Symbolic Reachability for Process Algebras with Recursive Data Types

Stefan Blom and Jaco van de Pol*

Formal Methods and Tools, Department of Computer Science, University of Twente, P.O. Box 217, 7500AE Enschede, The Netherlands
{sccblom,vdpol}@cs.utwente.nl

Abstract. In this paper, we present a symbolic reachability algorithm for process algebras with recursive data types. Like the various saturation based algorithms of Ciardo et al, the algorithm is based on partitioning of the transition relation into events whose influence is local. As new features, our algorithm supports recursive data types and allows unbounded non-determinism, which is needed to support open systems with data. The algorithm does not use any specific features of process algebras. That is, it will work for any system that consists of a fixed number of communicating processes, where in each atomic step only a subset of the processes participate. As proof of concept we have implemented the algorithm in the context of the μCRL toolset. We also compared the performance of this prototype with the performance of the existing explicit tools on a set of typical case studies.

1 Introduction

High level formalisms, such as Petri Nets and Process Algebras are powerful languages for specifying systems. When combined with recursive data types, they become even more powerful. However, we have to pay a price for this expressiveness. Analyzing these specifications with symbolic techniques is difficult, because it is not easy to translate them to a formalism where the state is a vector of booleans. Thus, toolsets for process algebras with data, such as CADP [1,2], FDR [3] and μCRL [4] rely on explicit state techniques. The former two exploit compositional techniques to extend the size of the state space that can be dealt with. For the latter, we present a symbolic technique based on decision diagrams in this paper.

Computing the set of reachable states is a good way to perform model checking tasks such as the verification of safety properties. But it can be an expensive computation. Therefore much work has gone into avoiding doing so. The entire fields of bounded model checking and on-the-fly model checking are devoted to developing methods that can give useful results without having to perform a complete reachability analysis. However, any exhaustive technique will need to perform a reachability analysis somehow, so we use this as a first step towards building a complete symbolic tool chain.

* This work has been partially funded by the EU under grant number FP6-NEST STREP 043235 (EC-MOAN).

Contribution and related work. In the area of quantitative evaluation, symbolic techniques already exist. The Petri Net based tool SMART [5] implements a powerful technique called saturation [6]. Also, symbolic techniques have been developed for a stochastic process algebra and implemented in the tool CASPA [7,8]. The most general algorithm is the one used in SMART. But, it does not have support for two features that are fundamental for the process algebra μCRL and an optional extension for Petri Nets: infinite data types and non-deterministic events.

Infinite data types are due to the fact that in μCRL one can define recursive data types. Non-deterministic events arise if one needs to model a random input while modeling an open system. For example, in μCRL, we can write a 1-place buffer as follows:

$$X = \sum_{x \in \mathbb{N}} read(x).write(x).X \text{ where } \mathbb{N} ::= 0 \mid \mathsf{succ}(\mathbb{N}) \ .$$

If a *read*-event happens then the argument must be chosen from an infinite set. In practice, choices like this will be from a finite set, which is not known a priori. Thus, we do not know a priori a limit on the branching degree. In a classical Petri Net, events are deterministic: once an event is chosen the result is fixed because no matter which tokens are selected the result will be identical. In a colored Petri Net, events can also be non-deterministic. If an event is chosen then selecting tokens with different colors can still lead to different results. If a finite superset of the used colors is known in advance a colored Petri Net can be encoded as a monochrome Petri Net. The algorithms, which are implemented in SMART rely on deterministic events. In our tool we extend the algorithms to deal with non-deterministic events. As the underlying data structure, we do not use MDDs with in-place updates, as in [9]. We use a decision diagram formalism which mixes features from MDDs and ZDDs instead. Moreover, we use a classical BDD style next state computation rather than in-place updates.

Overview. The remainder of this paper is organized as follows. In the next section, we discuss some of the basics of explicit state space generation and symbolic reachability analysis. In Sect. 3 we discuss the principle of event locality and how that notion leads to a partitioning of the symbolic transition relation and a refactoring of the next state code for explicit tools. This is followed by a presentation of our grey box reachability algorithm. The next section contains some remarks on how this algorithm was implemented for the μCRL toolset. Section 6 presents the results of a few experiments performed with that prototype and we conclude with a discussion of the results so far and future work.

2 Preliminaries

The semantics of modeling formalisms used in model checking are always some form of transition system. The basic structure of a transition system is defined below.

