Experimenting with Stubborn Sets on Petri Nets

Sami Evangelista[0000−0002−7666−583X]

LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité,
99, av. J.-B. Clément, 93430 Villetaneuse, France
sami.evangelista@lipn.univ-paris13.fr

Abstract. The implementation of model checking algorithms on real life systems usually suffers from the well known state explosion problem. Partial order reduction addresses this issue in the context of asynchronous systems. We review in this article algorithms developped by the Petri nets community and contribute with simple heuristics and variations of these. We also report on a large set of experiments performed on the models of a Model Checking Contest hosted by the Petri Nets conference since 2011. Our study targets the verification of deadlock freeness and liveness properties.

1 Introduction

System verification based on an exhaustive simulation suffers from the well known state explosion problem: the system state space often grows exponentially with respect to the system structure, making it hard if not impossible to apply it to real life systems. One major source of this problem lies in the concurrent execution of system components that often leads to a blowup of possible interleavings.

When dealing with asynchronous systems, it is often the case that the execution order of system transitions is irrelevant because their occurences can be swapped without consequences on the observed system. This observation has led to the development of some partial order reduction algorithms [18,15,10] that exploit this independence relation between transitions. Although they differ in their implementation of this general principle, they rely on a selective search within the state space: when considering a system state only a subset of allowed transitions are considered to pursue the exploration while the execution of other allowed transitions is postponed to a future state. Such a subset is called stubborn [18], persistent [10] or ample [15] in the literature.

Ignoring some transitions has the consequence of ruling out some system states and building a reduced state space that is more suitable for verification purposes. The filtering mechanism must however fulfill some conditions for the reduced state space to be of any use. Hence, several variations of the method have been designed depending on the property being investigated.

We focus in this article on deadlock freeness and liveness properties for which we review several algorithms. For deadlock freeness we consider Petri nets tailored algorithms while for liveness properties the algorithms usually operate on the underlying reduced state space and are thus language independent.

The contribution of this article is twofold. First, we introduce several simple heuristics and optimisations for existing algorithms. Second, in order to evaluate some of

these algorithms, in particular the benefits of our contributions, we report on a large series of experiments performed on the Petri net models of the Model Checking Contest [] which resulted in approximately 150,000 runs.

The rest of this paper is organised as follows. Background on Petri nets and partial order reduction is given in Section 2. Section 3 recalls the elements of the stubborn set theory for deadlock detection, reviews some algorithms developped for that purpose, and presents our experimental evaluation of these. Likewise, in Section 4 we review partial order reduction algorithms for liveness verification and present experimental observations on these. Section 5 concludes our work and introduces some perspectives.

Acknowledgments We thank the organizers of the Model Checking Contest and all people that contribute to its model database for providing such a database. Experiments presented in this paper were carried out using the Grid'5000 [4] testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organisations (see `https://www.grid5000.fr`). We thank Laure Petrucci for her comments on the first version of this paper.

2 Background

This section introduces notations and definitions used in the remainder of the paper.

Definition 1. *A* Petri net *is a tuple* (P,T,W)*, where* P *is a set of* places *;* T *is a set of* transitions *such that* $T \cap P = \emptyset$ *; and* $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ *is a* weighting function.

From now on, we assume a Petri net $N = (P,T,W)$. For any $n \in P \cup T$, ${}^\bullet n$ and $n^\bullet$ respectively denote the sets $\{o \in P \cup T \mid W(o,n) > 0\}$ and $\{o \in P \cup T \mid W(n,o) > 0\}$.

Definition 2. *The set* $\mathcal{M} = \{m \in P \to \mathbb{N}\}$ *is the set of* markings *of* N*. Let* $m \in \mathcal{M}$ *and* $t \in T$*. If* $W(p,t) \geq m(p), \forall p \in P$ *then* t *is* firable *at* m *(*$m[t\rangle$ *for short). The* firing *of* t *at* m *leads to* $m' \in \mathcal{M}$ *(*$m[t\rangle m'$ *for short) defined by* $m'(p) = m(p) - W(p,t) + W(t,p), \forall p \in P$*. The set* $en(m) = \{t \in T \mid m[t\rangle\}$ *is the set of* firable *(or enabled) transitions at* m*. A* deadlock *is a marking* m *such that* $en(m) = \emptyset$*.*

The firing rule is extended to sequences of transitions (i.e., elements of T^*). Let $m \in \mathcal{M}$ and $\sigma \in T^*$. σ is *firable* at m, ($m[\sigma\rangle$ for short) if $\sigma = \varepsilon$ or if $\sigma = t.\sigma'$, $m[t\rangle m'$ and $m'[\sigma'\rangle$ where $t.\sigma'$ is the concatenation of $t \in T$ and $\sigma' \in S^*$; and ε the empty sequence.

Definition 3. *The* state space *of* (N,m_0) *(with* $m_0 \in \mathcal{M}$ *an* initial marking*) is a couple* (R,A) *such that* R *and* A *are the smallest sets respecting:* $m_0 \in R$ *and if* $m \in R$ *and* $m[t\rangle m'$ *for some* $t \in T$ *then* $m' \in R$ *and* $(m,t,m') \in A$*.*

Stubborn, ample, or persistent sets reductions rely on the use of a reduction function that filters transitions to be used to generate the successors of a marking, leading to the construction of a reduced state space.

Definition 4. *A* reduction function *is a mapping* f *from* $\mathcal{M}$ *to* 2^T*. The* reduced state space *of* (N,m_0) *with respect to* f *is a couple* (R_f,A_f) *such that* R_f *and* A_f *are the smallest sets respecting:* $m_0 \in R_f$ *and if* $m \in R_f$ *and* $m[t\rangle m'$ *for some* $t \in f(m)$ *then* $m' \in R_f$ *and* $(m,t,m') \in A_f$*. If* $m[t\rangle$ *and* $t \in f(m)$ *then we note* $m[t\rangle_f$*. Likewise,* $m[t\rangle_f m'$ *denotes that* $m[t\rangle_f$ *and* $m[t\rangle m'$*.*

If a marking m is such that $en(m) \cap f(m) \subset en(m)$, then it is said to be *reduced*. Otherwise, it is said to be *fully expanded*. An exhaustive state space construction algorithm can be modified to build a reduced state space, simply by considering $en(m) \cap f(m)$ instead of $en(m)$ when processing a marking m.

Obviously, a reduction function must respect some conditions for the reduced state space to be of any use. We review in subsequent sections sufficient conditions to preserve deadlocks and liveness properties.

3 Stubborn sets for deadlock state detection

We recall in this section the theoretical background of the stubborn set theory for deadlock detection. We then review different algorithms that can be used in that context before presenting our experimental results.

3.1 Stubborn set theory for deadlock detection

Dynamic stubborness is a key concept in the stubborn sets theory. Whatever the property being investigated, it is used as the starting point to define reduction functions.

Definition 5. *Let $m \in \mathcal{M}$. $S \subseteq T$ is dynamically stubborn at m [20] if conditions* **D1** *and* **D2** *hold, where:*

D1 $\forall \sigma \in (T \setminus S)^*, t \in S$: $m[\sigma.t\rangle \Rightarrow m[t.\sigma\rangle$
D2 *if* $en(m) \neq \emptyset$ *then* $\exists k \in S \mid \forall \sigma \in (T \setminus S)^*$: $m[\sigma\rangle \Rightarrow m[\sigma.k\rangle$

A transition k in condition D2 is called a *key transition*. If all transitions of S are key transitions, then S is *strongly dynamically stubborn* at m. A reduction function producing dynamically stubborn sets preserves all deadlocks [19] in the reduced state space. Such a reduction function is also characterised as *dynamically stubborn*.

The two conditions of Def. 5 rely on a notion of dependency as defined below.

Definition 6. *A* dependency relation *$\mathcal{D}$ is a symmetric and reflexive relation over $T \times T$ such that $(t,u) \notin \mathcal{D}$ implies that for all $m \in R$: $m[t\rangle m' \wedge m[u\rangle \Rightarrow m'[u\rangle$.*

Given a dependency relation $\mathcal{D}$, we will note $\mathcal{D}(t)$ the set $\{t' \in T \mid (t,t') \in \mathcal{D}\}$.

