Guard-based Partial Order Reduction

Alfons Laarman, Elwin Pater, Jaco van de Pol, and Michael Weber

Formal Methods and Tools, University of Twente, The Netherlands
elwin.pater@gmail.com
{laarman,vdpol,michaelw}@cs.utwente.nl

Abstract. This paper aims at making partial order reduction independent of the modeling language. Our starting point is the stubborn set algorithm of Valmari (see also Godefroid's thesis), which relies on necessary *enabling* sets. We generalize it to a guard-based algorithm, which can be implemented on top of an abstract model checking interface.
We extend the generalized algorithm by introducing necessary *disabling* sets and adding a heuristics to improve state space reduction. The effect of the changes to the algorithm are measured using an implementation in the LTSMIN model checking toolset. We compare our results to the SPIN model checker, both on the benchmarks from the BEEM database, as well as on a number of PROMELA models.
In many cases, the reduction obtained by our algorithm surpasses the ideal upper bound on the reduction obtained by the ample set method, as established empirically by Geldenhuys, Hansen and Valmari.

1 Introduction

Model checking is an automated method to verify the correctness of concurrent systems by examining all possible execution paths for incorrect behavior. The main difficulty 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 has emerged since the 1980s [3] and several advances have pushed its boundaries. Partial order reduction is among those.

Partial order reduction (POR) exploits independence and commutativity between transitions in concurrent systems. Exhaustive verification needs to consider only a subset of all possible concurrent interleavings, without losing the global behavior of interest to the verified property. In practice, the state space is pruned by considering a sufficient subset of successors in each state.

The idea to exploit commutativity between concurrent transitions has been investigated by several researchers, leading to various algorithms for computing a sufficient successor set. The challenge is to compute this subset during state space generation (on-the-fly), based on the structure of the specification.

Already in 1981, Overman [19] suggested a method to avoid exploring all interleavings, followed by Valmari's [27,30,29] *stubborn sets* in 1988, 1991 and 1992. Also from 1988 onwards, Peled [16] developed the *ample set* [22,23], later extended by Holzmann and Peled [14,24], Godefroid and Pirottin [8,10] the *persistent set* [9], and Godefroid and Wolper [11] *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 either petri-nets [34] or parallel components with local program counters, called processes [14,9]. While this allows the exploitation of certain formalism-specific properties, like *fairness* [23] and token conditions [32], is also complicates the application to other formalisms, for instance, rule-based systems [12]. Moreover, current implementations are tightly coupled to a particular specification language in order to compute a good syntactic approximation of a sufficient successor set. In recognition of these problems, Valmari started early to generalize the stubborn set theory for "transition/variable systems" [28,30].

To address the same problem for symbolic and parallel model checking algorithms, we earlier proposed the PINS interface [2,18], 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 partial order reduction as PINS2PINS wrapper once and for all allows every combination of language and algorithm to benefit.

An important question is whether and how an abstract interface like PINS *can support the partial reduction theory.* We propose a solution that is based on the stubborn set theory. This theory stipulates how to choose a subset of transitions, enabled and disabled, based on a careful analysis of their dependency relations. These relations have been described on the abstract level of transition systems before [30]. Additionally, within the context of petri-nets, the relations were refined to include multiple enabling conditions, a natural distinction in this formalism [32].

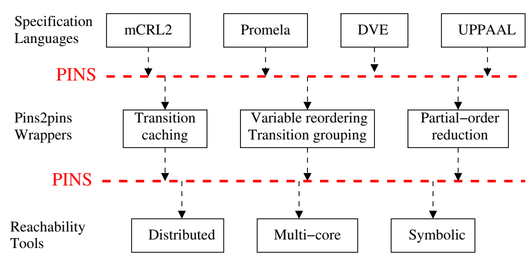


Fig. 1. Modular PINS architecture of LTSMIN

We generalize Valmari's work to a complete language-agnostic setting, by assuming that every transition consists of a number of guard conditions, both enabling and disabling, and an assignment to state variables (Section 3). In Section 4, we extend PINS with the necessary information: a co-enabled 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 [32], our heuristic finds reasonably effective stubborn sets in a short time, hopefully leading to smaller state spaces. In Section 5, we show how LTL can be supported.

Our implementation resides in the LTSMIN toolset [2], 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 LTSMIN's DVE and PROMELA [1] front-ends with guards and their matrices, providing us excellent opportunities to evaluate our solution on the DVE models of the BEEM repository [21] and compare against SPIN [13], as Section 6 shows.

Compared to SPIN, the new algorithm generally provides more reduction and uses less memory, but takes more time to do so. We also show that our implementation of guard-based stubborn sets yields more reduction than the theoretically best reduction using ample sets, as reported by Geldenhuys et al. [7] on a series of BEEM benchmarks.

Summarizing, 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 based on necessary enabling sets;
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;
4. An *empirical evaluation* of guard-based partial order reduction in LTSMIN:
   (a) A comparison on resource consumption and effectiveness of partial order reduction between LTSMIN [2] and SPIN [13] on 16 PROMELA models.
   (b) An impact analysis of necessary disabling sets and the heuristic selection.
   (c) A comparison with the best possible reduction achieved with the ample set method, as reported by Geldenhuys et al. [7], on BEEM models.

