Heuristics for Checking Liveness Properties with Partial Order Reductions

A. Duret-Lutz[1], F. Kordon[2,3], D. Poitrenaud[3,4], E. Renault[1]

[1] LRDE, EPITA, Kremlin-Bicêtre, France
[2] Sorbonne Universités, UPMC Univ. Paris 06, France
[3] CNRS UMR 7606, LIP6, F-75005 Paris, France
[4] USPC, Université Paris Descartes, Paris, France

Abstract. Checking liveness properties with partial-order reductions requires a cycle proviso to ensure that an action cannot be postponed forever. The proviso forces each cycle to contain at least one fully expanded state. We present new heuristics to select which state to expand, hoping to reduce the size of the resulting graph. The choice of the state to expand is done when encountering a "dangerous edge". Almost all existing provisos expand the source of this edge, while this paper also explores the expansion of the destination and the use of SCC-based information.

1 Introduction

The automata-theoretic approach to explicit LTL model checking explores a Labeled Transition System (LTS). Among the various techniques that have been suggested to tackle the well known state explosion problem, partial-order reductions (POR) reduce the size of the LTS by exploiting the interleaving semantics of concurrent systems. Under interleaved execution semantics, n independent actions (or events) lead to $n!$ possible interleavings. Numerous executions may only correspond to the permutation of independent actions: POR considers only some representative executions, ignoring all other ones [12, 9, 3].

The selection of the representative executions is performed on-the-fly while exploring the LTS: for each state, the exploration algorithm only considers a nonempty *reduced* subset of all *enabled* actions, such that all omitted actions are independent from those in the *reduced* set. The execution of omitted actions is then postponed to a future state. However if the same actions are consistently ignored along a cycle, they may never be executed. To avoid this *ignoring problem*, an extra condition called *proviso* is required. When checking liveness properties, the *proviso* forces every cycle of the LTS to contain at least one *expanded* state where all actions are considered.

This paper proposes several heuristics that can be combined to build new original provisos. Since POR reductions aim to reduce the number of states and transitions, we evaluate each proviso using these two criteria. This analysis reveals new provisos that outperform the state of the art [1, 9]. After the preliminaries of Section 2, we deconstruct a state-of-the-art proviso [1] in Section 3. In Section 4, we explore a new way to choose the state to be expanded among the cycle. Finally Section 5 presents improvements based on SCC information.

2 Preliminaries

A Labeled Transition System (LTS) is a tuple $L = \langle S, s^0, Act, \delta \rangle$ where S is a finite set of states, $s^0 \in S$ is a designated initial state, Act is a set of actions and $\delta \subseteq S \times Act \times S$ is a (deterministic) transition relation where each transition is labeled by an action. If $(s, \alpha, d) \in \delta$, we note $s \rightarrow d$ and say that d is a *successor* of s. We denote by $post(s)$ the set of all successors of s.

A *path* between two states $s, s' \in S$ is a finite and non-empty sequence of adjacent transitions $\rho = (s_1, \alpha_1, s_2)(s_2, \alpha_2, s_3) \ldots (s_n, \alpha_n, s_{n+1}) \in \delta^+$ with $s_1 = s$ and $s_{n+1} = s'$. When $s = s'$ the path is a *cycle*.

A non-empty set $C \subseteq S$ is a Strongly Connected Component (SCC) iff any two different states s, $s' \in C$ are connected by a path, and C is maximal w.r.t. inclusion. If C is not maximal we call it a *partial* SCC.

For the purpose of partial-order reductions, an LTS is equipped with a function $reduced : S \rightarrow 2^S$ that returns a subset of successors reachable via a reduced set of actions. For any state $s \in S$, we have $reduced(s) \subseteq post(s)$ and $reduced(s) = \emptyset \implies post(s) = \emptyset$. The *reduced* function must satisfy other conditions depending on whether we use *ample set*, *stubborn set* or *persistent set* [see 3, for a survey]. The algorithms we present do not depend on the actual technique used.

In this paper, we consider a DFS-based exploration of the LTS using a given *reduced* function. We survey different provisos that modify the exploration to ensure that at least one state of each cycle is expanded. We will first present simple provisos that capture cycles by detecting back-edges of the DFS (i.e., an edge reaching a state on the DFS stack), and always expanding one of its extremities. Then more complex provisos can be presented: to avoid some expansion around each back-edge, they also have to detect any edge that reachs a state that has been explored but is no longer on the stack, as this edge may be part of a cycle.

3 Provisos Inspired from Existing Work

This section presents two well known provisos solving the *ignoring problem* for liveness properties: the proviso introduced by Peled [9] and implemented in Spin [2], and the one of Evangelista and Pajault [1]. The latter proviso augments the former with several mechanisms to reduce the number of expansions. To show how each mechanism is implemented and its effect on the number of expansions, we introduce each mechanism incrementally as a new proviso.

Source expansion Algorithm 1, that we call SOURCE, corresponds to the proviso of Peled [9]. The global variable v stores the set of visited states. Each state on v has a Boolean flag to distinguish states that are on the DFS stack (IN) from those that left it (OUT).

