Dynamic Partial Order Reduction Using Probe Sets*

Harmen Kastenberg and Arend Rensink

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands

Abstract. We present an algorithm for partial order reduction in the context of a countable universe of deterministic actions, of which finitely many are enabled at any given state. This means that the algorithm is suited for a setting in which resources, such as processes or objects, are dynamically created and destroyed, without an *a priori* bound. The algorithm relies on abstract enabling and disabling relations among actions, rather than associated sets of concurrent processes. It works by selecting so-called *probe sets* at every state, and backtracking in case the probe is later discovered to have missed some possible continuation.

We show that this improves the potential reduction with respect to persistent sets. We then instantiate the framework by assuming that states are essentially sets of *entities* (out of a countable universe) and actions test, delete and create such entities. Typical examples of systems that can be captured in this way are Petri nets and (more generally) graph transformation systems. We show that all the steps of the algorithm, including the estimation of the missed actions, can be effectively implemented for this setting.

1 Introduction

Explicit state model checking is, by now, a well-established technique for verifying concurrent systems. A strong recent trend is the extension of results to *software* systems. Software systems have, besides the problems encountered in the traditional concurrent automata, the additional problem of unpredictable dynamics, for instance in the size of the data structures, the depth of recursion and the number of threads.

Typically, the number of components in concurrent software systems is fairly large, and the actions performed by those components, individually or together (in case of synchronization), can be interleaved in many different ways. This is the main cause of the well-known *state space explosion problem* model checkers have to cope with. A popular way of tackling this problem is by using so-called *partial order reduction*. The basic idea is that, in a concurrent model of system behaviour based on *interleaving* semantics, different orderings of independent actions, e.g., steps taken by concurrent components, can be treated as *equivalent*, in which case not all possible orderings need to be explored.

In the literature, a number of algorithms have been proposed based on this technique; see, e.g. [2, 3, 4, 12, 13]. These are all based upon variations of two core techniques:

* This work has been carried out in the context of the GROOVE project funded by the Dutch NWO (project number 612.000.314).

F. van Breugel and M. Chechik (Eds.): CONCUR 2008, LNCS 5201, pp. 233–247, 2008.

persistent (or *stubborn*) *sets* [3, 12] and *sleep sets* [3]. In their original version, these techniques are based on two important assumptions:

1. The number of actions is finite and *a priori* known.
2. The system consists of a set of concurrent processes; the orderings that are pruned away all stem from interleavings of actions from distinct processes.

Due to the dynamic nature of software, the domain of (reference) variables, the identity of method frames and the number of threads are all impossible to establish beforehand; therefore, the number of (potential) actions is unbounded, meaning that assumption 1 is no longer valid. This has been observed before by others, giving rise to the development of *dynamic* partial order reduction; e.g., [2, 5]. As for assumption 2, there are types of formalism that do not rely on a pre-defined set of parallel processes but which do have a clear notion of independent actions. Our own interest, for example, is to model check graph transformation systems (cf. [7, 11]); here, not only is the size of the generated graphs unbounded (and so assumption 1 fails) but also there is no general way to interpret such systems as sets of concurrent processes, and so assumption 2 fails as well.

In this paper, we present a new technique for partial order reduction, called *probe sets*, which is different from persistent sets and sleep sets. Rather than on concurrent processes, we rely on abstract *enabling* and *disabling* relations among actions, which we assume to be given somehow. Like persistent sets, probe sets are subsets of enabled actions satisfying particular local (in)dependence conditions. Like the existing dynamic partial order reduction techniques, probe sets are optimistic, in that they underestimate the paths that have actually to be explored to find all relevant behaviour. The technique is therefore complemented by a procedure for identifying *missed actions*.

We show that probe set reduction preserves all traces of the full transition system system modulo the permutation of independent actions. Moreover, we show that the probe set technique is capable of reducing systems in which there are no non-trivial persistent sets, and so existing techniques are bound to fail.

However, the critical part is the missed action analysis. In principle, it is possible to miss an action whose very existence is unknown. To show that the detection of such missed actions is nevertheless feasible, we further refine our setting by assuming that actions work by manipulating (reading, creating and deleting) *entities*, in a rule-based fashion. For instance, in graph transformation, the entities are graph nodes and edges. Thus, the actions are essentially rule applications. Missed actions can then be conservatively predicted by overestimating the applicable rules.

The paper is structured as follows. In Section 2, we introduce an abstract framework for enabling and disabling relations among actions in a transition system. In Section 3 we define missed actions and probe sets, give a first version of the algorithm and establish the correctness criterion. Section 4 then discusses how to identify missed actions and construct probe sets, and gives the definitive version of the algorithm. All developments are illustrated on the basis of a running example introduced in Section 2. Section 5 contains an evaluation and discussion of related and future work.

2 Enabling, Disabling and Reduction

Throughout this paper, we assume a countable universe of *actions* Act, ranged over by $a, b, \ldots$, with two binary relations, an irreflexive relation $\triangleright$ and a reflexive relation $\blacktriangleleft$:

Stimulation: $a \triangleright b$ indicates that a stimulates b;
Disabling: $a \blacktriangleleft b$ indicates that a disables b.

The intuition is that a stimulates b if the effect of a fulfills part of the precondition of b that was not fulfilled before (meaning that b cannot occur directly before a), whereas it disables b if it violates part of b's precondition (meaning that b cannot occur directly after a). If b neither is stimulated by a nor disables a then it is independent of a (meaning that it might occur concurrently with a). In the theory of *event structures* (e.g., [14]), $\triangleright$ roughly corresponds to a notion of (direct) *cause* and $\blacktriangleleft$ to *asymmetric conflict* (e.g., [8]; see also [6] for a systematic investigation of event relations).[1] Fig. 1 shows an example.

