The Implementation of Mazurkiewicz Traces in POEM

Peter Niebert and Hongyang Qu

Laboratoire d'Informatique Fondamentale de Marseille
CMI, 39, rue Joliot Curie, 13453 Marseille Cedex 13, France
{niebert,hongyang}@cmi.univ-mrs.fr

Abstract. We present the implementation of the trace theory in a new model checking tool framework, POEM, that has a strong emphasis on Partial Order Methods. A tree structure is used to store trace systems, which allows sharing common prefixes among traces and therefore, reduces memory cost. This structure is easy to extend to incorporate additional features. Two applications are shown in the paper: An extended structure to support an adequate order for Local First Search, and an acceleration of event zone based state space search for timed automata.

1 Introduction

POEM (Partial Order Environment of Marseille) is a new model checking tool (framework) that has a strong emphasis on Partial Order Methods [17, 8, 15, 9, 12, 16, 7, 13, 11]. The motivation for adding POEM to the world of model checkers is based on the authors work on algorithms that have a common basis concerning concurrency, but which are not reflected in a single existing tool. Moreover, by allowing commonly used specification languages as input languages and allowing decent connections to analysis backends, we aim to build a platform that allows direct comparisons of different algorithms on the same model.

The purpose of POEM is to allow the application of certain partial order oriented algorithms to a number of input languages with different sets of features, while allowing at the same time basic analysis algorithms. This gives a basic structure of POEM derived tools as "compilers" consisting of a *frontend* (syntactic and semantic analysis), a *middle* (model transformation), and a *backend* which passes the model to the aimed analysis algorithm and allows to interpret results. For instance, "if2c" consists of a frontend for (a variant of) Verimag's IF2.0 [3] language, static analysis for identifying the transitions and dependency, and finally a backend generating C-code for exploration. This kind of architecture is frequently used in model checkers and originally introduced in Spin [10]. The implementation language of POEM is Objective Caml (OCaml). This choice is due to the advantages of functional programming languages for compiler writing, the efficiency of OCaml and the availability of non-functional features.

We also intend to build POEM to be a common framework for several input languages and analysis methods:

– On the specification language side, we consider modeling languages for discrete concurrent models, such as Promela [10], Petri net based languages [?] but also timed automata specifications, in particular IF2.0 and UppAal [1]. These together with, for now, safety specifications or simply reachability.
– On the analysis side, we consider state exploration with partial order based reduction methods and symbolic state exploration for timed automata.

The goal driving the design of POEM is to have as much reuse of code as possible given these different front ends and backends. It is achieved in the following ways:

1. Given that most of the mentioned specification languages use some kind of interleaving model of automata with shared variables and certain kinds of communication, POEM uses a common data structure as an abstract specification language (it does not have a concrete syntax) that allows comparatively easy translation of specification languages into a unique metalanguage.
2. Since many partial order methods are based on Mazurkiewicz trace theory [4, Chapter 2], such as [15, ?,2, 11], a well designed and implemented trace structure can be reused by such methods. and thus, save time in software development.

This paper focuses on the design and the implementation details of traces in POEM, and presents how to extend it to implement Local First Search [2, 13]. Moreover, We show the improvement of time consuming of Event Zone approach [11] by combining it with the trace structure.

This paper is structured as follows. Section 2 gives the introduction of the trace theory. Section 3 explains the design and the implementation of basic trace structure in POEM. Section 4 describes the theory and the implementation of the extension to the structure for implementing Local First Search. The combination of event zone approcah and the trace structure, as well as the experiment to show effect of improvement, is presented in Section 5. We conclude the paper in Section 6.

2 Basic trace theory

Let Σ be an alphabet, $(\Sigma^*, \circ)$ the free monoid. We write letters $a, b, c \in \Sigma$, and words $u, v, w, \ldots \in \Sigma^*$. The concatenation of a word u and a letter a is denoted by $u \circ a$. Let $I \subseteq \Sigma \times \Sigma$ be an irreflexive and symmetric independence relation, and $D = \Sigma \times \Sigma - I$. For two words $u, v \in \Sigma^*$, write $u \equiv_I v$ if there exist words w_1, w_2 and letters a, b such that $(a, b) \in I$, $u = w_1abw_2$ and $v = w_1baw_2$, i.e. if u is obtained from v by exchanging the order of two adjacent independent letters. Let $\equiv$ be the reflexive and transitive closure of the relation $\equiv_I$. We say that u and v are *trace equivalent* [4, Chapter 2] over (Σ, I) if $u \equiv v$. That is, u is trace equivalent to v if u can be obtained from v by repeatedly commuting adjacent independent letters. $\equiv$ is a congruence with respect to concatenation and we call the quotient monoid $\Sigma/\equiv$ the trace monoid of (Σ, I). We write $[u] = \{v \mid u \equiv v\}$ for the equivalence classes and for the traces.

