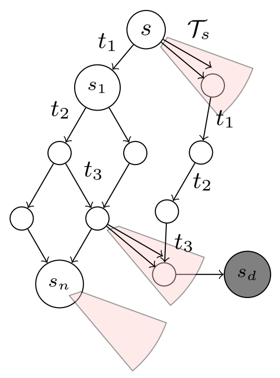

Figure 2. Stubborn set

set, we write $\mathcal{T}_s^k$ for the subset of $\mathcal{T}_s$ of transitions that satisfy D2.

D1 guarantees that we can delay the execution of non-stubborn transitions without losing the reachability of any deadlock states. Figure 2 illustrates this; since s is not a deadlock state, s_d is still reachable after executing a key transition from $\mathcal{T}_s$. The benefit is that, for the moment, we avoid exploring (and storing) states such as $s_1, \ldots, s_n$. "For the moment", because these states may still be reachable via other stubborn paths. Incidentally, this is the reason that smaller stubborn sets are only a heuristic for obtaining smaller state spaces.

This theoretical notion of stubborn sets is a semantic and *dynamic* definition, as it refers to executions starting from a given state. The future of the current state is of course not known until it is explored, so we need some *static* information that allows for computing a stubborn set. Furthermore, our definition identifies the so-called *weak* stubborn sets. Weak sets are more general than strong stubborn sets, which increases the chances of finding a set which yields better reduction [51, Sec. 7.4].

The *strong* notion of stubborn sets – which is more or less equal to ample and persistent sets - requires the D2 condition to hold for all enabled transitions of the stubborn set, or conversely, no omitted sequence is allowed to disable *any* stubborn transition. A stubborn set $\mathcal{T}_s$ is said to be a strong stubborn set if $\mathcal{T}_s \cap en(s) = \mathcal{T}_s^k$. Strong stubborn sets coincide with the stubborn sets defined in [14].

Definition 8. First, we define *strong according with* as those coenabled transitions that commute and do not disable eachother:
$A \subseteq \{(t,t') \in \mathcal{T} \times \mathcal{T} \mid \forall s, s', s_1 \in S \colon s \xrightarrow{t} s' \wedge s \xrightarrow{t'} s_1 \Rightarrow \exists s_1' \colon s' \xrightarrow{t'} s_1' \wedge s_1 \xrightarrow{t} s_1'\}$, or illustrated graphically:

$$\begin{tikzcd}
s \arrow[r, "t'"] \arrow[d, "t"] & s_1 & & s \arrow[r, "t'"] \arrow[d, "t"] & s_1 \arrow[d, "t"] \\
s' & & \Rightarrow & s' \arrow[r, "t'"] & s_1'
\end{tikzcd}$$

Its complement is the *do-not-accord* relation: $\mathcal{DNA} = \mathcal{T}^2 \setminus A$. We denote $\mathcal{DNA}_t = \{t' \mid (t,t') \in \mathcal{DNA}\}$.

Second, we define *left according with* as:
$B \subseteq \{(t,t') \in \mathcal{T} \times \mathcal{T} \mid \forall s, s', s_1' \in S \colon s \xrightarrow{t'} s' \wedge s' \xrightarrow{t} s_1' \Rightarrow \exists s_1 \colon s \xrightarrow{t} s_1 \wedge s_1 \xrightarrow{t'} s_1'\}$, or illustrated graphically:

$$\begin{tikzcd}
s \arrow[r, "t'"] & s' \arrow[d, "t"] & & s \arrow[r, "t'"] \arrow[d, "t"] & s' \arrow[d, "t"] \\
& s_1 & \Rightarrow & s_1 \arrow[r, "t'"] & s_1'
\end{tikzcd}$$

Its complement is the do-not-left-accord relation: $\mathcal{DNB} = \mathcal{T}^2 \setminus B$. We denote $\mathcal{DNB}_t = \{t' \mid (t,t') \in \mathcal{DNB}\}$ and $\mathcal{DNB}^{-1}$ for the inverse relation: $\{(t',t) \mid (t,t') \in \mathcal{DNB}\}$.

Please note the direction of the relation $\mathcal{DNB}_t$; it is vitally important for Lemma 1.
Each of the following criteria on $t, t' \in \mathcal{T}$ is sufficient to conclude strong accordance:

1. shared variables $Vs(t) \cap Vs(t')$ are disjoint from the write sets $Ws(t) \cup Ws(t')$,
2. t and t' are never co-enabled, e.g. have different program counter guards, or
3. t and t' do not disable each other, and their actions commute, e.g. write and read to a FIFO buffer, or perform atomic increments/decrements of the same variable.

The criterion 1 is sufficient for left accordance as well, but criteria 2 and 3 are not; left accordance is asymmetric, and, for instance, if t' enables t, then t' does not left accord with t.

1. If t is never enabled after t' is executed, then t' left accords with t. E.g., when the t' sets a variable to a value that always makes the guard of t false.

We defined the do-not-accord relations instead of relying on a definition of "dependent", to underline the fact that transitions modifying the same variables, for instance, can "accord", even though they are in some superficial sense "dependent". The definition of strong do-not-accord is equivalent to Godefroid's definition of do-not-accord for enabled transitions. We also need a necessary enabling relation:

Definition 9 (necessary enabling set [14]). Let $t \in \mathcal{T} \setminus en(s)$ be a disabled transition in state $s \in S_{\mathcal{T}}$. A necessary enabling set for t in s is a set of transitions $\mathcal{N}_t$, such that for all sequences of the form $s \xrightarrow{t_1,\ldots,t_n} s' \xrightarrow{t}$, there is at least one transition $t_i \in \mathcal{N}_t$, for some $1 \leq i \leq n$.

To find a necessary enabling set for a disabled transition t, which we denote with *find_nes*(t,s), Godefroid uses fine-grained analysis, which depends crucially on program counters. The analysis can be roughly described as follows:

1. If t is not enabled in global state s, because some local program counter has the "wrong" value, then use the set of transitions that assign the "right" value to that program counter as necessary enabling set;
2. Otherwise, if some guard g for transition t evaluates to **false** in s, take all transitions that write to the *test set* of that guard as necessary enabling set, i.e. include all transitions that might change g.

In Section 4, we show how to avoid program counters with guard-based POR.

Note that the above relations can be safely over-approximated: We may choose larger do-not-accord or necessary enabling relations, if our static analysis does not find the exact relation.

Lemma 1. *A set $\mathcal{T}_s$ of transitions is stubborn in a state s, if the following conditions hold for every $t \in \mathcal{T}_s$*

1. *If t is disabled in s, then $\exists \mathcal{N}_t \subseteq \mathcal{T}_s$ (multiple sets $\mathcal{N}_t$ can exist), and*
2. *If t is enabled in s, then either $\mathcal{DNA}_t \subseteq \mathcal{T}_s$, or $\mathcal{DNB}_t \subseteq \mathcal{T}_s$*
3. *$en(s) = \emptyset$ or $\exists t \in \mathcal{T}_s \cap en(s) \colon \mathcal{DNA}_t \subseteq \mathcal{T}_s$*

Proof. Assume that $\mathcal{T}_s$ satisfies the above conditions, and that $t_1, t_2, \ldots, t_n \notin \mathcal{T}_s$.

Firstly, let $t \in \mathcal{T}_s$, and $s \xrightarrow{t_1,\ldots,t_n t} s'_n$. If t is disabled in s, then $\mathcal{N}_t \subseteq \mathcal{T}_s$, by the above. But by definition of $\mathcal{N}_t$, for at least one $1 \leq i \leq n$, $t_i \in \mathcal{N}_t$, which leads to contradiction. Therefore, t must be enabled in s. $s \xrightarrow{tt_1,\ldots,t_n} s'_n$ follows by induction: when $n = 1$, this follows from the definitions of both $\mathcal{DNA}_t$ and $\mathcal{DNB}_t$. Suppose then that $s \xrightarrow{t_1,\ldots,t_i t} s'_i$ implies that $s \xrightarrow{tt_1,\ldots,t_i} s'_i$ for $i \leq n$. If $\mathcal{DNA}_t \subseteq \mathcal{T}_s$, then $s \xrightarrow{t_1,\ldots,t_i t} s'_i$ holds for every i, in particular for $i = n - 1$ and $t_n \notin \mathcal{DNA}_t$ gives D1. On the other hand, if $\mathcal{DNB}_t \subseteq \mathcal{T}_s$, then $s \xrightarrow{t_1,\ldots,t_n t}$ implies $s \xrightarrow{t_1,\ldots,t_{n-1} t}$ must hold, because $t_n \notin \mathcal{DNB}_t$, which combined with the inductive hypotheses implies D1.

If $en(s) = \emptyset$, D2 holds trivially. Otherwise there exists $t \in en(s) \cap \mathcal{T}_s$, so that $\mathcal{DNA}_t \subseteq \mathcal{T}_s$. Therefore, for every $t_1, t_2, \ldots, t_n \notin \mathcal{T}_s$, $s \xrightarrow{t_1,\ldots,t_n t}$ must hold, by the same reasoning as for D1, and therefore D2 holds. □

If we take $\mathcal{DNB} = \mathcal{T}^2$, the second condition of Lemma 1 makes the third one redundant and directly gives strong stubborn sets. It should be noted that Lemma 1 does not fully characterize stubborn sets, and neither does it mean that stubborn sets that do not satisfy the lemma are necessarily impractical. The relation $\mathcal{DNB}$ is by its definition stronger than needed, and under certain assumptions can be replaced by weaker relations. For instance in the case of P/T nets, such as in [56][Section 4.2], arbitrary orderings of transitions result in the same marking, as long as these orderings can be executed, making it possible to calculate sets that do not conform to the lemma.

In what follows, we will use the term *dependencies* to refer to both an accordance relation (in connection to an enabled transition), and a necessary enabling set (in connection to disabled transitions).

3.2 Stubborn Set Calculation

We now turn our attention to calculation of stubborn sets, and we give two algorithms to this end.

Algorithm 1 from [14] implements the closure method from [51, Sec. 7.4]. It builds a stubborn set incrementally by making sure that each new transition added to the set fulfils the sufficient stubborn set conditions of Lemma 1. Algorithm 1 only makes use of $\mathcal{DNA}$, so that it builds a strong stubborn set; Line 11 could also choose (nondeterministically) to add the transitions from $\mathcal{DNB}_t$ instead of $\mathcal{DNA}_t$. (Provided that it ends with at least one $\mathcal{DNA}_t$.)

Example 2. Suppose Figure 2 is a partial run of Algorithm 1 on state s, and transition t_3 does not accord with some transition $t \in \mathcal{T}_s$. The algorithm will proceed with processing t and add all transitions that do-not-accord, including t_3, to the work set. Since t_3 is disabled in state s, we add the necessary enabling set for t_3 to the work set. This could for instance be $\{t_2\}$, which is then added to the work set. Again, the transition is disabled and a necessary enabling set for t_2 is added, for instance, $\{t_1\}$. Since t_1 is enabled in s, and has no other dependent transitions in this example, the algorithm finishes. Note that in this example, t_1 now should be part of the stubborn set.

In Section 4.3, we extend the standard closure algorithm with heuristic selection, based on the guard-based approach presented there.