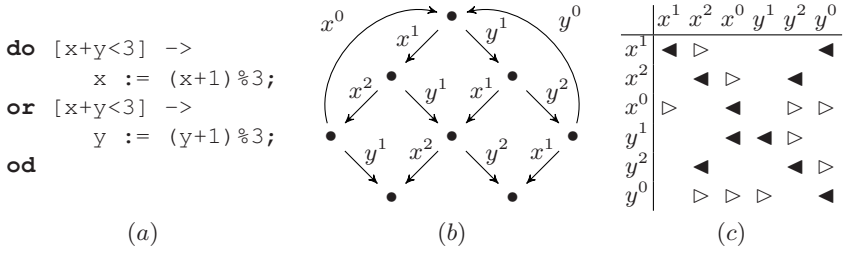


Fig. 1. A non-deterministic process (a), its transition system (b) and the stimulus and disabling relations (c). Action x^i $[y^i]$ assigns i to x $[y]$, with pre-condition $x + y < 3$.

We also use *words*, or sequences of actions, denoted $v, w \in \mathsf{Act}^*$. The empty word is denoted ε. The set of actions in w is denoted A_w. With respect to stimulation and disabling, not all words are possible computations. To make this precise, we define a derived *influence* relation over words:

$$v \rightsquigarrow w \quad :\Leftrightarrow \quad \exists a \in A_v, b \in A_w : a \triangleright b \vee a \blacktriangleright b.$$

(where $a \blacktriangleright b$ is equivalent to $b \blacktriangleleft a$.) $v \rightsquigarrow w$ is pronounced "v influences w." Influence can be positive or negative. For instance, in Fig. 1, $x^1 \cdot x^2 \rightsquigarrow y^1 \cdot y^2$ due to $x^2 \blacktriangleright y^2$ and $x^1 \cdot y^1 \rightsquigarrow x^2 \cdot y^2$ due to $x^1 \triangleright x^2$, whereas x^1 and $y^1 \cdot y^2$ do not influence one another.

Definition 1 (word feasibility). *A word w is* feasible *if*

- *for all sub-words $a \cdot v \cdot b$ of w, if $a \blacktriangleleft b$ then $\exists c \in A_v : a \triangleright c \triangleright b$;*
- *for all sub-words $v_1 \cdot v_2$ of w, $v_2 \rightsquigarrow v_1$ implies $v_1 \rightsquigarrow v_2$.*

[1] This analogy is not perfect, since in contrast to actions, events can occur only once.

The intuition is that infeasible words do not represent possible computations. For instance, if $a \blacktriangleleft b$, for the action b to occur after a, at least one action in between must have "re-enabled" b; and if v_2 occurs directly after v_1, but v_1 does *not* influence v_2, then v_2 might as well have occurred before v_1; but this also rules out that v_2 influences v_1.

For many purposes, words are interpreted up to permutation of independent actions. We define this as a binary relation over words.

Definition 2 (equality up to permutation of independent actions). $\simeq \subseteq \mathsf{Act}^* \times \mathsf{Act}^*$ *is the smallest transitive relation such that* $v{\cdot}a{\cdot}b{\cdot}w \simeq v{\cdot}b{\cdot}a{\cdot}w$ *if* $a \not\leadsto b$.

Some properties of this equivalence, such as the relation with feasibility, are expressed in the following proposition.

Proposition 3

1. *If* v *is feasible and* $v \simeq w$, *then* w *is feasible;*
2. $\simeq$ *is symmetric over the set of feasible words.*
3. $v{\cdot}w_1 \simeq v{\cdot}w_2$ *if and only if* $w_1 \simeq w_2$.

It should be noted that, over feasible words, the setup now corresponds to that of (Mazurkiewicz) *traces*, which have a long tradition; see, e.g., [1, 9]. The main difference is that our underlying notion of influence, built up as it is from stimulation and disabling, is more involved than the symmetric binary dependency relation that is commonly used in this context — hence for instance the need here to restrict to feasible words before $\simeq$ is symmetric.

We also define two prefix relations over words, the usual "hard" one ($\preceq$), which expresses that one word is equal to the first part of another, and a "weak" prefix ($\precsim$) up to permutation of independent actions. It is not difficult to see that, over feasible words, both relations are partial orders.

$$v \preceq w \;:\Leftrightarrow\; \exists u : v{\cdot}u = w \tag{1}$$

$$v \precsim w \;:\Leftrightarrow\; \exists u : v{\cdot}u \simeq w. \tag{2}$$

2.1 Transition Systems

We deal with transition systems labelled by Act. As usual, transitions are triples of source state, label and target state, denoted $q \xrightarrow{a} q'$. We use $q_0 \xrightarrow{w} q_{n+1}$ with $w = a_0 \cdots a_n$ as an abbreviation of $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_n} q_{n+1}$. Formally:

Definition 4. *A transition system is a tuple* $S = \langle Q, \rightarrow, \iota \rangle$ *such that* $\iota \in Q$ *and* $\rightarrow \subseteq Q \times \mathsf{Act} \times Q$, *with the additional constraints that for all* $q, q_1, q_2 \in Q$:

- *All traces are feasible; i.e.,* $\iota \xrightarrow{w} q$ *implies* w *is feasible;*
- *The system is deterministic up to independence; i.e.,* $q \xrightarrow{w_1} q_1$ *and* $q \xrightarrow{w_2} q_2$ *with* $w_1 \simeq w_2$ *implies* $q_1 = q_2$.
- *All out-degrees are finite; i.e.,* $enabled(q) = \{a \mid \exists q \xrightarrow{a} q'\}$ *is a finite set.*