2 The Computational Model of Guarded Transitions

In the current section, we provide a model of computation comparable to [7], on purpose leaving out the notion of processes. 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 $S = E_1 \times E_2 \times \ldots \times E_n$ be a set of vectors of elements. Each element E_i represents some finite domain. A state $s = \langle e_1, e_2, \ldots, e_n \rangle \in S$ 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, $\mathbb{B} = \{true, false\}$. 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 *$t \in T$ 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 $en(s) := \{t \in T \mid \bigwedge_{g \in \mathcal{G}_t} g(s)\}$. 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 T be the set of transitions. The state space from s_0 induced by T is $M_T = (S_T, s_0, \Delta)$, where $s_0 \in S$ is the* initial state*, and* $S_T \subseteq S$ *is the set of* reachable states*, and* $\Delta \subseteq S_T \times T \times S_T$ *is the set of* semantic transitions*. These are defined to be the smallest sets such that* $s_0 \in S_T$*, and if* $t \in T$*,* $s \in S_T$ *and* $s \xrightarrow{t} s'$*, then* $s' \in S_T$ *and* $(s, t, s') \in \Delta$*.*

Valmari and Hansen [32, Def. 6] also define guards (conditions), which take the role of enabling conditions for disabled transitions. We later generalize this role to enabled transitions as well for our necessary disabling sets (Section 4.2).

In the rest of the paper, we fix an arbitrary set of vectors $S = E_1 \times E_2 \times \ldots \times E_n$, initial state $s_0 \in S$, and set of transitions T, with induced reachable state space $M_T = (S_T, s_0, \Delta)$. We often just write "transition" for elements of T.

It is easy to see that our model generalizes the setting including processes (as in [7]). One can view the program counter of each process as a normal state variable, check for its current value in a separate guard, and update it in the transitions. But our definition is more general, since it can also be applied to models without a natural notion of a fixed set of processes, for instance rule-based systems, such as the linear process equations in mCRL [12].

Besides guarded transitions, structural information is required on the exact involvement of state variables in a transition. Analogous to [30], we define that some predicate g depends on index i, we test whether $g(s)$ is different from $g(s')$ for some s and s' that only differ at index i.

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* $t = (\mathcal{G}, a) \in T$ *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* $Ws(t) \supseteq \bigcup_{s \in S_T} \delta(s, s')$ *with* $s \xrightarrow{t} s'$*,*
4. *the read set of* t *is* $Rs(t) \supseteq \{i \mid \exists s, s' \in S : \delta(s, s') = \{i\} \wedge s \xrightarrow{t} \wedge s' \xrightarrow{t} \wedge Ws(t) \cap \delta(a(s), a(s')) \neq \emptyset\}$*, 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 $s \in S = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, consider the following transition: $t :=$ *IF* $(s[1] = 0 \wedge s[2] < 10)$ *THEN* $s[3] := s[1] + 1$. This transition 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, the test set of the transition is $Ts(t) = \{1, 2\}$. The write set $Ws(t) = \{3\}$, so the variable set $Vs(t) = \{1, 2, 3\}$. The 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 rephrase partial order reduction theory based on stubborn sets. We follow the definitions from Godefroid's thesis [9], but avoid the notion of processes.

To distinguish which transitions may interfere with one another we use the dependency relation. Recall that $M_T = (S_T, s_0, \Delta)$ is a fixed state space.

Definition 7 (dependency relation). *A* dependency relation $D \subseteq T \times T$ *for* S_T *is a symmetric, reflexive relation such that* $(t_1, t_2) \notin D$ *implies that for all states* $s \in S_T$*, the following hold:*

1. *If* $s \xrightarrow{t_1} s'$*, then* $s \xrightarrow{t_2}$ *iff* $s' \xrightarrow{t_2}$ *(independent transitions neither enable nor disable each other).*
2. *If* $s \xrightarrow{t_1 t_2} s'$ *and* $s \xrightarrow{t_2 t_1} s''$ *then* $s' = s''$ *(commutativity of enabled independent transitions).*

The following is a sufficient condition for a dependency relation [9]:

$$(t_1, t_2) \in D \text{ if } Ws(t_1) \cap Vs(t_2) \neq \emptyset \text{ or } Ws(t_2) \cap Vs(t_1) \neq \emptyset$$

Two transitions t, t' are independent if $(t, t') \notin D$. The dependency relation enables the definition of a persistent set (See Figure 2), which is a condition to preserve deadlocks in the reduced state space.

Definition 8 (persistent set [9]). *A set of transitions* $\mathcal{T}$ *enabled in state* $s \in S_T$ *is persistent in* s *iff for all non-empty sequences of transitions* $s \xrightarrow{t_1, \ldots, t_{n-1}} s_{n-1} \xrightarrow{t_n}$ *including only transitions* $t_i \notin \mathcal{T}$*,* $1 \leq i \leq n$*,* t_n *is independent with all transitions in* $\mathcal{T}$*. Contrary to Godefroid, we also require* $\mathcal{T} = \emptyset \iff en(s) = \emptyset$*.*

By limiting exploration of each state s to a subset of $en(s)$ which a persistent set, the state space is reduced. Researchers have attempted to weaken this theoretical definition as much as possible allowing it to include more sets, increasing the chance to find one which yields a larger reduction [31, Sec. 7.4].