This proviso expands any state s (the *source*) that has a successor s' (the *destination*) on the stack. This amounts to augmenting *todo* (line 11) with all the successors in $post(s)$ that were skipped by $reduced(s)$. The Boolean e prevents states from being expanded multiple times. Overall, this proviso can be implemented with two extra bits per state (one for e, and one for IN/OUT).

Algorithm 1. The Source proviso, adapted from Peled [9].

```
1  Procedure Source(s ∈ S)
2     todo ← reduced(s)
3     v.add(s)
4     v.setColor(s, IN)
5     e ← |todo| ≠ |post(s)|
6     while (¬todo.empty()) do
7        s′ ← todo.pick()
8        if (¬v.contains(s′)) then
9           Source(s′)
10       else if (e ∧ v.color(s′) = IN)
         then
11          todo.add(post(s) \ reduced(s))
12          e ← FALSE
13    v.setColor(s, OUT)
```

Algorithm 2. Conditional source expansion.

```
1  Procedure CondSource(s ∈ S)
2     todo ← reduced(s)
3     v.add(s)
4     v.setColor(s, (|todo| ≠ |post(s)| ?
5                 IN : OUT))
6     while (¬todo.empty()) do
7        s′ ← todo.pick()
8        if (¬v.contains(s′)) then
9           CondSource(s′)
10       else if (v.color(s) = IN ∧
11                v.color(s′) = IN) then
12          todo.add(post(s) \ reduced(s))
13          v.setColor(s, OUT)
14    v.setColor(s, OUT)
```

This proviso relies on the fact that each cycle contains a back-edge, and therefore expanding the source of each back-edge will satisfy the constraint of having at least one expanded state per cycle.

Conditional Source Expansion Some expansions performed by Source could be avoided: the expansion of the source s of a back-edge need only to be performed when its destination s' is not already expanded.

Algorithm 2 shows that this conditional expansion can be achieved by simply changing the semantic of IN and OUT. The IN status now means that a state is on the DFS stack and is not expanded. When a state s is discovered, its color is set to OUT instead of IN (line 5) whenever *reduced*(s) did not produce a set smaller than *post*(s). Doing so allows getting rid of the e variable.

Prioritizing already known successors In Source and CondSource, the decision to expand a state s occurs only when a back-edge has been discovered. However this discovery may occur after having visited several other successors of s, and the recursive calls on these successors are unaware that s will eventually be expanded. This may cause superfluous expansions as shown in Fig. 1.

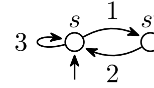


Fig. 1. If edges 1, 2, 3, are explored in that order, CondSource will expand both states. Prioritizing back-edges (i.e., 3, 1, 2) only expands s.

Algorithm 3 shows how this could be fixed. Among the successors of s, the known states are processed first, making sure that s is expanded (if it has to) before processing its other successors. CondSourceKnown forces that ordering by using a set *postponed* to delay the visit of unknown successors; another implementation would be to reorder *todo* to keep known states first. This latter implementation does not require additional memory (the set *postponed*) but it doubles the number of tests of the form *v.contains*(s').

Detecting expanded states on the DFS When a back-edge $s \rightarrow s'$ is detected, the DFS stack contains the states forming a path between s' and s. Some

Algorithm 3. Prioritizing known successors

```
1  Procedure CondSourceKnown(s ∈ S)
2    todo ← reduced(s)
3    v.add(s)
4    v.setColor(s, (|todo| ≠ |post(s)|  ? IN : OUT))
5    postponed ← ∅
6    while (¬todo.empty()) do
7      s′ ← todo.pick()
8      if (¬v.contains(s′)) then
9        postponed.add(s′)
10     else if (v.color(s) = IN ∧ v.color(s′) = IN) then
11       todo.add(post(s) \ reduced(s))
12       v.setColor(s, OUT)
13   while (¬postponed.empty()) do
14     s′ ← postponed.pick()
15     if (¬v.contains(s′)) then
16       CondSourceKnown(s′)
17   v.setColor(s, OUT)
```

Algorithm 4. Detecting expanded states on the DFS using weights

```
1  Procedure WeightedSource(s ∈ S)
2    todo ← reduced(s)
3    v.add(s)
4    v.setColor(s, ORANGE)
5    v.setWeight(s, w)
6    if (|todo| = |post(s)|) then
7      todo ← Expand(s, todo)
8    while (¬todo.empty()) do
9      s′ ← todo.pick()
10     if (¬v.contains(s′)) then
11       WeightedSource(s′)
12       if (v.color(s) = ORANGE ∧ v.color(s′) = RED) then
13         v.setColor(s, PURPLE)
14     else if (v.color(s) ∈ {ORANGE, PURPLE}) then
15       if (v.color(s′) = RED) then
16         todo ← Expand(s, todo)
17       else if (v.color(s′) ∈ {ORANGE, PURPLE}) then
18         if (v.weight(s′) = w) then
19           todo ← Expand(s, todo)
20         else
21           v.setColor(s, PURPLE)
22   switch (v.color(s)) do
23     case GREEN : w ← w − 1
24     case ORANGE : v.setColor(s, GREEN)
25     case PURPLE : v.setColor(s, RED)
26 Function Expand(s ∈ S, succ ⊆ S)
27   succ.add(post(s) \ reduced(s))
28   v.setColor(s, GREEN)        /* scan stack here in WeightedSourceScan */
29   w ← w + 1
30   return succ
```

of these states could already be fully expanded. A generalization of the optimization implemented in CondSource would therefore be to expand s only if there is no expanded state between s' and s. A consequence is that we might have back-edges in which neither the source nor the destination have been expanded. If we decide not to expand s, there might exist another path between s' and s (but not on the current DFS) that will later form a cycle without expanded state [cf. 1, Fig. 6]. Therefore a different way of ensuring that each cycle contains an expanded state is required. Evangelista and Pajault [1] fixed this problem by marking such states as dangerous so that they can trigger an expansion when encountered on another cycle without expanded state.