Algorithm 2 implements the deletion algorithm [56]. It starts with an initial set that is trivially stubborn: $\mathcal{T}_s = \mathcal{T}_n \cup \mathcal{T}_k = \mathcal{T}$. Hence, all enabled transitions are key transitions: $T_k = en(s)$ (see Lines 2–3). Then it recursively deletes *implied transitions* starting from all enabled transitions. Implied transitions are those transitions which no longer satisfy Lemma 1. To delete those transitions, the algorithm does a reverse, or backward, search of the (asymmetric) enabling and accordance relations (a version of the algorithm which makes these relations explicit via an and/or graph was presented in [47]). The postcondition of the *Delete* function adheres to the conditions of Lemma 1:

1. $\mathcal{T}_k \subseteq en(s)$ and non-empty,
2. for $t \in \mathcal{T}_k$, $\mathcal{DNA}_t \subseteq \mathcal{T}_k \cup \mathcal{T}_n$,
3. for $t \in \mathcal{T}_n \cap en(s)$, $\mathcal{DNB}_t \subseteq \mathcal{T}_k \cup \mathcal{T}_n$,
4. and for $t \in \mathcal{T}_s \setminus en(s)$, there is some $\mathcal{N}_t \in find\_nes(t, s)$ such that $\mathcal{N}_t \subseteq \mathcal{T}_k \cup \mathcal{T}_n$.

```
1  function $stubborn_{closure}(s)$
2      $\mathcal{T}_{work} = \{t\}$ for some $t \in en(s)$
3      $\mathcal{T}_s = \emptyset$
4      while $\mathcal{T}_{work} \neq \emptyset$ do
5          $\mathcal{T}_{work} = \mathcal{T}_{work} \setminus \{t\}$, $\mathcal{T}_s = \mathcal{T}_s \cup \{t\}$ for some
           $t \in \mathcal{T}_{work}$
6          if $t \in en(s)$ then
7              $\mathcal{T}_{work} = \mathcal{T}_{work} \cup \mathcal{DNA}_t \setminus \mathcal{T}_s$
8          else
9              $\mathcal{T}_{work} = \mathcal{T}_{work} \cup \mathcal{N} \setminus \mathcal{T}_s$ for some
               $\mathcal{N} \in find\_nes(t, s)$
10     return $\mathcal{T}_s$
```

Algorithm 1: The *closure* algorithm for finding stubborn sets

```
1  function $stubborn_{deletion}(s)$
2      $\mathcal{T}_k := en(s)$
3      $\mathcal{T}_n := \mathcal{T}$
4      forall the $t \in en(s)$ do
5          $(\mathcal{T}_k', \mathcal{T}_n') := Delete(s, t, \mathcal{T}_k, \mathcal{T}_n)$
6          if $\mathcal{T}_k' \neq \emptyset$ then
7              $(\mathcal{T}_k, \mathcal{T}_n) := (\mathcal{T}_k', \mathcal{T}_n')$
8      return $\mathcal{T}_n \cup \mathcal{T}_k$
9  function $Delete(s, t, \mathcal{T}_k, \mathcal{T}_n)$
10     $\mathcal{T}_k := \mathcal{T}_k \setminus \{t\}$
11     $\mathcal{T}_n := \mathcal{T}_n \setminus \{t\}$
12     forall the $t' \in \mathcal{DNA}_t \cap \mathcal{T}_k$ do
13         $\mathcal{T}_k := \mathcal{T}_k \setminus \{t'\}$
14         if $t' \notin \mathcal{T}_n$ then
15             $(\mathcal{T}_k, \mathcal{T}_n) := Delete(s, t', \mathcal{T}_k, \mathcal{T}_n)$
16     forall the $t' \in \mathcal{DNB}_t^{-1} \cap \mathcal{T}_n \cap en(s)$ do
17         $\mathcal{T}_n := \mathcal{T}_n \setminus \{t'\}$
18         if $t' \notin \mathcal{T}_k$ then
19             $(\mathcal{T}_k, \mathcal{T}_n) := Delete(s, t', \mathcal{T}_k, \mathcal{T}_n)$
20     forall the $t' \in \mathcal{T}_n \setminus en(s)$ such that
       $\exists \mathcal{N} \in find\_nes(t', s) : t \in \mathcal{N}$ do
21         if $\forall \mathcal{N}' \in find\_nes(t', s) \colon \mathcal{N}' \not\subseteq (\mathcal{T}_k \cup \mathcal{T}_n)$ then
22             $(\mathcal{T}_k, \mathcal{T}_n) := Delete(s, t', \mathcal{T}_k, \mathcal{T}_n)$
23     return $(\mathcal{T}_k, \mathcal{T}_n)$
```

Algorithm 2: The *deletion* algorithm for finding stubborn sets

This is easily verified by examining the conditions under which *Delete* is called; Once t has been removed, the enabled transitions in $\mathcal{DNA}_t$ are removed from $\mathcal{T}_k$ (note that by its symmetry, we have $\mathcal{DNA} = \mathcal{DNA}^{-1}$). If $t \in \mathcal{DNB}_{t'}$ for some enabled t', and $t' \notin \mathcal{T}_k$, then t' no longer satisfies the conditions of Lemma 1, and must be deleted. If $t \in \mathcal{N}_{t'}$ of some disabled t', and no other necessary enabled set $\mathcal{N}'$ of t' is a subset of the stubborn set, then t' must be deleted as well. This cascade will continue while the conditions 1 and 2 are violated. If deletion causes $\mathcal{T}_k$ to become empty, deletion is cancelled, and the previous stubborn set reverted at Line 7, because condition 3 could not be satisfied, which gives the correctness of the algorithm.

The deletion algorithm has been mostly of theoretical interest, and it has some attractive theoretical properties. In [52], it was proven that when restricted to strong stubborn sets, no proper subset of the stubborn set returned by Algorithm 2 can be (strongly) stubborn, if the relations $\mathcal{DNA}$ and necessary enabling sets are fixed. A similar condition holds for Algorithm 2, which we prove here.

Lemma 2. *The set $\mathcal{T}_s = \mathcal{T}_n \cup \mathcal{T}_k$ maintained by Algorithm 2 is maximal among sets that contain the same enabled transitions and satisfy Lemma 1.*

Proof. The invariant holds in the beginning. *Delete* removes only transitions that directly violate the conditions in Lemma 1. □

Theorem 1. *Let $\mathcal{T}_s$ be returned by Algorithm 2. There is no $\mathcal{T}_s' \cap en(s) \subset \mathcal{T}_s \cap en(s)$, such that $\mathcal{T}_s'$ satisfies the conditions of Lemma 1.*

Proof. Assume that $\mathcal{T}_s' \cap en(s) \subset \mathcal{T}_s \cap en(s)$. The algorithm iterates over the enabled transitions on Line 4. Let $t \in en(s) \cap \mathcal{T}_s$ and $t \notin \mathcal{T}_s' \cap en(s)$, and assume that it is the first such transition which the iteration on Line 4 passes to *Delete*. However, just before t is passed, the set $\mathcal{T}_n \cup \mathcal{T}_k$ maintained by the algorithm must be a superset of $\mathcal{T}_s'$, by Lemma 2. Removal of t is not possible without violating condition 3 of Lemma 1, as otherwise t would not be in $\mathcal{T}_s$. Therefore $\mathcal{T}_s'$ cannot satisfy Lemma 1. □

The previous theorem is our main motivation of including Algorithm 2 as a point of comparison, as we want to show that the guard-based heuristic approach is a good compromise between fast but inaccurate, and powerful but slow reduction.

4 Computing Necessary Enabling Sets for Guards

The current section investigates how necessary enabling sets can be computed purely based on guards, without reference to program counters. We proceed by introducing necessary enabling on guards, we then show how this relation can be improved by using disabling sets, and also introduce a heuristic for efficient stubborn set calculation. Finally, it is shown how the PINS interface can be extended to support guard-based partial-order reduction by exporting guards, test sets, and the relations from the previous section. By making some relations optional, and overestimating them using e.g. the test sets, the burden of implementation in the language frontends remains proportional to the increase in reduction power.

4.1 Guard-based Necessary Enabling Sets

We refer to all guards in the state space $M_\mathcal{T} = (S_\mathcal{T}, s_0, \Delta)$ as: $\mathcal{G}_\mathcal{T} := \bigcup_{t \in \mathcal{T}} \mathcal{G}_t$.

Definition 10 (necessary enabling set for guards). Let $g \in \mathcal{G}_\mathcal{T}$ be a guard that is disabled in some state $s \in S_\mathcal{T}$, i.e. $\neg g(s)$. A set of transitions $\mathcal{N}_g$ is a *necessary enabling set* for g in s, if for all states s' with some sequence $s \xrightarrow{t_1,\ldots,t_n} s'$ and $g(s')$, for at least one transition t_i $(1 \leq i \leq n)$ we have $t_i \in \mathcal{N}_g$.

Given $\mathcal{N}_g$, a concrete necessary enabling set on transitions in the sense of Definition 9 can be retrieved as follows (notice the non-determinism):

$$find\_nes(t, s) \in \{\mathcal{N}_g \mid g \in \mathcal{G}_t \wedge \neg g(s)\}$$

Proof. Let t be a transition that is disabled in state $s \in S_\mathcal{T}$, $t \notin en(s)$. Let there be a path where t becomes enabled, $s \xrightarrow{t_1,\ldots,t_n} s' \xrightarrow{t}$, On this path, all of t's disabled guards, $g \in \mathcal{G}_t \wedge \neg g(s)$, need to be enabled, for t to become enabled (recall that $\mathcal{G}_t$ is a conjunction). Therefore, any $\mathcal{N}_g$ is a $\mathcal{N}_t$. □

Example 3. Let ch be the variable for a *rendez-vous channel* in a PROMELA model. A channel read can be modeled as a PROMELA statement ch? in some process $P1$. A channel write can be modeled as a PROMELA statement ch! in some process $P2$. As the statements synchronise, they can be implemented as a single transition, guarded by process counters corresponding to the location of the statements in their processes, e.g.: $P1.pc = 1$ and $P2.pc = 10$. The set of all transitions that assign $P1.pc := 1$, is a valid necessary enabling set for this transition. So is the set of all transitions that assign $P2.pc := 10$.

Instead of computing the necessary enabling set on-the-fly, we statically assign each guard a necessary enabling set by default. Only transitions that write to state vector variables used by this guard need to be considered (as in [37]):

$$\mathcal{N}_g^{\min} := \{t \in \mathcal{T} \mid Ts(g) \cap Ws(t) \neq \emptyset\}$$

4.2 Necessary Disabling Sets

Consider the computation of a stubborn set $\mathcal{T}_s$ in state s along the lines of Algorithm 1. If a disabled t gets in the stubborn set, a necessary enabling set is required. This typically contains a predecessor of t in the control flow. When that one is not yet enabled in s, its predecessor is added as well, until we find a transition enabled in s. So a whole path of transitions between s and t ends up in the stubborn set.

Example 4. Assume a system with several parallel processes, two of which are P_1 and P_2, shown in Figure 3 with $\mathcal{DNA}(t_1, t_7)$ and $\mathcal{DNA}(t_6, t_7)$. We use Algorithm 1 to construct the set, starting from $t = t_1$. We have $\{t_1, t_7\} \subseteq en(s_0)$, and both end up in the stubborn set, since they do-not-accord and may be co-enabled. Then t_7 in turn adds t_6, which is disabled. Now working backwards, the enabling set for t_6 is t_5, for t_5 it is t_4, etc, eventually resulting in the stubborn set $\{t_1, \ldots, t_7\}$. If one of those transitions, say t_3 has, not only t_2 but also some t^* (not shown) in the same necessary enabling set, then also t^* gets added to the set; the reduction is made worse if t^* is enabled.

How can this unnecessary growth of stubborn set be avoided? The crucial insight is that to enable a disabled transition t, it is necessary to disable any enabled transition t' which cannot be co-enabled with t. Quite likely, t' could be a successor of the starting point s, leading to a smaller stubborn set.

Example 5. Consider again the situation after adding $\{t_1, t_7, t_6\}$ to $\mathcal{T}_s$, in the previous example. Note that t_1 and t_6 cannot be co-enabled, and t_1 is enabled in s_0. So it must be disabled in order to enable t_6. Note that t_1 is disabled by itself. Hence t_1 is a necessary enabling set of t_6, and the algorithm can directly terminate with

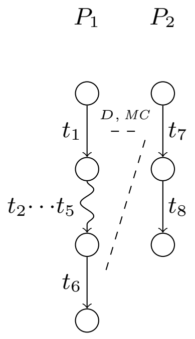

Figure 3. Two process example

the stubborn set $\{t_1, t_7, t_6\}$, avoiding adding t^* into the stubborn set. Clearly, using disabling information saves time and can lead to better reduction.