Generally speaking, it is also required for $\mathcal{D}$ that t and u commute (that their execution order is irrelevant) but we have left out this condition, as it is obviously superfluous in the case of Petri nets.

We will use the following proposition to serve as a basis for the implementation of the stubborn set computation algorithms that we will experiment with.

Proposition 1. *Let $\mathcal{D}$ be a dependency relation, $m \in \mathcal{M}$ and $S \subseteq T$ be such that:*

1. *if $en(m) \neq \emptyset$ then $S \cap en(m) \neq \emptyset$;*
2. *if $t \in S \cap en(m)$ then $\mathcal{D}(t) \subseteq S$;*
3. *if $t \in S \setminus en(m)$ then $\exists p \in P \mid m(p) < W(p,t)$ and $\{t \in T \mid W(t,p) > W(p,t)\} \subseteq S$.*

Then S is strongly dynamically stubborn at m.

According to Item 1 a non deadlock marking may not have an empty stubborn set. If an enabled transition is stubborn then so are all its dependent transitons (Item 2). Last, Item 3 states that if a disabled transition is stubborn then there is a place that disables its firing and such that all transitions that could increase its marking are also stubborn. It is easy to prove that the firing of any transition outside S cannot alter the firability of transitions of S and conversely. Hence, stubborn sets respecting conditions of Prop. 1 are indeed strongly dynamically stubborn sets. Note however, that conditions could be relaxed to ensure dynamic stubborness, see [20].

Prop. 1 is parametrized by $\mathcal{D}$. We define below two such dependency relations.

Definition 7. *The* exact dependency relation $\mathcal{D}_e$ *is such that* $(t,u) \in \mathcal{D}_e$ *if and only if* $\exists m \in R$ *such that* $m[t\rangle m' \wedge m[u\rangle \wedge \neg m'[u\rangle$. *The* static dependency relation $\mathcal{D}_s$ *is such that* $(t,u) \in \mathcal{D}_s$ *if and only if* $\exists p \in P$ *such that* $min(W(t,p),W(u,p)) < min(W(p,t),W(p,u))$.

It is straightforward to show that $\mathcal{D}_e$ and $\mathcal{D}_s$ are dependency relations (as defined by Def. 6) and that $\mathcal{D}_e \subseteq \mathcal{D}_s$. Relation $\mathcal{D}_e$ is the smallest dependency relation as its definition is based on the state space. It is however useless in practice since our goal is precisely to avoid the construction of this state space. Nevertheless, we will use it in our experiments (as first done in [9]) for comparison purposes in situations where the full state space can be computed with available resources. Relation $\mathcal{D}_s$ (from [19,20]) might be larger and may thus have a smaller reduction power but it has the advantage to only rely on the structure of the net and can therefore be the basis of a practical implementation of the stubborn set reduction.

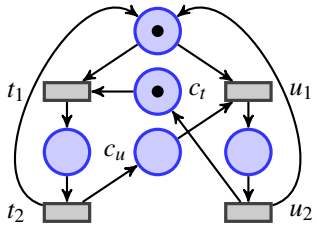

Fig. 1. $\mathcal{D}_e \subset \mathcal{D}_s$ for this net

It is noted in [9] that Def. 5 actually considers for a marking m only its possible futures rather than the full state space. It is thus possible to refine $\mathcal{D}_e$ and define a context-dependent relation (see [9], Sec. 3.1, page 44). We have not considered such a possibility and leave it for future experiments.

The net depicted on Fig. 1 shows a simple example net for which $\mathcal{D}_e$ and $\mathcal{D}_s$ differ. Places c_t and c_u force an alternation between transitions $t_1.t_2$ and transitions $u_1.u_2$. So there actually is no conflict between t_1 and u_1 whereas the structure of the net tells us a different story. Therefore, we have $\mathcal{D}_e = \emptyset$ and $\mathcal{D}_s = \{(t_1,u_1),(u_1,t_1)\}$.

3.2 Stubborn set algorithms for deadlock detection

We chose to introduce and experiment with three algorithms: the closure algorithm, the deletion algorithm and a combination of these two. The first one has been chosen for its simplicity and because one of our contributions is to introduce a simple optimisation to this one that occurs to be quite helpful for some models. The second algorithm has been chosen for its ability to produce minimal sets (with respect to inclusion and according to the conditions of Prop. 1) despite its quadratic complexity.

The closure algorithm This first algorithm (see Alg. 1) is a straightforward implementation of Prop. 1. It initiates the stubborn set (S) construction by picking an enabled transition (condition 1). Based on conditions 2 and 3, it then inserts new transitions in S until all transitions of S have been treated.

Algorithm 1 *clo*, the closure algorithm

```
 1: S := { pick from en(m) } ; Q := S
 2: while Q ≠ ∅ do
 3:     t := pick from Q ; Q := Q \ {t}
 4:     if t ∈ en(m) then
 5:         U := D(t)
 6:     else
 7:         C := {p ∈ P | m(p) < W(p,t)}
 8:         s := pick from C
 9:         U := {t ∈ T | W(t,s) > W(s,t)}
10:     Q := Q ∪ (U \ S) ; S := S ∪ U
11: return S
```

When examining a disabled transition t, Alg. 1 chooses a place that disables its firing. Such a place is called a *scapegoat* place in the literature because it is considered as responsible of t being disabled. The choice of this scapegoat largely impacts the construction of the stubborn set. To limit as much as possible the choices of undesirable scapegoats (in the sense that it may produce unnecessary large sets), we introduce a modification of this algorithm that exploits the past of the construction (see Alg. 2). A counter I is associated with each place. For any $p \in P$, $I[p]$ is initialised with the number of transitions that increase the marking of p (lines 1–2). During the construction, each time a transition t is put in the stubborn set, we decrement the counter of any place p of which the marking is increased by t (lines 25–26). When the counter reaches 0 for some $p \in P$, then we know that p cannot gain any token without a transition of S occurring first. Consequently, all output transitions of p that are disabled by p can be put in S (lines 27–30). However, a transition u put in S by this way does not need to be further considered by the algorithm (this is the purpose of the *enqueue* parameter of procedure *new_stub*) since we know that p is already a valid scapegoat place for t: all transitions that may increase its marking are already stubborn.

Algorithm 2 $clo^\star$, an optimised closure algorithm

```
 1: for p in P do
 2:     I[p] := |{t ∈ T | W(t,p) > W(p,t)}|
 3: init_stub()
 4: while Q ≠ ∅ do
 5:     t := pick from Q
 6:     Q := Q \ {t}
 7:     if t ∈ en(m) then
 8:         U := D(t)
 9:     else
10:         C := {p ∈ P | m(p) < W(p,t)}
11:         s := pick from C
12:         U := {t ∈ T | W(t,s) > W(s,t)}
13:     for u in U \ S do
14:         new_stub(u, true)
15: return S
16: procedure init_stub() is
17:     t0 := pick from en(m)
18:     S := ∅
19:     Q := ∅
20:     new_stub(t0, true)
21: procedure new_stub(t, enqueue) is
22:     S := S ∪ {t}
23:     if enqueue then
24:         Q := Q ∪ {t}
25:     for p in {p ∈ P | W(t,p) > W(p,t)} do
26:         I[p] := I[p] − 1
27:         if I[p] = 0 then
28:             for u in p• \ S do
29:                 if W(p,u) > m(p) then
30:                     new_stub(u, false)
```

We illustrate the principle of our modification with the help of Fig. 2. Let us first see how the basic algorithm proceeds. Suppose that the algorithm is instantiated with $\mathcal{D}_s = \{(t,u),(u,t)\}$ and that the stubborn set construction is initiated with t. Since $u \in \mathcal{D}_s(t)$, u must also be put in the stubborn set. When processing u, the algorithm has to choose among two scapegoats: r and s. Choosing r causes the insertion of v while choosing s does not cause any new transition to be put in the stubborn set and halts the construction. Hence, the algorithm may produce either $\{t,u,v\}$ or either $\{t,u\}$ depending on the scapegoat choice.

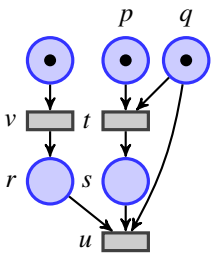

Fig. 2. An example net illustrating Alg. 2

With our modification, if the construction is initiated with t, the insertion of t in the stubborn sets causes $I[s]$ to reach 0. This causes the insertion of u to S without u being put in Q, and immediately stops the construction. Hence, with the same starting transition, our modified algorithm can only produce $\{t,u\}$ as a resulting set.