The second condition implies (among other things) that the actions in Act are fine-grained enough to deduce the successor state of a transition entirely from its source state

and label. Although this is clearly a restriction, it can always be achieved by including enough information into the actions. Some more notation:

$$q \vdash w \;:\Leftrightarrow\; \exists q' : q \xrightarrow{w} q'$$
$$q \uparrow w \;:=\; q' \quad \text{such that } q \xrightarrow{w} q'.$$

$q \vdash w$ expresses that q *enables* w, and $q \uparrow w$ is q *after* w, i.e., the state reached from q after w has been performed. Clearly, $q \uparrow w$ is defined (uniquely, due to determinism) iff $q \vdash w$. In addition to determinism modulo independence, the notions of stimulation and disabling have more implications on the transitions of a transition system. These implications are identified in the following definition.

Definition 5 (dependency consistency and completeness). *A transition system S is called* dependency consistent *if it satisfies the following properties for all $q \in Q$:*

$$q \vdash a \;\wedge\; a \rhd b \Longrightarrow q \not\vdash b \tag{3}$$
$$q \vdash a \;\wedge\; a \blacktriangleleft b \Longrightarrow q \not\vdash a{\cdot}b. \tag{4}$$

S is called dependency complete *if it satisfies:*

$$q \vdash a{\cdot}b \;\wedge\; a \not\rhd b \Longrightarrow q \vdash b \tag{5}$$
$$q \vdash a \;\wedge\; q \vdash b \;\wedge\; a \not\blacktriangleleft b \Longrightarrow q \vdash a{\cdot}b. \tag{6}$$

Dependency consistency and completeness are illustrated in Fig. 2.

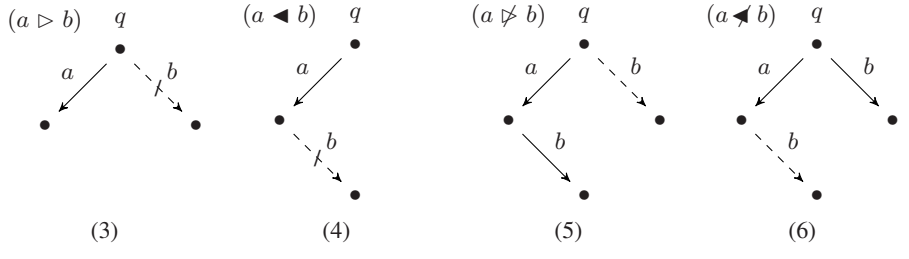


Fig. 2. The consistency and completeness properties of Def. 5. The (negated) dashed arrows are implied by the others, under the given dependency relations.

The following property states an important consequence of dependency completeness, namely that weak prefixes of traces are themselves also traces. (Note that this does not hold in general, since weak prefixes allow reshuffling of independent actions).

Proposition 6. *If S is a dependency complete transition system, then $q \vdash w$ implies $q \vdash v$ for all $q \in Q$ and $v \precsim w$.*

The aim of this paper is to *reduce* a dependency complete transition system to a smaller transition system (having fewer states and transitions), which is no longer dependency complete but from which the original transition system can be reconstructed by completing it w.r.t. (5) and (6). We now define this notion of reduction formally.

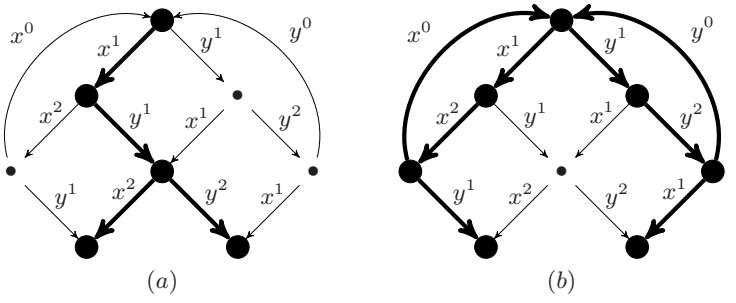


Fig. 3. An incorrect (a) and a correct (b) reduction of the transition system in Fig. 1. The fat nodes and arrows are the states and transitions of the reduced system.

Definition 7 (reduction). *Let R, S be two dependency consistent transition systems. We say that R* reduces *S if $Q_R \subseteq Q_S$, $T_R \subseteq T_S$, $\iota_R = \iota_S$, and for all $w \in \mathsf{Act}^*$*

$$\iota_S \vdash_S w \implies \exists v \in \mathsf{Act}^* : w \precsim v \ \wedge \ \iota_R \vdash_R v.$$

We will often characterise a reduced transition system only through its set of states Q_R.

For example, Fig. 3 shows two reductions of the transition system in Fig. 1, one invalid (a) and one valid (b). In (a), among others the trace $x^1 \cdot x^2 \cdot x^0$ is lost.