Definition 11 (may be co-enabled for guards). The *may be co-enabled* relation for guards, $MC_g \subseteq \mathcal{G_T} \times \mathcal{G_T}$ is a symmetric, reflexive relation. Two guards $g, g' \in \mathcal{G_T}$ may be co-enabled if there exists a state $s \in S_\mathcal{T}$ where they both evaluate to **true**: $\exists s \in S_\mathcal{T} : g(s) \wedge g'(s) \Rightarrow (g, g') \in MC_g$.

Example 6. Two guards that can never be co-enabled are: $g_1 := v = 0$ and $g_2 := v \geq 5$. In e.g. PROMELA, these guards could implement the channel empty and full expressions, where the variable v holds the number of buffered messages. In e.g. mCRL2, the conditions of a *summand* can be implemented as guards.

Note that it is allowed to over-approximate the maybe co-enabled relation. Typically, transitions within a sequential system component can never be enabled at the same time. They never interfere with each other, even though their test and write sets share at least the program counter.

Definition 12 (necessary disabling set for guards). Let $g \in \mathcal{G_T}$ be a guard that is enabled in some state $s \in S_\mathcal{T}$, i.e. $g(s)$. A set of transitions $\overline{\mathcal{N}}_g$ is a *necessary disabling set* for g in s, if for all states s' with some sequence $s \xrightarrow{t_1,\ldots,t_n} s'$ and $\neg g(s')$, for at least one transition t_i $(1 \leq i \leq n)$ we have $t_i \in \overline{\mathcal{N}}_g$.

The following disabling set can be assigned to each guard. Similar to enabling sets, only transitions that change the state indices used by g are considered.

$$\overline{\mathcal{N}}_g^{\min} := \{t \in \mathcal{T} \mid Ts(g) \cap Ws(t) \neq \emptyset\}$$

Using disabling sets, we can find an enabling set for the current state s:

Theorem 2. *If $\overline{\mathcal{N}}_g$ is a necessary disabling set for guard g in state s with $g(s)$, and if g' is a guard that may not be co-enabled with g, i.e. $(g, g') \notin MC_g$, then $\overline{\mathcal{N}}_g$ is also a* necessary enabling set *for guard g' in state s.*

Proof. Guard g' is disabled in state s, since $g(s)$ holds and g' cannot be co-enabled with g. In any state reachable from s, g' cannot be enabled as long as g holds. Thus, to make g' true, some transition from the disabling set of g must be applied. Hence, a disabling set for g is an enabling set for g'. □

Given $\mathcal{N}_g$ and $\overline{\mathcal{N}}_g$, we can find a necessary enabling set for a particular transition $t = (\mathcal{G}, a) \in \mathcal{T}$ in state s, by selecting one of its disabled guards. Subsequently, we can choose between its necessary enabling set, or the necessary disabling set of any guard that cannot be co-enabled with it. This spans the search space of our new *find_nes* algorithm, which is called by Algorithm 1:

$$\mathit{find\_nes}(t,s) \in \{\mathcal{N}_g \mid g \in \mathcal{G}_t \wedge \neg g(s)\} \cup \bigcup_{g' \in \mathcal{G}_\mathcal{T}} \{\overline{\mathcal{N}}_{g'} \mid g'(s) \wedge g' \notin MC_g \wedge g \in \mathcal{G}_t\} \quad (1)$$

4.3 Heuristic Selection for Stubborn Sets

Even though the stubborn set conditions of Lemma 1 are stronger than the dynamic stubborn set, it still allows many different sets to be computed, as both the choice of an initial transition t at Line 2 and the *find_nes* function in Algorithm 1 are non-deterministic. It is well known that the resulting reductions depend strongly on a smart choice of the necessary enabling set [52]. A known approach to resolve this problem is to run an SCC algorithm on the complete search space for each enabled transition t [51]. The complexity of this solution can be somewhat reduced by choosing a 'scapegoat' for t [56]. In Algorithm 2, the choice of order in which enabled transitions are taken out from the set is still nondeterministic, but it completely avoids the nondeterminism of *find_nes*. However, Algorithm 2 is potentially too expensive to be of practical use unless the added reduction is clearly superior.

We propose here a practical solution that avoids the complexities of both the scapegoat approach, and the deletion algorithm. Using a heuristic, we explore all possible scapegoats, while limiting the search by guiding it towards a local optimum. (This makes the algorithm deterministic, which has other benefits, cf. Section 8). Even though choosing stubborn sets as small as possible (in terms of number of transitions) is not a perfect solution, it is an often-effective heuristics for large partial-order reductions [14,33]. To this end, we define a heuristic function h that associates some cost to adding a new transition to the stubborn set. Here enabled transitions weigh more than disabled transitions. Transitions that do not lead to additional work (already selected or going to be processed) do not contribute to the cost function at all. Below, $\mathcal{T}_s$ and $\mathcal{T}_{work}$ refer to Algorithm 1.

$$h(\mathcal{N}, s) = \sum_{t \in \mathcal{N}} \mathit{cost}(t,s), \text{ where}$$

$$\mathit{cost}(t,s) = \begin{cases} 1 & \text{if } t \notin en(s) \wedge t \notin \mathcal{T}_s \cup \mathcal{T}_{work} \\ n & \text{if } t \in en(s) \wedge t \notin \mathcal{T}_s \cup \mathcal{T}_{work} \\ 0 & \text{otherwise} \end{cases}$$

Here n is the maximum number of outgoing transitions (degree) in any state, $n = \max_{s \in S}(|en(s)|)$, but it can be over-approximated (for instance by $|\mathcal{T}|$).
We restrict the search to the cheapest necessary enabling sets:

$$\mathit{find\_nes}'(t,s) \in \{\mathcal{N} \in \mathit{find\_nes}(t,s) \mid \forall \mathcal{N}' \in \mathit{find\_nes}(t,s) \colon h(\mathcal{N}, s) \leq h(\mathcal{N}', s)\}$$

4.4 A Pins Extension to Support Guard-based POR

In model checking, the state space graph of Definition 4 is constructed only implicitly by iteratively computing successor states. A generic next-state interface hides the details of the specification language, but exposes some internal structure to enable efficient state space storage or state space reduction.

The Partitioned Interface for the Next-State function, or Pins [3], provides such a mechanism. The interface assumes that the set of states S consists of vectors of fixed length N, and transitions are partitioned disjunctively in M partition groups T. Pins also supports K state predicates L for model checking. In order to exploit locality in symbolic reachability, state space storage, and incremental algorithms, Pins exposes a dependency matrix DM, relating transition groups to indices of the state vector. This yields orders of magnitude improvement in speed and compression [3,2]. The following functions of Pins are implemented by the language front-end and used by the exploration algorithms:
– InitState: S
– NextStates: $S \to 2^{T \times S}$ and
– StateLabel: $S \times L \to \mathbb{B}$
– DM: $\mathbb{B}_{M \times N}$

Extensions to Pins. POR works as a state space transformer, and therefore can be implemented as a Pins2Pins wrapper (cf. Figure 1), both using and providing the interface. This *POR layer* provides a new NextStates(s) function, which returns a subset of enabled transitions, namely: $\mathit{stubborn}(s) \cap en(s)$. It forwards the other Pins functions. To support the analysis for guard-based partial-order reduction in the POR layer, we introduced four essential extensions to Pins:
– StateLabel additionally exports guards: $\mathcal{G}_\mathcal{T} \subseteq L$,
– a $K \times N$ label dependency matrix is added for Ts,
– DM is split into a read and a write matrix representing Rs and Ws,
– an $M \times M$ do-not-accord matrix is added.

Mainly, the language front-end must do some static analysis to estimate the do-not-accord relation on transitions based on the criteria listed below Definition 8 While Criterion 1 allows the POR layer to estimate the relation without help from the front-end (using Rs and Ws), this will probably lead to poor reductions.

Tailored Necessary Enabling/Disabling Sets. To support necessary disabling sets, we also extend the PINS interface with an optional maybe co-enabled matrix. Without this matrix, the POR layer can rely solely on necessary enabling sets.

Both $\mathcal{N}^{\min}$ and $\overline{\mathcal{N}}^{\min}$ can be derived via the refined PINS interface (using Ts and Ws). In order to obtain the maximal reduction performance, we extend the PINS interface with two more optional matrices:

- a $K \times M$ necessary enabling set $\mathcal{N}_g^{\text{PINS}}$, and
- a $K \times M$ necessary disabling set $\overline{\mathcal{N}}_g^{\text{PINS}}$.

The language front-end can now provide more fine-grained dependencies by inspecting the syntax as in Example 3.

The POR layer actually uses the following intersections:

$$\mathcal{N}_g := \mathcal{N}_g^{\min} \cap \mathcal{N}_g^{\text{PINS}}$$

$$\overline{\mathcal{N}}_g := \overline{\mathcal{N}}_g^{\min} \cap \overline{\mathcal{N}}_g^{\text{PINS}}$$

A simple insight shows that we can compute both $\mathcal{N}_g^{\text{PINS}}$ and $\overline{\mathcal{N}}_g^{\text{PINS}}$ using one algorithm. Namely, for a transition to be *necessarily disabling* for a guard g, means exactly the same as for it to be *necessarily enabling* for the inverse: $\neg g$. Or by example: to disable the guard $pc = 1$, is the same as to enable $pc \neq 1$.

Weak Stubborn Sets. To facilitate the use of weak stubborn sets, the left-accordance relation is required in the POR layer. This matrix can also be derived from other matrices. From the explanation in Section 3.1, the following follows:

$$\mathcal{DNB} \subseteq (\mathcal{DNA} \cup \{(t, t') \mid \exists g \in \mathcal{G}_t \colon t' \in \mathcal{N}_g\}) \setminus \mathcal{M},$$

where $\mathcal{M}$ is the transition *must-disable set*:

$$\mathcal{M} \subseteq \{(t, t') \mid \forall s, s' \in S \colon s \xrightarrow{t} s' \wedge t' \in en(s) \wedge t' \notin en(s')\}$$

So only an additional $\mathcal{M}$ matrix is required for weak sets. We finally also allow an additional (optional) do-not-left-accords matrix to be exported, as it could be that the combined static analysis yields a better estimation:

- an $M \times M$ must disable matrix $\mathcal{M}$, or
- an $M \times M$ do-not-left-accord matrix $\mathcal{DNB}$.

In the following section, we demonstrate how these relations can also be exploited to implement LTL model checking with POR in a language-independent fashion.

5 Partial-Order Reduction for On-The-Fly LTL Checking

A more or less standard specification logic for liveness properties is Linear Temporal Logic (LTL) [41]. An example LTL property is $\Box\Diamond p$, expressing that from any state in an execution ($\Box$ = always), eventually ($\Diamond$) a state s can be reached s.t. $p(s)$ holds, where p is a predicate over a state $s \in S_\mathcal{T}$, similar to our definition of guards in Definition 2.

In the automata-theoretic approach, an LTL property φ is transformed into a Büchi automaton $\mathbb{B}_\varphi$ whose ω-regular language $\mathcal{L}(\mathbb{B}_\varphi)$ represents the set of all infinite traces the system should adhere to. $\mathbb{B}_\varphi$ is an automaton $(S_\mathbb{B}, \Sigma, \mathcal{F})$ with additionally a set of transition labels Σ, made up of the predicates, and accepting states: $\mathcal{F} \subseteq S_\mathbb{B}$. Its language is formed by all infinite paths visiting an accepting state infinitely often. Since $\mathbb{B}_\varphi$ is finite, a lasso-formed trace exists, with an accepting state on the cycle. The system $M_\mathcal{T}$ is likewise interpreted as a set of infinite traces representing its possible executions: $\mathcal{L}(M_\mathcal{T})$. The model checking problem is now reduced to a *language inclusion* problem: $\mathcal{L}(M_\mathcal{T}) \subseteq \mathcal{L}(\mathbb{B}_\varphi)$.