Example 2. Suppose $\mathcal{T}$ is a persistent set in some state s. All transitions outside the persistent set, are independent with all transitions in the persistent set. Hence, by Definition 7 of dependence, it is possible to swap the order of execution of the transitions in the persistent set, with the transitions not in the persistent set, without disabling any of the transitions in the persistent set. Therefore, after the sequence $t_1, t_2, t_3 \notin \mathcal{T}$, all transitions in the persistent set are still enabled, and, because $\mathcal{T}$ is not empty, no deadlocks can occur in between. Hence any (gray) deadlock state reachable after t_1 is also reachable after a $\mathcal{T}$-successor of s.

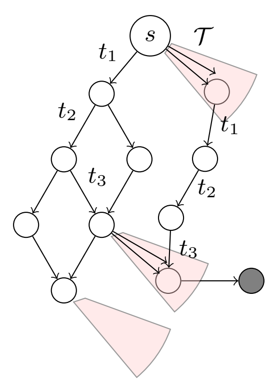


Fig. 2. Persistent set

The theoretical notion of a persistent set is not suitable to compute persistent sets by itself; its reference to future parts of the exploration would still cause the whole state space to be explored. Therefore, we now preset the notion of a *(static) stubborn set*, as developed by Valmari. While this definition is stronger than the persistent set, it can be computed more efficiently.

Definition 9 (may be co-enabled [9]). *A symmetric, reflexive relation $MC \subseteq T \times T$ is a valid* may be co-enabled *relation, if it contains all transitions pairs (t_1, t_2) enabled in the same state: $(\exists s \in S_T : t_1, t_2 \in en(s)) \implies (t_1, t_2) \in MC$.*

Definition 10 (necessary enabling set [9]). *Let $t \in T$ be a disabled transition in state $s \in S_T$, $t \notin en(s)$. 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 $1 \leq i \leq n$).*

Note that it is allowed to over-approximate the may be 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. Using necessary enabling sets, the dependency relation and the may be co-enabled relation, we can define a stubborn set as follows:

Definition 11 (stubborn set [9]). *A set $\mathcal{T}_s$ of transitions is a* stubborn set *in a state s if $\mathcal{T}_s \cap en(s) = \emptyset \iff en(s) = \emptyset$, and for all transitions $t \in \mathcal{T}_s$, the following conditions hold:*

1. *If t is disabled in s, then all transitions in some necessary enabling set for t are also in $\mathcal{T}_s$;*
2. *If t is enabled in s, then all transitions t' that are dependent and may be co-enabled with t are also in $\mathcal{T}_s$.*

Weaker definitions of (static) stubborn sets exists [30,31]. These require however more dependencies, e.g. *writeup* sets. For the sake of a simpler interface we chose the above. The following theorem states how stubborn sets help to identify a persistent set:

Theorem 1 (from stubborn set to persistent set [9]). *Let $\mathcal{T}_s$ be a stubborn set in state s. Then $\mathcal{T} := \mathcal{T}_s \cap en(s)$ is a persistent set in s.*

Algorithm 1 from [9] implements the closure method from [31, Sec. 7.4]. It builds a stubborn set incrementally by making sure that each new transition added to the set fulfills the stubborn set conditions (Definition 11). Starting with some enabled transition, it adds all dependent and possibly co-enabled transitions to a work set. For each new transition in the work set it, either adds a necessary enabling set if it is disabled, or in case of an enabled transition it adds all dependent transitions that may be co-enabled.

Example 3. Suppose Figure 2 is a partial run of the algorithm, and transition t_3 is dependent with some transition $t \in \mathcal{T}$. At some point, when a stubborn set for a state s is calculated, the algorithm will process t and add all dependent transitions, 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 persistent set.

```
function stubborn(s)
    T_work = {t̂} such that t̂ ∈ en(s)
    T_s = ∅
    while T_work ≠ ∅ do
        T_work = T_work − t, T_s = T_s ∪ {t} for some t ∈ T_work
        if t ∈ en(s) then
            T_work = T_work ∪ {t' ∈ Σ | (t, t') ∈ D ∩ MC} \ T_s
        else
            T_work = T_work ∪ N \ T_s where N ∈ find_nes(t, s)
    return T_s
```

Algorithm 1: The stubborn set algorithm

To find a necessary enabling set for a disabled transition t (i.e. $\mathit{find\_nes}(t, s)$), 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 those transitions that can possibly change g to *true*).

In the next section, we will show how we get around program counters in guard-based partial order 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 and disabling sets on guards, and a heuristic selection function. Next, it is shown how the Pins interface can be extended to support guard-based partial order reduction. Finally, we devise an optional extension for language modules to provide fine-grained structural information. Providing this optional information further increases the reduction power.