Note that, as illustrated by our example, our modification does not improve on the basic closure algorithm as any set produced by the former can also be produced by the latter. Our modification must therefore be thought as a way to equip the basic algorithm with a mechanism that can avoid the choice of inappropriate scapegoat places.

The nondeterminism of the closure algorithm stems from the two choices done for the transition picked to initiate the construction of the set at l. 1 of Alg. 1 ; and for the scapegoat place picked at l. 8 of Alg. 1.

We considered 4 strategies to choose the starting transition t to compute a stubborn set S. They rely on a bijective mapping $ord : P \cup T \to \{1,\ldots,|P \cup T|\}$ and two heuristics h_t^e and h_t^f where $h_t^e(t)$ and $h_t^f(t)$ are respectively the number of enabled transitions and the number of forward transitions (i.e., enabled transitions for which the firing leads to an undiscovered marking) in S if t is chosen to initiate S. These 4 strategies are:

- rnd_t — Pick t randomly.
- fst_t — Pick t st. $ord(t)$ is minimal.
- min_t^e — Pick t st. $(h_t^e(t), ord(t))$ is minimal.
- min_t^f — Pick t st. $(h_t^f(t), h_t^e(t), ord(t))$ is minimal.

Note that, using strategies min_t^e, min_t^f we trade the linear complexity of the algorithm for a quadratic complexity since the closure algorithm is now invoked on every possible starting transition.

For the choice of a scapegoat place s we considered 8 strategies. They rely on 3 heuristics h_s^t, h_s^e and h_s^f where $h_s^t(s)$, $h_s^e(s)$ and $h_s^f(s)$ are respectively the number of transitions, the number of enabled transitions and the number of forward transitions inserted in S if s is chosen as a scapegoat. These 8 strategies are:

- rnd_s — Pick s randomly.
- fst_s — Pick s st. $ord(s)$ is minimal.
- min_s^t and max_s^t — Pick s st. $(h_s^t(s), ord(s))$ is minimal (maximal).
- min_s^e and max_s^e — Pick s st. $(h_s^e(s), h_s^t(s), ord(s))$ is minimal (maximal).
- min_s^f and max_s^f — Pick s st. $(h_s^f(s), h_s^e(s), h_s^t(s), ord(s))$ is minimal (maximal).

In the following, $clo(t,s)$ and $clo^\star(t,s)$ denote algorithms Alg. 1 and Alg. 2 respectively, instanciated with strategies t and s for choosing the starting transition and scapegoat places respectively.

The deletion algorithm This second algorithm avoids the necessity of choosing scapegoats. It relies on the construction of a graph capturing transition dependencies.

Definition 8. *Let $\mathcal{D}$ be a dependency relation, and $m \in \mathcal{M}$. A* dependency graph *for m is a directed graph (V,E) with $V = P \cup T$ and $E = E_1 \cup E_2 \cup E_3$ where:*

$$E_1 = (en(m) \times T) \cap \mathcal{D}$$
$$E_2 = \{(t,p) \in (T \setminus en(m)) \times P \mid m(p) < W(p,t)\}$$
$$E_3 = \{(p,t) \in P \times T \mid W(t,p) > W(p,t)\}$$

A dependency graph is nothing more than a reformulation of Prop. 1 as a graph structure. The deletion algorithm iteratively tries to delete enabled transitions from this graph. When a node is deleted then so are its immediate predecessors that are places or enabled transitions. Disabled transitions are deleted when they do not have any successor remaining in V. If, after a deletion step, the graph does not contain an enabled transition anymore, then the deletion is undone and the algorithm tries to delete another transition. Termination occurs when no transition can be further deleted. Set $V \cap T$ is then a valid stubborn set. Indeed, after a successful deletion, edges of E_1 ensure that, for any $t \in en(m) \cap V$, all its dependent transitions are still in V ; and edges of E_2 ensure that, for any $t \in (T \setminus en(m)) \cap V$, there is at least one $p \in P \cap V$ such that $m(p) < W(p,t)$, and, due to to E_3, all transitions that increase the marking of p are also in V.

The only source of nondeterminism of the deletion algorithm is in the choice of the transition t to delete at each step. We considered 6 strategies to make that choice. They rely on two heuristics h^e_d and h^f_d where $h^e_d(t)$ and $h^f_d(t)$ are respectively the number of enabled transitions and the number of forward transitons deleted by the algorithm if t is picked as the transition to be deleted from the graph. These 6 strategies are:

- rnd_t — Pick t randomly.
- fst_t — Pick t st. $ord(t)$ is minimal.
- min^e_t and max^e_t — Pick t st. $(h^e_d(t), ord(t))$ is minimal (maximal).
- min^f_t and max^f_t — Pick t st. $(h^f_d(t), h^e_d(t), ord(t))$ is minimal (maximal).

For the four last strategies, the algorithm has to simulate the deletion of all enabled transitions remaining in the graph before picking one. This however does not impact the algorithm complexity since in the worst case every transition has to be checked anyway.

In the following, $del(t)$ denotes the deletion algorithm instanciated with strategy t for choosing the transition to be deleted at each iteration.

The clodel algorithm It is also possible to chain both algorithms: a stubborn set is first computed using the closure algorithm ; then the deletion algorithm is used on the resulting set to try to further eliminate transitions. We call this combination the clodel algorithm. Its principle has been given by Valmari and Hansen [21] (section 7, page 58).

The clodel algorithm can be instanciated as the closure algorithm is. We could also consider in the instanciation the strategy followed by the deletion algorithm to pick transitions to delete but, to avoid a blowup of experimented configurations, we only considered the fst_t strategy.

In the following, $clodel(t,s)$ denotes the stubborn set construction algorithm that first invokes $clo^\star(t,s)$ and then tries to reduce it with $del(fst_t)$.

3.3 Experimentation context

We have implemented the algorithms introduced in the previous section in the Helena [6] tool and we have performed experiments on models of the MCC model database. We also experimented with the Prod [23] tool that implements the deletion algorithm and the incremental algorithm based on strongly connected components and that is parametrised as is the closure algorithm. In the following, $inc(t,s)$ denotes the incremental algorithm instanciated with strategies t and s for choosing starting transitions and scapegoat places respectively.

All our experimental data are available on the following web page:
`https://www.lipn.univ-paris13.fr/~evangelista/recherche/por-xp`

Input models The MCC model database [1] comprises 128 Petri net models ranging from simple ones used for educational purposes to complex models corresponding to real life systems. Most of these are obtained from parametrised higher level descriptions (e.g., colored Petri nets) and can be instanciated. Although we have experimented with instances of 130 models (all models of the MCC database as well as two models of our own) we have voluntarily left out some of these. Several reasons can explain this: inability of partial order reduction to reduce the state space, timeout in the state space exploration, timeout in the model compilation, ... The reader may find on the aforementioned web page the details on our selection process. As a result, our report deals with 0 models. For each of these we considered two of its instances, or a single one for non parametrized models. This resulted in 0 instances.

Algorithmic configurations With Helena, we have experimented with all algorithm instances considered in Section 3.2: 32 instances for algorithms clo, $clodel$ and $clo^\star$ and 6 instances for algorithm del. Moreover, we have experimented with the $\mathcal{D}_s$ and, when available, the $\mathcal{D}_e$ dependency relations. Since the computation of $\mathcal{D}_e$ required to first perform a full state space exploration to store the state space on disk (as done in [9]) we could not experiment with $\mathcal{D}_e$ on instances for which this operation was not feasible.

With Prod, we experimented with the 8 algorithmic configurations it provides: $del(d)$, $\forall d \in \{fst_t, rnd_t\}$, and $inc(t,s)$, $\forall (t,s) \in \{fst_t, rnd_t, min_t^e\} \times \{fst_s, rnd_s\}$.

Randomness and static node ordering All algorithm instances rely either on a random selection of nodes, or either on a static ordering of nodes computed prior to the exploration (even strategies based on, e.g., a minimisation process, rely on a static ordering

[1] See `https://mcc.lip6.fr/models.php` for the list of models. All models from 2011 to 2022 (included) have been considered.

when several sets of minimal size are available). It is therefore relevant to explore to which extent these mechanisms alter the reduced state space size. Thus, for each model instance, we randomly shuffled the net description 5 times to generate as many different static orderings of nodes and launched each algorithm instance with these 5 settings.