Since the number of cycles in $M_\mathcal{T}$ is exponential in its size, it is more efficient to invert the problem and look for error traces. The error traces are captured by the negation of the property: $\neg\varphi$. The new problem is a *language intersection and emptiness* problem: $\mathcal{L}(M_\mathcal{T}) \cap \mathcal{L}(\mathbb{B}_{\neg\varphi}) = \emptyset$. The intersection can be solved by computing the synchronous cross product $M_\mathcal{T} \otimes \mathbb{B}_{\neg\varphi}$ The states of $S_{M_\mathcal{T} \otimes \mathbb{B}_{\neg\varphi}}$ are formed by tuples (s, s') with $s \in S_{M_\mathcal{T}}$ and $s' \in S_{\neg\varphi}$, with $(s, s') \in \mathcal{F}$ iff $s' \in \mathcal{F}_{\neg\varphi}$. The transitions in $\mathcal{T}_{M_\mathcal{T} \otimes \mathbb{B}_{\neg\varphi}}$ are formed by synchronising the propositions Σ on the states $s \in S_{M_\mathcal{T}}$. For an exact definition of $\mathcal{T}_{M_\mathcal{T} \otimes \mathbb{B}_{\neg\varphi}}$, we refer to [54]. The construction of the cross product can be done *on-the-fly*, without computing (*and storing!*) the full state space $M_\mathcal{T}$. Therefore, the NDFS [6] algorithm is often used to find accepting cycles (= error traces) as it can do so on-the-fly as well. In the absence of accepting cycles, the original property holds.

To combine LTL model checking with POR, so that all behaviours characterized by an LTL formula are preserved, the reduction function needs to fulfil some additional constraints, which we discuss here. For more comprehensive treatment of LTL and stubborn sets, we refer the reader to [51].

First, it should be noted that LTL needs to be slightly restricted, in order to make reduction possible. In the general LTL, the *next-state* operator $\bigcirc$ (next-state) is used, for indicating that some subformula holds in the state immediately following the current state. This makes partial-order reduction problematic. Consider the formula $\bigcirc\neg p$, that should hold in the current state. Any two transitions, one of which changes the truth value of p and one which does not, have the possibility of leading to a violation of this formula, therefore, the reduction would have to include both kinds of transitions. But

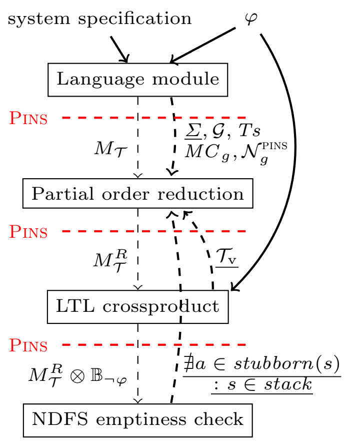
system specification
φ
Language module
Pins
$M_\mathcal{T}$
$\underline{\Sigma}, \mathcal{G}, Ts$
$MC_g, \mathcal{N}_g^{\text{PINS}}$
Partial order reduction
Pins
$M_\mathcal{T}^R$
$\underline{\mathcal{T}_\text{v}}$
LTL crossproduct
Pins
$M_\mathcal{T}^R \otimes \mathbb{B}_{\neg\varphi}$
$\underline{\nexists a \in stubborn(s)}$
$\underline{: s \in stack}$
NDFS emptiness check

Figure 4. Pins w. LTL POR

there are no other kinds of transitions in the system. Therefore, we have to assume that $\bigcirc$ is not used as an operator. Repetition of the same propositional values in an execution is referred to as *stuttering*, and LTL without the next-state operator can express only the properties that are *stuttering insensitive*, i.e., invariant under finite stuttering.

Second, even without $\bigcirc$, the propositional statements in an LTL formula cannot in general appear in arbitrary order. For example, formulas such as $\Box(p \Rightarrow (\neg q \wedge \Diamond q))$ are sensitive to whether p or q hold in a given state. Even if two transitions would otherwise accord, the omitted states may be important, if the values of p and q change in between. Consider two transitions t_1 and t_2, that accord strongly. Assume that $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s'$ and $s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s'$. p holds only at s, and q holds only at s_2 . On the path ss_1s', the formula is not satisfied, but on the path ss_2s' it does hold. Transitions that change the truth-value of some proposition that appears in φ, are called *visible transitions*, and denoted $\mathcal{T}_\text{v}$. Transitions that are not visible are *invisible*. If the order between visible transitions is not preserved, there is risk that some paths are missed. A *visibility proviso* is needed to ensure that the traces included in $\mathbb{B}_{\neg\varphi}$ are not pruned from $M_\mathcal{T}$ when reducing.

Third, cyclical executions of transitions correspond to infinite executions, and all (relevant) such executions need to be preserved. For instance, consider the property $\Diamond p$, and a system with two states: In the first, p does not hold and in the second, it holds. t_1 is a self-loop in both states, and t_2 leads from the first to the second. t_1 and t_2 accord strongly, but omitting t_2 misses the execution where $\Diamond p$ is satisfied. This is known as the *ignoring problem*, where some essential transition is postponed infinitely. An *ignoring proviso* is needed to ensure that important transitions are included in the reduced state space.

Fourth, even if LTL properties are assumed to be insensitive to finite stuttering, *infinite stuttering* must still

Table 1. POR provisos for the LTL model checking of $M_\mathcal{T}$ with a property φ

visibility	**V**	$\mathcal{T}_s \cap en(s) \cap \mathcal{T}_\text{v} = \emptyset$, or $\mathcal{T}_\text{v} \subseteq \mathcal{T}_s$
invisibility	**I**	$en(s) \setminus \mathcal{T}_\text{v} \neq \emptyset \Rightarrow \mathcal{T}_s^k \cap (en(s) \setminus \mathcal{T}_\text{v}) \neq \emptyset$
visibility+ invisibility	**C2**	$\mathcal{T}_s \cap en(s) \cap \mathcal{T}_\text{v} = \emptyset$, or $\mathcal{T}_s = en(s)$
ignoring	**C3**	$\nexists t \in \mathcal{T}_s$: closing a cycle, or $\mathcal{T}_s = en(s)$
ignoring	**C3'**	$\nexists t \in \mathcal{T}_s$: closing a cycle, or $\mathcal{T}_\text{v} \subseteq en(s)$

be preserved. We can consider the same two-state system as before, but with the property $\Box\neg p$. If we ignore t_1 - an invisible, and thus seemingly non-essential, transition in the first state, no execution in the reduced system satisfies $\Box\neg p$. For this we need an *invisibility proviso*.

Classically, many partial-order reduction methods combine visibility and invisibility provisos, but strictly speaking this is not necessary. Table 1 lists some of the conditions found in the literature that ensure LTL properties are preserved. With stubborn sets, we can use **C3** to resolve ignoring, and the combination of **I** and **V** for visibility. The condition **C2** is a stronger alternative to using the combination of **I** and **V**. Please note that the set $\mathcal{T}_s^k$ mentioned in proviso **I** refers to the set of key transitions in $\mathcal{T}_s$.

We state two ignoring provisos, **C3** and **C3'**, both use the 'closing of a cycle' as premise. This proposition is purposefully a bit vague, as it is up to the state space exploration algorithm to identify at least one transition per infinite path in the reduced states space. The simplest way to do this, is by running a DFS algorithm and mark all transitions that end in a state on the current search stack as 'cycle closing'. To find the minimum number of transitions satisfying the required condition, is obviously a hard problem as there can be exponentially many cycles in the reduced states space. However, alternative algorithms exist that can find good estimates for practical problems [11].

Stubborn sets can use the weaker **C3'** proviso, potentially yielding better reductions. All these different provisos, including slight variants not mentioned here, have been extensively discussed in [49,53,51], but have never been evaluated in practice on real-world variable/transition systems. Section 7 provides an evaluation of the performance of these different provisos.

These conditions can easily be integrated in Algorithm 1. The integration requires $\mathcal{T}_\text{v}$ and information whether the target state of a given transition is in the DFS stack. The reduced state space $M_\mathcal{T}^R$ is constructed on-the-fly, while the LTL cross product and emptiness check algorithm run on top of the reduced state space [40]. Figure 4 shows the Pins stack with POR and LTL as Pins2Pins wrappers.

We extend the NextStates function of Pins with a boolean, that can be set by the caller to pass the information needed for **C3**. For **C2**, or **V** and **I** , we extend PINS with $\mathcal{T}_\mathrm{v}$, to be set by the LTL wrapper based on the predicates Σ in φ:

$$\mathcal{T}_\mathrm{v}^{\mathrm{min}} := \{t \in \mathcal{T} \mid \mathit{Ws}(t) \cap \bigcup_{p \in \Sigma} \mathit{Ts}(p) \neq \emptyset\}$$

However, this is a coarse over-approximation, which we can improve by inputting φ to the language module, so it can export Σ as state labels, i.e. $\Sigma \subseteq \mathcal{G}$, and thereby obtain $\mathcal{N}/\overline{\mathcal{N}}$ for it:

$$\mathcal{T}_\mathrm{v}^{\mathrm{nes}} := \bigcup_{p \in \Sigma} \mathcal{N}_p \cup \overline{\mathcal{N}}_p$$

To summarise, we can combine guard-based partial-order reduction with on-the-fly LTL model checking with limited extensions to PINS: a modified NEXTSTATES function and a visibility matrix $\mathcal{T}_\mathrm{v} \colon \mathcal{T} \to \mathbb{B}$. For better reduction, the language module needs only to extend the exported state labels from $\mathcal{G}$ to $\mathcal{G} \cup \Sigma$ and calculate the MC (and $\mathcal{N}^{\mathrm{PINS}}/\ \overline{\mathcal{N}}^{\mathrm{PINS}}$) for these labels as well.

6 Implementation

The closure algorithm has been implemented using a form of Beam search [36] to facilitate the heuristic search. Traditional Beam search performs BFS with an fixed-sized, ordered work queue to prioritize successors and discard paths that are less promising according to the heuristic.For the closure algorithm (see Algorithm 3), instead, we use one search context for each enabled state c (Lines 2–4). The search contexts represent independent closure searches, each with their own work set ($\mathcal{T}_{\mathit{work}}^c$) and visited set ($\mathcal{T}_s^c$).

The search contexts are scheduled sequentially always running the one which found the fewest enabled transitions yet (see Line 6). The (Beam) search continuous until the context with the minimum number of enabled transitions has finished (the closure algorithm), then the stubborn set is returned (Line 8). All contexts search different transition dependency spaces, because the heuristic selection criterion dependents on its current visited set (see Line 13, which uses the cost function from Section 4.3 implicitly passing $\mathcal{T}_s = \mathcal{T}_s^c$ and $\mathcal{T}_{\mathit{work}} = \mathcal{T}_{\mathit{work}}^c$).

Note that all nondeterminism is resolved: the nondeterminism of selecting an initial state, by starting Beam searches from all initial transitions, and the nondeterminism of choosing a necessary enabling set for disabled transitions via the heuristic selection criterion.

The advantages of this approach is that the search tries to find local optima starting from all enabled transitions and may terminate early when a best result is found. The worst-case complexity is $c^2|T|^2$, where c is a reasonably small constant bounded by the size of the transition's test set, which is limited due to the locality of transitions in a parallel system (c^2 represents the dependencies that have to be considered per transition, this is explained in detail in [49]). The factor $|T|$ arises because a search may consider all transitions, and the other $|T|$ is caused by the different search contexts. Because in practice the number of enabled transitions is much lower than $|T|$, we can expect the complexity to lie closer to to the SCC algorithm presented in [49].