4.1 Guard-based Necessary Enabling Sets

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

Definition 12 (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 $s \in S_T$ where they both evaluate to true: $\exists s \in S_T : g(s) \wedge g'(s) \implies (g, g') \in MC_g$.*

Given MC_g, a concrete may be co-enabled relation on transitions in the sense of Definition 9 can be retrieved by defining:

$$MC := \{(t_1, t_2) \mid \forall g \in \mathcal{G}_{t_1}, g' \in \mathcal{G}_{t_2} : (g, g') \in MC_g\}$$

Proof. According to Definition 3, we have: $t \in en(s) \Leftrightarrow \bigwedge_{g \in \mathcal{G}_t} g(s)$. Assume that there is $\exists s \in S_T : t_1, t_2 \in en(s)$ and $(t_1, t_2) \notin MC$. According to the definition of MC there must be guards $g \in \mathcal{G}_{t_1}, g' \in \mathcal{G}_{t_2}$ such that $(g, g') \notin MC_g$. However, according to Definition 12 this contradicts our assumption. □

Note that the above definition introduces an imprecision, for instance: let $g_1 := x < 1$,$g_2 := x > 0$ and $g' := x = 1 \vee x = 3$, now a transition $t = (a, \{g_1, g_2\})$ will be defined co-enabled with $t' = (a, \{g'\})$ because $(g_1, g') \in MC_g$ and $(g_2, g') \in MC_g$, whereas $(t, t') \notin MC$. This might be avoided by only splitting a conjunctive g into separate guards if the conjuncts are independent, e.g. $Ts(g_1) \cap Ts(g_2) = \emptyset$. Though, we did not implement this yet.

Example 4. An example of two guards that can never be co-enabled is: $g_1 := v = 0$ and $g_2 := v \geq 5$. In a language like Promela, these guards could implement the channel empty and full operations, where v is some variable that the number of messages in the channel. In a language like mCRL2, the conditions in a summand can be implemented as guards.

We delegate the computation of MC_g to the language front-end (Section 4.4).

Definition 13 (necessary enabling set for guards). *Let $g \in \mathcal{G}_T$ be a guard that is disabled in some state $s \in S_T$, 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 10 can be retrieved as follows (notice the non-determinism):

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

Proof. Let t be a transition that is disabled in state $s \in S_T$, $t \notin en(s)$. Let there be a path where t becomes enabled, $s \xrightarrow{t_1,\ldots,t_n} s' \xrightarrow{t}$ (for some $1 \leq i \leq n$), 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 $\mathcal{N}_t$. □

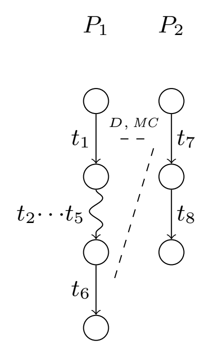
Example 5. Let ch be the variable for a *rendez-vous channel* in a Promela model. A channel read can be modeled as a Promela statement ch? in some process $P1$, guarded by some process counter, e.g. $P1.pc = 1$. A channel write can be modeled as a Promela statement ch! in some process $P2$, guarded by some process counter, e.g. $P2.pc = 10$. The set of all transitions that assign $P1.pc := 1$, are a valid necessary enabling set for both statements (transitions). 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 [20]):

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

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 basically a whole path of transitions between s and t ends up in the stubborn set.



Example 6. Assume two parallel processes P_1 and P_2, with $D(t_1, t_7)$ and $D(t_6, t_7)$. Initially $en(s_0) = \{t_1, t_7\}$; both end up in the stubborn set, since they are dependent 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 fat stubborn set $\{t_1, \ldots, t_7\}$. □

How can this lengthy deduction 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 slim stubborn set.

Example 7. 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. However, t_1 is disabled by itself. From this it can be concluded that t_1 is a necessary enabling set of t_6, and the algorithm can directly terminate with the stubborn set $\{t_1, t_7, t_6\}$. Clearly, using disabling information can save time and can lead to smaller stubborn sets. □

Definition 14 (necessary disabling set for guards). *Let $g \in \mathcal{G}_T$ be a guard that is enabled in some state $s \in S_T$, 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.

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

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 $t = (\mathcal{G}, a) \in T$ 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:

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

4.3 Heuristic Selection for Stubborn Sets

Even though the static stubborn set of Definition 11 is stronger than the persistent set or the *dynamic stubborn set*, its non-determinism still allows many different sets to be computed, as both the choice of an initial transition $\hat{t}$ at Line 2 and the *find_nes* function in Algorithm 1 are non-deterministic. In fact, it is well known that the resulting reductions depend strongly on a smart choice of the necessary enabling set [32]. Known approaches to resolve this problem either search for SCCs in the complete search space [31], or use even more complicated means (like the *deletion algorithm* in [34]). The complexity of these solutions can be somewhat reduced by choosing a 'scapegoat' for $\hat{t}$ [34].

We propose here a practical solution that does neither; using a heuristics, we explore all possible scapegoats, while limiting the search by guiding it towards a local optimum. (This makes the the algorithm deterministic, which has other benefits, cf. Section 7). An effective heuristics for large partial order reductions should select small persistent sets [9]. 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.

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

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 $|T|$).