Detecting the presence of expanded states along the cycle is done by assigning each state s of the DFS a weight that represents the number of expanded states seen since the initial state (s excluded). WeightedSource (Algorithm 4) maintains this count in the global variable w.

The dangerousness of each state is indicated with four colors:

- green means that any cycle through this state already contains an expanded state, so reaching this state does not require any more extension. A state can be marked as green if it is expanded or if all its successors are green.
- orange and purple states are unexpanded states on the DFS stack (their successors have not all been visited). The purple states are those for which a non-green successor has been seen.
- red states are considered dangerous and should trigger an expansion when reached. A purple state becomes red once its successors have been all visited.

In Algorithm 4, two situations trigger an expansion. A source s is expanded when processing an edge $s \to s'$ where s' is marked red (line 16), or when $s \to s'$ is a back-edge and there is no expanded state between s' and s (line 18).

While Algorithm 4 stores the weights in v it is only needed for the states on the DFS. The states on the stack need two bits to store one of the four colors, but states outside the DFS require only one bit as they are either red or green.

Combining prioritization and detection of expanded states on DFS The proviso $\mathsf{C2}^{\mathsf{L}}_{\mathsf{c}}$ presented by Evangelista and Pajault [1] (renamed WeightedSourceKnown, see Algorithm 5) corresponds to the combination of the last two ideas. The main difference is that the second loop (line 21) working on successors ignored by the first loop also performs an expansion (line 28) whenever it discovers a red successor. This was not the case in Algorithm 3 because in CondSourceKnown the only dangerous successors are those on the DFS stack.

Early propagation of green in the DFS stack Evangelista and Pajault [1] also introduce a variant of WeightedSourceKnown in which the green color of a state can be propagated to its predecessors in the DFS stack before the actual backtrack. This propagation could prevent other states from being colored in red [cf. 1, Fig. 8]. As soon as a state is expanded (i.e., in the Expand function), the DFS stack is scanned backward and all orange states that are ready to be popped (i.e., they do not have any pending successors left to be processed) can be marked as green. This backward scan stops on the first state

Algorithm 5. Combining WeightedSource and CondSourceKnown [1]

```
1  Procedure WeightedSourceKnown(s ∈ S)
2    todo ← reduced(s)
3    v.add(s)
4    v.setColor(s, ORANGE)
5    v.setWeight(s, w)
6    if (|todo| = |post(s)|) then
7      todo ← Expand(s, todo)                        /* defined in Algorithm 4 */
8    postponed ← ∅
9    while (¬todo.empty()) do
10     s' ← todo.pick()
11     if (¬v.contains(s')) then
12       postponed.add(s')
13     else if (v.color(s) ∈ {ORANGE, PURPLE}) then
14       if (v.color(s') = RED) then
15         todo ← Expand(s, todo)
16       else if (v.color(s') ∈ {ORANGE, PURPLE}) then
17         if (v.weight(s') = w) then
18           todo ← Expand(s, todo)
19         else
20           v.setColor(s, PURPLE)
21   while (¬postponed.empty()) do
22     s' ← postponed.pick()
23     if (¬v.contains(s')) then
24       WeightedSourceKnown(s')
25       if (v.color(s) = ORANGE ∧ v.color(s') = RED) then
26         v.setColor(s, PURPLE)
27     else if (v.color(s) ∈ {ORANGE, PURPLE} ∧ v.color(s') = RED) then
28       postponed ← Expand(s, postponed)
29   switch (v.color(s)) do
30     case GREEN : w ← w − 1
31     case ORANGE : v.setColor(s, GREEN)
32     case PURPLE : v.setColor(s, RED)
```

that is either GREEN or PURPLE, or that has some unprocessed successors. This idea can be applied to all Weighted algorithms.

eBecause it has to scan the stack, this algorithm may not be presented as a recursive procedure like we did so far. However if WeightedSource or WeightedSourceKnown were implemented as non-recursive procedures, the place to perform the stack scanning would be in function Expand, as defined on page 4. The modification also requires keeping track of whether a state is GREEN because it has been expanded, or because it has been marked during such a stack scanning: an additional bit is needed for this.

We call these two variants WeightedSourceScan and WeightedSourceKnownScan. The latter one corresponds to the proviso $\mathsf{C2}^{\mathsf{L}}_{\mathsf{c}\star}$ presented by Evangelista and Pajault [1].