The disadvantage of our approach is that the heuristic has to be maintained independently in each search context. We partly solved this problem by incrementally updating the initial cost values, which is possible due to the relatively small set of enabled transitions at each state. Still the costs have to be copied to – and maintained at – each search context which becomes active. We suspect it is further possible to update costs more lazily, however we did not implement this.

The deletion algorithm was implemented without the explicit recursion shown in Algorithm 2. Also several optimizations were added to crucially improve the algorithm's performance:

1. If a (backward) search from a transition t fails, and the stubborn set has to be reverted at Line 7, t is marked as a fail transition.
2. If a fail transition is encountered, the search is terminated early at the beginning of the *Delete* function.
3. The same is done when the set of key transitions T_k becomes empty at Line 10 or 13.
4. The fulfilment of condition 4 in Section 3.2 is complicated, as necessary enabling set $\mathcal{N}$ has to be found from which the deleted t is a part of (see Line 20), and subsequently that no disabled transitions t' depending on $\mathcal{N}$ end up without any *stubborn* necessary enabling set (see Line 21). That is both a backward and a forward search. The additional forward search can however be eliminated by counting. Each $\mathcal{N}_g$, i.e. each guard g, gets a counter of how many transitions are removed from it. Deleted transitions in $\mathcal{N}_g$ increment the counter. The first transition that increments

```
1  function $\mathit{stubborn}_{\mathit{heur}}(s)$
2      forall the $c \in \mathit{en}(s)$ do
3          $\mathcal{T}_{\mathit{work}}^c := \{c\}$
4          $\mathcal{T}_s^c := \emptyset$
5      while $\mathit{true}$ do
6          $\mathcal{T}_s := \mathcal{T}_s^c$, $\mathcal{T}_{\mathit{work}} := \mathcal{T}_{\mathit{work}}^c$ for some $c \in \mathit{en}(s)$
           such that $|(\mathcal{T}_{\mathit{work}}^c \cup \mathcal{T}_s^c) \cap \mathit{en}(s)|$ is minimal
7          if $\mathcal{T}_{\mathit{work}} = \emptyset$ then
8              return $\mathcal{T}_s$
9          $\mathcal{T}_{\mathit{work}} := \mathcal{T}_{\mathit{work}} \setminus \{t\}$, $\mathcal{T}_s := \mathcal{T}_s \cup \{t\}$ for some
           $t \in \mathcal{T}_{\mathit{work}}$
10         if $t \in \mathit{en}(s)$ then
11             $\mathcal{T}_{\mathit{work}} := \mathcal{T}_{\mathit{work}} \cup \mathcal{DNA}_t \setminus \mathcal{T}_s$
12         else
13             $\mathcal{T}_{\mathit{work}} := \mathcal{T}_{\mathit{work}} \cup \mathcal{N} \setminus \mathcal{T}_s$ for some
               $\mathcal{N} \in \mathit{find\_nes}(t, s)$
```

Algorithm 3: The Beam search algorithm for finding stubborn sets with the heuristic function

it, i.e. makes it incomplete, has to decrement a second counter on those transitions t' that rely on it. This counter is initialized to the number of necessary enabling sets that t' has. If it reaches zero (and the transition is disabled), t' has to be deleted as well.

These reflect all optimizations known to us.

The deletion algorithm also has a complexity of $c^2|T|^2$ according to [49]. Instead of finding only local optima, it guarantees a subset minimal result (see Section 3.2). We suspect therefore that it is less likely to terminate early, unless a stubborn set of size one is found early on.

Note that subset minimal sets are not necessarily the smallest stubborn set possible at a state, even though the converse holds. There could exist a partly overlapping set that is smaller and still satisfies the stubborn set constraints as set forth in Section 3. Therefore, the deletion algorithm can be further extended to yield almost smallest stubborn sets by combining it with the *incomplete minimization* algortihm [55]. However, this increase the complexity with another factor $|T|$, so we do not implement it here.

7 Experimental Evaluation

7.1 Experimental Setup

The LTSMIN toolset implements Algorithm 1 as a language-independent PINS layer since version 1.6. We implemented the deletion algorithm as well to evaluate the performance of the heuristic better.

We experimented with BEEM and PROMELA models. To this end, first the DIVINE front-end of LTSMIN was extended with the new PINS features in order to export the necessary static information. In particular, it supports guards, R/W-dependency matrices, the do-not-accord matrix, the co-enabled matrix, and disabling- and enabling sets. Later the PROMELA front-end SpinS [2] was extended, with relatively little effort.

We performed experiments and indicate performance measurements with LTSMIN 2.0[1] and SPIN version 6.2.1[2]. All experiments ran on a dual Intel E5335 CPU with 24GB RAM memory, restricted to use only one processor, 8GB of memory and 3 hours of runtime. None of the models exceeded these bounds.

The first goal of this evaluation is to obtain a better understanding of the performance of the heuristic selection criterion. In the prequel work [29], we showed that guard-based partial-order reduction could compete with state of the art model checkers such as SPIN, despite its language independence. The obtained reductions were shown to be consistently better than those of the ample-set approach in SPIN. The runtimes of SPIN were however several times faster. A symptom which we attributed to the more elaborate heuristic selection algorithm and the many guards and transitions which our models included.

Previously, we did however not succeed completely to isolate the performance of the necessary disabling sets, nor did we have a reference point for the evaluation of the heuristic selection criterion. To tackle the latter, the following section first details the implementation and complexity of both stubborn-set calculation algorithms. It turns out that both algorithms have the same worst case complexity, but the heuristic closure algorithm has a better potential to scale better. This is then verified with experiments in Section 7.2.

Since our implementation has been improved somewhat, we also include a new comparison of the guard-based stubborn method with the ample-set method, both theoretically and experimentally. For the theoretical comparison the same BEEM models were used as in [12] to establish the best possible reduction with ample sets. For the experimental comparison, we used a rich set of PROMELA models[3], which were also run in SPIN with partial-order reduction. While POR is only useful for models which cannot be explored fully, we focus here on smaller models in order to be able to study the obtained reductions. In [2], the authors reported how the same method was used to fully explore the GARP model [24], which previously was only analyzed with incomplete verification methods [24].

In the following, when we discuss *necessary enabling sets*, we mean the enabling sets extended with the necessary disabling sets as explained in Section 4.2, unless stated otherwise. In Section 7.3, we investigate the impact of the necessary disabling set method.

7.2 Algorithms for Stubborn-Set Calculation

First, we compare the two algorithms for stubborn set calculation, to establish the effectiveness of the heuristic selection (Section 4.3) criterion as implemented with the Beam search (Section 6). We focus on a subset of PROMELA models that show interesting reductions, i.e. excluding those without reduction and toy examples with obscene reductions.

To investigate the reductions, we currently only look at the reduction in the number of states. For performance, we choose the metric of *number of states per second*. This facilitates some insights in the performance of the algorithms despite their difference in reductions. Furthermore, the number of *new* states is the major work unit here, as it corresponds to the number of calls of the stubborn-set algorithm. Other metrics of the same models, such as absolute runtimes, are shown in the experimental comparison with other tools, in a subsequent sections.

Table 2 shows the obtained results in the second and third columns. The heuristic selection ('Beam heur.')

[1] http://fmt.cs.utwente.nl/tools/ltsmin/

[2] http://spinroot.com

[3] http://www.albertolluch.com/research/promelamodels

Table 2. Reductions and speed of the deletion algorithm and the closure algorithm with heuristic selection, both implementing the strong stubborn set definition.

	no POR		Deletion		Beam heur.		Closure heur.		Beam no heur.		Closure	
	$\|S\|$	$\|\mathcal{T}\|$	$\varDelta\|S\|$	$\|S\|$/sec	$\varDelta\|S\|$	$\|S\|$/sec	$\varDelta\|S\|$	$\|S\|$/sec	$\varDelta\|S\|$	$\|S\|$/sec	$\varDelta\|S\|$	$\|S\|$/sec
brp.prm	3.280.269	7.058.556	34,8%	91.816	28,0%	254.404	28,9%	224.147	36,5%	198.827	40,0%	244.647
garp	48.363.145	247.135.869	3,8%	30.922	3,6%	59.965	49,5%	99.594	8,4%	28.970	72,8%	37.065
iprotocol0	9.798.465	45.932.747	6,1%	23.062	5,7%	70.706	36,8%	100.268	12,5%	26.255	80,2%	27.428
iprotocol2	14.309.427	48.024.048	17,6%	73.194	16,1%	189.562	55,2%	218.248	36,3%	66.796	92,7%	117.433
p117.pml	354	828	41,8%	n/a	46,3%	n/a	41,5%	n/a	93,2%	n/a	100,0%	n/a
peterson4	12.645.068	47.576.805	3,0%	70.698	2,9%	212.681	50,8%	247.946	2,9%	156.367	60,1%	186.736
philo.pml	1.640.881	16.091.905	4,9%	29.171	7,9%	31.683	55,4%	81.263	43,6%	7.668	89,8%	37.726
SMALL1	36.970	163.058	17,8%	n/a	17,8%	n/a	60,4%	n/a	17,8%	n/a	80,5%	n/a
smcs	5.066	19.470	6,6%	n/a	7,6%	n/a	31,6%	n/a	56,0%	3.594	96,8%	7.908
snoopy	81.013	273.781	9,2%	12.034	15,2%	21.964	37,0%	46.881	10,3%	11.599	66,0%	30.400
X.509.prm	9.028	35.999	7,6%	n/a	12,7%	n/a	76,4%	n/a	94,9%	12.604	100,0%	n/a

Table 3. Branching factors of the state spaces generated by the different algorithms with and without weak stubborn sets and necessary disabling sets. The different algorithm variants are encoded as follows: B = Beam search, c = Closure algorithm, +h/-h = with/without heuristic necessary enabling set selection.

	No	Strong					Weak		No NDS	
	POR	deletion	B+h	c+h	B-h	c-h	del	heur	del	heur
brp.prm	2,15	1,15	1,16	1,15	1,28	1,40	1,17	1,16	1,15	1,16
garp	5,11	2,10	2,11	2,79	2,01	3,12	2,14	2,04	2,10	2,12
iprotocol0	4,69	1,84	1,83	2,42	1,96	2,85	1,91	1,82	1,84	1,96
iprotocol2	3,36	1,94	1,99	2,19	1,96	2,48	1,90	1,84	1,94	1,98
p117.pml	2,34	1,22	1,19	1,22	2,12	2,13	1,22	1,19	1,22	1,19
peterson4	3,76	1,23	1,23	1,74	1,23	1,77	1,23	1,23	1,23	1,23
philo.pml	9,81	3,90	4,23	5,94	7,82	8,79	3,90	4,31	3,90	3,90
SMALL1	4,41	2,34	2,34	2,45	2,34	2,53	2,34	2,34	2,34	2,34
smcs	3,84	1,45	1,37	1,94	2,84	3,45	1,45	1,37	1,45	1,38
snoopy	3,38	1,24	1,31	1,43	1,36	2,22	1,24	1,28	1,24	1,28
X.509.prm	3,99	1,59	1,41	2,37	3,61	3,95	1,59	1,40	1,59	1,41

Table 4. Reductions and speed of the algorithms with and without weak stubborn sets and necessary disabling sets.