Definition 1. *A transition system (TS) is a tuple $\langle S, R, s^0 \rangle$, where S is a set of states, $R \subseteq S \times S$ is the transition relation and $s^0 \in S$ is the initial state*

For practical purposes, one needs to attach labels to either states (e.g. atomic propositions) or the transitions (e.g. actions) or both. These labels are not essential to the presentation in this paper, so we omit them. What is essential is that states have a vector structure. In this paper, we assume that a state is a fixed length vector. From now on N will stand for the length of the vector and D_i for $i = 1 \cdots N$ will be the domain of the i^{th} element. Thus the set of states S is given as

$$S = D_1 \times \cdots \times D_N \ .$$

When the set of states is defined as a tuple like this, it is inevitable that not all states are reachable. In fact, for μCRL many sets D_i are the semantics of a recursive data type and hence infinite. In the remainder, we assume a *single initial state* and a *finite set of reachable states.*

Definition 2. *Given a transition system $\mathcal{L} \equiv \langle S, R, s^0 \rangle$. The set of reachable states is*

$$V = \{s \in S \mid s^0 \ R^* \ s\} \ .$$

A state $s \in S$ is reachable if $s \in V$.

Usually, we build the set of reachable states using a breadth first strategy. Thus, we define level i (denoted L_i) as the set of states whose distance to the initial state is i. The set of states at distance less than or equal to i is denoted V_i. The union of all V_i is V:

Proposition 1. *Given a transition system $\mathcal{L} \equiv \langle S, R, s^0 \rangle$. Let*

$$\begin{array}{llll} L_0 & = \{s^0\} & V_0 & = \{s^0\} \\ L_{i+1} & = \{s' \in S \mid \exists s \in L_i : \ s \, R \, s' \wedge s' \notin V_i\} & V_{i+1} & = V_i \cup L_{i+1} \end{array}$$

then

$$V = \bigcup_{i=0}^{\infty} V_i \ .$$

A simple algorithm that computes the set of reachable states is given in Table 1. How this algorithm is implemented depends on how the transition system is given. Next, we shortly review explicit state space generation from an on-the-fly interface and symbolic reachability.

For explicit state model checking, the fundamental way of implementing a transition system is by implementing the two functions in Table 2. The first function simply returns the initial state and the second returns the set of successors $\{s' \mid s \ R \ s'\}$ of a given state s as a list. The data structure used to implement sets is typically a hash table. Using this representation, the implementation of line 5 of Table 1 boils down to a simple loop that calls GetNext many times. Because this interface does not give away any details about the internal structure, we call it a black box interface.

Table 1. Basic reachability algorithm

```
1  proc reach()
2    $V := \{s^0\}$
3    $L := V$
4    while $L \neq \emptyset$ do
5      $L := \{y \mid \exists x \in L : x\ R\ y\}$
6      $L := L \setminus V$
7      $V := V \cup L$
8    end
9    return $V$
10 end
```

Table 2. Black Box on-the-fly API

```
state       GetInitial();
state list  GetNext(state s);
```

For symbolic reachability, the data structures for both sets and the transition relation are some form of decision diagram. That is, a set $S' \subseteq S$ is represented by a boolean expression $\mathcal{S}'(\boldsymbol{x})$ such that

$$\boldsymbol{x} \in S' \Leftrightarrow \mathcal{S}'(\boldsymbol{x})$$

where the expression is stored as a decision diagram and $\boldsymbol{x}$ stands for the vector $x_1, \cdots x_n$. Similarly the transition relation is stored as a boolean expression $\mathcal{R}(\boldsymbol{x}, \boldsymbol{x'})$, such that

$$\boldsymbol{x}\ R\ \boldsymbol{x'} \Leftrightarrow \mathcal{R}(\boldsymbol{x}, \boldsymbol{x'})$$

Given a level as a formula $\mathcal{L}(\boldsymbol{x})$, we can compute the next level using the expression:

$$(\exists \boldsymbol{x}.(\mathcal{L}(\boldsymbol{x}) \wedge \mathcal{R}(\boldsymbol{x}, \boldsymbol{x'})))[\boldsymbol{x'} := \boldsymbol{x}]$$

Which provides us with the symbolic implementation of line 5.

The major advantage of symbolic techniques is that the representation of a set of states can be very compact. For example, the set of bit vectors with even parity can be represented by a decision diagram with a number of nodes that is linear in the length of the vector. The same holds for transition relations. Of course not every relation is represented easily. For example, multiplication of two n-bit numbers will produce a diagram that is exponential in n. (See [10].)

Although it is easy to apply explicit techniques to a symbolic model, it is not so easy to apply symbolic techniques to a given explicit model. The problem is that to compute a symbolic version of the transition relation, we must basically enumerate all possible transitions and collect them in a symbolic structure. In the next section, we explain how to modify the explicit interface in such a way that symbolic techniques can be applied efficiently.