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

We restrict the search to the cheapest necessary enabling sets:

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

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 [2], 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 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 [2,1]. The original functionality of PINS is:

- INITSTATE: S
- NEXTSTATES: $S \times T \to 2^S$ and
- STATELABEL: $S \times L \to \mathbb{B}$
- DM: $\mathbb{B}_{M \times N}$

The functions of PINS are implemented by the language front-end and used by the exploration algorithms. Note that the POR layer both uses and provides the PINS interface, since it is a state space transformer, i.e. a PINS2PINS wrapper in Figure 1.

Extensions to PINS. We introduced three essential extensions of the PINS concept to support guard-based partial order reduction:

- STATELABEL additionally exports $\mathcal{G}_T$,
- DM is refined to expose the affect sets Ts, Rs and Ws, and
- The May be Co-enabled matrix MC_g is introduced, filled by the front-end (see Example 4), which may over-approximate it (Definition 12).

Tailored Necessary Enabling/Disabling Sets. We can statically derive $\mathcal{N}^{\max}$ and $\overline{\mathcal{N}}^{\max}$ via the refined PINS interface. In order to obtain the maximal reduction performance, we extend the PINS interface with two optional matrices, called $\mathcal{N}_g^{\text{PINS}}$ and $\overline{\mathcal{N}}_g^{\text{PINS}}$. The language front-end can now provide more fine-grained dependencies by inspecting the syntax as in Example 5 and filling in these matrices. The stubborn set algorithm actually uses the following intersections:

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

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 to to be *necessarily enabling* for the inverse: $\neg g$. Or by example: to disable the guard $pc = 1$, is the same as to enable $pc \neq 1$.

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

Liveness properties can be expressed in Linear Temporal Logic (LTL) [25]. An example LTL property is $\Box\Diamond p$, expressing that from any state in a trace ($\Box$ = generally), eventually ($\Diamond$) a state s can be reached s.t. $p(s)$ holds, where p is a predicate over a state $s \in S_T$, 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

Table 1. POR provisos for the LTL model checking of M_T with a property φ

C2	No $a \in \mathit{stubborn}(s)$ is *visible*, except when $\mathit{stubborn}(s) = \mathit{en}(s)$.
C3	$\nexists a \in \mathit{stubborn}(s)$: $a(s)$ is on the DFS stack, except when $\mathit{stubborn}(s) = \mathit{en}(s)$.

infinite traces the system should adhere to. $\mathbb{B}_\varphi$ is an automaton $(M_\mathbb{B}, \Sigma, \mathcal{F})$ 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, due to its finiteness these are all lasso-formed with an accepting state on the cycle. The system M_T is likewise interpreted as a set of infinite traces representing its possible executions: $\mathcal{L}(M_T)$. The model checking problem is now reduced to a *language inclusion* problem: $\mathcal{L}(M_T) \subseteq \mathcal{L}(\mathbb{B}_\varphi)$.

However the number of cycles in M_T is exponential in its size, therefore 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: $\mathcal{L}(M_T) \cap \mathcal{L}(\mathbb{B}_{\neg\varphi}) = \emptyset$. The intersection can be solved by computing the synchronous cross product $M_T \otimes \mathbb{B}_{\neg\varphi}$ The states of $S_{M_T \otimes \mathbb{B}_{\neg\varphi}}$ are formed by tuples (s, s') with $s \in S_{M_T}$ and $s' \in S_{\neg\varphi}$, with $(s, s') \in \mathcal{F}$ iff $s' \in \mathcal{F}_{\neg\varphi}$. The transitions in $T_{M_T \otimes \mathbb{B}_{\neg\varphi}}$ are formed by synchronizing the propositions Σ on the states $s \in S_{M_T}$. For an exact definition of $T_{M_T \otimes \mathbb{B}_{\neg\varphi}}$, we refer to [33]. The construction of the cross product can be done *on-the-fly*, without computing (*and storing!*) the full state space M_T. Therefore, the NDFS [4] algorithm is often used to find accepting cycles (= error traces) as it can do so on-the-fly as well. No trace implies language emptiness.

To combine partial order reduction with LTL model checking, the reduced state space M_T^R is constructed on-the-fly, while the LTL cross product and emptiness check algorithm run on top of the reduced state space [24]. Figure 3 shows the PINS stack with POR and LTL as PINS2PINS wrappers.

To preserve all traces that are captured by the LTL formula, POR needs to fulfill two additional constraints: the *visibility proviso* ensures that traces included in $\mathbb{B}_{\neg\varphi}$ are not pruned from M_T, the *cycle proviso* ensures the necessary fairness. The visible transitions T_{vis} are those that can enable or disable a proposition of φ ($p \in \Sigma$). Table 1 shows sufficient conditions to ensure both provisos ([31] presents weaker conditions). These can easily be integrated in Algorithm 1, which now also requires T_{vis} and access to the DFS stack.