Evaluation We evaluate the above 7 provisos (as well as more provisos we shall introduce in the next sections) on state-spaces generated from 38 models

Table 1. Comparison of the provisos of section 3. Columns present the number of states and transitions (by million) summed over all runs, their ratio compared to the non-reduced graphs, and the number of states investigated per milliseconds. Provisos with a reference correspond to state-of-the-art algorithms.

	states (10^6)		transitions (10^6)		st/ms
FULL	784.45	100.00%	2,677.73	100.00%	17.90
SOURCE [9]	303.21	38.65%	679.16	25.36%	12.33
WEIGHTEDSOURCE	263.43	33.58%	537.56	20.08%	11.68
WEIGHTEDSOURCEKNOWN [1]	262.63	33.48%	534.35	19.96%	11.77
CONDSOURCE	252.83	32.23%	518.80	19.37%	11.85
CONDSOURCEKNOWN	251.05	32.00%	510.91	19.08%	11.89
WEIGHTEDSOURCESCAN	250.49	31.93%	505.98	18.90%	11.67
WEIGHTEDSOURCEKNOWNSCAN [1]	248.11	31.63%	498.68	18.62%	11.70
NONE	57.58	7.34%	97.65	3.65%	22.65

from the BEEM benchmark [7]. We selected models[1] such that every category of Pelánek's classification [8] is represented.

We compiled each model using a version of DiVinE 2.4 patched by the LTSmin team[2]. This tool produces a shared library that allows on-the-fly exploration of the state-space, as well as all the information required to implement a *reduced* function. This library is then loaded by Spot[3], in which we implemented all the provisos described here. Our *reduced*(s) method implements the stubborn-set method from Valmari [12] as described by Pater [5, p. 21] in a deterministic way: for any state s, *reduced*(s) always returns the same set.

Because provisos can be sensitive to the exploration order (Fig. 1 is one such example), we ran each model 100 times with different transition orders. Table 1 sums these runs for all models, and shows:

- the size of the full (non-reduced) state-space (FULL),
- the size of the reduced state-space using each of the above proviso,
- the size of the reduced state-space, applying just *reduced* without any proviso (NONE). Even if this graph that cannot be used for verification in practice (it ignores too many runs), NONE was used as a lower bound by Evangelista and Pajault [1].

In addition to showing the contribution of each individual idea presented in the above section, Table 1 confirms state-of-the-art results [1]. However, since these values are sums, they are biased towards the largest models. Section 5 will present the most relevant provisos after normalizing the results model by model, in order to be less sensitive to their size.

We observe that WEIGHTEDSOURCEKNOWNSCAN outperforms (18% fewer states) SOURCE as measured by Evangelista and Pajault [1]. We note that

[1] The full benchmark can be found at: `https://www.lrde.epita.fr/~renault/benchs/ATVA-2016/results.html`

[2] `http://fmt.cs.utwente.nl/tools/ltsmin/#divine`

[3] `https://spot.lrde.epita.fr`

Source processes more states per millisecond, because it maintains less information than WeightedSourceKnownScan.

Surprisingly, CondSource, despite its simplicity, is more efficient than WeightedSourceKnown. This might be due to red states introduced in WeightedSourceKnown, as they can generate additional expansions. WeightedSourceKnown can only be competitive with other provisos when combined with the scan of the DFS stack as integrated in WeightedSourceKnownScan. The additional implementation complexity required to update the weights and to scan the stack only provides a very small benefit in term of size; however it can be seen in the last column that the runtime overhead is negligible: all provisos process the same number of states per millisecond.

4 New Provisos Based on Destination Expansion

The Source proviso relies on the fact that each cycle contains a back-edge, so expanding the source of this edge guarantees that each cycle will have an expanded state. This guarantee would hold even if the *destination* of each back-edge was expanded instead. This idea, already proposed by Nalumasu and Gopalakrishnan [4] in a narrower context, brought promising results. This section investigates this idea more systematically yielding many new proviso variants.

Destination expansion The simplest variant, called Dest (Algorithm 6) is a modification of Source that expands the destination of back-edges instead of the source. This requires a new Boolean per state to mark (line 10) whether a state on the stack should be expanded (line 12) during backtrack.

As previously, it is possible to perform a conditional expansion (not marking the destination if the source is already expanded) and to prioritize the visit of some successors. Contrary to Source, where it is preferable to consider known states first, it is better to visit unknown successors (or self-loops) first with Dest, since those successors might ultimately mark the current state for expansion, therefore avoiding the need to expand the destinations of this state's back-edges.

In Dest, the recursive visit of unknown successors could mark the current state for later expansion: in this case, successors that are on the DFS stack have been marked uselessly. The next algorithm avoids these pointless expansions.

Algorithm 7, called CondDestUnknown, implements the prioritization of successors (lines 8–13) as well as the conditional expansion (line 12). The main loop investigates new successors first (through recursive calls), handles self-loops, and postpones the processing of dangerous states. Then, either the current state is marked and must be expanded, or all the dangerous direct successors of the current state are marked to be expanded later (when backtracking these states, after returning from the recursive calls, line 14).