It follows from Proposition 6 that the reduction of a dependency complete transition system is essentially lossless: if R reduces S and S is complete, then the reachable part of S can be reconstructed from R up to isomorphism. In particular, it immediately follows that deadlock states are preserved by reduction:

Proposition 8. *If R, S are dependency consistent transition systems such that S is dependency complete and R reduces S, then for any reachable deadlock state $q \in Q_S$ (i.e., such that $\forall a \in \mathsf{Act} : q \not\vdash a$) it holds that $q \in Q_R$.*

2.2 Entity-Based System Specifications

Above we have introduced a very abstract notion of actions and dependencies. We will now show a way to instantiate this framework. In the following, Ent is a countable universe of *entities*, ranged over by $e, e_1, e', \ldots$

Definition 9. *An action a is said to be* Ent-based *if there are associated finite disjoint sets*

- $R_a \subseteq \mathsf{Ent}$*, the set of entities* read *by a;*
- $N_a \subseteq \mathsf{Ent}$*, the set of entities* forbidden *by a;*
- $D_a \subseteq \mathsf{Ent}$*, the set of entities* deleted *by a;*
- $C_a \subseteq \mathsf{Ent}$*, the set of entities* created *by a.*

The set of Ent-based actions is denoted $\mathsf{Act}[\mathsf{Ent}]$. For Ent-based actions a, b we define

$$a \rhd b :\Leftrightarrow C_a \cap (R_b \cup D_b) \neq \emptyset \vee D_a \cap (C_b \cup N_b) \neq \emptyset \tag{7}$$

$$a \lhd b :\Leftrightarrow D_a \cap (R_b \cup D_b) \neq \emptyset \vee C_a \cap (C_b \cup N_b) \neq \emptyset \tag{8}$$

Since Ent and Act may both be infinite, we have to impose some restrictions to make sure that our models are effectively computable. For this purpose we make the following important assumption:

Enabling is finite. *For every finite set $E \subseteq \mathsf{Ent}$, the set of potentialle applicable actions $\{a \in \mathsf{Act} \mid R_a \cup D_a \subseteq E\}$ is finite.*

A transition system S is called Ent-based if $A \subseteq \mathsf{Act}[\mathsf{Ent}]$ and for every $q \in Q$ there is an associated finite set $E_q \subseteq \mathsf{Ent}$, such that $E_q = E_{q'}$ implies $q = q'$.

Definition 10 (Entity-based transition systems). *A transition system S is called* Ent*-based if all transitions are labelled by* Ent*-based actions, and for all $q \in Q$:*

- *There is a finite set $E_q \subseteq \mathsf{Ent}$, such that $E_q = E_{q'}$ implies $q = q'$;*
- *For all $a \in \mathsf{Act}[\mathsf{Ent}]$, $q \vdash a$ iff $(R_a \cup D_a) \subseteq E_q$ and $(N_a \cup C_a) \cap E_q = \emptyset$;*
- *For all $a \in enabled(q)$, $q \uparrow a$ is determined by $E_{q\uparrow a} = (E_q \setminus D_a) \cup C_a$.*

It can be shown that these three conditions on the associated events, together with the assumption that enabling is computable, actually imply feasibility, determinism and finite out-degrees. The following (relatively straightforward) proposition states that this setup guarantees some further nice properties.

Proposition 11. *Every* Ent*-based transition system is dependency complete and consistent, and has only feasible words as traces.*

Models whose behaviour can be captured by entity-based transition systems include: Turing machines (the entities are symbols at positions of the tape), Petri nets (the entities are tokens), term and graph rewrite systems (the entities are suitably represented subterms and graph elements, respectively). Computability of enabling is guaranteed by the *rule-based* nature of these models: all of them proceed by attempting to instantiate a finite set of rules on the given finite set of entities, and this always results in a finite, computable set of rule applications, which constitute the actions.

For instance, the transition system of Fig. 1 is obtained (*ad hoc*) by using entities $e_{x>0}$, $e_{x>1}$, $e_{y>0}$ and $e_{y>1}$, setting $E_\iota = \emptyset$ and defining the actions as follows:

a	R_a	N_a	D_a	C_a
x^1				$e_{x>0}$
x^2	$e_{x>0}$	$e_{y>0}$		$e_{x>1}$
x^0		$e_{y>1}$	$e_{x>0}, e_{x>1}$	

a	R_a	N_a	D_a	C_a
y^1				$e_{y>0}$
y^2	$e_{y>0}$	$e_{x>0}$		$e_{y>1}$
y^0		$e_{x>1}$	$e_{y>0}, e_{y>1}$	

(9)

3 Missed Actions and Probe Sets

All *static* partial order reduction algorithms explore subsets of enabled transitions in such a way that they guarantee *a priori* not to rule out any relevant execution path of the system. *Dynamic* partial order reduction algorithms, such as e.g. [2], on the other hand, potentially "miss" certain relevant execution paths. These missed paths then have to be added at a later stage. The resulting reduction may be more effective, but additional resources (i.e. time and memory) are needed for the analysis of missed execution paths.

For instance, in the reduced system of Fig. 3(a), the transitions are chosen such that all actions that run the danger of becoming disabled are explored. Nevertheless, actions x^0 and y^0 are missed because they have never become enabled.

Our dynamic partial order reduction algorithm selects the transitions to be explored on the basis of so-called *probe sets*. We will now introduce the necessary concepts.

3.1 Missed Actions

We define a *vector* in a transition system as a tuple consisting of a state and a trace leaving that state. Vectors are used especially to characterise their *target* states, in such a way that not only the target state itself is uniquely identified (because of the determinism of the transition system) but also the causal history leading up to that state.