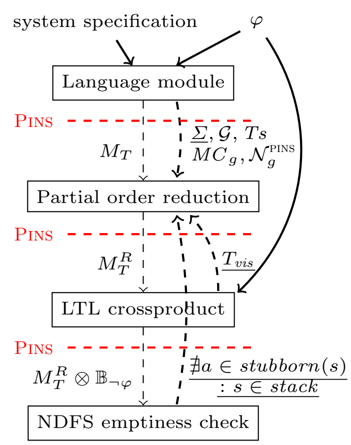


Fig. 3. PINS w. LTL POR

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**, we extend PINS with T_{vis}, to be set by the LTL wrapper based on the predicates Σ in φ:

$$T_{vis} := \{t \in T \mid \mathit{Ws}(t) \cap \bigcup_{p \in \Sigma} \mathit{Ts}(p) \neq \emptyset\}$$

Peled [22, Sec. 4.1] shows how to prove this. However, this is a coarse over-approximation, which we can improve by inputting φ to the language module, so it can export Σ, i.e. $\Sigma \subseteq \mathcal{G}$, and thereby obtain $\mathcal{N}/\overline{\mathcal{N}}$ for it:

$$T_{vis}^{\max} := \bigcup_{p \in \Sigma} \mathcal{N}(p) \cup \overline{\mathcal{N}}(p) \cap T_{vis}$$

We could also make this definition dynamic, by only selecting $\mathcal{N}$ in states where a p is disabled and $\overline{\mathcal{N}}$ where it is enabled, but we did not implement this yet. Finally, we can improve the heuristic (Section 4.3) to avoid visible transitions:

$$cost'(t,s) = \begin{cases} n^2 & \text{if } t \in en(s) \cap T_{vis} \text{ and } t \notin \mathcal{T}_s \cup \mathcal{T}_{work} \\ cost(t,s) & \text{otherwise} \end{cases}$$

To summarize, 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 $T_{vis} : T \to \mathbb{B}$. 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 $\mathcal{N}^{\text{PINS}}$/ $\overline{\mathcal{N}}^{\text{PINS}}$) for these labels as well.

6 Experimental Evaluation

Experimental Setup The LTSMIN toolset implements Algorithm 1 as a language-independent PINS layer since version 1.6. 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 co-enabled matrix, disabling- and enabling sets; see [20] for details. Later the PROMELA front-end SpinS [1] 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.

We compared our 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 [7] 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.

BEEM Models Table 2 shows the results obtained on those models from the BEEM database [21] that were selected by Geldenhuys, Hansen and Valmari [7]. The results in Table 2 are ordered by the best theoretical ample set reduction (best first). These numbers (column AMPLE) are taken from their paper [7, column AMPLE2 Df/Rf]. They indicate the experimentally established best possible reduction that can be achieved with the deadlock preserving ample set method, while considering conditional dependence and full information on the state space.

[1] http://fmt.cs.utwente.nl/tools/ltsmin/
[2] http://spinroot.com
[3] http://www.albertolluch.com/research/promelamodels

Table 2. Comparison of guard-based POR results with [7] (split in two columns)

Model	AMPLE	nes	nes +h	nes 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%

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 interesting to see that guard-based stubborn sets can provide more reduction than ample sets. One explanation is that the ample set algorithm with a dependency relation based on the full state space (Df/Rf, [7]) is still coarse. However, further comparison reveals that many models yield also better reductions than those using dynamic relations (Dd/Rd, [7]), e.g. `protocols.3` with 7% vs 70%. This prompted us to verify our generated stubborn sets, but we found no violations of the persistent set definition. So we suspect that either the relations deduced in [7] are not entirely optimal or the POR heuristic of selecting the smallest ample set fails in these cases.

We also investigated the effects of the necessary disabling sets (Sec. 4.2) and heuristic selection (Sec. 4.3). Heuristic selection improves reductions (column nes+h). For instance, for `cyclic_scheduler.1` it achieves a similar reduction as the optimal ample set method. The reduction improves in nearly all cases, and it improves considerably in several cases. Using Necessary Disabling Sets (nes+nds) in itself did not yield an improvement compared to plain nes,

Table 3. Guard-based POR in LTSMIN vs ample set POR in SPIN

	No Partial-Order Reduction			Guard-based POR LTSMIN			Ample-set POR SPIN		
Model	States $\lvert S_T\rvert$	Trans $\lvert\Delta\rvert$	Time	$\%\lvert S_T\rvert$	$\%\lvert\Delta\rvert$	Time	$\%\lvert S_T\rvert$	$\%\lvert\Delta\rvert$	Time
garp	48,363,145	247,135,869	95.6	3%	1%	35.5	18%	9%	15.5
i-protocol2	14,309,427	48,024,048	15.5	16%	10%	22.7	24%	16%	4.5
peterson4	12,645,068	47,576,805	13.8	3%	1%	2.4	5%	2%	0.3
i-protocol0	9,798,465	45,932,747	17.3	6%	2%	12.5	44%	29%	8.7
brp.prm	3,280,269	7,058,556	3.7	100%	100%	13.5	58%	39%	1.6
philo.pml	1,640,881	16,091,905	5.2	5%	2%	3.3	100%	100%	6.3
sort	659,683	3,454,988	1.9	1660	1660	0.0	112	112	0.0
i-protocol3	388,929	1,161,274	0.6	14%	7%	0.5	26%	16%	0.1
i-protocol4	95,756	204,405	0.2	28%	18%	0.2	38%	28%	0.0
snoopy	81,013	273,781	0.2	13%	5%	0.3	17%	7%	0.0
peterson3	45,915	128,653	0.1	8%	3%	0.0	10%	4%	0.0
SMALL1	36,970	163,058	0.1	18%	9%	0.1	48%	45%	0.0
SMALL2	7,496	32,276	0.0	19%	10%	0.0	48%	44%	0.0
X.509.prm	9,028	35,999	0.1	8%	4%	0.0	68%	34%	0.0
dbm.prm	5,112	20,476	0.0	100%	100%	0.1	100%	100%	0.0
smcs.promela	5,066	19,470	0.0	18%	7%	0.1	25%	11%	0.0