Considering all parameters, our experiments resulted in 0 runs. We checked that all runs on the same model instance produced the same number of deadlock states.

3.4 Experimental observations

We start by general observations before presenting a sample of our results. We only consider for now, the static dependency relation $\mathcal{D}_s$. The comparison with the exact relation $\mathcal{D}_e$ will be addressed later in this section.

First, strategies based on a random selection process perform generally worse. For the deletion algorithm, the rnd_t strategy outperformed others for three instances only (airplane(20), airplane(50) and erk(100)). Likewise, for the three variants of the closure algorithm, selecting the starting transition randomly was the best strategy for only one instance (erk(100)). On all other instances it performed (sometimes significantly) worse. The same remark applies to the choice of the scapegoat place. It is somehow surprising that, all things being equal, choosing the first node according to some static ordering is generally preferable to choosing the node randomly. We conjecture that, unlike random strategies, choosing the first node leads to compute similar stubborn sets when processing similar markings (i.e., whose marking differs on a small number of places) which is probably preferable.

Algorithm $clo^\star$ outperforms clo on several non trivial instances while we did not find out any instance for which the converse holds. Moreover, the scapegoat choice strategy has a lesser impact with algorithm $clo^\star$, which is not surprising considering that the goal of $clo^\star$ is precisely to restrict the number of scapegoat candidates. Nevertheless, when both perform comparably, clo can be significantly faster than $clo^\star$.

For the clo and $clo^\star$ algorithms strategy fst_t (and rnd_t as said above) for choosing the starting transition is largely outperformed by strategies based on a minimisation process. We only found out one model instance for which always choosing the first candidate transition to build the stubborn set produced a smaller state space (qcertifprotocol(6)). Nevertheless when a run based on that strategy could terminate with a number of markings in the same order of magnitude as those based on strategies min_t^e or min_t^f, it was usually much faster due to its linear complexity.

For the closure algorithm and its variants, minimisation based strategies (min_s^t, min_s^e and min_s^f) are clearly preferable for the choice of the scapegoat. They exhibit similar performances. As one could expect maximisation based strategies (max_s^t, max_s^e and max_s^f) perform the worse. As noted above, the strategy used to pick a scapegoat place has clearly a lesser impact with algorithm $clo^\star$.

Identifying forward transitions and using this information generally has a small impact. Moreover, this identification has a non-negligible cost as it requires to execute all enabled transitions and check for the existence of successors in the state space. Hence, algorithmic instances relying on that process are generally slower by approximately 20% compared to strategies that only require to count enabled transitions. On

four model instances (those of models diffusion2d and neighborgrid) they significantly outperformed other algorithmic instances, even guaranteeing the success of the run in two cases.

For the deletion algorithm, strategies max_t^e and max_t^f are clearly the best. On only 13 instances (over 0) did none of these two perform the best (compared to the four other strategies). Moreover, when this occured, the differences observed were negligible whereas strategies max_t^e and max_t^f often significantly outperformed their rivals, sometimes making the run successful.

Sadly, $clodel$ does not bring an improvement with respect to $clo^\star$. For the models for which $clodel$ built smaller state spaces (e.g., aslink, lamport, shieldrvt) the gain was very negligible in terms of reduction (typically less than 10% w.r.t. $clo^\star$) and it often led to an important increase of the search time (remember that $clodel$ first invokes $clo^\star$ then tries to reduce the stubborn set with del). This seems to indicate that $clo^\star$ already often produces stubborn sets that are minimal (w.r.t. inclusion).

The way nodes are ordered can have a large impact on the reduction. In a few pathological situations, we observed that an unfortunate ordering could lead to a state explosion. However, this observation seems more valid for "toy" examples, although there still are real life models (tagged as industrial on the MCC webpage) for which significant differences could be observed according to the ordering (e.g., gpufp or shieldrvt) whatever the algorithm used.

Algorithms $clo^\star$ (using strategies min_t^e or min_t^f) and del (using strategies max_t^e or max_t^f) have, on the average, comparable performances regarding both the reduction power and the search time. Nevertheless, significant differences can be observed when using both algorithms on the same instance.

We conclude our observations with a comparison of relations $\mathcal{D}_s$ and $\mathcal{D}_e$. We could compute the exact relation $\mathcal{D}_e$ for 0 model instances (over 0). For most of these 0 instances the use of $\mathcal{D}_e$ was useless or of very little help (with an additional reduction typically less than 5%). Table 1 gives, for the 12 instances for which relation $\mathcal{D}_e$ performed the best (w.r.t. $\mathcal{D}_s$) the minimal numbers of states in the reduced state space over all runs using the static (column $min(\mathcal{D}_s)$) and exact relations (column $min(\mathcal{D}_e)$). Table is sorted according to the ratio $\frac{min(\mathcal{D}_e)}{min(\mathcal{D}_s)}$. This observation is somehow disappointing as it seems to indicate that there is not much thing that can be expected from refining the dependency relation. Data reported in [9] (see Table 2, p. 49) exhibit better performances of relations based on the analysis of the full state space. We believe the difference with respect to our results can be explained by the DVE modelling language used in [9]. DVE processes synchronise through shared variables or rendez-vous and it is hard, in contrast to Petri nets, to perform a precise static analysis of such models which can in turn explain why semantic based relations space can fill that gap. Moreover algorithms in [9] are parametrized by two relations: the dependency and precedence relations while we only considered the first one here.

3.5 Experimental results sample

To back up our observations, we present in this section a sample of our experimental results. We only consider here the static dependency relation $\mathcal{D}_s$. For comparison

purposes, we computed a *state score* (or more simply score) defined for an algorithm $alg \in \mathcal{A}$ ($\mathcal{A}$ being the set of all algorithmic instances) and a model instance *inst* as:

$$score(alg, inst) = \sum_{i \in \{1,\ldots,5\}} 20 \cdot \frac{S_{\min}(inst)}{S(alg, inst, i)} \quad (1)$$

where $S(alg, inst, i)$ is the number of states in the reduced state space built by algorithm alg on model instance *inst* during run $i \in \{1,\ldots,5\}$ if the run terminated within our time limit (30 min.), or ∞ otherwise ; and $S_{\min}(inst) = \min_{alg \in \mathcal{A}, i \in \{1,\ldots,5\}} S(alg, inst, i)$. Thus a score ranges from 0 if the algorithm did not terminate on the instance for any of the 5 runs to 100 if the algorithm performed the best on all its 5 runs.

Table 2 provides scores for 15 non trivial model instances as well as the average over the 0 model instances we experimented with. The bottom row gives the number of successful runs of an algorithmic instance over all model instances, which is at most 700 (5 runs $\times$ 0 model instances). On the basis of our previous observations and to lighten the table, we voluntarily ruled out several algorithmic configurations. We provide next to each model name, the minimal number of visited states over all algorithms ($S_{\min}$).

In general, Prod's implementation of algorithm *del* performs better than Helena's as evidenced by a comparison of columns $del(fst_t)$ of the two tools. We conjecture that this may be due to a finer implementation of the dependency graph (see, e.g., [22], Def. 3.5, page 136) that permits the computation of smaller sets. Instance smhome(8) is an interesting case from that perspective as Prod, using $del(fst_t)$, significantly outperforms all its competitors. Nevertheless, using the max_t^e and max_t^f strategies, Helena's deletion algorithm usually performs better than Prod's. It could be worthwhile experimenting with these two strategies on a refined dependency graph as computed by Prod.

Instance ibmb2s565s3960 is one the few for which the simplest algorithm (*clo* with strategy fst_t) is competitive with other algorithmic instances. Moreover it naturally significantly outperforms these regarding the execution time due to its linear complexity.

Instances aslink(1,a), deploy(3,a), or lamport(4) illustrate that $clo^\star$ can significantly outperform *clo* reducing further the state space by a factor of approximately 2.

Instance aslink(1,a) illustrates the impact of the scapegoat choice strategy with algorithm *clo* and its lesser importance with algorithm $clo^\star$. With the former, using the same starting transition choice strategy, strategy fst_s performs clearly worse that min_s^e and min_s^f while this observation is less valid when using algorithm $clo^\star$.