Let $<_{alph}$ be the lexicographical order defined over Σ. For any two different letters $a, b \in \Sigma$, either $a <_{alph} b$ or $b <_{alph} a$. We also extend $<_{alph}$ for words, i.e. for two words u and v, $u <_{alph} v$ iff $u = wau'$ and $v = wbv'$ and $a <_{alph} b$. Thus, a lexicographically least representive t of a trace $[u]$ is defined as follows:

$$t \in [u] \text{ and for any word } v \in [u] \text{ with } t \neq v, t <_{alph} v.$$

For a trace u, we concider occurrences of letters such that $u = a_1 a_2 \dots a_n$. Let $E = \{a_1, a_2, \dots, a_n\}$ be the set of occurrences of letters in u, and λ the function mapping occurrences to letters. Let $(E, \prec, \lambda)$ be a finite (Σ-labeled) partial order such that for any two occurrences $a_i, a_j \in E$ $(i < j)$, $a_i \prec a_j$ iff

– either $\lambda(a_i)\ D\ \lambda(a_j)$ or
– there exists a sequence $a_{k_1} \dots a_{k_m}$ with $i < k_1 < \dots < k_m < j$ and $(\lambda(a_i)\ D\ \lambda(a_{k_1})) \wedge (\lambda(a_{k_1})\ D\ \lambda(a_{k_2})) \wedge \dots \wedge (\lambda(a_{k_m})\ D\ \lambda(a_j))$.

Therefore, traces can be viewed as partial orders based on the one-to-one correspondence [4, Chapter 2]. An element $e \in E$ is *maximal* if there is no $f \in E$ such that $e \prec f$.

3 The basic data structure for trace systems

We aimed to design a data structure that uses memory as small as possible. Prefix sharing is a key idea to reduce the memory cost, i.e. for any two traces that have a common prefix w, only one copy of w is allowed to be allocated in memory. Therefore, it is natural to choose to a tree structure to build a trace system. Any node in a tree, except the *root*, is associated with an action and has a predecessor (father). A node may or may not have successors (children). A path in a tree starting at the root, to the node associated with action a_1, then to the node a_2, until the node a_n, represents the trace $a_1 a_2 \dots a_n$.

Figure 1 illustrates the tree structure. A node has three fields: "predecessor" is a pointer to its predecessor, "lastaction" is the associated action, and "children" is a pointer to a link list such that each element in the list has two fields: "first" points to a child node and "rest" points to the next element. This structure is easy to be extended to facilitate complex trace systems by adding more fields into a node. We will see in Section 4 a kind of extension.

In this trace structure, a trace $t = a_1 a_2 \dots a_n$ is accessed through its last node a_n and following the predecessor pointer of each node of the trace. A trace system is generated from an initial trace t_0, which includes only the root node — an artificial node respresenting an empty trace, by extending t_0 one action after another. Algorithm 1 describes the general steps to extend a trace t by an action a. In the algorithm, a stack S and three stack functions are used: $POP(S)$ gets rid of the top element of S; $TOP(S)$ accesses the top element; $PUSH(a, S)$ puts the action a onto the top of S. In order to generating a lexicographical least trace $t' = t \circ a$ where $t = a_1 \dots a_n$, the extension is done in three steps in the algorithm:

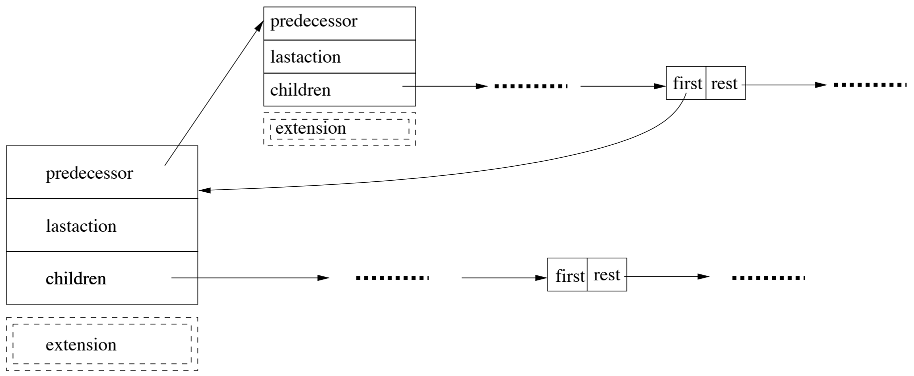

Fig. 1. The basic data structure

1. Find an i $(0 \leq i \leq n)$ such that $a_i \; D \; a$ and for any $i < j \leq n$, $\neg(a_j \; D \; a)$. Note that $i = 0$ means all actions in t are independent of a and $i = n$ means they are dependent on a.
2. Find a k $(k \geq i)$ such that for any $i < j < k$, $a_j \leq a$ and $a_k > a$.
3. Insert a between a_{k-1} and a_k. Moreover, the actions $a_k, \ldots, a_n$ are inserted into the trace again as $a_{k+1}, \ldots, a_{n+1}$.

Note that in Algorithm 1, a variable t represents both a trace conceptually and its last node when we access the trace.

Algorithm 1 Extending a trace t by an action a using a stack S

```
1: $et \leftarrow t$, $pos \leftarrow 0$, $S \leftarrow$ empty
2: while $t \neq root$ and $\neg(t.lastaction \; D \; a)$ do
3:     $pos \leftarrow pos + 1$, $PUSH(t.lastaction, S)$, $t \leftarrow t.predecessor$
4: end while
5: while $pos > 0$ and $a < TOP(S)$ do
6:     $POP(S)$, $pos \leftarrow pos - 1$
7: end while
8: for all $i$ such that $0 \leq i \leq pos$ do
9:     $et \leftarrow et.predecessor$
10: end for
11: $PATH\_SUCCESSOR(et, a)$
12: while $S$ is not empty do
13:     $eptr \leftarrow TOP(S)$, $POP(S)$, $et \leftarrow PATH\_SUCCESSOR(et, eptr)$
14: end while
15: return et
```

The function $PATH\_SUCCESSOR$ in Algorithm 2 inserts a node with a given action into its father's children list. The list is sorted in the ascending order on children's associated actions. When there is a node in the list that has been associated with the action already, this node is returned by the function. Otherwise, a new node is created, inserted into the list and returned. In this way, prefix sharing is maintained.

Algorithm 2 Function $PATH\_SUCCESSOR(t : trace, a : action)$

```
1:  $tl \leftarrow t.children$, $previous \leftarrow NULL$
2:  while $tl \neq NULL$ and $tl.first.lastaction < a$ do
3:      $previous \leftarrow tl$, $tl \leftarrow tl.rest$
4:  end while
5:  if $tl \neq NULL$ and $tl.first.lastaction = a$ then
6:      return $tl.first$
7:  end if
8:  $new\_t.predecessor \leftarrow t$, $new\_t.lastaction \leftarrow a$, $new\_t.children \leftarrow NULL$
9:  $new\_child.rest \leftarrow tl$, $new\_child- > first = new\_t$
10: if $previous \neq NULL$ then
11:     $previous.rest \leftarrow new\_child$
12: else
13:     $t.children \leftarrow new\_child$
14: end if
15: return $new\_t$
```

COMPLEXITY ANALYSIS?

Proposition 1. *Algorithm 1 and 2 1) maintain the lexicographical least represention for every feasible trace and 2) preserve prefix sharing.*

Proof. 1)
2) □

4 Extending trace systems for Local First Search

Local First Search (LFS) [2, 13] is a partial order method to seach for *local properties*. For a property φ, a *visible* action causes the system to move from a state not satisfying φ to a state satisfying it, or vice versa. When all visible actions are pairwise dependent, such a property is a local property. In [13], it is shown that *prime traces*, i.e. traces with a single maximal element, suffice to search for local properties; in turn, to approximate all prime traces, it suffices to consider only traces with a logarithmic number of maximal elements (compared to the overall parallelism in the system); this number is called *LFS*-bound.