hence we didn't include the results in the table. Combined with the heuristic selection, necessary disabling sets provide an improvement of the reduction in some cases (column nes+h+d). In particular, for `leader_election` the reduction doubles again. Also some other examples show a small improvement.

We can explain this as follows: Although nds allows smaller stubborn sets (cf. Example 7), there is no reason why the eager algorithm would find one. Only with the heuristic selection, the stubborn set algorithm tends to favour small stubborn sets, harvesting the potential gain of nds.

We conclude that, the heuristic selection is more important to improve reductions, than the necessary disabling sets. In terms of computation time the situation is reversed: the selection heuristics is costly, but the disabling sets lower the computation time. In the next section, we investigate computation times.

PROMELA *Models* 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 `-O3 -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 turned had to turn off control flow optimizations (`-o1/-o2/-o3`) for some models (see `ltsmin/spins/test/`).

Table 3 shows the results we obtained. Overall, we witness consistently better reductions by the guard-based algorithm (using nes+h+d). The only exception being `sort` and `brp`. The former is a synthetic example that uses multiple processes and channels to sort an array and can be reduced to a single path in SPIN, while the latter models a protocol and also yields little reduction in SPIN. The reductions are significantly larger than the ample set approach in the cases of `garp`, `peterson`, dining philosophers (`philo.pml`) and `iprotocol`. As a consequence, guard-based partial order reduction in LTSMIN uses considerably less memory than ample-based partial order reduction in SPIN (it makes little sense

Table 4. Reductions ($\%|S_T|$) and runtimes (sec) obtained for LTL model checking

Model	States $\|S_T\|$	LTSMIN ($\%\|S_T\|$) T_{vis}	$T_{vis}^{\max}$	color	SPIN $\%\|S_T\|$	LTSMIN (sec) NoPOR	T_{vis}	$T_{vis}^{\max}$	color	SPIN (sec) NoPOR	POR
garp	72,318,749	21.8%	14.5%	3.8%	18.3%	1,162	421	262	71	2,040	127
i-protocol	20,052,267	100.0%	29.7%	28.8%	41.4%	193	271	86	87	103	37
leader	89,771,572	100.0%	0.6%	0.4%	1.2%	3,558	4,492	18	14	1,390	5

to compare the memory usage in our setting, as the state vector representation in PINS can be several times larger than in SPIN).

On the other hand, the additional computational overhead of our algorithm is clear from the runtime figures. The was expected, as the stubborn-set algorithm considers all transitions whereas the ample-set algorithm only choses amongst components of the system and there might be far fewer processes than transitions. Moreover, the heuristic search still considers all possible initial transitions – we do not select a scapegoat – so a considerable portion of the transitions might still be evaluated. Finally, the choice the store the information on a guard basis requires our implementation to iterate over all guards of a transition at times. This unfortunately cannot be mitigated by combining this information on a transition basis, as enabled guards are treated differently than disabled guards.

However, the runtimes never exceed the runtimes of benchmarks without partial order reduction by a great margin. This is achieved by backtracking in the heuristic search space as soon as $en(s) \subseteq \mathcal{T}_s$.

LTL Model Checking To compare the reductions under LTL model checking with SPIN, we used 3 models that were verified for absence of livelocks, using an LTL property $\Box\Diamond progress$. Implementation details still caused in a slight difference in state counts, so we included these numbers for both tools in Table 4.

In LTSMIN, we used both implementations of the visibility matrix (see Section 5) and the color proviso [6] (`--proviso=color`). To obtain T_{vis}, we defined *progress* with a predicate referencing the right program counters ($Proc.\_pc = 1$). For $T_{vis}^{\max}$, we exported a `np_` label through pins and defined $\varphi := \Box\Diamond\neg$`np_`. SPIN also predefines this label, hence we used the same property (though negated [13]).

The results in Table 4 show that T_{vis} is indeed too coarse an over-approximation. Reductions with $T_{vis}^{\max}$ are much better and even lower than in SPIN. The color proviso shows even better results.

7 Conclusions

We proposed guard-based partial order reduction, as a language-agnostic stubborn set method. It extends Valmari's stubborn sets for transition systems [30] with an abstract interface (PINS) to language modules. It also generalizes previous notions of guards [32], by considering them as disabling conditions as well. The main advantage is that a single implementation of partial order reduction can serve multiple specification languages front-ends and multiple high-performance model checking back-ends. This requires only that it exports guards, guarded transitions, affect sets, and the MC_g matrix. Optional extensions are matrices $\mathcal{N}^{\text{PINS}}$ and $\overline{\mathcal{N}}^{\text{PINS}}$ (computing the latter merely requires negating the guards), which expose more static information to yield better reduction.