	Strong stubborn set				No NDS				Weak stubborn set			
	deletion		heuristic		deletion		heuristic		deletion		heuristic	
	$\varDelta\|S\|$	$\|S\|$/sec	$\varDelta\|S\|$	$\|S\|$/sec	$\varDelta\|S\|$	$\|S\|$/sec	$\varDelta\|S\|$	$\|S\|$/sec	$\varDelta\|S\|$	$\|S\|$/sec	$\varDelta\|S\|$	$\|S\|$/sec
brp.prm	34,8%	91.816	28,0%	254.404	34,8%	203.763	28,0%	308.187	35,6%	91.359	28,0%	235.486
garp	3,8%	30.922	3,6%	59.965	3,8%	58.695	3,6%	86.436	2,1%	27.360	2,6%	37.032
iprotocol0	6,1%	23.062	5,7%	70.706	6,1%	67.642	7,3%	82.416	11,3%	23.264	5,9%	28.142
iprotocol2	17,6%	73.194	16,1%	189.562	17,6%	154.766	18,0%	190.713	30,3%	74.959	14,8%	65.773
p117.pml	41,8%	n/a	46,3%	n/a	41,8%	n/a	46,3%	n/a	41,8%	n/a	46,3%	n/a
peterson4	3,0%	70.698	2,9%	212.681	3,0%	125.135	2,9%	256.689	3,0%	71.642	2,9%	191.852
philo.pml	4,9%	29.171	7,9%	31.683	4,9%	40.548	5,1%	39.764	4,9%	25.827	9,7%	23.523
SMALL1	17,8%	n/a	17,8%	n/a	17,8%	n/a	17,8%	n/a	17,8%	n/a	17,8%	n/a
smcs.promela	6,6%	n/a	7,6%	n/a	6,6%	n/a	7,6%	n/a	6,6%	n/a	7,6%	n/a
snoopy	9,2%	12.034	15,2%	21.964	9,2%	n/a	12,8%	n/a	9,2%	12.034	8,4%	11.745
X.509.prm	7,6%	n/a	12,7%	n/a	7,6%	n/a	13,1%	n/a	7,8%	n/a	13,0%	n/a

shows the potential to yield even better reductions than the deletion algorithm, i.e. `brp` and `garp`. However, in most cases, the deletion algorithm shows better results, sometimes significantly better: `snoopy` and `X.509`.

The performance of the Beam heuristic search algorithm is however consistently better than that of the deletion algorithm (where the runtimes were too small to measure accurately, we give 'n/a'). In some cases, the difference is around a factor 3, i.e. `peterson4` and `brp`. This confirms our expectation that the algorithm has more potential for early termination.

To gain better insight into the different aspects of the algorithm that influence its performance, we also disabled the Beam search by using only one search context. This corresponds to running the basic closure algorithm with heuristics for necessary enabling set selection. To compensate for the loss of resolving the nondeterminism at the initial selection of an enabled transition, we use random selection there (we found that without random selection, reductions are worse). Alternately, we also completely disabled the heuristic selection, again resolving the nondeterminism of selecting necessary enabling sets randomly. Essentially this corresponds to the Beam search running the plain closure algorithm. Finally, we also disabled both heuristic selection and Beam search, yielding a plain closure algorithm.

These different variants of the algorithm are shown in the last 3 columns of Table 2: 'Closure heur.', 'Beam no heur.' and 'Closure'. The conclusion from these results can be drawn unambiguously, as reductions are consistently worse than our Beam search with heuristics. Only the closure algorithm with heuristics can yield some acceptable reductions showing that a good selection of the necessary enabling sets is indispensable and that our heuristics do a good job at it.

What is even more surprising is that the runtime performance of the simpler versions of these algorithms does not notably increase. Without heuristic selection, we even witness much slowdown. Depending on how valid we assume the relative performance metric of number of states per second (it may be the case that the performance of the underlying exhaustive exploration algorithm depends on the number of states processed, although we think that in LTSMIN's case this is barely so), we could draw 2 interesting conclusions:

1. The cost of maintaining the heuristic outweighs the costs of going through larger search spaces as a result of bad necessary enabling set selection. In other words, also for (*relative*) performance it is crucial to make good choices for the necessary enabling sets.
2. The search scheduling in the beam search does a good job at avoiding searches from bad 'scape goats', i.e. bad choices of the initial enabled transition. In other words, the algorithm indeed seems to 'terminate early' often.

At least for these problems, we thus find that the performance of the $c^2|T|^2$ Beam search algorithm with heuristics is closer to the (almost linear) $c^2|T|$ closure algorithm, than to the deletion algorithm with the same complexity.

7.3 Necessary Disabling Sets

To investigate the effects of the necessary disabling sets (Section 4.2), we turned off the extension of necessary enabling sets (NESs) with the disabling sets (NDSs) from Section 4.2.

Table 4 shows the results without NDSs in the middle column. The original results of the strong stubborn set with NDSs is again included for easy lookup. If we inspect the columns of the deletion algorithm, we see that it yields the same reductions regardless of the presence of NDSs in the NESs.

Consider that the algorithm delivers a subset minimal result as shown in Theorem 1, starting from the first enabled transition and deterministically processing all others. Because the NDSs only added NES dependencies in to the dependency graph searched by the deletion algorithm, thus weakening the dependencies (the chance should be lower that a 'disappearing' NES causes disables states to be deleted). Theoretically it could thus potentially add smaller subsets that are still valid, i.e. still have a key transition. But here we do not find any (we checked that the state counts are exactly the same), thus indicating some strong relation between NESs and NDSs. At the same this explains why the runtimes of the deletion algorithm increase.

For the heuristics approach, the situation is different even if the NDSs have little chance to allow for more reductions. As Example 5 showed, NDSs allow the algorithm to find smaller stubborn sets quicker. The heuristic *forward* search approach might benefit from this. The experiments show that the reductions can be improved, though inconsistently so, indicating that the heuristic nature of the algorithm is the cause again. Unfortunately, we do not find any faster runtimes. This does not surprise us, as the implementation of the heuristic with NDSs doubles the arrays containing the costs. Therefore, we still believe that the NDSs can aid stubborn set computation.

Similar results can be shown in Table 5 for DVE BEEM models. First, the heuristic selection improves reductions (column Beam+h). For instance, for `cyclic_scheduler.1`. The reduction improves in nearly all cases, and it improves considerably in several cases. Combined with the heuristic selection, NDSs provide some improvement of the reduction though not consistently (column Beam+h+d). In particular, for `leader_election` the reduction doubles again.

7.4 Smaller Stubborn Sets and State Space Size

The difference in state space reductions however may give an obscured impression of the real algorithm's reduction power, as smaller sets are only likely to yield smaller state spaces, but do not guarantee it. In order to obtain a more objective look in the reduction potential of the algorithm, we investigate the *branching factor*, i.e. the average number of outgoing transitions at a state, of the reduced state spaces for all approaches (strong and weak theory, and no necessary disabling sets).

Table 3 shows these results for the deletion algorithm and the Beam heuristic search (Column ‘Strong’). Indeed, we can identify models for which the deletion algorithm finds, on average, smaller stubborn sets, while still yielding a larger state space. Examples are: `brp` and `iprotocol2`. But also the other way around: `p117`, `smcs` and `X.509`, showing indeed that smaller sets are only a heuristics.

Nonetheless, these results do not invalidate the conclusions drawn in the previous subsection, as the differences in branching factors between the different algorithms, correlate highly with the differences in state space reduction of these algorithms.

7.5 Weak Stubborn Sets

Table 4 shows the results obtained using the weak definition of stubborn sets. In some important cases this leads to good improvements in the reductions, e.g.: `garp`. In one case, it leads to the opposite: `iprotocol`. This holds for both algorithms.

The performance remains largely the same for both algorithms, except that the heuristic approach shows some cases with performance decrease. These cases correspond to the same models that have a better and worse reduction, indicating that the algorithm has processed more dependencies, which could indeed be a symptom of both better and worse reductions (for better reductions, new reduction paths have been considered).

7.6 Comparison with Empirical Evaluation of Ample Sets

Table 5 shows the results obtained on those models from the BEEM database [38] that were selected by Geldenhuys, Hansen and Valmari [12]. The table is sorted by the best theoretical process-based ample-set reduction (best first). These numbers (column AMPLE) are taken from [12, column AMPLE2 Df/Rf]. They indicate the experimentally established best possible reduction that can be achieved with the deadlock-preserving process-based ample-set method (without **C2/C3**), while considering conditional dependencies based on full information on the state space.

The amount of reduction is expressed as the percentage of the reduced state space compared to the original state space (100% means no reduction). The next three columns show the reduction achieved by the guard-based stubborn approach, based on necessary enabling sets only (nes), the heuristic selection function (nes+h), and the result of including the necessary disabling sets (nes+h+d).

The results vary a lot. For instance, the best possible ample-set reduction in `cyclic_scheduler.1` is far better than the actual reduction achieved with stubborn sets (nes). However, for `cyclic_scheduler.2` the situation is reversed. Other striking differences are `mcs.1` versus `leader_election`. Since we compare *best case* ample sets (using global information) with *actual* stubborn sets (using only static information), it is quite informative to see that guard-based stubborn sets can provide more reduction than ample sets. It might be possible that the ample-set algorithm with a dependency relation based on the full state space (Df/Rf, [12]) is still coarse, but this is unlikely. Further comparison reveals that many models yield also better reductions than those using dynamic relations (Dd/Rd, [12]), e.g. `protocols.3` with 7% vs 70%. This prompted us to verify our generated stubborn sets, but we found no violations of the stubborn set definition. There are two differences in the implementations of the two methods. Firstly, the guard-based approach constructs the stubborn sets one transition at a time, whereas the ample-set method used in [12] builds them one *process* at a time. Secondly, the ample-set method does not consider multiple necessary enabling sets. Instead, it is equivalent to using one necessary enabling set for each transition that are enabled by a given program counter, but disabled for other reasons. We cannot say for certain, how much of the loss in reduction is caused by each factor, but our other experiments suggest that the coarse necessary enabling sets are a more likely cause.

7.7 Comparison with Ample Sets in SPIN

Additionally, we compared our partial-order reduction results to the ample-set algorithm as implemented in SPIN. Here we can also compare time resource usage. We ran LTSMIN with arguments `--strategy=dfs -s26 --por`, and we compiled SPIN with `-O2 -DNOFAIR -DNOBOUNDCHECK -DSAFETY`, which enables POR by default. We ran the `pan`-verifier with `-m10000000 -c0 -n -w26`. To obtain the same state counts in SPIN, we had to turn off control flow optimizations (`-o1/-o2/-o3`) for some models (see `ltsmin/spins/test/`).

Table 6 shows the results. Overall, we witness consistently better reductions by the guard-based algorithm (using Beam search with heuristics and additional necessary disabling sets). The reductions are significantly larger than the ample-set approach in the cases of `garp`, dining philosophers (`philo.pml`) and `iprotocol`. As a consequence, guard-based POR in LTSMIN reduces memory usage considerably more than ample-based POR in SPIN. (Though we included memory use for completeness’ sake, it only provides an indirect comparison, due to a different state representation and compression in LTSMIN [30]).