Mixing destination expansion and dangerousness Previous provisos can still perform useless expansions. When an edge $s \rightarrow d$ returning to the DFS is detected, the destination d is marked to be expanded. However during the backtrack of the DFS stack, we might encounter another marked state q that is

Algorithm 6. Expanding destination instead of source

```
1  Procedure Dest(s ∈ S)
2    todo ← reduced(s)
3    v.add(s)
4    v.setMark(s, FALSE)
5    while (¬todo.empty()) do
6      s′ ← todo.pick()
7      if (¬v.contains(s′)) then
8        Dest(s′)
9      else
10       v.setMark(s′, TRUE)

11   if (v.mark(s)) then
12     todo ←
13         post(s) \ reduced(s)
14     while (¬todo.empty())
       do
15       s′ ← todo.pick()
16       if (¬v.contains(s′))
         then
17         Dest(s′)
```

Algorithm 7. Prioritizing unknown successors with conditional expansion of destination

```
1  Procedure CondDestUnknown(s ∈ S)
2    todo ← reduced(s)
3    v.add(s)
4    v.setMark(s, |todo| = |post(s)|)
5    postponed ← ∅
6    while (¬todo.empty()) do
7      s′ ← todo.pick()
8      if (¬v.contains(s′)) then
9        CondDestUnknown(s′)
10     else if (s = s′) then
11       v.setMark(s, TRUE)
12     else if (¬v.mark(s) ∧ ¬v.mark(s′)) then
13       postponed.add(s′)
14   if (v.mark(s)) then
15     todo ← post(s) \ reduced(s)
16     while (¬todo.empty()) do
17       s′ ← todo.pick()
18       if (¬v.contains(s′)) then
19         CondDestUnknown(s′)
20   else
21     while (¬postponed.empty()) do
22       s′ ← postponed.pick()
23       v.setMark(s′, TRUE)
24   v.setMark(s, TRUE)
```

expanded because it belongs to another cycle. Thus d and q are both expanded, but since q belongs to the two cycles, the expansion of d was superfluous.

ColoredDest (Algorithm 8) proposes a solution to this problem. It reuses the color mechanism introduced in WeightedSource (all Weighted algorithms use colors), but without the weights. Here, useless expansions are also tracked by propagating GREEN (line 17); the difference is that only the PURPLE states that are marked will be expanded (lines 19–25), not the ORANGE ones.

As done previously, we can prioritize unknown states, resulting in a new variant: ColoredDestUnknown. This avoids useless markings (line 14). However, mixing this variant with the stack scanning technique is not interesting. Indeed, propagating the GREEN color as early as possible is pointless since the expansion is done when backtracking (i.e., as late as possible): the color will be naturally propagated anyway when it has to be used.

Of course, weights can also be used in addition to colors. In WeightedDest (Algorithm 9), we use a slightly different implementation of weights than in WeightedSource: instead of storing the number of expanded states seen above any state of the DFS stack, we store the depth of each state, and maintain a stack of the depths of all expanded states on the DFS stack. This alternate representation of weights is not necessary in WeightedDest, but will be useful for the next extension we present.

Algorithm 8. Mixing destination expansion and dangerousness.

```
1  Procedure ColoredDest(s ∈ S)
2    todo ← reduced(s)
3    v.add(s)
4    v.setColor(s, (|todo| ≠ |post(s)| ? ORANGE : GREEN))
5    v.setMark(s, FALSE)
6    while (¬todo.empty()) do
7      s' ← todo.pick()
8      if (¬v.contains(s')) then
9        ColoredDest(s')
10       if (v.color(s) = ORANGE) ∧ (v.color(s') = RED) then
11         v.setColor(s, PURPLE)
12     else if (v.color(s) ∈ {ORANGE, PURPLE}) ∧ (v.color(s') ≠ GREEN) then
13       v.setColor(s, PURPLE)
14       v.setMark(s', TRUE)
15   switch (v.color(s)) do
16     case ORANGE :
17       v.setColor(s, GREEN)
18     case PURPLE :
19       if (v.mark(s)) then
20         v.setColor(s, GREEN)
21         todo ← post(s) \ reduced(s)
22         while (¬todo.empty()) do
23           s' ← todo.pick()
24           if (¬v.contains(s')) then
25             ColoredDest(s')
26       else
27         v.setColor(s, RED)
```

In WeightedDest, when a back-edge $s \to s'$ discovers a dangerous state s' on the DFS stack (lines 19–21), the algorithm can use the additional stack e to decide whether s' actually needs to be marked for expansion: if the depth of s' is less than the depth of the last expanded state, then a state has been expanded between s' and s, and the marking can be avoided. However, and as in WeightedSource, when an edge $s \to s'$ reaches a RED state s', the source has to be expanded immediately (lines 23–25) since there is no way to know whether this edge could be part of a cycle without expanded state.

The reason we introduced the depth-based representation of weights is for another heuristic we call DeepestDest. If a state s has several back-edges $s \to s_1$, $s \to s_2$, ..., $s \to s_n$ to different states $s_1, s_2, \ldots, s_n$ on the DFS stack, then all these back-edges close cycles that all pass through the deepest of these states, which is the only one needing to be marked for (possible) expansion. Note that in this situation (one source, with n back-edges), Source would immediately expand one state (the source), ColoredDest and WeightedDest would mark n states for (possible) expansion, while DeepestDest would mark only one.