3 Locality

The key to applying symbolic techniques to on-the-fly models is *event locality*. The notion of event locality refers to the fact that even though in a state several events could be enabled, each event separately affects just a small part of the state vector. For example, if one has a system which is composed of several processes running in parallel then in many models there are just two kinds of events: events in which one of the processes performs a step and events in which two of the processes synchronize to perform a step. In these cases, the enabledness and result of steps is decided by looking at the global variables (if any) and the local variables of the processes involved.

As an example, let us consider two ways of solving the 8-queens problem. The efficient way of solving the problem with a non-deterministic program is

```
for $i = 1$ to 8 do
  put queen $i$ in any row in column $i$
  if for some $0 < j < i$ queen $i$ on the same row or diagonal as queen $j$ then
    fail
  end
end
success
```

A solution to the problem is a path that ends in success. Putting the first queen is a very local event: we just put the queen somewhere in the first column. This affect the counter i and the position of the first queen only. The 8th step however is completely non-local: we need to check the counter i, test the positions of queens $1, \cdots, 7$ and write the position of queen 8.

To get event locality, we may rewrite the problem as follows:

```
for $i = 1$ to 8 do put queen $i$ on row 1 of column $i$ end
while true
  if $\exists 0 < i, j \leq 8$: queen $i$ on the same row or diagonal as queen $j$ then
      move queen $i$ to any row in column $i$
  end
end
```

A solution to the problem is a path to deadlock: if no two queens are in the same row, the same diagonal or the same column then no move is possible. However, every step is local: every move requires testing the positions of two queens and if enabled writing one of them.

To exploit event locality, one can partition the set of possible transitions into groups of transitions, such that each group affects part of the state only.

For the algorithm we present in the next section, we need to extend the black box interface to support group information. Because the extension exposes more structure than the black box interface, we have called it the grey box interface. The grey box interface is presented in Table 3 and uses five functions. The first function returns the length of the state vector (N). The second returns the number of groups, which from now on we will refer to as K. The third function

Table 3. Grey Box State Space API

```
int              GetStateLength();
int              GetGroupCount();
int list         GetGroupInfluenced(int group);
data list        GetInitialState();
data list list   GetNext(data list src,int group);
```

returns (a list representation of) the set of indices that is influenced (either read or written) while computing the enabledness and next states of the given group. This set of indices will be referred to as I_g, for any group g. The fourth function returns the initial state as a list of length N. The fifth function returns a list of projected next states, given a projected state and a group. That is, the length of src and each of the next states is the size of I_g if the group is g.

In terms of symbolic algorithms, event locality means that we can partition the transition relation into a disjunction (over the separate groups) of conjunctions (collections of local transitions) as follows:

$$\mathcal{R}(\boldsymbol{x},\boldsymbol{x'}) = \bigvee_{g=1}^{K} \mathcal{D}_g(\boldsymbol{x},\boldsymbol{x'}) = \bigvee_{g=1}^{K} \left(\mathcal{R}_g(\pi_g(\boldsymbol{x}),\pi_g(\boldsymbol{x'})) \wedge \bigwedge_{i \notin I_g} [x_i = x'_i] \right),$$

where we define the projection to a group as $\pi_g(\boldsymbol{x}) = (x_j)_{j \in I_g}$.

4 Grey Box Reachability Algorithm

In this section, we describe our symbolic reachability algorithm for grey box models. The variables and constant used are listed in Table 4. The set operations are listed in Table 5 and the algorithm itself is presented as Table 6.

The algorithm performs a breadth first analysis. That is, the visited set and the current level are set to singleton initial state and the visited part of each group is set to empty. Next, we repeat the main loop in which we replace the current level by the new states reachable from that level until the current level is empty.

In each iteration of the main loop we first extend the local symbolic transition relations of the groups to include all necessary transitions. (See lines 9-15.) That is, for every group we project the current level to the sub-vector used by the group and for all of the new sub-vectors we explore the next states and insert any transition found into the local transition relation. The second half of the main loop is building the new level by computing the next states of the level set according to each of the group relations. (See lines 16-21.) This involves a symbolic next state computation for each group that changes the members of the sub-vector and leaves all other variables unchanged. From this the next level is then computed.