Model instance	$min(\mathcal{D}_s)$	$min(\mathcal{D}_e)$
hexagonalgrid(1,2,6)	111,684	901
triangulargrid(1,50,0)	81,198	764
triangulargrid(1,20,0)	13,563	288
hexagonalgrid(1,1,0)	6,708	196
robot(5)	196	10
mapk(8)	3,483	619

Model instance	$min(\mathcal{D}_s)$	$min(\mathcal{D}_e)$
anderson(5)	219,420	104,406
anderson(4)	12,519	6,753
safebus(3)	3,052	2,784
egfr(20,1,0)	162	159
shieldsppp(1,a)	5,453	5,423
deploy(4,a)	571,200	568,234

Table 1. Comparison of relations and $\mathcal{D}_s$ and $\mathcal{D}_e$

Table 2. Scores (according to Eq. (1)) of selected algorithmic instances on 15 model instances and average scores over all experimented model instances

Helena																						Prod		
clo										*clo**										*del*			*del*	*inc*	
fst_t			min_t^e			min_t^f				fst_t			min_t^e			min_t^f				fst_t	max_t^e	max_t^f	fst_t	fst_t	min_t^e
fst_s	min_s^e	min_s^f	fst_s	min_s^e	min_s^f	fst_s	min_s^e	min_s^f		fst_s	min_s^e	min_s^f	fst_s	min_s^e	min_s^f	fst_s	min_s^e	min_s^f						fst_s	fst_s
Model instance aslink(1,a), $S_{\min} = 960{,}868$ states																								
1.6	14.1	14.8	7.8	46.8	50.2	6.8	46.9	51.3		18.4	36.5	32.3	79.2	93.8	93.3	80.3	94.6	94.9		74.0	97.9	99.0	80.9	5.2	5.3
Model instance deploy(3,a), $S_{\min} = 28{,}510$ states																								
16.9	16.8	17.3	32.5	34.8	38.8	32.9	35.4	39.9		24.3	30.7	25.7	58.3	55.9	56.5	60.9	57.4	57.6		77.6	87.0	98.1	76.0	25.9	26.2
Model instance des(5,a), $S_{\min} = 1{,}752{,}989$ states																								
0.6	1.9	2.7	51.2	63.7	68.2	55.1	70.0	75.2		9.9	9.9	6.2	80.2	79.1	79.4	85.1	83.8	84.4		72.0	78.8	84.7	66.9	22.3	45.3
Model instance flexbar(4,b), $S_{\min} = 2{,}017{,}473$ states																								
0.6	0.6	0.0	70.1	82.1	84.0	70.9	83.2	85.2		0.8	0.8	0.0	85.2	86.7	86.8	86.1	87.9	87.9		60.7	85.2	87.0	82.2	59.2	70.5
Model instance gpufp(08,a), $S_{\min} = 47{,}410$ states																								
25.9	25.9	25.9	82.2	82.2	82.2	70.9	70.2	70.3		26.0	25.9	25.9	82.2	82.2	82.2	70.2	70.3	70.3		82.0	96.2	90.5	83.6	71.4	84.4
Model instance ibmb2s565s3960, $S_{\min} = 1{,}500{,}964$ states																								
63.7	63.7	63.7	68.1	68.1	68.1	67.2	67.2	67.2		63.7	63.7	63.7	68.1	68.1	68.1	67.2	67.2	67.2		54.9	66.4	65.4	79.2	54.3	67.1
Model instance lamport(4), $S_{\min} = 206{,}527$ states																								
15.4	17.3	19.5	23.7	26.6	31.2	23.7	27.4	32.9		27.6	31.1	31.2	76.2	85.2	85.8	78.2	88.4	88.3		81.4	97.0	99.4	88.9	23.1	23.1
Model instance peterson(3), $S_{\min} = 102{,}371$ states																								
11.2	14.2	14.5	82.9	89.1	90.0	82.9	89.2	90.0		13.1	14.1	14.3	87.7	90.5	91.4	88.7	91.2	91.5		94.9	99.6	100.0	96.7	39.5	39.5
Model instance raft(3), $S_{\min} = 7{,}262{,}240$ states																								
27.6	28.2	17.9	65.1	74.9	76.2	65.1	75.1	76.5		28.2	28.4	17.9	73.7	75.5	76.4	73.9	75.7	76.6		73.5	76.4	76.6	90.9	70.4	72.8
Model instance satmem(1000,32), $S_{\min} = 1{,}880{,}803$ states																								
36.8	36.8	36.8	50.2	50.2	50.2	50.2	50.2	50.2		36.8	36.8	36.8	50.2	50.2	50.2	50.2	50.2	50.2		50.1	50.1	50.1	68.9	50.2	50.2
Model instance shieldsrv(3,a), $S_{\min} = 386{,}893$ states																								
32.6	36.3	40.7	50.5	68.8	72.4	50.7	73.2	77.0		38.4	44.7	45.1	76.8	84.1	84.0	88.9	97.2	97.0		72.7	86.4	95.7	82.5	38.0	40.3
Model instance smhome(8), $S_{\min} = 14{,}434$ states																								
0.0	0.0	0.0	19.8	18.3	18.3	20.4	18.8	18.8		0.0	0.0	0.0	18.3	18.3	18.3	18.8	18.8	18.8		7.5	18.4	18.6	76.0	13.0	19.0
Model instance stigcomm(2,b), $S_{\min} = 165{,}682$ states																								
0.5	0.6	0.6	34.1	34.1	34.2	34.1	34.1	34.2		2.2	1.3	1.8	53.1	38.3	79.7	53.1	38.4	79.8		40.3	95.3	92.2	40.8	28.2	34.2
Model instance stigelec(4,b), $S_{\min} = 133{,}892$ states																								
0.0	0.0	0.0	12.5	12.7	14.0	12.5	12.8	14.1		0.5	0.5	0.5	19.5	17.2	26.0	19.5	17.2	26.2		31.2	83.9	63.7	54.4	9.4	12.5
Model instance tcp(5), $S_{\min} = 601{,}458$ states																								
29.2	30.1	29.9	90.3	92.4	92.5	93.9	97.4	97.4		29.6	30.2	30.0	93.1	93.4	93.4	97.7	98.2	98.2		60.2	87.8	91.9	69.5	42.1	87.3
Average scores and total number of successful runs over 140 instances																								
22.1	23.4	23.7	59.8	67.2	68.4	62.8	70.3	72.0		24.6	25.5	25.3	70.8	72.9	73.8	74.4	76.4	77.4		51.7	72.1	75.1	61.5	39.2	59.6
605	613	594	684	688	688	681	685	691		622	625	598	698	698	698	700	699	700		681	687	685	675	663	691

4 Stubborn sets for liveness verification

The principle of state space reduction based on stubborn set reduction is somehow to reorder transitions in such a way that only stubborn transitions are considered to generate the successors of a marking while the firing of non stubborn transitions is postponed to a future marking. However, such a marking may never occur due to the so called *ignoring problem*. To illustrate this situation, let us assume a net having a transition t disconnected from the rest of the net (i.e., $\forall p \in P, W(p,t) = W(t,p) = 0$). Then, since $\mathcal{D}_s(t) = \emptyset$ and $m_0[t\rangle m_0$, a dynamically stubborn reduction function may build a reduced state with a single self loop marking. In other words, t hides the dynamics of the rest of the net. The reduced state space is therefore of very little use besides the one of proving that the system does not halt.

As in the previous section, we first recall in this section the theoretical background of the stubborn set theory for liveness verification. We then review different algorithms that can be used in that context and introduce two variations of previous algorithms before presenting our experimental results. Algorithms considered here are not specific to Petri nets. Therefore we will often use here more generic terms, such as *state* instead of marking, or *action* instead of transition.