Definition 12 (vector). *A vector (q, w) of a transition system S consists of a state $q \in Q$ and a word w such that $q \vdash w$.*

Missed actions are actions that would have become enabled along an explored execution path if the actions in the path had been explored in a different order. To formalise this, we define the (weak) *difference* between words, which is the word that has to be concatenated to one to get the other (modulo independence), as well as the *prime cause* within w of a given action a, denoted $\downarrow_a w$, which is the smallest weak prefix of w that includes all actions that influence a, directly or indirectly:

$$w - v := u \text{ such that } v{\cdot}u \simeq w$$
$$\downarrow_a w := v \text{ such that } w - v \not\rightarrow a \ \wedge\ \forall v' \precsim w : (w - v' \not\rightarrow a \ \Rightarrow\ v \precsim v').$$

Clearly, $w - v$ exists if and only if $v \precsim w$; in fact, as a consequence of Proposition 3.3, it is then uniquely defined up to $\simeq$. The prime cause $\downarrow_a w$, on the other hand, is always defined; the definition itself ensures that it is unique up to $\simeq$. A representative of $\downarrow_a w$ can in fact easily be constructed from w by removing all actions, starting from the tail and working towards the front, that do not influence either a or any of the actions *not* removed. For instance, in Fig. 1 we have $x^1{\cdot}y^1{\cdot}y^2 - y^1 = x^1{\cdot}y^2$ whereas $x^1{\cdot}y^1{\cdot}y^2 - y^2$ is undefined; furthermore, $\downarrow_{y^2} x^1{\cdot}y^1 = y^1$.

Definition 13 (missed action). *Let (q, w) be a vector. We say that an action a is* missed along (q, w) *if $q \not\vdash w{\cdot}a$ but $q \vdash v{\cdot}a$ for some $v \precsim w$. The missed action is characterised by $\downarrow_a v{\cdot}a$ rather than just a; i.e., we include the prime cause. A missed action is said to be* fresh *in (q, w) if $w = w'{\cdot}b$ and a is not a missed action in (q, w').*

The set of fresh missed actions along (q, w) is denoted $fma(q, w)$. It is not difficult to see that $v{\cdot}a \in fma(q, w)$ implies $w = w'{\cdot}b$ such that $b \rightsquigarrow a$; otherwise a would already have been missed in (q, w').

A typical example of a missed action is $(y^1{\cdot}y^2) \in fma(\iota, x^1{\cdot}x^2{\cdot}y^1)$ in Fig. 1: here $\iota \not\vdash x^1{\cdot}x^2{\cdot}y^1{\cdot}y^2$ but $\iota \vdash y^1{\cdot}y^2$ with $y^1 \precsim x^1{\cdot}x^2{\cdot}y^1$. Note that indeed $y^1 \rhd y^2$.

3.2 Probe Sets

The most important parameter of any partial order reduction is the selection of a (proper) subset of enabled actions to be explored. For this purpose, we define so-called *probe*

Algorithm 1. Probe set based partial order reduction, first version

```
1: let Q ← ∅;  // result set of states, initialised to the empty set
2: let C ← {(ι_S, ε)};  // set of continuations, initialised to the start state
3: while C ≠ ∅ do  // continue until there is nothing left to do
4:    choose (q, w) ∈ C;  // arbitrary choice of next continuation
5:    let C ← C \ {(q, w)};
6:    if q↑w ∉ S then  // test if we saw this state before
7:       let Q ← Q ∪ {q↑w};  // if not, add it to the result set
8:       for all v·m ∈ fma(q, w) do  // identify the fresh missed actions
9:          let Q ← Q ∪ {q↑v' | v' ⪯ v};  // add intermediate states
10:         let C ← C ∪ {(q↑v·m, ε)};  // add a continuation
11:      end for
12:      choose p ∈ P_{q,w};  // choose a probe set for this continuation
13:      let C ← C ∪ {(q↑p(a), w·a − p(a)) | a ∈ dom(p)};  // add next continuations
14:   end if
15: end while
```

sets, based on the disabling among the actions enabled at a certain state (given as the target state $q \uparrow w$ of a vector (q, w)). Furthermore, with every action in a probe set, we associate a part of the causal history that can be discharged when exploring that action. (Thus, our probe sets are actually partial functions).

Definition 14 (probe set). *For a given vector (q, w), a* probe set *is a partial function p: $enabled(q \uparrow w) \rightharpoonup \mathsf{Act}^*$ mapping actions enabled in $q \uparrow w$ onto words, such that the following conditions hold:*

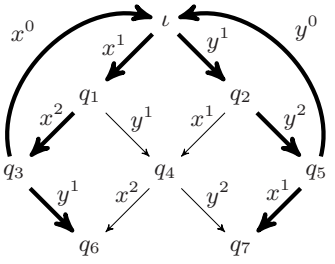
1. *For all $a \in dom(p)$ and $b \in enabled(q \uparrow w)$, $b \blacktriangleleft a$ implies $b \in dom(p)$;*
2. *For all $a \in dom(p)$ and $b \in enabled(q \uparrow w)$, $p(a) \not\precsim \downarrow_b w$ implies $b \in dom(p)$;*
3. *For all $a \in dom(p)$, $p(a) \precsim \downarrow_a w$.*

We use $\mathcal{P}_{q,w}$ to denote the set of all probe sets for a vector (q, w). We say that an action a is *in* the probe set p if $a \in dom(p)$. The first condition states that probe sets are closed under inverse disabling. The second and third conditions govern the discharge of the causal history: the fragment that can be discharged must be contained in the prime cause of any action not in the probe set (Clause 2) and of a itself (Clause 3). Of these, Clause 2 is the most involved: if we discharge any action that does not contribute to (the cause of) some b, then we must probe b as well, so that missed actions stimulated by b can still be identified. Section 4.3 gives some guidelines on how to select probe sets.

Algorithm 1. gives a first version of the reduction algorithm. In Fig. 4 we apply this algorithm to the example system of Fig. 1, obtaining the reduced system in Fig. 3(b). The first column shows the value of C at the beginning of the loop; the second column represents the choice of continuation; the third is the resulting set of fresh missed actions; the fourth column gives the increase in the result set Q; and the final column shows the choice of probe set.