Table 4. Reachability Variables and Constants

K	Number of groups	constant number
I_i	Indices in the state vector, which are Influenced by group i	explicit list
V	Visited states	symbolic
L	current Level	symbolic
L^p	projection of current level to influenced variables of current group	symbolic
V_i^p	projected states Visited for group i	symbolic
R_i^p	projected transition Relation for group i	symbolic
N	Next level	symbolic
s^p	projected state	explicit vector

Table 5. Operations on sets

$\cdot \setminus \cdot$	set minus	symbolic
$\cdot \cup \cdot$	set union	symbolic
project$(\cdot,\cdot)$	projection to a sub-vector	symbolic
step$(\cdot,\cdot,\cdot)$	result of one step in a relation applied to a sub-vector	symbolic
next$_i^p(\cdot)$	next state function of the i^{th} group	explicit
$\{\cdot \mid \cdot\}$	building a set by inserting elements one element at a time	mixed

The set operations project and step can be written as symbolic set operations as follows:

$$\begin{aligned}
&\text{project}(\mathcal{S}(\boldsymbol{x}), I) = \exists (x_i)_{i \notin I}.\mathcal{S}(\boldsymbol{x})\\
&\text{step}(\mathcal{S}(\boldsymbol{x}), \mathcal{R}((x_i)_{i\in I}, (x_i')_{i\in I}), I) =\\
&\quad (\exists (x_i)_{i\in I}(\mathcal{S}(\boldsymbol{x}) \wedge \mathcal{R}((x_i)_{i\in I}, (x_i')_{i\in I})))[x_i' := x_i | i \in I]
\end{aligned}$$

The function call next$_i^p(s)$ is shorthand for the call GetNext(s,i).

5 Implementation

We have implemented a prototype of the algorithm presented in the previous section on top of the μCRL tool set ([4],[11]). The concept of this toolset is to take a specification and compile it into a linear process equation (LPE). An LPE is a process given as an initial state and a recursive equation:

$$X(\boldsymbol{x}) = \sum_{i=1}^{K} \underbrace{\sum_{e_i \in E_i} C_i \Rightarrow a(t_{i,0}).X(t_{i,1}, \cdots, t_{i,n})}_{\text{summand } i}$$

where C_i and $t_{i,j}$ are expressions over $e_i, x_1, \cdots, x_n$. The intended meaning of this equation is that to perform a step, one has to first non-deterministically select $1 \leq i \leq K$ (determining a summand), then non-deterministically select some $e \in E_i$, evaluate the condition C_i to see if the step is enabled and if it is enabled then the label of the step is the result of the expression $a(t_{i,0})$ and the next state is $t_{i,1}, \cdots, t_{i,n}$.

Table 6. Symbolic reachability algorithm for grey box models

```
proc mixed_reach()
  $V := \{s^0\}$
  $L := V$
  for $i = 1$ to $K$ do
    $V_i^p := \emptyset$
    $R_i^p := \emptyset$
  end
  while $L \neq \emptyset$ do
    for $i = 1$ to $K$ do
      $L^p := \text{project}(L, I_i)$
      for $s^p$ in $L^p \setminus V_i^p$ do
        $R_i^p := R_i^p \cup \{(s^p, d^p) \mid d^p \in \text{next}_i^p(s^p)\}$
      end
      $V_i^p := V_i^p \cup L^p$
    end
    $N := \emptyset$
    for $i = 1$ to $K$ do
      $N := N \cup \text{step}(L, R_i^p, I_i)$
    end
    $L := N \setminus V$
    $V := V \cup N$
  end
  return $V$
end
```

Hence, an LPE has a natural partitioning into groups by treating each summand as a group. Selecting this partitioning, the influenced variables of each summand are as follows:

$$I_i^X = \{x_j \mid t_{j,k} \neq x_j \vee \exists k \neq j : x_j \text{ occurs in } C_j \text{ or } t_{j,k}\}$$

We implemented this natural partitioning and we used it for the tests presented in this paper.

The μCRL toolset uses the ATerm library ([12]). To make interfacing with the decision diagrams easy we used a simple decision diagram library for manipulating sets, which we implemented on top of the ATerm library. The ATerm library was developed for the manipulation of large terms. It uses maximal sub-term sharing to keep its memory footprint minimal, which is the equivalent of a global unique table. It also provides garbage collection, but it does not provide advanced caching strategies. The resulting data structure for sets of vectors is a form of multi-way decision diagram (MDD). We call it List Decision Diagram (LDD), because instead of having one node with many edges we have a linked list.

By using the ATerm library, we automatically get maximal sharing (a global unique table), so there are no duplicate nodes. We can further classify our structure as a quasi-reduced version [13] rather than the fully-reduced version [14]. This choice was made because the set of possible values at each level is dynamic.