DeepestDest, which we do not present to save space, can be implemented by modifying Algorithm 9 as follows: instead of marking a destination for ex-

Algorithm 9. Adapting weights to the expansion of destination states.

```
1  Procedure WeightedDest(s ∈ S)
2    todo ← reduced(s)
3    v.add(s)
4    v.setColor(s, ORANGE)
5    v.setMark(s, FALSE)
6    d ← d + 1
7    v.setDepth(s, d)
8    if (|todo| = |post(s)|) then
9      v.setColor(s, GREEN)
10     e.push(d)
11   while (¬todo.empty()) do
12     s′ ← todo.pick()
13     if (¬v.contains(s′)) then
14       WeightedDest(s′)
15       if (v.color(s) = ORANGE) ∧ (v.color(s′) = RED) then
16         v.setColor(s, PURPLE)
17     else if (v.color(s) ∈ {ORANGE, PURPLE}) ∧ (v.color(s′) ≠ GREEN) then
18       v.setColor(s, PURPLE)
19       if (v.color(s′) ∈ {ORANGE, PURPLE}) then
20         if (e.empty() ∨ v.depth(s′) > e.top()) then
21           v.setMark(s′, TRUE)
22       else if (v.color(s′) = RED) then
23         v.setColor(s, GREEN)
24         e.push(d)
25         todo ← todo ∪ (post(s) \ reduced(s))
26   switch (v.color(s)) do
27     case GREEN :
28       e.pop()
29     case ORANGE :
30       v.setColor(s, GREEN)
31     case PURPLE :
32       if (v.mark(s)) then
33         v.setColor(s, GREEN)
34         e.push(d)
35         todo ← post(s) \ reduced(s)
36         while (¬todo.empty()) do
37           s′ ← todo.pick()
38           if (¬v.contains(s′)) then
39             WeightedDest(s′)
40         e.pop()
41       else
42         v.setColor(s, RED)
43   d ← d − 1
```

Table 2. Comparison of the provisos of section 4. For reference, we highlight the performance of WeightedSourceKnownScan, the best proviso of section 3.

	states (10^6)		transitions (10^6)		st/ms
DeepestDestUnknown	276.51	35.25%	570.52	21.31%	11.81
DeepestDest	275.31	35.10%	566.63	21.16%	11.87
WeightedDestUnknown	273.94	34.92%	563.61	21.05%	11.83
Dest	272.79	34.77%	508.17	18.98%	14.48
WeightedDest	272.68	34.76%	559.73	20.90%	11.80
WeightedSourceKnownScan [1]	248.11	31.63%	498.68	18.62%	11.70
CondDest	213.98	27.28%	413.15	15.43%	12.57
CondDestUnknown	213.92	27.27%	412.75	15.41%	12.52
ColoredDest	213.92	27.27%	412.93	15.42%	12.54
ColoredDestUnknown	213.83	27.26%	412.27	15.40%	12.46

pansion at line 21, simply collect the deepest destination, and mark that single destination in the same block as line 42.

Evaluation Table 2 presents the performance of the provisos presented in this section. Some provisos measured here, such as CondDest, WeightedDestUnknown, and DeepestDestUnknown have not been explicitly presented, but the techniques they combine should be obvious from their name. All WeightedDest and DeepestDest variants could also be combined with the Scan technique however these combinations did not achieve interesting performances.

As for the Source family of provisos, using a conditional expansion brings the most benefits. The Unknown variants generally show a very small effect (slightly positive or slightly negative) on a proviso, so this does not seem to be an interesting heuristic. The Weighted and Deepest variants are disappointing. We believe this is due to mixing destination expansions (for back-edges) and source expansions (for red states). However, next section will show that, when combined with others techniques, they bring promising results.

The better provisos of this table are therefore CondDest and ColoredDest (with or without Unknown) with very close results. Note that both provisos are easy to implement, and have a small memory footprint: CondDest requires one additional bit per state, while ColoredDest needs three bits. This is smaller than what WeightedSourceKnownScan requires.

5 Improving Provisos With SCCs

To test the emptiness of the product between a state-space and a specification, an explicit model checker can use two kinds of emptiness checks: those based on Nested Depth First Search (NDFS) [11], and those based on enumerating the Strongly Connected Components (SCC) [10].

All provisos presented so far apply to both NDFS or SCC-based setups. In this section, we present two ideas that are only relevant to model checkers using SCC-based emptiness checks, since they exploit the available information about (partial) SCCs.

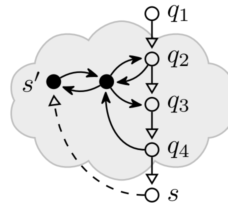
In all SCC-based emptiness checks, states may be partitioned in three sets: live states, dead states, and unknown states. Unknown states are states that have not yet been discovered. Dead states are states that belong to SCCs that have been entirely visited. The remaining states are live, and their SCCs might be only partially known.

Using dead SCCs The first idea is rather trivial. In the COLORED or WEIGHTED provisos presented so far, RED states are always considered dangerous. When we discover an edge $s \to s'$ to a RED state s', we either expand the source s (all WEIGHTED provisos), or propagate the RED color to s (for COLOREDDEST). But these actions are superfluous when the state s' is known to belong to a dead SCC: in that case s and s' are in different SCCs so they cannot appear on the same cycle, and the edge may be simply ignored.