3.3 Correctness

In order to have a correct reduction of a transition system, we must select sufficiently many probe sets and take care of the missing actions. Let us define this formally.



	x^1	x^2	x^0	y^1	y^2	y^0
x^1	◀	▷				◀
x^2		◀	▷		◀	
x^0	▷		◀		▷	▷
y^1			◀	◀	▷	
y^2		◀			◀	▷
y^0		▷	▷	▷		◀

iteration	C	$(q,v) \in C$	$fma(q,v)$	ΔQ	$p \in \mathcal{P}_{q,v}$
1	(ι, ε)			ι	(x^1, ε)
2	(ι, x^1)	(ι, x^1)		q_1	(x^2, ε)
3	$(\iota, x^1{\cdot}x^2)$	$(\iota, x^1{\cdot}x^2)$		q_3	$(x^0, x^1{\cdot}x^2), (y^1, \varepsilon)$
4	$(q_3, x^0), (\iota, x^1{\cdot}x^2{\cdot}y^1)$	(q_3, x^0)			
5	$(\iota, x^1{\cdot}x^2{\cdot}y^1)$	$(\iota, x^1{\cdot}x^2{\cdot}y^1)$	$y^1{\cdot}y^2$	q_6, q_2	
6	(q_5, ε)	(q_5, ε)		q_5	$(x^1, \varepsilon), (y^0, \varepsilon)$
7	$(q_5, x^1), (q_5, y^0)$	(q_5, x^1)		q_7	
8	(q_5, y^0)	(q_5, y^0)			

Fig. 4. Step-by-step execution of Algorithm 1. on the example of Fig. 1

Definition 15 (probing). *Let S be a transition system. A* probing *for S is a K-indexed set $P = \{p_{q,w}\}_{(q,w)\in K}$ where*

1. *K is a set of vectors of S such that $(\iota, \varepsilon) \in K$;*
2. *For all $(q,w) \in K$, $p_{q,w}$ is a probe set such that $(q \uparrow p_{q,w}(a), w{\cdot}a - p_{q,w}(a)) \in K$ for all $a \in dom(p_{q,w})$.*
3. *for all $(q,w) \in K$ and all $v{\cdot}a \in fma(q,w)$, there is a word $u \precsim w - v$ such that $(q \uparrow v{\cdot}a, u) \in K$ and $u \not\rightsquigarrow a$.*

We write $(q,w) \uparrow p(a)$ for the vector $(q \uparrow p_{q,w}(a), w{\cdot}a - p_{q,w}(a))$ in Clause 2. P is called fair *if for all $(q,w) \in K$ there is a function $n_{q,w}$: $enabled(q \uparrow w) \to \mathbb{N}$, assigning a natural number to all actions enabled in $q \uparrow w$, such that for all $a \in enabled(q \uparrow w)$, either $a \in dom(p_{q,w})$, or $n_{(q,w)\uparrow p(b)}(a) < n_{q,w}(a)$ for some $b \in dom(p)$.*

Clause 2 guarantees that, from a given probe set, all regular explored successors (of actions in the probe set) are indeed also probed; Clause 3 takes care of the missed actions. Fairness ensures that every enabled action will eventually be included in a probe set. In Section 4.3 we will show how to guarantee fairness.

The following is the core result of this paper, on which the correctness of the algorithm depends. It states that every fair probing gives rise to a correct reduction. The proof can be found in the appendix.

Theorem 16. *If P is a fair probing of a transition system S, then the transition system R characterized by $Q_R = \{q \uparrow w \mid (q,w) \in dom(P)\}$ reduces S.*

If we investigate Algorithm 1. in this light, it becomes clear that this is not yet correct. The total collection of vectors and probe sets produced by the algorithm give

rise to a correct probing in the sense of Def. 15 (where the u of Clause 3 is always set to ε), and also generates a probing; however, this probing is not fair. As a result, Algorithm 1. suffers from the so-called "ignoring problem" well-known from other partial order reductions.

4 The Algorithm

In this section, we put the finishing touch on the algorithm: ensuring fairness, identifying missed actions, and constructing probe sets. For this, we take the entity-based setting from Section 2.2.

4.1 Identifying Missed Actions

As we have discussed in Section 3, finding the missed actions $fma(q, v)$ by investigating all weak prefixes of v negates the benefits of the partial order reduction. In the the entity-based setting of Section 2.2, however, a more efficient way of identifying missed actions can be defined on the basis of an *over-approximation*. We define the over-approximation of the target state of a vector (q, w), denoted $q{\Uparrow}w$, as the union of all entities that have appeared along that vector, and the *weak enabling* of an action a by a set of entities E, denoted $E \Vdash a$, by only checking for the presence of read and deleted entities and not the absence of forbidden and created entities.

$$q{\Uparrow}w \;:=\; E_q \cup \bigcup\nolimits_{a \in A_w} C_a$$
$$E \Vdash a \;:\Leftrightarrow\; (R_a \cup D_a) \subseteq E$$

This gives rise to the set of *potentially missed actions*, which is a superset of the set of fresh missed actions.