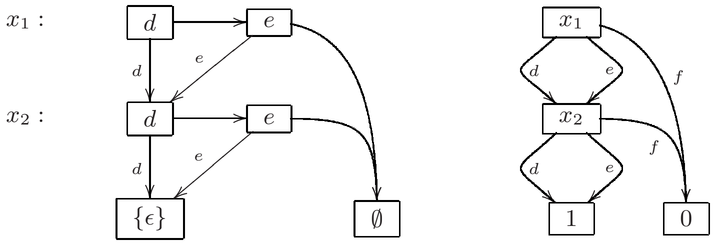

Fig. 1. The set {dd,de,ed,ee} as LDD and MDD

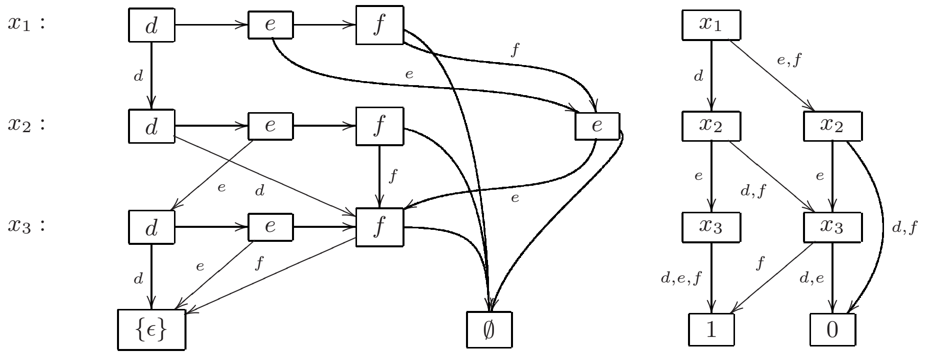

Fig. 2. The set {def,eef,fef,ddf,dff,ded,dee} as LDD and MDD

In a fully-reduced setting every extension of the set of values requires an update of every diagram, in the quasi-reduced setting this operation has no impact at all. For the same reason we do not use nodes with multiple successors, but just nodes with 2 successors that form a sorted list.

An LDD is a DAG. In this dag we have three types of nodes. The node types $\{\epsilon\}$ (or true) and $\emptyset$ (or false) do not have successors. That is, they are constants. The third type of node has a label (a) and two successors (n_1, n_2) and is written as node(a, n_1, n_2). The semantics $[\![S]\!]$ of an LDD S is as follows:

$$\begin{aligned}
[\![\{\epsilon\}]\!] &= \{\epsilon\} \\
[\![\emptyset]\!] &= \emptyset \\
[\![\text{node}(a, n_1, n_2)]\!] &= \{a\,w \mid w \in [\![n_1]\!]\} \cup [\![n_2]\!]
\end{aligned}$$

To illustrate the relation between MDDs and our own LDDs, we have drawn the set $\{dd, de, ed, ee\}$ $(\{d, e\} \times \{d, e\})$ and the set $\{def, eef, fef, ddf, dff, ded, dee\}$ (Hamming distance to def no more than 1) in both formats in Fig. 1 and 2 respectively. The MDD's are over the domain $D = \{d, e, f\}$. To make these four diagrams easier to read, we have added edge labels that allow us to check for membership of a vector by checking if there is a path from the root to $\{\epsilon\}$ or 1 such that the string of edge labels along the path is the vector. To save clutter

Table 7. Reachability for Distributed Lift

legs	states	mem	time	mem/state	states/sec
2	391	30,668	1.08	80,317.22	362.04
3	7,369	56,236	4.32	7,814.58	1,705.79
4	129,849	79,820	29.48	629.47	4,404.65
5	2,165,446	181,592	250.10	85.87	8,658.32
6	33,949,609	661,724	2,344.03	19.96	14,483.44
7	501,505,138	2,246,788	17,995.72	4.59	27,868.02

an edge labeled d, e means two edges, one labeled d and one labeled e. Note that the edges without label in the LDDs correspond to the linked lists "building" the MDD nodes.

The sizes of LDDs and MDDs can only differ by a constant: it can be proven that the number of nodes in an MDD is less than or equal to the number of nodes in the corresponding LDD, which is in turn less than or equal to the number of edges in the MDD.

6 Experiments

In this section, we describe two sets of experiments that have been carried out. The first set of experiments measures the performance of our symbolic reachability analysis on two parametrized problems. The second set of experiments compares the performance of state space generation for a set of five problems and three tools.

To test the performance of our symbolic reachability analysis, we used two series of problems:

– A distributed lift system [15]. This model describes a system that can lift large vehicles by using one leg for each wheel of the vehicle. These legs are connected in a ring topology. The number of legs is a parameter.
– A version of the sliding window protocol [16]. This model is parametrized by both the number of data elements in the alphabet and the windows size. (The buffer is a fixed 1 place lossy buffer.)