4.1 Stubborn set theory for liveness verification

The above example has shown that dynamic stubborness is not sufficient for the verification of several properties, including liveness properties, because of the ignoring problem. Formally, transition ignoring occurs when a transition is enabled for some state of a cycle but never executed along that cycle. Otherwise, the reduced state space fullfils the *strong cycle proviso* defined below:

Definition 9. *Let f be a reduction function. The reduced state space has the* strong cycle proviso *property if, for any $m_1[t_1\rangle_f m_2[t_2\rangle_f \dots m_n[t_n\rangle_f m_1$, the following holds for any $t \in T$: $(\exists i \in \{1,\dots,n\}$ such that $m_i[t\rangle) \Rightarrow (\exists j \in \{1,\dots,n\}$ such that $m_j[t\rangle_f)$.*

To verify linear time temporal logic properties, an addition condition linked to the visibility of transitions is required [18,15] but this is out of the scope of our study.

A sufficient condition for a reduction function to ensure the strong cycle proviso is that along any cycle of the reduced state space, there is at least one fully expanded state:

Proposition 2. *Let f be a reduction function. If, for any $m_1[t_1\rangle_f m_2[t_2\rangle_f \dots m_n[t_n\rangle_f m_1$, there is $i \in \{1,\dots n\}$ such that $en(m_i) \subseteq f(m_i)$ then the reduced state space has the strong cycle proviso property.*

Hereafter, we refer to the condition of Prop. 2 as the *weak cycle proviso* or more simply *cycle proviso*. Checking that each cycle contains a fully expanded state is easier and can be done on-the-fly, i.e., during the construction of the reduced state space. Hence, while it is a stronger condition that may bring less reduction, this proposition serves as a basis for all the algorithms we review in the following.

4.2 Stubborn set algorithms for liveness verification

State of the art Most algorithms operate on-the-fly: they address the cycle proviso as they generate the state space. Therefore they are tightly linked to a specific search order.

For DFS, a sufficient condition to ensure the cycle proviso is to forbid a cycle-closing edge (i.e., an edge of which the destination is in the DFS stack) outgoing from a reduced state [15]. An alternate implementation for DFS has also been introduced in [7]. For BFS, a dual sufficient condition is that a reduced state only has successors in the BFS queue [2]. This principle has been generalised in [3] to any search order.

Several optimisations and variations for DFS (including Tarjan algorithm) have been proposed in [5] that lead to an improvement over [15] and [7] in practice. A lesson that can be drawned from the experimentation is that the full expansion of the destination state of a cycle-closing edge should be preferred in practice (rather than the full expansion of the source state, as done in [15] and [7]). Indeed, the destination state needs to be fully expanded when leaving the stack and at that moment, the algorithm may have discovered that the full expansion is no more required (e.g., if all its successors have been fully expanded). This can save useless full state expansions.

The algorithm of [1] alternates expansion phases, during which states are expanded without taking care of the cycle proviso, with topological sorts (efficiently performed in a distributed way) of the resulting reduced state space to detect states to be fully expanded to prevent action ignoring. The full expansion of these states may then lead to new states used to initiate a new expansion phase. The algorithm stops when the topological sort does not produce any new states.

We finally mention the static algorithm of [12] and the two-phase algorithm of [14].

New algorithms ensuring the cycle proviso

A BFS based on destination state revisit. The principle of the dst proviso of [5] can be combined with the proviso of [2] for BFS (see Alg. 3). S denotes in the algorithm a set of safe destination states, in the sense that any reduced state may have successors in S without endangering the cycle condition. S consists of all fully expanded states (see line 6). An invalid cycle may be closed each time the algorithm discovers an edge from an unsafe state to another state that is neither safe, neither in the queue. In that case, the destination state s' of the edge is reinserted in the queue to be fully reexpanded (see lines 15–16). This is the purpose of the second component of items put in Q: if set to **true**, the state must be fully expanded. Otherwise it can be reduced using function *stub* that can be any dynamically stubborn function (see line 5).

This proviso is especially suited for distributed model checking based on state space partitioning (as done in [17]). In that context, whenever a process p generates a state s' it puts s' in the queue of the owner process p' of s'. Ownership is determined using typically the hash value of the state. If the generating process is not the owner process, a communication is needed. With the proviso of [2], a round trip between the two processes would be necessary for p' to notify p whether s' is in its queue (or unvisited) which makes it unusable in that context. With a proviso based on the full expansion of the destination state, p now delegates the responsability of checking the cycle proviso to p'. Hence, it does not require additional communications.

Algorithm 3 Algorithm $\mathsf{BFS}^{\mathsf{dst}}$ ensuring the cycle proviso

```
1:  R := {m0} ; Q := {(m0, false)} ; S := ∅
2:  while Q ≠ ∅ do
3:      (m, fexp) := pick from Q
4:      Q := Q \ {(m, fexp)}
5:      U := if fexp then en(m) else stub(m)
6:      if U = en(m) then S := S ∪ {m}
7:      expand_marking(m, U)
8:  return S
9:  procedure expand_marking(m, U) is
10:     for t ∈ U ∩ en(m) do
11:         let m' be such that m[t⟩m'
12:         if m' ∉ R then
13:             R := R ∪ {m'}
14:             Q := Q ∪ {(m', false)}
15:         else if ¬(m ∈ S ∨ m' ∈ S ∪ Q) then
16:             Q := Q ∪ {(m', true)}
```

New offline optimal provisos. We propose two algorithms that perform optimally in the sense that they do not uselessly (fully) re-expand reduced states to verify the cycle proviso: if the algorithm fully re-expands a state, then it is because it is part of a cycle of reduced states. Such an algorithm will be characterised as *WCP-optimal* (Weak Cycle Proviso-optimal) hereafter. It is easy to find counter examples showing that all algorithms we previously reviewed are not WCP-optimal. Likewise, an algorithm is *SCP-optimal* (Strong Cycle Proviso-optimal) if it does not uselessly visit new transitions to verify the strong cycle proviso: if the algorithm forces the execution of some transition t at a state s, then it is necessarily because s is part of a cycle that ignores t.

The two algorithms have very little practical use as they operate offline and consume a significant additional amount of memory per state (the adjacency list) but can be used experimentally to evaluate how other algorithms perform. Both are a variation of the topological sort based algorithm [1] and rely on an alternation of an expansion phasis with a cycle proviso checking phasis.

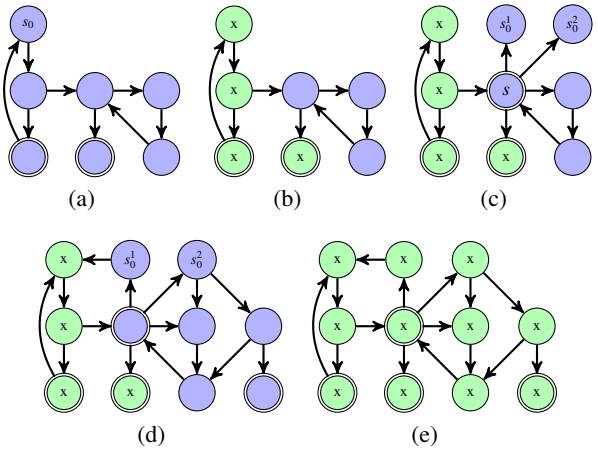

Fig. 3. Expansion and checking phases of our WCP-optimal algorithm. Fig. 3(a): after a 1st expansion step; Fig. 3(b): after a 1st checking step; Fig. 3(c): full expansion of s; Fig. 3(d): after a 2nd expansion step; Fig. 3(e): after a 2nd checking step.

Our WCP-optimal algorithm can be illustrated with the example depicted on Fig. 3. In a first step, the state space is generated starting from the initial state s_0 using a dynamically stubborn reduction function, (i.e., without taking care of transition ignoring). The reduced state space obtained after this first expansion step is depicted on Fig. 3(a) where states with double circles are fully expanded states. The second step, the checking step, consists of marking fully expanded states as *safe* meaning that these cannot be part of a cycle of reduced states. States of which all successors or all predecessors are safe are also marked as such and this procedure is repeated until no more state can be marked. This leads us to the configuration of Fig. 3(b) where safe states are green (and marked with a cross). The outcome of the checking step is to pick an unsafe state (s in our example) and fully expand it (see Fig. 3(c)). This may generate new states which are then used as initial states for a new expansion step (s_0^1 and s_0^2 in our example). After this one (see Fig. 3(d)) a new checking step is triggered and the algorithm may terminate if all states are safe (see Fig. 3(e)).

It is easy to see to that this algorithm is WCP-optimal. Indeed, if after an expansion step the reduced state space already has the weak cycle proviso property then all states will be marked as safe and the algorithm will immediately terminate whereas if it fully expands an unsafe state then it is because this state belongs to a cycle of reduced states.