Definition 17 (potentially missed actions). *Let $(q, w{\cdot}b)$ be a vector. Then, $a \in \mathsf{Act}$ is a* potentially missed action *if either $b \blacktriangleleft a$, or the following conditions hold:*

1. *a is weakly but not strongly enabled: $q{\Uparrow}w \Vdash a$ and $q{\uparrow}w \nvdash a$,*
2. *a was somewhere disabled: $\exists c \in A_w : c \blacktriangleleft a$;*
3. *a is freshly enabled: $b \rhd a$.*

We will use $pma(q, v)$ to denote the set of potentially missed actions in the vector (q, v). It is not difficult to see that $pma(q, v) \supseteq fma(q, v)$ for arbitrary vectors (q, v). However, even for a given $a \in pma(q, v)$ it is not trivial to establish whether it is really missed, since this still involves checking if there exists some $v' \precsim v$ with $q{\uparrow}v' \vdash a$, and we have little prior information about v'. In particular, it might be that v' is smaller than the prime cause $\downarrow_a v$. For instance, if $E_q = \{1\}$, $C_b = \{2\}$, $D_c = \{1, 2\}$ and $R_a = \{1, 2\}$ then $q \nvdash v{\cdot}a$ with $v = b{\cdot}c{\cdot}b$, and $\downarrow_a v = v$; nevertheless, there is a prefix $v' \precsim v$ such that $q \vdash v'{\cdot}a$, viz. $v' = b$.

In some cases, however, the latter question is much easier to answer; namely, if the prime cause $\downarrow_a v$ is the only possible candidate for such a v'. The prime cause can be computed efficiently by traversing backwards over v and removing all actions not (transitively) influencing a.

Definition 18 (reversing actions). *Two entity-based actions a, b are* reversing *if $C_a \cap D_b \neq \emptyset$ or $D_a \cap C_b \neq \emptyset$. A word w is said to be* reversing free *if no two actions $a, b \in A_w$ are reversing.*

We also use $rev_a(w) = \{b \in A_w \mid a, b \text{ are reversing}\}$ to denote the set of actions in a word w that are reversing with respect to a. Reversing freedom means that no action (partially) undoes the effect of another. For instance, in the example above b and c are reversing due to $C_b \cap D_c = \{1\}$, so v is not reversing free. The following result now states that for reversing free vectors, we can efficiently determine the fresh missed actions.

Proposition 19. *Let (q, v) is a vector with v reversing free.*

1. *For any action a, $q \vdash v' \cdot a$ with $v' \precsim v$ implies $v' = \downarrow_a v$.*
2. *$fma(q, v) = \{a \in pma(q, v) \mid q \vdash \downarrow_a v \cdot a\}$.*

4.2 Ensuring Fairness

To ensure that the probing we construct is fair, we will keep track of the "age" of the enabled actions. That is, if an action is *not* probed, its age will increase in the next round, and probe sets are required to include at least one action whose age is maximal. This is captured by a partial function $\alpha \colon \mathsf{Act} \rightharpoonup \mathbb{N}$. To manipulate these, we define

$$\begin{aligned} \alpha \oplus A &:= \{(a, \alpha(a) + 1) \mid a \in dom(\alpha)\} \cup \{(a, 0) \mid a \in A \setminus dom(\alpha)\} \\ \alpha \ominus A &:= \{(a, \alpha(a)) \mid a \notin A\} \\ \max \alpha &:= \{a \in dom(\alpha) \mid \forall b \in dom(\alpha) : \alpha(a) \geq \alpha(b)\} \\ A \text{ satisfies } \alpha &:\Leftrightarrow \alpha = \emptyset \text{ or } A \cap \max(\alpha) \neq \emptyset. \end{aligned}$$

So, $\alpha \oplus A$ initialises the age of the actions in A to zero, and increases all other ages; $\alpha \ominus A$ removes the actions in A from α; $\max \alpha$ is the set of oldest actions; and A satisfies the fairness criterion if it contains at least one oldest action, or α is empty.

4.3 Constructing Probe Sets

When constructing probe sets, there is a trade-off between the size of the probe set and the length of the vectors. On the one hand, we aim at minimising the size of the probe sets; on the other hand, we also want to minimise the size of the causal history. For example, probe sets consisting of pairs (a, ε) only (for which the second condition of Def. 14 is fulfilled vacuously, and the third trivially) are typically small, but then no causal history can be discharged. Another extreme case is when a probe set consists of pairs $(a, \downarrow_a w)$. In this case, the maximal amount of causal history is discharged that is still consistent with the third condition of Def. 14, but the probe set domain is likely to equal the set of enabled actions, resulting in no reduction at all.

The probe sets $p_{q,w}$ we construct will furthermore ensure that the vectors of the new continuation points are reversing free. Therefore, for every $p_{q,w}$ we additionally require that for all $a \in dom(p_{q,w}) : rev_a(w) \subseteq A_{p(a)}$. Since $rev_a(w) \subseteq A_{\downarrow_a w}$, this does not conflict with Def. 14.

Algorithm 2. Probe set based partial order reduction algorithm, definitive version.

let $Q \leftarrow \emptyset$;
let $C \leftarrow \{(\iota_S, \varepsilon, \emptyset)\}$; *// age function initially empty*
while $C \neq \emptyset$ **do**
  choose $(q, w, \alpha) \in C$;
  let $C \leftarrow C \setminus \{(q, w, \alpha))\}$;
  if $q \uparrow w \notin S$ **then**
    let $Q \leftarrow Q \cup \{q \uparrow w\}$;
    for all $v{\cdot}m \in fma(q, w)$ **do** *// calculated according to Proposition 19.2*
      let $Q \leftarrow Q \cup \{q \uparrow v' \mid v' \preceq v\}$;
      let $C \leftarrow C \cup \{(q \uparrow v{\cdot}m, \varepsilon, \emptyset)\}$;
    end for
    choose $p \in \mathcal{P}_{q,w}$ such that $dom(p)$ satisfies α, and $\forall a \in dom(p) : rev_a(w) \subseteq A_{p(a)}$;
    // choose a fair probe set, and ensure reversing freedom
    let $\alpha \leftarrow \alpha \oplus enabled(q \uparrow w) \ominus dom(p)$; *// update the age function*
    let $C \leftarrow C \cup \{(q \uparrow p(a), w{\cdot}a - p(a), \alpha) \mid a \in dom(p)\}$;
  end if
end while

An interesting probe set $p_{q,w}$ could be constructed such that $p_{q,w}$ satisfies the condition on disabling actions and furthermore $p_{q,w}(a) = \downarrow_a w$ except for one action, say a', which is mapped to the empty vector, i.e. $p_{q,w}(a') = \varepsilon$. This action a' then ensures that no further action needs to be included in the probe set. The selection of this action a' can be based on the length of its prime cause within w.