To test the performance for these models, we used a dual Intel Xeon E5335 (2.0GHz) machine with Intel 5000P chipset and 8GB memory. We ran one experiment at a time. The results for the lift problem are in Table 7. Those for the sliding window protocol are in Table 8. Both tables contain the number of states in each of the models and the time and memory usage. Time is measured in time elapsed and in number of states processed per second. Memory is measured in maximum resident set (RSS) in kB and in number of bytes per state. The first table contains both forms, the second table has just the second form of the time and memory numbers.

The explicit state space generator of the μCRL has a lower bound of 16 bytes per state, excluding hash tables and other overhead. Since a hash table easily accounts for another 8 bytes per state, anything below 24 is good. The reason for these high numbers is that we have used the ATerm library in 64-bit

Table 8. Reachability for Sliding Window Protocol

states mem/state (B) states/sec		→ windows size							
		1	2	3	4	5	6	7	8
→ elements		156	1,860	10,608	43,320	146,740	442,524	1,235,528	3,269,680
	1	36,785.23	6,355.41	1,749.53	499.90	149.17	50.05	20.62	12.32
		111.43	1,273.97	6,548.15	20,826.92	48,589.40	83,652.93	110,020.30	109,171.29
		390	10,156	126,138	1,132,248	8,487,750	56,793,060	351,503,922	
	2	28,262.40	1,821.34	174.08	24.31	7.28	4.79	9.79	
		267.12	6,269.14	46,545.39	114,600.00	140,688.71	96,365.59	34,239.62	
		708	32,124	719,460	12,075,000	174,187,380			
	3	15,955.89	657.04	34.54	5.55	5.19			
		468.87	18,149.15	125,779.72	189,560.44	89,441.99			
		1,110	78,156	2,829,570	79,474,200				
	4	9,819.33	280.12	10.87	3.31				
		760.27	37,575.00	201,536.32	189,694.00				
		1,596	161,812	8,746,248	375,691,704				
	5	7,114.11	135.50	5.88	6.91				
		1,093.15	67,421.67	269,945.93	113,410.54				
		2,166	299,820	22,789,098					
	6	5,137.96	73.29	4.28					
		1,483.56	103,030.93	301,802.38					
		2,820	512,076	52,280,988					
	7	3,683.50	43.42	3.45					
		1,931.51	139,530.25	314,094.25					
		3,558	821,644	108,715,890					
	8	3,143.95	30.87	2.52					
		2,356.29	179,398.25	294,822.75					

mode which uses about twice as much memory as in 32-bit mode. A carefully designed symbolic set library should work with half the memory or less. The rate of state exploration in the purely explicit instantiator is typically around a few thousand states per second. Values of over 100,000 obtained for some sliding window protocol instances are therefore a big improvement.

Our second experiment used the same set of 5 problems that we used in an earlier paper where we studied the performance of distributed state space generators [17]. We list the number of levels (iterations needed), the size of the state space and a brief description of each problem:

lift5. This model has 103 levels, 2,165,446 states and 8,723,465 transitions. It describes an elevator system with 5 legs in order to lift large vehicles [15].

SWP. This model has 61 levels, 19,466,100 states and 93,478,264 transitions. It is a version of the sliding window protocol [16]. This instance has 3 data elements, window size 2 and 3-place lossy buffers for communication.

1394fin. This model has 170 levels, 88,221,818 states and 152,948,696 transitions. It describes the physical layer service of the 1394 or firewire protocol and also the link layer protocol entities. [18,19] We use an instance with 3 links and 1 data element.

franklin53. This model has 82 levels, 84,381,157 states and 401,681,445 transitions. It describes a leader election protocol for anonymous processes along a bidirectional ring of asynchronous channels, which terminates with probability one [20,21]. We chose an instance with 5 nodes and 3 identities.

Table 9. Comparison of state space generation tools

problem/order		symbolic		mixed		explicit	
		time(s)	mem(kB)	time(s)	mem(kB)	time(s)	mem(kB)
lift5	G	861	378	535	195	408	230
lift5	R	217	181	398	101	319	226
swp	G	30,322	345	2,073	111	1,426	1,308
swp	R	29,232	342	1,850	107	1,264	1,308
1394fin	G	768	214	46,787	2,356	41,560	5,592
franklin53	G	out of memory		97,005	23,875	6,745	7,844
franklin53	R1	4,970	2,187	17,649	653	6,557	5,533
franklin53	R2	12,712	6,989	15,448	662	6,565	5,529
ccp33	G	out of memory		out of memory		44,855	7,741
ccp33	R	146,127	44,214	82,379	2,895	46,092	6,419

CCP33. This model has 297 levels, 97,451,014 states and 1,061,619,779 transitions. It describes and instance of the cache coherence protocol *Jackal* for Java programs with 3 processes and 3 threads [22].