For the strong proviso, this algorithm can be adapted by repeating the checking step for each transition. A state s is marked as safe for some transition t (denoted by *t-safe* hereafter) if either t is disabled at s, or either t has been executed at s (i.e., it is enabled and stubborn at s). The checking step then proceeds similarly: any state of which all the successors or all the predecessors are t-safe becomes t-safe. If some state is detected as not being t-safe after this step, the algorithm picks such a state and computes a stubborn set that includes t (both the closure and the deletion algorithms can be easily modified to compute stubborn sets including a specific transition). As in the WCP-optimal algorithm, new states that may be reached through this process are used as initial states for a new expansion step. The algorithm may terminate if, after a checking step, all states are marked as t-safe for each transition t.

4.3 Experimentation context

We experimented with the following algorithms:

- $\mathsf{DFS}^{\mathsf{src}}$([7]): DFS + full expansion of source states of cycle closing edges
- $\mathsf{DFS}^{\mathsf{dst}}$([5]): DFS + full expansion of destination states of cycle closing edges [2]
- $\mathsf{BFS}^{\mathsf{src}}$([2]): BFS + full expansion of source states of backward edges
- $\mathsf{BFS}^{\mathsf{dst}}$: BFS + full expansion of destination states of backward edges (i.e., Alg. 3)
- $\mathsf{OPT}^{\mathsf{wcp}}$: the WCP-optimal algorithm presented above

All these algorithms have been integrated in Helena.

Based on the outcome of our first experiment, we selected the following dynamically stubborn functions: $clo^{\star}(t,s)$, $\forall (t,s) \in \{min_t^f, min_t^e\} \times \{min_s^t, min_s^f, min_s^t\}$, and

[2] We implemented the ColoredDest variant of [5] which makes use of state tagging mechanisms to avoid useless full expansions.

$del(t)$, $\forall t \in \{max_t^e, max_t^f, fst_t\}$. These were among the best strategies for algorithms Clo* and Del.

We also included to this second experiment a few additional model instances that were left out in our first experiment (for the reason that all dynamically stubborn function computed the exact same reductions for these) and removed some for which no run could terminate within our time limit (set, as in our first experiment, to 30 min.). Overall this second experiment was performed on 0 model instances of 0 models (over 130) resulting in 0 runs.

4.4 Experimental observations

As in the previous section, we start with general observations before presenting some selected results.

First, as noted elsewhere [2], BFS based provisos perform significantly worse than DFS based provisos. On only one model instance (MAPK(20)) did a BFS based proviso perform significantly better than its DFS analogous. BFS based provisos may however still be useful for specific contexts such as distributed model checking which do not allow a depth-first search order.

Our results confirm those of [5]: in DFS, the full expansion of destination states of cycle closing edges (rather that source states) is preferable, i.e, DFS$^{\text{dst}}$ outperforms DFS$^{\text{src}}$ in general. This also holds for BFS: BFS$^{\text{dst}}$ performs better than BFS$^{\text{src}}$.

We observe that OPT$^{\text{wcp}}$ does not bring any improvement with respect to DFS$^{\text{dst}}$. When both algorithms could terminate, they performed comparably — DFS$^{\text{dst}}$ being even slightly better. This seems to indicate that DFS$^{\text{dst}}$ is already close to optimal in the sense that it never uselessly reexpands states. Moreover, when it does, it "picks" better states (destinations of cycle closing edges) than OPT$^{\text{wcp}}$, that picks them randomly.

4.5 Experimental results sample

Table 3 provides scores of the 5 search algorithms for 15 selected model instances and 4 selected dynamically stubborn functions. Average scores and total number of successful runs (over 685) are also provided at the bottom of the table.

Instance deploy(3,a) illustrates that choosing stubborn sets reducing forward transitions may be unappropriate for liveness properties as it tends to trigger more state reexpansions. Indeed, we observe that, regardless of the search algorithm used, $del(max_t^e)$ and $clo^\star(min_t^e, min_s^e)$ perform better than $del(max_t^f)$ and $clo^\star(min_t^f, min_s^f)$ respectively.

A single run of the OPT$^{\text{wcp}}$ algorithm on instance ibmb2s565s3960 coupled with reduction function $del(fst_t)$ (not shown on the table) could produce a reduced graph with an order of magnitude smaller — which explains the low scores reported in the table. We plan to further investigate the net structure and the conditions that made such a drastic reduction feasible.

Instance shieldtppp(1,b) is one of the few instances for which DFS$^{\text{src}}$ competes favorably against DFS$^{\text{dst}}$.

As said above, algorithm OPT$^{\text{wcp}}$ does not improve on DFS$^{\text{dst}}$. It performed slightly better on 5 of the 15 model instances selected: aslink(1,a), eisenbergmcguire(4), lamport(4), raft(2) and shieldtppp(1,b). Note however that the average score computed over

Table 3. Scores (according to Eq. (1)) of selected algorithmic instances on 15 model instances and average scores over all experimented model instances

BFS^{src} (from [2])				BFS^{dst} (from Alg. 3)				DFS^{src} (from [7])				DFS^{dst} (from [5])				OPT^{wcp} (from Fig. 3)			
$clo^\star$		del		$clo^\star$		del		$clo^\star$		del		$clo^\star$		del		$clo^\star$		del	
min_t^e	min_t^f	max_t^e	max_t^f	min_t^e	min_t^f	max_t^e	max_t^f	min_t^e	min_t^f	max_t^e	max_t^f	min_t^e	min_t^f	max_t^e	max_t^f	min_t^e	min_t^f	max_t^e	max_t^f
min_s^e	min_s^f			min_s^e	min_s^f			min_s^e	min_s^f			min_s^e	min_s^f			min_s^e	min_s^f		
Model instance aslink(1,a), $S_{\min} = 975{,}971$ states																			
0.0	0.0	0.0	0.0	51.0	40.5	43.2	39.7	93.4	94.5	97.5	98.9	92.7	93.9	96.2	97.5	93.0	94.5	97.2	98.5
Model instance deploy(3,a), $S_{\min} = 67{,}212$ states																			
45.0	35.4	40.4	35.8	75.4	36.0	65.3	41.7	92.8	57.3	88.8	82.5	94.5	57.6	95.5	85.8	87.0	50.4	81.1	72.0
Model instance des(5,a), $S_{\min} = 1{,}900{,}083$ states																			
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	76.4	82.0	75.2	80.5	77.8	83.4	75.5	80.6	0.0	0.0	0.0	0.0
Model instance eisenbergmcguire(4), $S_{\min} = 282{,}317$ states																			
20.1	19.0	20.7	19.7	65.5	50.5	71.2	70.6	89.9	91.7	99.3	99.6	90.2	92.7	99.6	99.8	90.4	93.0	99.7	100.0
Model instance flexbar(6,a), $S_{\min} = 148{,}514$ states																			
5.0	5.0	5.0	5.0	10.1	5.4	8.2	6.4	14.8	15.1	14.7	22.5	66.2	30.8	67.4	65.3	57.2	32.7	60.0	68.1
Model instance gpufp(08,b), $S_{\min} = 500{,}514$ states																			
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	66.0	66.2	66.6	67.1	66.9	66.9	68.2	68.2	60.7	60.7	62.1	62.1
Model instance ibmb2s565s3960, $S_{\min} = 1{,}567{,}329$ states																			
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.7	8.4	14.8	17.5	6.5	9.1	15.6	16.5	0.0	0.0	0.0	0.0
Model instance lamport(4), $S_{\min} = 208{,}895$ states																			
12.9	12.6	12.7	12.5	56.9	42.7	73.0	65.6	73.1	74.8	87.0	89.5	83.3	85.9	95.9	98.3	85.1	87.5	97.0	99.5
Model instance peterson(3), $S_{\min} = 125{,}748$ states																			
6.6	5.3	6.6	5.8	54.5	28.5	58.6	53.0	87.3	87.5	93.8	94.1	92.5	92.6	99.7	100.0	84.8	84.5	90.8	91.1
Model instance raft(2), $S_{\min} = 3{,}116$ states																			
60.9	59.6	50.8	59.1	82.9	68.7	80.5	80.4	93.3	78.1	92.2	94.4	98.7	79.6	97.8	98.7	98.8	80.0	99.3	99.7
Model instance shieldsrv(3,b), $S_{\min} = 1{,}832{,}300$ states																			
0.0	0.0	0.0	0.0	1.7	0.0	0.7	0.0	83.0	93.8	75.6	80.5	87.5	97.2	81.7	86.5	0.0	0.0	0.0	0.0
Model instance shieldtppp(1,b), $S_{\min} = 7{,}450$ states																			
0.0	0.1	0.0	0.1	0.5	0.2	0.2	0.1	77.6	78.9	51.6	54.9	56.4	56.3	39.5	41.0	63.2	63.3	42.0	43.1
Model instance smhome(8), $S_{\min} = 155{,}000$ states																			
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	19.3	42.9	0.0	2.0	98.5	82.1	10.5	12.6	97.5	81.7	0.0	0.0
Model instance stigcomm(2,b), $S_{\min} = 468{,}130$ states																			
0.0	0.0	0.0	0.0	4.7	1.5	5.0	0.5	33.8	55.1	55.5	66.5	35.8	59.1	61.2	62.6	20.0	28.0	43.1	37.1
Model instance tcp(5), $S_{\min} = 787{,}898$ states																			
26.7	26.9	26.5	26.6	42.7	30.5	34.7	34.6	37.6	36.1	36.2	36.3	95.2	98.0	88.3	91.3	0.0	0.0	0.0	0.0
Average scores and total number of successful runs over 137 instances																			
23.4	22.8	22.4	22.4	37.1	31.8	35.1	34.7	69.2	69.5	65.0	69.1	82.5	79.6	76.4	79.1	70.8	65.9	65.2	65.6
509	512	492	485	573	537	543	526	665	664	643	645	671	660	656	653	553	541	531	538