LFS uses a breadth-first search algorithm, which is described as follows. Consider a state in the search queue is explored with an enabled action in this state. Let t be the trace leading to the state and a the action. If the number of maximal elements of $t \circ a$ succeed the LFS-bound, then the trace $t \circ a$ is abandoned; else, a state s reached by $t \circ a$ is generated. If s is not seen by other traces, it is put into the queue. Otherwise, let u be the trace reaching s with $u \neq t \circ a$. We need to compare u and $t \circ a$ with respect to a total adequate order and use the smaller trace to explore s. The adequate order used in POEM is described as follows.

4.1 The adequate order for POEM

An *adequate order* on $\Sigma^*/\equiv$ is a partial order $\sqsubseteq \subseteq (\Sigma^*/\equiv \times \Sigma^*/\equiv)$ such that the following properties are satisfied:

- $[u] \sqsubseteq [uv]$, i.e. it refines the prefix relation on traces;
- $[u] \sqsubseteq [v]$ implies $[uw] \sqsubseteq [vw]$;
- $\sqsubseteq$ is well-founded, i.e. there is no infinite strictly descending chain $[u_1] \sqsupset [u_2] \sqsupset \ldots$.

The most straight forward partial orders are:

- The prefix relation itself, i.e. $[u] \sqsubseteq [v]$ iff there exists v_1 with $[v] = [uv_1]$.
- The length order : $[u] \sqsubseteq [v]$ iff $|u| \sqsubseteq |v|$.

The first order is included in the second order. For application purposes, let us just say here that the bigger the order (in ordering more pairs), the better. The ideal case is that of total adequate orders. Total adequate orders were proposed in [5, 6].

Here we propose a new adequate order for the implementation in POEM. The difference compared to previously proposed orders is that it is based on interleavings rather than partial orders and is thus potentially better suited for use with Local First Search.

The order is constructed in several steps based on some total order $\leq_{alph}$ on Σ. Moreover, let $|[u]| = |u|$ denote the length of u, and let $|[u]|_a = |u|_a$ denote the number of occurrences of a in u (a property invariant under $\equiv$). The Parikh vector [14] $p(u)$ of u or $[u]$ is the function $p(u) : \Sigma \longrightarrow N$ such that $p(u)(a) := |u|_a$. The $\leq_{alph}$-induced lexicographical order on Parik-vectors is defined as follows: $u <_p v$: iff

- either $|u| < |v|$
- or $|u| = |v|$ and for some $b \in \Sigma$ it holds that
  - $|u|_b < |v|_b$ and
  - for all $a \in \Sigma$ with $a <_{alph} b$ it holds that $|u|_a = |v|_a$.

If neither $u <_p v$ nor $v <_p u$ then obviously $p(u) = p(v)$.

On the other hand, $<_{alph}$ induces a lexicographical order on Σ^*, here simply denoted by $<_{lex}$: $u <_{lex} v$ if either $|u| < |v|$ or $|u| = |v|$ and $u = u_1 a u_2$, $v = u_1 b v_2$ with $u_1, u_2, v_2 \in \Sigma^*$ and $a, b \in \Sigma$ with $a <_{lex} b$.

$<_{lex}$ is a total order on Σ^*, which allows us to identify *unique representatives* of traces: Let $lex([u])$ denote the $v \equiv u$ such that for all $w \equiv u$ it holds that $v \leq_{lex} w$.

Lemma 1. *Let $u = lex([u])$ be the unique representant of $[u]$. Then for $a \in \Sigma$ we get $lex([ua]) = w_1 a w_2$ such that $u = w_1 w_2$ and for all b with $|w_2|_b > 0$ we have $a\ I\ b$.*

Proof. First note that there is a unique decomposition of $lex([ua])$ with $lex([ua]) = w_1 a w_2$ with $[u] = [w_1 w_2]$ and for all b with $|w_2|_b > 0$ we have $a\ I\ b$. By definition, $u \leq_{lex} w_1 w_2$ and hence $ua \leq_{lex} w_1 w_2 a$, but on the other hand $w_1 a w_2 \leq_{lex} ua$. Let $u = u_1 u_2$ such that $|u_1| = |w_1|$. We obtain from the above inequalities that $u_1 \leq_{lex} w_1 \leq_{lex} u_1$, hence $u_1 = w_1$. Hence, $[u_2] = [w_2]$ and by the definition of lex, it is easy to see that $u_2 = lex([u_2])$ and $w_2 = lex([w_2])$. Hence $u_2 = w_2$. $\square$