Using live SCCs through highlinks In WEIGHTED provisos, we can derive additional insights about cycles in live SCCs. When we discover an edge $s \to s'$ to a RED state s' that is also live, then s' necessarily belongs to the same SCC as s. This means that $s \to s'$ closes at least one cycle, even if s' is not on the DFS stack: therefore one state on the cycles including s' and s has to be marked for expansion, and only states from the DFS can be marked as such. The default solution used by WEIGHTED provisos would be to expand the source s, but we have also seen previously that expanding states that are that are higher (i.e., less deep) in the DFS stack improves results.

In order to expand higher states, we equip each live state x with a pointer called *highlink*(x) that gives a DFS state (preferably the highest) that is common to all known cycles passing through x. Figure 2 shows a snapshot of an algorithm computing the SCC, where a partial SCC is highlighted. In this configuration, *highlink*$(s') = q_3$. When an edge $s \to s'$ reaches a state s' that is live and RED, we therefore have to ensure that some state between *highlink*(s') and s is expanded: since these two states are on the stack, and s is deeper than *highlink*(s'), we prefer to expand the latter. Furthermore, using the same weight implementation as Algorithm 9, we can easily check whether there exists an expanded state between *highlink*(s') and s to avoid additional work.

Fig. 2. White states and edges with white arrows denote the DFS stack. Black states have been fully visited. The cloud represents the only (partial, non trivial) SCC that has been discovered so far. Dashed-edge has not yet been visited.



In the example of Figure 2, once s, q_4, and q_3 are popped from the DFS stack *highlink*(s') should be updated to value of *highlink*(q_3) which is q_2. In our implementation, these updates are performed lazily in a way that is similar to the path-compression technique used in the union-find data structure [6]: when we query the highlink of a state and find that it points to a state q that is not on the DFS stack, we update it to *highlink*(q).

Because it would require introducing an SCC-based algorithm, and because we consider that the fine details of how to update *highlink*(x) efficiently in this

Table 3. Comparison of the provisos of section 5. For reference, we recall the performances of DeepestDest,WeightedDest that are the support of heuristics presented in this section, and those of ColoredDest, the best proviso so far.

	states (10^6)		transitions (10^6)		st/ms
DeepestDest	275.31	35.10%	566.63	21.16%	11.87
DeadDeepestDest	269.10	34.30%	543.64	20.30%	11.92
WeightedDest	272.68	34.76%	559.73	20.90%	11.80
DeadWeightedDest	270.62	34.50%	554.91	20.72%	11.88
DeadWeightedSourceKnownScan	247.68	31.57%	497.79	18.59%	11.67
ColoredDest	213.92	27.27%	412.93	15.42%	12.54
DeadColoredDest	213.87	27.26%	412.80	15.42%	12.53
HighlinkWeightedDest	207.41	26.44%	393.22	14.68%	12.44
HighlinkWeightedDestScan	206.23	26.29%	391.05	14.60%	12.41
HighlinkWeightedSourceKnown	203.20	25.90%	386.84	14.45%	12.20
HighlinkWeightedSourceKnownScan	203.08	25.89%	386.60	14.44%	12.12
HighlinkDeepestDest	192.84	24.58%	349.89	13.07%	13.20
HighlinkDeepestDestScan	191.78	24.45%	347.95	12.99%	13.21

context is not necessary to reach our conclusion, we have decided to not present this algorithm formally. Our implementation is however publicly available.[1]

Evaluation Table 3 presents the performances of the provisos presented in this section. We prefix by Dead and Highlink the provisos of previous sections when combined with the two SCC-based heuristics. Note that dead states are also ignored in Highlink variants.

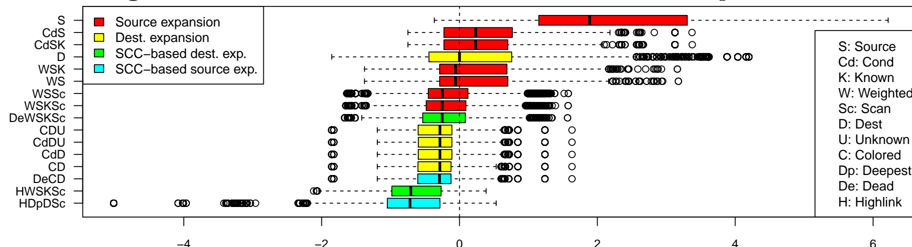
We observe that the Dead variants only improve the original non-Dead variants by 3%. On the contrary, the Highlink variants bring an important benefit. For instance the addition of Highlink to DeadWeightedDest reduces the number of states by 25% and the number transitions by 30%. The improvements are similar when using Highlink on top of the state-of-the-art WeightedSourceKnown variants. These results confirm that the case where an edge leading to a (non-dead) red state is well handled by this Highlink.

Note that while DeepestDest combinations did not achieve interesting performances so far, it outperforms all provisos presented in this paper when combined with Highlink and Scan techniques.

Among the 46 provisos we implemented and benched[1], we selected the 16 most relevant: all the Source-based strategies (to see the contribution of each optimization), the bests Dest-based ones (i.e., without weights), and finally the best of each SCC-based strategy.