The main goal of this test was to compare the performance of our symbolic prototype and the sequential state space generator of the μCRL toolset. However, we also wanted to have an indication of the memory cost of the operations on decision diagrams. Thus, we implemented a mixed state space generator. This mixed tool is a direct implementation of reachability in which we enumerate the successors of each state explicitly, but which uses the symbolic set structure instead of a hashtable for the set of visited states and the level sets.

All three tools were using version 2.18.0 of the μCRL toolset. To run these tests, we used a server with dual Intel Xeon X5365 (3.00GHz) processor and an Intel 5000P chipset with 64GB memory. We ran one experiment at a time.

The problems as they were formulated for the testing of the distributed tools, could only be dealt with by the symbolic tool in 3 out of 5 cases. The remaining cases ran out of memory. To fix this, we reordered the variables using the heuristic that the distance between variables that interact should be low.

The data collected in this test is summarized in Table 9. For each of the tools, we have two columns: time (in seconds) and maximum memory (in kB). The first column of the table indicates the problem, the second column contains the variable order, where G means given and R means reordered.

The given variable ordering of the franklin problem was first all processes then all channels. We changed this to process variables and channel variables interleaved, either starting with a process (R1) or starting with a channel (R2). While the reordering didn't affect the mixed tools very much, it did have a large influence on the symbolic tool. It is left as future work to find out if a reason for this difference can be found.

To compare the performance of the various tools, we compare the best runs for each tool within each of the groups for lift5, swp, 1394fin, franklin and ccp33. For memory the score (symbolic vs mixed vs explicit) is 1-4-0. For time the score is 3-0-2.

What can we learn from these experiments? Looking at the data of the first test, it seems that the bigger the model the better the performance in both states/sec and mem/state of the symbolic tools. This is a clear improvement over the explicit tools where performance usually decreases slightly as the models grow. It should be noted however, that this trend is broken for the three largest instances of SWP. We think that this is due to performance issues in our symbolic set implementation, but that is a conjecture only.

From the second test, we get some data to compare approaches. If we compare the memory usage of the symbolic tool with the mixed tool then we see that with one exception, the mixed tool uses less memory. This is to be expected because the mixed tool uses the symbolic sets for storage only, whereas the symbolic tool computes with these sets, which requires additional memory for the operation cache and intermediate results. If we compare the mixed tool and the explicit tool on time then the mixed tool looses. This is not surprising because insert/lookup is much more expensive for the symbolic set than for the hash table set. Comparing the symbolic tool with the explicit tool, then we get a mixed result. In two cases (lift5,franklin) the results do not differ substantially. In the other three cases the symbolic tool is substantially better in memory and time (1394fin), substantially worse in time but better in memory (swp) and substantially worse in both time and memory (ccp33). The conclusion is that the symbolic tool even in its current form is a very useful addition, but not capable of replacing the explicit tool. Replacing the explicit tool will not be an option for some time anyway, as there are a number of issues left open, which will be discussed in the next section.

7 Conclusion

In this section, we discuss some of the open issues that need to be solved to make the symbolic tool more useful and summarize the results.

On the implementation side, the most important task is to replace the current ATerm based symbolic set implementation with a much higher performance decision diagram package. This is needed not only for performance, but also to allow our back-end to interface with other modeling formalisms. For example, we plan on writing an interface to NIPS (see [23]) which is a virtual machine which is compatible with SPIN (see [24]). To do this, we also need to extend the implementation from fixed vector length to variable vector length.

Independently, we will investigate the effects of the search order on the size of the intermediate structures. So far, we have used a breadth first search (BFS) strategy. One of the strategies we need to look at is saturation. Ciardo et al. have shown that this strategy is much better for Petri Nets. As explanation they state that it reduces the difference between the size (in numbers of nodes) of the reachable state space and the peak size of the set of visited states considerably. If we look at the peak/final differences for our typical models then we find they are small. Thus, it is not a priori clear if saturation will work well.

By implementing a symbolic tool for a modeling language that has until now had support for explicit exploration only, we are in a situation where we can compare the performance of explicit and symbolic tools. One of the things that we need to do is see how far we can get with applying the symbolic tool to existing models. We have started this in the second test by looking at five models, but all five of the problems were message passing systems. Such systems have a high degree of locality. This is not always the case. In timed systems one usually has global synchronisation steps which involve nearly the entire state. We will study examples with global steps in order to find common features that allow an efficient embedding into our tool.

Another issue is that so far, we have written models in a way that is optimized for explicit enumeration. For example consider the 8-queens problem. Finding all solutions using the first algorithm (directed search) requires looking at 2,058 states. If we use the second algorithm (random moves) we get 16,777,216 states. Using explicit tools we would never consider the second approach for practical purposes. However, the worst size of the diagram representing the set of states for the first approach is 2,655 nodes and the worst size for the second approach is 166. And there are many other modeling techniques that are good for explicit exploration but bad for symbolic techniques, such as path reduction by making sequences of steps atomic.