Based on the unique representatives $lex([v])$, we define $\sqsubseteq \subseteq (\Sigma^* / \equiv \times \Sigma^* / \equiv)$ as follows:

$[u] \sqsubseteq [v]$ iff

- either $u <_p v$ (Parikh order).
- or $p(u) = p(v)$ and $lex([u]) \leq_{lex} lex([v])$

Proposition 2. $\sqsubseteq$ *is a total adequate order*

Proof. First observe that $<_p$ is an adequate order.

Second, observe that $\leq_{lex}$ is total on Σ^* such $lex([u]) \leq_{lex} lex([v])$ defines a total order on traces, in particular those with the same Parikh vector. Hence $\sqsubseteq$ is total. Wellfoundedness of $\sqsubseteq$ results from the fact that $<_p$ is wellfounded, that the number of traces with the same Parikh vector is finite (permutations) and that $lex([u]) \leq_{lex} lex([v])$ defines a total order on traces.

$[u] \sqsubseteq [uv]$ is also obvious since either $v = \varepsilon$ (the empty sequence, obviously $\sqsubseteq$ is reflexive) or $|u| < |uv|$.

The difficult step is to prove that $[u] \sqsubset [v]$ implies $[uw] \sqsubset [vw]$ in the case that $p(u) = p(v)$ (otherwise, the fact that $<_p$ is adequate is sufficient). It is sufficient to check that $[u] \sqsubset [v]$ implies $[ua] \sqsubset [va]$ and use induction for the general case.

So let $[u] \sqsubset [v]$, $p(u) = p(v)$ and for simplicity assume that $u = lex([u])$ and $v = lex([v])$, i.e. u and v are the lexicographically least representatives of $[u]$ and $[v]$ respectively. Let $u = wbu'$ and $v = wcv'$ with $b <_{alph} c$.

Obviously $p([ua]) = p([va])$. Let $lex([ua]) = u_1 a u_2$ with $u = u_1 u_2$ and $lex([va]) = v_1 a v_2$ with $v = v_1 v_2$ according to Lemma 1.

Now we have to compare the different decompositions of $v_1 v_2 = wcv'$. If $|v_1| \leq |w|$ then let $u = v_1 u_2'$ the according decomposition of u where $p(v_2) = p(u_2')$ and hence $ua \equiv v_1 a u_2'$ (the importance of the same Parikh-vector here is that a commutes with all letters in u_2') and we obtain $u_1 a u_2 \leq_{lex} v_1 a u_2'$ and we know that $u_2' <_{lex} v_2$ hence $u_1 a u_2 <_{lex} v_1 a v_2$. If $|v_1| > |w|$ then $v_1 = wcv_1'$ and we obtain $u_1 a u_2 \leq_{lex} ua = wbu'a <_{lex} wcv_1' a v_2 = v_1 a v_2$. $\square$

4.2 The extended data structure

In order to support the adequate order, the basic trace structure needs to be extended. Figure 2 depicts the extension, where a node has three additional fields:

"parikh_vector_sum" records the number of actions in the trace that is from the root to the current node, "parikh_vector" points a dynamically allocated memory to store the parikh vector, "peak_vector" has the same structure as parikh_vector. The parikh vector has a field "length" and an array "vector". The length field records the length of the array, and each element in the array is the number of occurrences of an action in the trace. The array in the peak vector stores maximal actions in the trace.

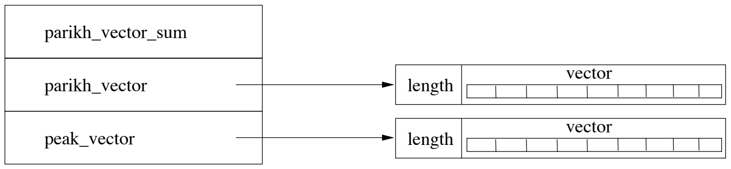

Fig. 2. The extension to the basic data structure

LFS requires to compare two traces with respect to the adequate order during state space search. In a comparison, one trace is an "old" one that has been explored, while the other is the "new" one currently being explored, i.e. it is created by appending an action to a trace. The procedure of comparison is shown in Algorithm 3 according to the definition of the adequate order:

1. Compare the length of two traces. If they are equal, go to the next step.
2. Compare their parikh vectors. If they are still equal, go to the next step.
3. Compare the lexicographical representives of these traces.