On the other hand, the additional computational overhead of our algorithm is clear from the runtimes. In Section 7.2, we identified that this is not a problem of the

Table 5. Comparison of guard-based POR with strong stubborn sets and heuristic (+h) Beam search and NDS (+d) with [12]

	AMPLE	Beam	Beam	Beam
Model		-h-d	+h-d	h+d
cyclic_scheduler.1	1%	58%	1%	1%
mcs.4	4%	16%	16%	16%
firewire_tree.1	6%	8%	8%	8%
phils.3	11%	14%	16%	16%
mcs.1	18%	87%	85%	85%
anderson.4	23%	58%	46%	46%
iprotocol.2	26%	19%	17%	16%
mcs.2	34%	64%	64%	64%
phils.1	48%	60%	48%	48%
firewire_link.2	51%	24%	21%	19%
krebs.1	51%	94%	93%	93%
leader_election.3	54%	13%	12%	6%
telephony.2	60%	95%	95%	95%
leader_election.1	61%	23%	22%	11%
szymanski.1	63%	68%	65%	65%
production_cell.2	63%	26%	24%	24%
at.1	65%	96%	95%	95%
szymanski.2	66%	66%	64%	64%
leader_filters.2	66%	57%	53%	53%
lamport.1	66%	95%	95%	95%
protocols.2	68%	18%	13%	13%
collision.1	68%	88%	59%	56%
driving_phils.1	69%	99%	68%	78%
protocols.3	71%	13%	7%	7%
peterson.2	72%	82%	82%	82%
driving_phils.2	72%	99%	45%	45%
collision.2	74%	75%	40%	39%
production_cell.1	74%	23%	19%	19%
telephony.1	75%	95%	95%	95%
lamport.3	75%	96%	95%	96%
firewire_link.1	79%	42%	37%	33%
pgm_protocol.4	81%	93%	56%	55%
bopdp.2	85%	90%	73%	73%
fischer.1	87%	87%	87%	87%
bakery.3	88%	99%	96%	96%
exit.2	88%	94%	94%	94%
brp2.1	88%	95%	80%	79%
public_subscribe.1	89%	81%	79%	76%
firewire_tree.2	89%	84%	63%	47%
pgm_protocol.2	89%	96%	72%	72%
brp.2	96%	76%	42%	42%
extinction.2	96%	25%	24%	21%
cyclic_scheduler.2	99%	46%	28%	27%
synapse.2	100%	93%	93%	93%

Table 6. Guard-based POR in LTSMIN vs ample-set POR in SPIN (seconds and MB)

	No Partial-Order Reduction				Guard-based POR				Ample-set POR			
			LTSMIN	SPIN	LTSMIN				SPIN			
Model	States $\|S_{\mathcal{T}}\|$	Trans $\|\Delta\|$	time	time	$\|S_{\mathcal{T}}\|$	$\|\Delta\|$	mem	time	$\|S_{\mathcal{T}}\|$	$\|\Delta\|$	mem	time
garp	48.363.145	247.135.869	166,1	267,0	4%	1%	20,3	28,8	18%	9%	932,1	25,2
i-protocol2	14.309.427	48.024.048	27,7	30,3	16%	10%	29,2	12,1	24%	16%	239,9	6,0
peterson4	12.645.068	47.576.805	22,7	17,0	3%	1%	5,7	1,8	5%	2%	37,0	0,5
i-protocol0	9.798.465	45.932.747	29,3	37,7	6%	2%	6,8	7,8	44%	29%	361,7	12,3
brp.prm	3.280.269	7.058.556	6,0	5,6	28%	15%	14,6	3,6	58%	39%	160,9	2,4
philo.pml	1.640.881	16.091.905	9,8	10,2	8%	3%	1,9	4,1	100%	100%	125,4	10,7
sort	659.683	3.454.988	2,8	3,8	182	181	0,0	0,3	182	181	0,3	0,0
i-protocol3	388.929	1.161.274	1,0	0,7	14%	7%	0,9	0,6	26%	16%	6,6	0,1
i-protocol4	95.756	204.405	0,5	0,1	27%	18%	0,4	0,5	38%	28%	2,5	0,0
snoopy	81.013	273.781	0,6	0,2	15%	6%	0,3	0,6	17%	7%	1,2	0,0
peterson3	45.915	128.653	0,4	0,0	8%	3%	0,1	0,4	10%	4%	0,5	0,0
SMALL1	36.970	163.058	0,5	0,0	18%	9%	0,1	0,4	48%	45%	0,9	0,0
SMALL2	7.496	32.276	0,4	0,0	19%	10%	0,0	0,4	48%	44%	0,4	0,0
X.509.prm	9.028	35.999	0,4	0,0	13%	4%	0,0	0,4	68%	34%	1,1	0,0
dbm.prm	5.112	20.476	0,4	0,0	100%	100%	0,1	0,5	100%	100%	0,7	0,0
smcs	5.066	19.470	0,4	0,1	8%	3%	0,0	0,3	25%	11%	0,7	0,0

Table 7. Reductions obtained for LTL model checking problems with deletion algorithm and the heuristic Beam search

	No POR		Strong with heur. Beam				**C2** instead of **V+I**				**C3** instead of **C3'**			
			deletion		heuristic		deletion		heuristic		deletion		heuristic	
	$\|S\|$	$\|\mathcal{T}\|$	$\Delta\|S\|$	$\|\mathcal{T}\|/sec$	$\Delta\|S\|$	$\|\mathcal{T}\|/sec$	$\Delta\|S\|$	$\|\mathcal{T}\|/sec$	$\Delta\|S\|$	$\|\mathcal{T}\|/sec$	$\Delta\|S\|$	$\|\mathcal{T}\|/sec$	$\Delta\|S\|$	$\|\mathcal{T}\|/sec$
garp	7E+7	4E+8	14,4%	23.679	18,3%	48.875	16,3%	66.408	20,1%	199.149	16,7%	26.873	20,0%	69.286
iprotocol2	2E+7	6E+7	34,4%	80.237	32,8%	64.186	32,8%	181.935	33,0%	330.651	37,4%	66.076	32,9%	73.049
pacemaker-con	2E+7	5E+7	41,4%	24.427	46,1%	41.013	45,0%	33.339	63,8%	79.318	70,6%	27.381	63,1%	51.331
pacemaker-dis	7E+7	2E+8	27,9%	28.065	46,1%	83.239	28,7%	34.014	46,7%	121.933	48,4%	30.834	46,7%	93.566
peterson4	8E+7	3E+8	12,6%	46.543	13,5%	81.356	12,6%	69.853	13,9%	138.624	14,8%	48.505	13,9%	91.553

way we compute stubborn sets (by means of heuristics and Beam search). The explanation can therefore be found in the fact that the stubborn-set algorithm considers all transitions whereas the ample-set algorithm only chooses amongst the less numerous process components of the system.However, the runtimes never exceed the runtimes of benchmarks without partial-order reduction by a great margin.

7.8 LTL Model Checking

To evaluate the performance of guard-based POR under LTL model checking, we compare the results here with SPIN, and also use different provisos to assess their benefits. For this purpose, we use the following set of 5 PROMELA models and LTL formulae:

`garp` the GARP protocol[24], with livelock: $\Box\Diamond$progress,

`i-protocol2` the i-protocol [8], with livelock: $\Box\Diamond$progress,

`pacemaker-concurrent` a concurrent model of a Cardiac pacemaker [43] with the invariant property $\Box(p\wedge(q \Rightarrow r))$,

`pacemaker-distributed` the same model with the property $\Box((p \Rightarrow q) \wedge (r \Rightarrow s))$, and

`peterson` the peterson mutual exclusion protocol as included in the SPIN distribution, with the livelock $\Box\Diamond$progress.

Some of these properties are rather simple LTL formulae. In fact, the invariants can be checked with specific safety provisos [51]. However, this does not matter for the analysis here, as the ignoring provisos is anyway triggered by cycles in the state space, and the formulae contain several propositions that the visibility provisos must take into account.

Comparing the deletion algorithm with heuristic Beam heuristic search in the second column of Table 7, we see this time that the deletion algorithm almost always achieves better reductions. This can be attributed to the tight integration of the visibility and ignoring proviso with the algorithm. For example, the visibility proviso can simply be implemented as constraint between visible transitions [51, Sec. 7.4]. While the same is true for the heuristic algorithm, it has no way to backtrack from bad choices, i.e. the inclusion of a visible enabled transition in the set. And it has only limited capabilities of avoiding this via the heuristic that includes extra costs for visible transitions (see Section 5).

We also experimented with the different visibility and ignoring provisos (middle and last column of the table). Choosing the stronger provisos negative impact on the reductions for both algorithms. This shows that indeed often visible transitions do not have to be included in the stubborn set, and that when they are, for example via **C3'**, this does not have to lead automatically to the inclusion of many enabled transitions. But also that the inclusion of the invisible transitions is not so costly in these cases.

While the performance of the Deletion algorithm often remains the same, the heuristic approach clearly benefits from the stronger provisos. Surprisingly also when the reduction is similar, e.g. see `iprotocol2`. This is likely because the additional checks for the weak provisos are expensive.

Table 8. Reductions obtained for LTL model checking problems with SPIN's ample set

	$\|S\|$	$\|\mathcal{T}\|$	$\Delta\|S\|$	$\|\mathcal{T}\|/sec$	$\frac{1}{\Delta}\|S_{Deletion}\|$
garp	4E+7	2E+8	16,3%	585.734	88,3%
iprotocol2	2E+7	6E+7	45,1%	644.272	76,1%
pacemaker-con	2E+7	5E+7	90,0%	983.618	46,0%
pacemaker-dis	7E+7	2E+8	93,9%	820.532	29,7%
peterson4	1E+8	5E+8	9,9%	1.128.331	127,3%

Table 8 shows the same measurements for SPIN's ample-set implementation for LTL [40,44]. Unfortunately, we were unable to consistently obtain exactly the same state counts due to the different way that SPIN processes the LTL properties (with reachability, we do obtain the same state count for all models). For this reason, we also included the reduced state space size of the Deletion algorithm relative to SPIN ($\frac{1}{\Delta}S_{Deletion}$), figuring that the difference in state counts will largely be negated by both reduction techniques. In all cases, the stubborn set method reduces state spaces better, even when the unreduced state space is already smaller in SPIN, e.g. `garp`. Because the ample set works on the level of processes, we again see that the number of generated transitions per second is much higher for SPIN.

7.9 Validation

Implementing POR methods is a detailed, and error-prone task. For validation of the results, we implemented a small checker that verifies the constraints D1 and D2 (and non-emptiness) from Section 3.1 for each explored state. This checker is implemented as a PINS2PINS wrapper itself (see Section 5), which solely considers the structure of the full state space and the stubborn set of transitions that it obtains from the higher POR layer.

In order to check the constraints for a state s, the checker explores all states s' reachable from that state over *non-stubborn* transitions t' using DFS. At the same time, at each location in the DFS stack, all stubborn transitions are tried (stubborn at s that is). Each stubborn transition t is checked for commutativity with the t' along the DFS stack, or not if t' has become disabled as the weak sets allow. This amounts to checking left-accordance according to Definition 8 for all transitions on the stack (non-stubborn), with all stubborn transitions.

We ran this checker on most of the PROMELA and some of the DVE models discussed here. This gives a good confidence that at least the reachability results presented here are correct. So much cannot be said for the LTL results, for which it is much harder to checker whether indeed all infinite paths are preserved.

8 Conclusions

We proposed guard-based partial-order reduction, as a language-agnostic implementation of the stubborn set method. It extends Valmari's stubborn sets for transition systems [50] with an abstract interface (PINS) to language modules. It also makes more use of previous notions of guards [52], by considering them as disabling conditions as well. The main advantage is that a single implementation of POR can serve multiple specification languages front-ends and multiple high-performance model checking back-ends. This requires only that the front-end, or language module, exports guards, guarded transitions, affect sets, and the do-not-accord matrix ($\mathcal{DNA}$). Optional extensions are matrices MC_g, $\mathcal{N}^{\text{PINS}}$, $\overline{\mathcal{N}}^{\text{PINS}}$ (computing the latter merely requires negating the guards), and for weak sets a do-not-left-accord matrix ($\mathcal{DNB}$). These expose more static information to yield better reduction.

We implemented these functions for the DVE and PROMELA language modules in LTSMIN. It should now be a trivial exercise to add partial-order reduction to the mCRL2 [9] and UPPAAL [32] language modules [17,7, 28]. Since the linear process of mCRL2 is rule-based and has no natural notion of processes, our generalization is crucial.

We introduced two improvements to the basic stubborn set method. The first uses necessary disabling sets to identify necessary enabling sets of guards that cannot be co-enabled. This allows for the existence of smaller stubborn sets. Most of the reduction power of the algorithm is harvested by the heuristic selection function, which actively favours small stubborn sets.

To do a cost-benefit analysis of the heuristic selection criterion, we worked out a version of the deletion algorithm for guard-based POR. Experimental comparison with this algorithm provides strong evidence that the heuristic selection is a powerful tool to generate small stubborn sets at relatively low costs. Indeed the heuristic selection algorithm is consistently faster, over a factor 4, than the deletion algorithm, while yielding similar reductions. Experiments further show that a good resolution of the non-determinism in the stubborn-set definition is indispensable. While the deletion algorithm solves this, it carries great costs. The heuristic selection with Beam search implemented here was shown to circumvent much of this problem. We also experimented with the SCC algorithm from [49,51], but the results were meagre, especially since it does not combine well with heuristic selection.