all instances must be taken with care since, as witnessed by the last row, runs of algorithm $\mathsf{OPT}^{\mathsf{wcp}}$ timed out more frequently. It is likely that with a higher time limit, the average scores of $\mathsf{OPT}^{\mathsf{wcp}}$ and $\mathsf{DFS}^{\mathsf{dst}}$ would have been very close.

5 Conclusion

We have contributed with this paper with a number of simple algorithmic variants and heuristics for stubborn set construction algorithms. In the context of deadlock detection, we introduced for the closure algorithm an optimisation used to avoid the selection of unappropriate scapegoats and, for the deletion algorithm, we introduced very simple heuristics to choose the transition to delete. For liveness properties we introduce a BFS algorithm based on [2,5] and an offline algorithm that has the property to fully expand states only when absolutely needed.

A second contribution of our work is a large experimentation of these algorithms and variants on models of the MCC database resulting in approximately 150 000 runs using both Prod and Helena tools. These showed that our algorithmic contributions, despite their simplicity, can bring significant results on many instances.

We plan to pursue these experiments in several directions in order to identify room for improvements or to design new heuristics.

First, we would like to experiment with other forms of dependence and precedence relations (as done in [9]) based on the full state graph in order identify if there are still room for improvement in that perspective.

We also plan to implement, for the deletion algorithm, a finer dependency graph, as done by Prod, to study the impact of our heuristics (or others) on it.

Stubborn set construction algorithms may also exploit other informations than the net structures, such as place invariants, or unit decomposition [8] and we plan to investigate how these can be used.

There also exists some algorithms based on integer linear programming techniques [13]. Comparing them to algorithms discussed here is relevant.

For liveness properties, we have only considered the resolution of the ignoring problem, putting aside other conditions required for, e.g., LTL model checking, regarding the visibility of transitions. It seems worth experimenting with such conditions.

Considering the ignoring problem, our experiments are somehow disappointing in the sense that the $\mathsf{DFS}^{\mathsf{dst}}$ algorithm seems hard to outperform. Still we have not experimented with algorithm $\mathsf{OPT}^{\mathsf{scp}}$ and plan to do so in order to check whether new algorithms reasoning on the strong cycle proviso could be of practical use.

Last we have not considered safety properties in our study and we would like to experiment with an algorithm tailored to these, e.g., [16,11], and compare them to general purpose algorithms, i.e., stubborn sets construction algorithm coupled with a cycle proviso and conditions on transition visibility.

References

1. J. Barnat, L. Brim, and P. Rockai. Parallel Partial Order Reduction with Topological Sort Proviso. In *SEFM'2010*, pages 222–231. IEEE Computer Society Press, 2010.

2. D. Bosnacki and G. J. Holzmann. Improving Spin's Partial-Order Reduction for Breadth-First Search. In *SPIN'2005*, volume 3639 of *LNCS*, pages 91–105. Springer, 2005.
3. D. Bosnacki, S. Leue, and A. Lluch-Lafuente. Partial-Order Reduction for General State Exploring Algorithms. In *SPIN'2006*, volume 3925 of *LNCS*, pages 271–287. Springer, 2006.
4. F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Vicat-Blanc Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing CD*, pages 99–106, Seattle, Washington, USA, November 2005. IEEE/ACM.
5. A. Duret-Lutz, F. Kordon, D. Poitrenaud, and É. Renault. Heuristics for Checking Liveness Properties with Partial Order Reductions. In *Automated Technology for Verification and Analysis*, volume 9938 of *LNCS*, pages 340–356. Springer, 2016.
6. S. Evangelista. High Level Petri Nets Analysis with Helena. In *ATPN'2005*, volume 3536 of *LNCS*, pages 455–464. Springer, 2005.
7. S. Evangelista and C. Pajault. Solving the Ignoring Problem for Partial Order Reduction. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2), 2010.
8. H. Garavel. Nested-unit Petri nets. *J. Log. Algebraic Methods Program.*, 104:60–85, 2019.
9. J. Geldenhuys, H. Hansen, and A. Valmari. Exploring the Scope for Partial Order Reduction. In *Automated Technology for Verification and Analysis*, volume 5799 of *LNCS*, pages 39–53. Springer, 2009.
10. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *LNCS*. Springer, 1996.
11. L. M. Kristensen and A. Valmari. Improved Question-Guided Stubborn Set Methods for State Properties. In *ATPN'2000*, volume 1825 of *LNCS*, pages 282–302. Springer, 2000.
12. R.P. Kurshan, V. Levin, M. Minea, D.A. Peled, and H. Yenigün. Static Partial Order Reduction. In *TACAS'1998*, volume 1384 of *LNCS*, pages 345–357. Springer, 1998.
13. A. Lehmann, N. Lohmann, and K. Wolf. Stubborn Sets for Simple Linear Time Properties. In *PETRI NETS 2012*, volume 7347 of *LNCS*, pages 228–247. Springer, 2012.
14. R. Nalumasu and G. Gopalakrishnan. An Efficient Partial Order Reduction Algorithm with an Alternative Proviso Implementation. *FMSD*, 20(3):231–247, 2002.
15. D. Peled. All from One, One for All: on Model Checking Using Representatives. In *CAV'1993*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.
16. K. Schmidt. Stubborn Sets for Standard Properties. In *ATPN'1999*, volume 1639 of *LNCS*, pages 46–65. Springer, 1999.
17. U. Stern and D. L. Dill. Parallelizing the Murphi Verifier. In *CAV'1997*, volume 1254 of *LNCS*, pages 256–278. Springer, 1997.
18. A. Valmari. A Stubborn Attack On State Explosion. In *CAV'1990*, volume 531 of *LNCS*, pages 156–165. Springer, 1990.
19. A. Valmari. Stubborn Sets for Reduced State Space Generation. In *ATPN'1991*, volume 483 of *LNCS*, pages 491–515. Springer, 1991.
20. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer, 1998.
21. A. Valmari and H. Hansen. Can stubborn sets be optimal? In *PETRI NETS 2010*, volume 6128 of *LNCS*, pages 43–62. Springer, 2010.
22. K. Varpaaniemi. Finding Small Stubborn Sets Automatically. In *Proceedings of the 11th International Symposium on Computer and Information Sciences*, pages 133–142, 1996.
23. K. Varpaaniemi, K. Heljanko, and J. Lilius. prod 3.2: An Advanced Tool for Efficient Reachability Analysis. In *CAV'1997*, volume 1254 of *LNCS*, pages 472–475. Springer, 1997.