Note that a temporary trace stored in a stack is generated for $t2 \circ a$ during the comparison. If the result shows $t1 \sqsubset t2 \circ a$, the temporary trace is discarded. Otherwise, it is written into the tree structure. In this case, it is easy to know that 1 and $t2 \circ a$ have the same length, and therefore, the last node of $t1$ is in the search queue waiting for process. Removing this node, naming it as x, from the queue first and then appending a new node, say y, for $t2 \circ a$ to the end of queue cause difficulties on maintenance of the queue. During the implementation of LFS, we chose to reuse the space of x for y, and afterwards, remove x from the children list of the father node of x.

5 Trace systems in Event Zone approach

Event zone automata [11] are a partial order based approach to reduce one source of clock explosion, interleaving semantics. It uses vectors of event (action) occurrences, namely, event zones, instead of classical clock zones, to express clock constraints. The independence relation in event zone approach is based on reading and writing of shared variables: If for some clock x, transition a resets x

Algorithm 3 Compare two traces $t1$ and $t2 \circ a$ w.r.t. the adequate order

```
**Return Value**: $1 \Rightarrow [t1] > [t2 \circ a]$; $0 \Rightarrow [t1] = [t2 \circ a]$; $-1 \Rightarrow [t1] < [t2 \circ a]$
if $t1.parikh\_vector\_sum > t2.parikh\_vector\_sum + 1$ then
  return 1
else if $t1.parikh\_vector\_sum < t2.parikh\_vector\_sum + 1$ then
  return -1
end if
generate a new parikh vector $new\_pv$ for $t2 \circ a$
for all $i$ such that $i \geq 0 \wedge i \leq t1.parikh\_vector.length \wedge i \leq new\_pv.length$ do
  if $t1.parikh\_vector.vector[i].act < new\_pv.vector[i].act$ then
    return -1
  else if $t1.parikh\_vector.vector[i].act > new\_pv.vector[i].act$ then
    return 1
  end if
  if $t1.parikh\_vector.vector[i].num < new\_pv.vector[i].num$ then
    return -1
  else if $t1.parikh\_vector.vector[i].num > new\_pv.vector[i].num$ then
    return 1
  end if
end for
$temp\_trace \leftarrow t2 \circ a$
while $t1.predecessor \neq temp\_trace.predecessor$ do
  $t1 \leftarrow t1.predecessor$, $temp\_trace \leftarrow temp\_trace.predecessor$
end while
if $t1.lastaction < temp\_trace.lastaction$ then
  return -1
else if $t1.lastaction > temp\_trace.lastaction$ then
  return 1
else
  return 0
end if
```

and transition b has a condition on x or if both a and b reset x, then they must be dependent. Based on Mazurkiewicz trace theory and the independence relation, event zone approach successfully avoids zone splitting in a typical situation: transitions a resetting clock x and b resetting y are independent, and both enabled in a state. Executing the sequence ab and ba results two incomparable clock zones, while only one event zone.

Event zone approach also uses a breath-first search algorithm. The one implemented in [11] works as follows. When an enabled action a in a state s in the search queue is explored, let s' be the state reached by a, and $\mathcal{Z}$ the event zone after executing a. $\mathcal{Z}$ is computed according to the trace leading to s'. If the symbolic state $(s', \mathcal{Z}$ is not visited by other traces, it is put into the queue. Otherwise, the trace to s' is discarded.

Event zone approach has been improved during implementing it in POEM. The improvement came from the following proposition [11].

Proposition 3. *A trace has a unique canonical equivalent event zone.*

In the implementation of event zone in [11], two equivalent traces are detected by testing equivalence of their symbolic states. Compute event zones, which is highly time consuming, has to be done before performing this testing, which thus slow down the state space search. Due to Propositon 3, one of two equivalent traces can be removed without loss of event zones. The trace structure in POEM supports automatic detection of equivalent trace by a minor modification of Algorithm 1 and 2: $PATH\_SUCCESSOR$ sets a flag is_old_trace if it finds out that there is a node in the children list that has been associated with the given action, and Algorithm 1 sets another flag to indicate an equivalent trace is found by checking whether each calling of $PATH\_SUCCESSOR$ sets is_old_trace.