To evaluate the limits of our approach, we also worked out weak sets and benchmarked their benefits. Results show foremost the heuristic the nature of POR itself. While overall an improvement, reductions are not consistently better, and in some isolated cases much worse. Additionally, evaluation of the results is hard, as the nature of the stubborn sets is heuristic: Finding the smallest set is well-known to be as expensive as the exhaustive exploration itself, therefore any reasonable algorithm needs to make use of approximations. Moreover, the smallest stubborn set is still no guarantee for smaller state spaces, as a smaller set might lead to different states in the exhaustive exploration, than some other larger set. Our experimental comparison of branching factors with state space sizes indeed confirms that smaller average sets, might still result in more state, and vice versa. The next step in obtaining better reductions would be to evaluate the effects of different small stubborn sets at each state on the entire state space. This could lead to a heuristic to replace the current smallest-set heuristic.

Further evaluation shows the competitiveness to other tools. Compared to the best possible ample set with conditional dependencies derived from prior knowledge about the state space, the stubborn set yields better reductions in a number of cases. Compared to SPIN's ample set, LTSMIN consistently provides more reduction, but takes more time to do so, probably because of the additional complexity of the stubborn set method that operates on guards and transitions, as opposed to the process components used by the ample-set.

Recently, LTSMIN's parallel algorithms [10,26] fully support LTL model checking with POR, thanks to a new *parallel cycle proviso* implementing the **C3/C3'** proviso [25]. One benefit for the parallel algorithms is that both the heuristic selection and deletion algorithms can find small stubborn sets deterministically, which avoids well-known problems with possible re-explorations [20, 42,27].

Challenges remain, as not all of LTSMIN's algorithmic backends can fulfil the POR layer's requirements. For example, LTSMIN's symbolic algorithms [34] do not yet support state-local implementation of POR in LTSMIN's PINS2PINS wrapper, as they operate on sets of partial states [3]. In general, symbolic algorithms are hard to combine with classic POR methods [21], though some solutions do exists that take different approach [21,1].

Since the first publication of our guard-based partial-order reduction method in [29], some related work has seen the light of day.

The heuristic selection algorithm been used to solve planning problems. Wehrle and Helmert [57] show that the algorithm performs well in this different setting. They also propose orthogonal methods to improve stubborn sets for finding the planning goal. To this extend, they define *envelopes* of relevant transitions leading to goal states, a method that can also be exploited by our algorithm, as goal finding translates to error/invariant detection in the model checking setting.

Siegel [45] and Kokkarinen et al. [23] show elegant ways to further improve the visibility proviso. The latter is orthogonal to our definition of visibility $\mathcal{T}_{\mathrm{v}}^{\mathrm{nes}}$, while the former generalizes and further weakens the visibility proviso (allowing more different stubborn sets and therefore possibly better reductions). Chu and Jaffar [4] take a more rigorous approach and refine the persistent set method specifically for safety properties, resulting in a weaker notion of commutativity analogous to the subsumption-based definition for timed automata proposed in [17].

Acknowledgments. We are grateful to Antti Valmari, Patrice Godefroid and Dragan Bošnački for useful discussions and advice.

References

1. R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-order reduction in symbolic state space exploration. In Orna Grumberg, editor, *CAV*, volume 1254 of *LNCS*, pages 340–351. Springer, 1997.
2. F.I. van der Berg and A.W. Laarman. SpinS: Extending LTSmin with Promela through SpinJa. In *PDMC 2012, London, UK*, ENTCS. Spinger, September 2012.
3. S.C.C. Blom, J.C. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In *CAV*, volume 6174 of *LNCS*, pages 354–359. Springer, 2010.
4. Duc-Hiep Chu and Joxan Jaffar. A framework to synergize partial order reduction with state interpolation. In Eran Yahav, editor, *HVC*, volume 8855 of *LNCS*, pages 171–187. Springer, 2014.
5. E.M. Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer, 2008.
6. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In *CAV*, volume 531 of *LNCS*, pages 233–242. Springer, 1990.
7. Andreas E. Dalsgaard et al. Multi-core Reachability for Timed Automata. In *FORMATS 2012*, volume 7595 of *LNCS*, pages 91–106. Springer, 2012.
8. Yifei Dong, Xiaoqun Du, Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky, Eugene W. Stark, and David S. Warren. Fighting livelock in the i-protocol: A comparative study of verification tools. In W. Rance Cleaveland, editor, *TACAS*, volume 1579 of *LNCS*, pages 74–88. Springer, 1999.
9. J.F. Groote et al. The mCRL2 toolset. In *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques*, WASDeTT, 2008.
10. S. Evangelista, A. Laarman, L. Petrucci, and J. van de Pol. Improved Multi-core Nested Depth-First Search. In *ATVA*, LNCS 7561, pages 269–283. Springer, 2012.
11. S. Evangelista and C. Pajault. Solving the Ignoring Problem for Partial Order Reduction. *STTT*, 12:155–170, 2010.
12. J. Geldenhuys, H. Hansen, and A. Valmari. Exploring the scope for partial order reduction. In *ATVA'09*, LNCS, pages 39–53. Springer, 2009.
13. P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *CAV*, volume 531 of *LNCS*, pages 176–185. Springer, 1990.
14. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer, 1996.
15. P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *CAV*, volume 697 of *LNCS*, pages 438–449. Springer, 1993.
16. P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *FMSD*, 2:149–164, 1993.
17. Henri Hansen, Shang-Wei Lin, Yang Liu, Truong Khanh Nguyen, and Jun Sun. Diamonds are a girl's best friend: Partial order reduction for timed automata with abstractions. In Armin Biere and Roderick Bloem, editors, *CAV*, volume 8559 of *LNCS*, pages 391–406. Springer, 2014.
18. G.J. Holzmann. The model checker SPIN. *IEEE TSE*, 23:279–295, 1997.
19. G.J. Holzmann and D. Peled. An Improvement in Formal Verification. In *IFIP WG6.1 ICFDT VII*, pages 197–211. Chapman & Hall, Ltd., 1995.
20. G.J. Holzmann, D Peled, and M. Yannakakis. On Nested Depth First Search. In *SPIN*, pages 23–32. American Mathematical Society, 1996.
21. V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, LNCS, pages 398–413. Springer, 2009.
22. S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *REX Workshop*, volume 354 of *LNCS*, pages 489–507. Springer, 1988.
23. I. Kokkarinen, D. Peled, and A. Valmari. Relaxed visibility enhances partial order reduction. In Orna Grumberg, editor, *CAV*, volume 1254 of *LNCS*, pages 328–339. Springer, 1997.

24. I. Konnov and O.A. Letichevsky Jr. Model Checking GARP Protocol using Spin and VRS. *International Workshop on Automata, Algorithms, and Information Technologies*, May 2010.
25. A. W. Laarman and Anton J. Wijs. Partial-Order Reduction for Multi-core LTL Model Checking. In Eran Yahav, editor, *HVC 2014*, volume 8855 of *LNCS*, pages 267–283. Springer, 2014.
26. A.W. Laarman. *Scalable Multi-Core Model Checking*. PhD thesis, University of Twente, 2014.
27. A.W. Laarman and Fárago D. Improved On-The-Fly Livelock Detection. In *NFM*, accepted for publication in LNCS. Springer, 2013.
28. A.W. Laarman, M. Chr. Olesen, A.E. Dalsgaard, K.G. Larsen, and J.C. van de Pol. Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 968–983. Springer, 2013.
29. A.W. Laarman, E. Pater, J.C. van de Pol, and M. Weber. Guard-based partial-order reduction. In Ezio Bartocci and C.R. Ramakrishnan, editors, *Model Checking Software*, volume 7976 of *LNCS*, pages 227–245. Springer, 2013.
30. A.W. Laarman, J.C. van de Pol, and M. Weber. Parallel Recursive State Compression for Free. In *SPIN*, LNCS, pages 38–56. Springer, 2011.
31. A.W. Laarman, J.C. van de Pol, and M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In *NFM*, LNCS 6617, pages 506–511. Springer, 2011.
32. K. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *STTT*, 1:134–152, '97.
33. A. Lehmann, N. Lohmann, and K. Wolf. Stubborn sets for simple linear time properties. In *Application and Theory of Petri Nets*, volume 7347 of *LNCS*, pages 228–247. Springer, 2012.
34. J. Meijer, G. Kant, S.C.C. Blom, and J.C. van de Pol. Read, write and copy dependencies for symbolic model checking. In Eran Yahav, editor, *Hardware and Software: Verification and Testing*, volume 8855 of *LNCS*, pages 204–219. Springer, 2014.
35. W.T. Overman. *Verification of concurrent systems: function and timing*. PhD thesis, University of California, Los Angeles, 1981. AAI8121023.
36. Peng Si Ow and Thomas E Morton. Filtered beam search in scheduling? *The International Journal Of Production Research*, 26(1):35–62, 1988.
37. E. Pater. Partial Order Reduction for PINS, Master's thesis, March 2011.
38. R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.
39. D. Peled. All from One, One for All: on Model Checking Using Representatives. In *CAV*, pages 409–423. Springer, 1993.
40. D. Peled. Combining Partial Order Reductions with On-the-Fly Model-Checking. In *CAV*, volume 818 of *LNCS*, pages 377–390. Springer, 1994.
41. Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
42. S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In *TACAS*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.
43. Asankhaya Sharma. End to End Verification and Validation with SPIN. *CoRR*, abs/1302.4796, 2013.
44. S.F. Siegel. Reexamining two results in partial order reduction. Technical report, University of Delaware, 2011.
45. S.F. Siegel. Transparent partial order reduction. *FMSD*, 40(1):1–19, 2012.
46. A. Valmari. Error Detection by Reduced Reachability Graph Generation. In *APN*, pages 95–112, 1988.
47. A. Valmari. Heuristics for Lazy State Generation Speeds up Analysis of Concurrent Systems. In *STeP-88*, volume 2, pages 640–650. Helsinki, 1988.
48. A. Valmari. Eliminating Redundant Interleavings During Concurrent Program Verification. In *PARLE*, volume 366 of *LNCS*, pages 89–103. Springer, 1989.
49. A. Valmari. A Stubborn Attack On State Explosion. In *CAV*, LNCS, pages 156–165. Springer, 1991.
50. A. Valmari. Stubborn Sets for Reduced State Space Generation. In *ICATPN/APN'90*, pages 491–515. Springer, 1991.
51. A. Valmari. The State Explosion Problem. In *LPN*, pages 429–528. Springer, 1998.
52. A. Valmari and H. Hansen. Can Stubborn Sets Be Optimal? In J. Lilius and W. Penczek, editors, *ATPN*, volume 6128 of *LNCS*, pages 43–62. Springer, 2010.
53. Antti Valmari. Stubborn set methods for process algebras. In *DIMACS workshop on Partial order methods in verification*, pages 213–231. AMS Press, Inc., 1997.
54. M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 332–344. IEEE, 1986.
55. K. Varpaaniemi. Finding small stubborn sets automatically. In Volkan Atalay, Uğur Halıcı, Kemal İnan, Neşe Yalabık, and Adnan Yazıcı, editors, *Proceedings of the Eleventh International Symposium on Computer and Information Sciences, ISCIS XI*, pages 133–142. Middle East Technical University, Ankara, Turkey, 1996.
56. K. Varpaaniemi. *On the Stubborn Set Method in Reduced State Space Generation*. PhD thesis, Helsinki University of Technology, 1998.
57. Martin Wehrle and Malte Helmert. Efficient stubborn sets: Generalized algorithms and selection strategies. In *International Conference on Automated Planning and Scheduling*. AAAI Publications, 2014.