Figure 3 shows box plots of standard score computed for selected provisos and all models. The standardization is performed as follows. For each model M, we take the set of 1600 runs generated (100 runs per proviso), and compute a mean number of states μ_M and a standard deviation σ_M. The standard score of a run r is $\frac{states(r)-\mu_M}{\sigma_M}$. Therefore a score of 2 signifies that the run is two standard deviations away from the mean (of selected provisos) for the given model. Figure 3 shows the distribution of these scores as box plots. Each line

Fig. 3. Distributions of standard scores for a selection of provisos.



shows a box that spans between the first and third quartiles, and is split by the median. The whiskers show the ranges of values below the first and above the third quartile that are not further away from the quartiles than 1.5 times the interquartile range. Other values are shown as outliers using circles.

The ranking of provisos in Figure 3 differs from previous tables that were biased toward large models. However, if we omit some permutations between provisos that have close median standard score, the order stays globally the same.

If we look at provisos that do not exploit SCCs, the best provisos appear to be all the CondDest variants, but they are very close to the state-of-the-art WeightedSourceKnownScan [1]. Introducing SCC-based provisos clearly brings another level of improvements, where, on the contrary to previous provisos, expanding the source or the destination does not make a serious difference.

6 Conclusion

Starting from an overview of state-of-the-art provisos for checking liveness properties, we have proposed new provisos based on the expansion of the destination instead of the source. These new provisos have been successfully combined with existing heuristics (Scan, (Un)Known, Weighted) and new ones (Colored, Deepest, Dead, and Highlink).

For source expansion, our results confirm and extend those of Evangelista and Pajault [1] who have shown that WeightedSourceKnown and WeightedSourceKnownScan were better than Source. However when deconstructing these provisos to evaluate each optimization independently, we discovered that most of the gain can be obtained by implementing a very simple proviso, CondSource, that does not require maintaining weights or scanning the stack.

Expanding the destination of edges, even in very simple implementations like CondDest, appears to be competitive with state-of-the-art provisos using source-based expansions. When using an NDFS-based emptiness check, we recommend to use CondDest since it remains very simple to implement, requires small memory footprint and achieves good results.

We have also shown how to exploit SCC-based information to limit the number of expansions: the use of Highlink brings a solid improvement to all provisos. When using an SCC-based emptiness check, our preference goes to HighlinkWeightedSourceKnown that does not require scanning the stack.

From this extensive analysis, we also observe: (1) the WEIGHTED-variants ruins the benefits of DEST-based provisos without HIGHLINKS, while they increase performances of SOURCE-based ones, (2) the (UN)KNOWN variants only bring a modest improvements while they double the number of visited transitions, (3) the SCAN heuristic is not of interest when combined with HIGHLINKS but is efficient otherwise. A scatter plot[1] comparing the best of SOURCE-based provisos with the best of DEST-based ones, shows that they are complementary.

Most of the heuristics presented in this paper are derived from state-of-the-art provisos which have been proven correct [1, 9]. Since reproducing the proof schemes for all the 46 provisos we presented in this paper would be laborious, and considering they were implemented, we opted for an extensive test campaign checking that, for randomly generated LTS, all provisos produce reduced graphs containing at least one expanded state per cycle.

Finally, note that SOURCE is for instance implemented in Spin. However, the *reduced* function implemented in Spin is different than ours: it returns either a single transition, or all transitions. With such a *reduced* function, some of the variants we presented make no sense (KNOWN, UNKNOWN, DEEPEST), and the results might be completely different. We leave the evaluation of the effect of different *reduced* functions on the provisos as a future work.

References

1. S. Evangelista and C. Pajault. Solving the ignoring problem for partial order reduction. STTT, 12(2):155–170, 2010.
2. G. J. Holzmann. The model checker Spin. IEEE Transactions on software Engineering, 23(5):279–295, May 1997.
3. A. Laarman, E. Pater, J. Pol, and H. Hansen. Guard-based partial-order reduction. STTT, pp. 1–22, 2014.
4. R. Nalumasu and G. Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. FMSD, 20(1):231–247, Jan. 2002.
5. E. Pater. Partial order reduction for PINS. Technical report, University of Twente, Mar. 2011.
6. M. M. A. Patwary, J. R. S. Blair, and F. Manne. Experiments on union-find algorithms for the disjoint-set data structure. In SEA'10, vol. 6049 of LNCS, pp. 411–423. Springer, 2010.
7. R. Pelánek. BEEM: benchmarks for explicit model checkers. In SPIN'07, vol. 4595 of LNCS, pp. 263–267. Springer, 2007.
8. R. Pelánek. Properties of state spaces and their applications. STTT, 10:443–454, 2008.
9. D. Peled. Combining partial order reductions with on-the-fly model-checking. In CAV'94, vol. 818 of LNCS, pp. 377–390. Springer, 1994.
10. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In LPAR'13, vol. 8312 of LNCS, pp. 668–682. Springer, Dec. 2013.
11. S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In TACAS'05, vol. 3440 of LNCS, Apr. 2005. Springer.
12. A. Valmari. Stubborn sets for reduced state space generation. In Advances in Petri Nets, vol. 483 of LNCS, pp. 491–515. Springer, 1991.