In order to demonstrate the effect of improvement, we made two experiments to compare the time cost before and after applying the improvement. The experiments were carried out in a machine with two 2.8GHz Xeon CPUs, 2GB memory and Fedora core 4 Linux.

The first experiment is a timed version of dining philosophers. There are a group of philosophers and a timestopper process. The automata of a philosopher and the timestopper are shown in Figure 3, respectively [1]. Each philosopher has five states: think, hungry, leftfork, eat and dropthefork. The timestopper has two states: onestate and finalstate. This process is used to stop the execution of the system when time progresses to a limit. hungernoticed, patience, starved, eatingtime, concentrated and timelimit are constant; foodless, myclock and time are local clocks; myindex is the process id; afork and done are program variables. Figure 4 shows the results generated by POEM. The data under the title "Testing" were obtained by testing if the current trace has been seen before; The data with "No Testing" were obtained without such testing.

The second experiment was performed on the following example. A multi lane highway with cars on each lane and a rabbit who wants to cross. The rabbit has some freedom of going slower or faster and so do the cars. Can - with the

[1] Figure 3 and 6 were produced by UppAal.

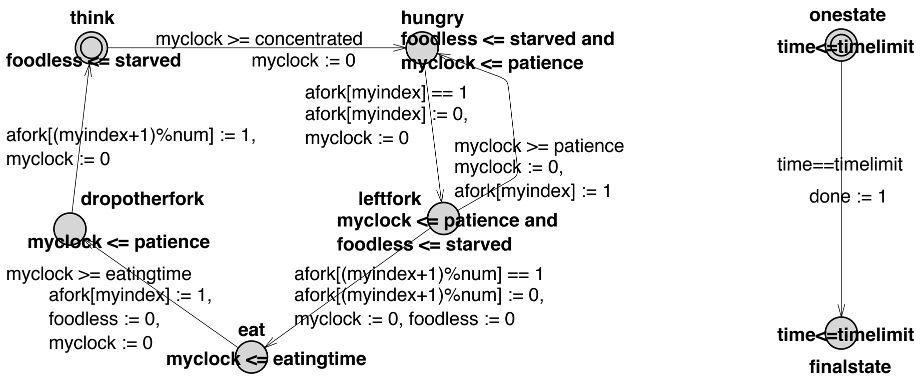

Fig. 3. The automata of a philosopher (left) and the timestopper (right)

Number of philosophers	Memory	Time	
		Testing	No Testing
2	16m	0.03s	0.02s
3	16m	0.05s	0.05s
4	17m	0.31s	0.52s
5	22m	5.01s	9.58s
6	72m	78.12s	173.33s
7	540m	1168.92s	2840.52s

Fig. 4. Results of the philosophers

Number of lanes	Memory	Time	
		Testing	No Testing
1	16m	0.03s	0.03s
2	16m	0.03s	0.03s
3	16m	0.03s	0.04s
4	17m	0.09s	0.12s
5	19m	1.27s	2.31s
6	36m	10.79s	22.96s
7	118m	87.46s	211.78s
8	466m	554.34	1490.43s

Fig. 5. Results of the highway

help of the car drivers - the rabbit reach the other side of the highway alive? To model this by a network of timed automata, we choose to model the highway as a checker board of lanes and positions on lanes as indicated in the picture, cars move in the horizontal direction and the rabbit in the vertical direction. Each car and the rabbit is realised by an individual automaton. The freedom of going slower or faster is modeled by a time interval in which the rabbit can advance by one lane and an interval in which the car can advance for one unit length on a discretized highway. If a car and the rabbit are in the same field of the checker board at the same time, an accident occurs. Figure 6 shows the automaton for the rabbit and an instance of the automata for cars. The results are listed in