There is a wide range of similar heuristics that use different criteria for selecting the first action from which to construct the probe set or for extending the causal history to be discharged. Depending on the nature of the transition system to be reduced, specific heuristics might result in more reduction. This is a matter of future experimentation.

Algorithm 2. now shows the definitive version of the algorithm. The differences with the original version are commented. Correctness is proved using Theorem 16. The proof relies on the fact that the algorithm produces a fair probing, in the sense of Def. 15.

Theorem 20. *For a transition system S, Algorithm 2. produces a set of states $Q \subseteq Q_S$ characterising a reduction of S.*

For our running example of Figs. 1 and 4, there are several observations to be made.

- The probe sets we constructed in Fig. 4 (on an ad hoc basis) are reversing free. Note that (in terms of (9)) x^0 reverses x^1 and x^2; likewise, y^0 reverses y^1 and y^2.
- The run in Fig. 4 is *not* fair: after the first step, the age of y^1 becomes 1 and hence y^1 should be chosen rather than x^2. This suggests that our method of enforcing fairness is too rigid, since the ignoring problem does not actually occur here.

5 Conclusion

Summary. We have proposed a new algorithm for dynamic partial order reduction with the following features:

- It can reduce systems that have no non-trivial persistent sets (and so traditional methods do not have an effect).
- It is based on abstract enabling and disabling relations, rather than on concurrent processes. This makes it suitable for, e.g., graph transformation systems.
- It uses a universe of actions that does not need to be finite or completely known from the beginning; rather, by adopting an entity-based model, enabled and missed actions can be computed on the fly. This makes it suitable for dynamic systems, such as software.
- It can deal with cyclic state spaces.

We have proved the algorithm correct (in a rather strong sense) and shown it on a small running example. However, an implementation is as yet missing.

Related Work. Traditional partial order reduction (see e.g. [3, 12]) is based on statically determined dependency relations, e.g. for constructing *persistent sets*. More recently, dynamic partial order reduction techniques have been developed that compute dependency relations on-the-fly. In [2], for example, partial order reduction is achieved by computing persistent sets dynamically. This technique performs a stateless search, which is the key problem of applying it to cyclic state spaces. In [5], Gueta et al. introduce a *Cartesian* partial order reduction algorithm which is based on reducing the number of context switches and is shown also to work in the presence of cycles. Both approaches are based on processes or threads performing read and/or write operations on local and/or shared variables. The setting we propose is more general in the sense that actions are able to create or delete entities that can be used as communication objects. Therefore, our algorithm is better suited for systems in which resources are dynamically created or destroyed without an a priori bound.

Future Work. As yet, there is no implementation of probe sets. Now that the theoretical correctness of the approach is settled, the first step is to implement it and perform experiments. We plan to integrate the algorithm in the Groove tool set [10], which will then enable partial order reduction in the context of graph transformations. The actual reduction results need to be compared with other algorithms, by performing some benchmarks; see, e.g., [5].

In the course of experimentation, there are several parameters by which to tune the method. One of them is obviously the choice of probe sets; a discussion of the possible variation points was already given in Section 4. However, the main issue, which will eventually determine the success of the method, is the cost of backtracking necessary for repairing missed actions, in combination with the precision of our (over-)estimation of those missed actions. If the over-estimation is much too large, then the effect of the partial order reduction may be effectively negated.

To improve this precision, analogous to the over-approximation of an exploration path, an under-approximation can be used for decreasing the number of potentially missed actions. Actions that create or forbid entities that are in the under-approximation can never be missed actions and do not have to be considered. Essentially, also including this under-approximation means we are introducing a three-valued logic for determining the presence of entities.

Other issues to be investigated are the effect of heuristics such as discussed in Section 4.3, alternative ways to ensure fairness, and also the combination of our algorithm with the *sleep set* technique [3].

Acknowledgment. We want to thank Wouter Kuijper for contributing to this work in its early stages, through many discussions and by providing a useful motivating example.

References

1. Diekert, V., Rozenberg, G.: The Book of Traces. World Scientific Publishing Co., Inc., Singapore (1995)
2. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proc. of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005), pp. 110–121. ACM Press, New York (2005)
3. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
4. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. Formal Methods in System Design 2(2), 149–164 (1993)
5. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bosnacki, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 5495, pp. 95–112. Springer, Heidelberg (2007)
6. Janicki, R., Koutny, M.: Structure of concurrency. Theor. Comput. Sci. 112(1), 5–52 (1993)
7. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 299–305. Springer, Heidelberg (2006)
8. Langerak, R.: Transformations and Semantics for LOTOS. PhD thesis, University of Twente (1992)
9. Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1987)
10. Rensink, A.: The GROOVE Simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
11. Rensink, A.: Explicit state model checking for graph grammars. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Festschrift for Ugo Montanari. LNCS, vol. 5065. Springer, Heidelberg (2008)
12. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)
13. Valmari, A.: On-the-fly verification with stubborn sets. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 397–408. Springer, Heidelberg (1993)
14. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)