We have shown that a generalized version of the conjunctive/disjunctive partitioning scheme implemented in SMART for Petri nets can be successfully extended to allow non-deterministic transitions and implemented for the process algebra μCRL. The initial results with the prototype show that symbolic exploration should be a part of the future of process algebra's, but also that retiring the explicit tools is not an option yet.

References

1. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
2. Fernandez, J.C., Garavel, H., Kerbrat, A., Mounier, L., Mateescu, R., Sighireanu, M.: CADP - A Protocol Validation and Verification Toolbox. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 437–440. Springer, Heidelberg (1996)
3. Roscoe, B.: The theory and practice of concurrency. Prentice-Hall, Englewood Cliffs (amended, 1998) (2005)
4. Blom, S., Fokkink, W., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.: μCRL: A Toolset for Analysing Algebraic Specifications. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 250–254. Springer, Heidelberg (2001)
5. Ciardo, G., Miner, A.S.: SMART: The Stochastic Model checking Analyzer for Reliability and Timing. In: QEST, pp. 338–339. IEEE Computer Society, Los Alamitos (2004)
6. Ciardo, G., Yu, A.J.: Saturation-Based Symbolic Reachability Analysis Using Conjunctive and Disjunctive Partitioning. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 146–161. Springer, Heidelberg (2005)

7. Kuntz, M., Siegle, M.: Deriving Symbolic Representations from Stochastic Process Algebras. In: Hermanns, H., Segala, R. (eds.) PROBMIV 2002, PAPM-PROBMIV 2002, and PAPM 2002. LNCS, vol. 2399, pp. 188–206. Springer, Heidelberg (2002)
8. Kuntz, M., Siegle, M., Werner, E.: Symbolic Performance and Dependability Evaluation with the Tool CASPA. In: Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F. (eds.) FORTE 2004. LNCS, vol. 3236, pp. 293–307. Springer, Heidelberg (2004)
9. Ciardo, G., Marmorstein, R.M., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. STTT 8, 4–25 (2006)
10. Bryant, R.E.: On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. IEEE Trans. Computers 40, 205–213 (1991)
11. Blom, S., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.: New developments around the μCRL tool set. In: Arts, T., Fokkink, W. (eds.) Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2003). ENTCS, vol. 80 (2003)
12. Brand, M.G.J.v.d., Jong, H.A.d., Klint, P., Olivier, P.A.: Efficient Annotated Terms. Software – Practice & Experience 30, 259–291 (2000)
13. Kimura, S., Clarke, E.: A parallel algorithm for constructing binary decision diagrams. Computer Design: VLSI in Computers and Processors. Proceedings. ICCD 1990, 220–223 (1990)
14. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers 35, 677–691 (1986)
15. Groote, J.F., Pang, J., Wouters, A.G.: A Balancing Act: Analyzing a Distributed Lift System. In: Gnesi, S., Ultes-Nitsche, U. (eds.) Proc. 6th Workshop on Formal Methods for Industrial Critical Systems, pp. 1–12 (2001)
16. Badban, B., Fokkink, W., Groote, J.F., Pang, J., van de Pol, J.: Verification of a sliding window protocol in μCRL and PVS. Formal Aspects of Computing 17, 342–388 (2005)
17. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A database approach to distributed state space generation. In: Haverkort, B., Černa, I. (eds.) Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verification, vol. 198 (2007)
18. Luttik, S.: Description and formal specification of the link layer of P1394. In: Technical Report SEN-R9706, Amsterdam, The Netherlands (1997)
19. Sighireanu, M., Mateescu, R.: Verification of the Link Layer Protocol of the IEEE-1394 Serial Bus (FireWire). An Experiment with E-LOTOS. STTT 2, 68–88 (1998)
20. Bakhshi, R., Fokkink, W., Pang, J., van de Pol, J.: Leader Election in Anonymous Rings: Franklin Goes Probabilistic. In: Accepted for 5th IFIP International Conference on Theoretical Computer Science (2008)
21. Franklin, W.R.: On an Improved Algorithm for Decentralized Extrema Finding in Circular Configurations of Processors. Commun. ACM 25, 336–337 (1982)
22. Pang, J., Fokkink, W.J., Hofman, R.F., Veldema, R.: Model checking a cache coherence protocol of a Java DSM implementation. JLAP 71, 1–43 (2007)
23. Weber, M.: An Embeddable Virtual Machine for State Space Generation. In: Bosnacki, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 168–186. Springer, Heidelberg (2007)
24. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2003)