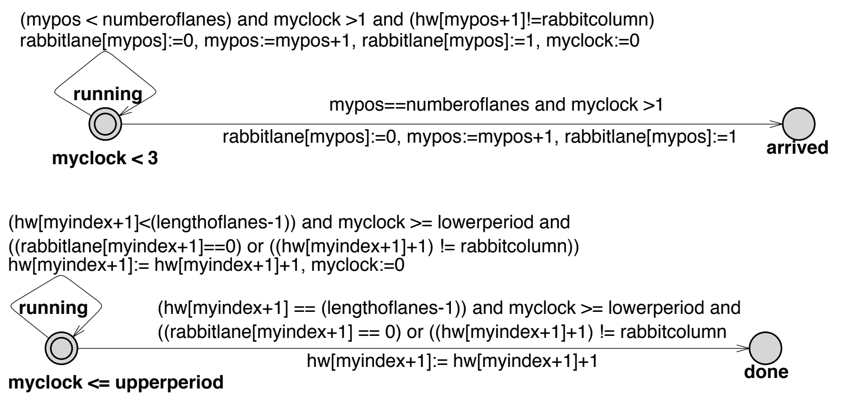
(mypos < numberoflanes) and myclock >1 and (hw[mypos+1]!=rabbitcolumn)
rabbitlane[mypos]:=0, mypos:=mypos+1, rabbitlane[mypos]:=1, myclock:=0
running
mypos==numberoflanes and myclock >1
rabbitlane[mypos]:=0, mypos:=mypos+1, rabbitlane[mypos]:=1
myclock < 3
arrived
(hw[myindex+1]<(lengthoflanes-1)) and myclock >= lowerperiod and
((rabbitlane[myindex+1]==0) or ((hw[myindex+1]+1) != rabbitcolumn))
hw[myindex+1]:= hw[myindex+1]+1, myclock:=0
running
(hw[myindex+1] == (lengthoflanes-1)) and myclock >= lowerperiod and
((rabbitlane[myindex+1] == 0) or ((hw[myindex+1]+1) != rabbitcolumn
hw[myindex+1]:= hw[myindex+1]+1
myclock <= upperperiod
done

Fig. 6. The automata of the rabbit (up) and a car (down)

Figure 5. The advantage of testing known traces in this experiment was more explicit than the first one.

6 Conclusion

References

1. G. Behrmann, A. David, K. G. Larsen, O. Moeller, P. Pettersson, and W. Yi. Uppaal - present and future. In *Proc. of 40th IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.
2. S. Bornot, Rèmi Morin, Peter Niebert, and Sarah Zennou. Black box unfolding with local first search. In *TACAS*, LNCS 2280, pages 386–400. Springer, 2002.
3. M. Bozga, S. Graf, and L. Mounier. If-2.0: A validation environment for component-based real-time systems. In *CAV*, LNCS 2404, pages 343–348. Springer, 2002.
4. Volker Diekert and Grzegorz Rozemberg, editors. *The Book of Traces*. World Scientific Publishing Co. Pte. Ltd., 1995.

5. J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In *CONCUR*, LNCS 1664, pages 2–20. Springer, 1999.
6. J. Esparza, S. Romer, and W. Vogler. An improvement of mcmillan's unfolding algorithm. In *TACAS*, LNCS 1055, pages 87–106. Springer, 1996.
7. J. Esparza, S. Römer, and W. Vogler. An improvement of mcmillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.
8. Javier Esparza, Stefan Romer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 87–106, 1996.
9. Patrice Godefroid and Pierre Wolper. A partial approach to model checking. In *Logic in Computer Science*, pages 406–415, 1991.
10. Gerard Holzmann and Doron Peled. Partial order reduction of the state space. In *First SPIN Workshop*, Montrèal, Quebec, 1995.
11. G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, Reading, Massachusetts, 2003.
12. D. Lugiez, P. Niebert, and S. Zennou. A partial order semantics approach to the clock explosion problem of timed automata. *Theoretical Computer Science*, 345(1):27–59, 2005.
13. K. L. McMillan. A technique of state space search based on unfolding. *Form. Methods Syst. Des.*, 6(1):45–65, 1995.
14. P. Niebert, M. Huhn, S. Zennou, and D. Lugiez. Local first search: a new paradigm in partial order reductions. In *CONCUR*, LNCS 2154, pages 396–410. Springer, 2001.
15. R. J. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.
16. Doron Peled. All from one, one for all: on model checking using representatives. In *CAV*, pages 409–423, 1993.
17. W. Penczek and R. Kuiper. Traces and logic. In Diekert and Rozemberg [4].
18. Antti Valmari. Stubborn sets for reduced state space generation. In *Applications and Theory of Petri Nets*, pages 491–515, 1989.