We implemented these functions for the DVE and PROMELA front-ends in LTSMIN. It should now be a trivial exercise to add partial order reduction to the mCRL2 and UPPAAL language front- ends. 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 favors small stubborn sets.

Compared to the best possible ample set with conditional dependencies, the stubborn set can reduce the state space more effectively in a number of cases. Compared to SPIN's ample set, LTSMIN generally provides more reduction, but takes more time to do so, probably because of the additional complexity of the stubborn set method, but also due to overhead in the guard-based abstraction.

Comparing our stubborn set computation against earlier proposals, we see the following. While other stubborn set computation methods require $\mathcal{O}(c|T|)$ [31, Sec. 7.4] using scapegoat selection and resolving the dependencies of *find_nes* arbitrarily (where c depends on the modeling formalism used), our algorithm resolves non-deterministic choices heuristically potentially reducing the search space. It would therefore be interesting to compare our heuristic algorithm to approaches other like the deletion algorithm [34], selecting a scapegoat [34] and the strongly connected components method [31], or one of these combined with heuristic. This would provide more insight in the trade-off between time spend on finding stubborn sets and state space reductions.

Finally, we note that heuristic selection algorithm makes POR deterministic. With re-explorations, different persistent sets for the same state can cause probelms [15]. While for NDFS, this problem has been solved efficiently [26], for parallel algorithms, like CNDFS [5], it still plays a role. Previously, we already exploited deterministic POR in the parallel DFS_{FIFO} algorithm [17, end of Sec. 5].

Acknowledgments. We are grateful to Antti Valmari, Patrice Godefroid and Dragan Bošnački for their useful feedback on this paper.

References

1. 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.
2. 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, Berlin, 2010. Springer.
3. Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer, Berlin, Heidelberg, 2008.
4. 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.

5. 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.
6. S. Evangelista and C. Pajault. Solving the Ignoring Problem for Partial Order Reduction. *STTF*, 12:155–170, 2010.
7. J. Geldenhuys, H. Hansen, and A. Valmari. Exploring the scope for partial order reduction. In *ATVA'09*, LNCS, pages 39–53, Heidelberg, 2009. Springer.
8. P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *CAV*, volume 531 of *LNCS*, pages 176–185. Springer, 1990.
9. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, Secaucus, NJ, USA, 1996.
10. P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *CAV*, volume 697 of *LNCS*, pages 438–449. Springer, 1993.
11. P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *FMSD*, 2:149–164, 1993.
12. J.F. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. van Weerdenburg, W. Wesselink, T. Willemse, and J. van der Wulp. The mCRL2 toolset. WASDeTT, 2008.
13. G.J. Holzmann. The model checker SPIN. *IEEE TSE*, 23:279–295, 1997.
14. G.J. Holzmann and D. Peled. An Improvement in Formal Verification. In *IFIP WG6.1 ICFDT VII*, pages 197–211. Chapman & Hall, Ltd., 1995.
15. G.J. Holzmann, D Peled, and M. Yannakakis. On Nested Depth First Search. In *SPIN*, pages 23–32. American Mathematical Society, 1996.
16. 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.
17. A.W. Laarman and Fárago D. Improved On-The-Fly Livelock Detection. In *NFM*, accepted for publication in LNCS. Springer, 2013.
18. 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.
19. W.T. Overman. *Verification of concurrent systems: function and timing*. PhD thesis, University of California, Los Angeles, 1981. AAI8121023.
20. E. Pater. Partial Order Reduction for PINS, Master's thesis, March 2011.
21. R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.
22. D. Peled. All from One, One for All: on Model Checking Using Representatives. In *CAV*, pages 409–423. Springer, 1993.
23. D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV*, LNCS, pages 377–390, London, UK, 1994. Springer.
24. D. Peled. Combining Partial Order Reductions with On-the-Fly Model-Checking. In *CAV*, volume 818 of *LNCS*, pages 377–390. Springer, 1994.
25. Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
26. S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In *TACAS*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.
27. A. Valmari. Error Detection by Reduced Reachability Graph Generation. In *APN*, pages 95–112, 1988.
28. A. Valmari. Eliminating Redundant Interleavings During Concurrent Program Verification. In *PARLE*, volume 366 of *LNCS*, pages 89–103. Springer, 1989.
29. A. Valmari. A Stubborn Attack On State Explosion. In *CAV*, LNCS, pages 156–165. Springer, 1991.
30. A. Valmari. Stubborn Sets for Reduced State Space Generation. In *ICATPN/APN'90*, pages 491–515, London, UK, 1991. Springer.

31. A. Valmari. The State Explosion Problem. In *LPN*, pages 429–528. Springer, 1998.
32. 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.
33. M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 332–344. IEEE, 1986.
34. Kimmo Varpaaniemi. *On the Stubborn Set Method in Reduced State Space Generation*. PhD thesis, Helsinki University of Technology, 1998.