Design and Evaluation of a Symbolic and Abstraction-Based Model Checker

Serge Haddad[1], Jean-Michel Ilié[2], and Kais Klai[2]

[1] Lamsade CNRS UMR 7024 Université de Paris Dauphine, Place du Maréchal de Lattre de Tassigny 75775 Paris Cedex 16 France
contact author tel: 33 1 44 05 41 20
haddad@lamsade.dauphine.fr
[2] Lip6 laboratory, Paris 6 University, 8 Rue du Capitaine Scott, 75015 Paris, France
{Jean-Michel.ilie,Kais.Klai}@lip6.fr

Abstract. Symbolic model-checking usually includes two steps: the building of a compact representation of a state graph and the evaluation of the properties of the system upon this data structure. In case of properties expressed with a linear time logic, it appears that the second step is often more time consuming than the first one. In this work, we present a mixed solution which builds an observation graph represented in a non symbolic way but where the nodes are essentially symbolic set of states. Due to the small number of events to be observed in a typical formula, this graph has a very moderate size and thus the complexity time of verification is neglectible w.r.t. the time to build the observation graph. Thus we propose different symbolic implementations for the construction of the nodes of this graph. The evaluations we have done on standard examples show that our method outperforms the pure symbolic methods which makes it attractive.

Keywords: OBDD, Model Checking, Abstraction

1 Introduction

Checking properties of a dynamic system often leads to the building of a graph corresponding either to the state graph of the system or to some synchronized product of it with an automaton. In both cases, one has to tackle with the state explosion problem, i.e. the exponential increasing in the number of states w.r.t. the number of system's components.

Among the numerous techniques proposed to cope with such an explosion, the ordered binary decision diagrams (OBDDs) approach can be described as follows [1,2]. Each potential state is viewed as a vector of boolean variables by choosing the appropriate variables describing the system. Then the set of reachable states is equivalent to the boolean function which returns true iff the input vector corresponds to a reachable state. The boolean expression associated with the function can now be represented in a compact way by factorising the multiple occurences of the same subexpression. Hence the final structure is a rooted directed acyclic graph (DAG) where the subgraph rooted at each node corresponds

F. Wang (Ed.): ATVA 2004, LNCS 3299, pp. 196–210, 2004.

to a subexpression and the root corresponds to the function to be represented. In case of multiple functions there is one "root" per different function (here the structure is sometimes called shared OBDD).
The benefit of OBDDs comes from the fact that a small OBBD can often represent a huge set of states, and the "symbolic" operations like the set operations (union, intersection, complementation) and the membership test are cheap as long as the OBDDs are small. Equally important are the operations associated to an event of the system and a set of states: the subset of states for which this event is enabled, the "image" of this set obtained by the occurrence of the event and the "preimage" of the set i.e. the set of states where the occurrence of this event leads to a state of the specified set. Generally these latter algorithms have a time complexity proportionnal to the size of the OBDD on which they are applied.

Once the OBBD has been built, the reachability problem is straightforwardly answered by the membership test. However checking a temporal formula requires a more complex algorithm. It is shown in [3] that checking CTL and LTL formulae can be essentially reduced to the search of particular cycles in either the reachability graph or in some synchronized product. Thus the key point for an efficient OBDD-based formulae checking is the design of an OBDD-algorithm for the search of such cycles. The earliest algorithm is known as the Emerson-Lei algorithm [4]. Its worst case time complexity is quadratic w.r.t. the size of the reachability space (n). So different improvements have been proposed and analyzed [5]. With a relaxed definition of a symbolic algorithm, the worst case time complexity has been first reduced to $\mathcal{O}(n \log(n))$ [6], then to $\mathcal{O}(n)$ [7]. However since these algorithms repeatedly build sets reduced to a singleton, their empirical complexity is often worse than the Emerson-Lei algorithm. In [3], the authors study variants of Emerson-Lei algorithm (e.g. CTY, OWCTY) with the same worst case time complexity but outperforming it on practical examples.

In this paper, we study the checking of an evenemential linear time formula (formulas over events). We propose an hybrid method which builds:

- a standard representation for the observation graph i.e. the abstraction of the reachability graph w.r.t. the events occurring in the formula,
- an OBDD representation of each node of this graph which is indeed the closure of some set of states under the occurrences of unobserved events.

Once this structure is obtained, a standard model-checking algorithm is applied on the observation graph. Even in case of a huge reachability graph, the observation graph is quite small and the execution time of this last step is negligible w.r.t. the execution time of the building of the observation graph.

So we have paid attention to an efficient building of the observation graph. The critical factor is the "accumulated" size of the OBDD corresponding to the sets associated to each node of this graph. Our goal is to reduce both the number of the nodes of the graph and the size of the set of states associated to each node.

- Two nodes corresponding to different sets of states can be regarded as equal if they have the same subset of states enabling observed events (thus the same

successors) and the same behavior w.r.t. the deadlock and the divergence properties.
- Once a new node is built, we substitute to the corresponding set of states, the subset consisting of one representant per initial strongly connected component (SCC) of the subgraph spanned by this set. Indeed these subsets are sufficient to check equality between the original sets. We call this step the canonization.

The evaluations on typical cases (see section 3) show that the size of this structure is either of the same magnitude order of the size of the OBDD of the reachablity graph or even of a smaller order. More importantly the maximum size of the intermediate OBDDs is often much smaller than the corresponding size for the reachability graph. Thus w.r.t. the space complexity our method achieves its goal.

The critical procedure w.r.t. the time complexity is the canonization step. We have observed that a standard symbolic search algorithm of initial SCCs of a graph (see for instance [8,6]) may have a bad time complexity. Thus we have developed a specific procedure which takes advantage of the parallelism of the system under observation. For this kind of systems, our procedure outperforms the standard procedure.

The paper is organized as follows. In the second section, we briefly introduce the model-cheking problem and we develop our method describing and analyzing the main algorithms. In the third section, we evaluate our method on a benchmark of problems. We also briefly discuss how to exhibit counter-examples when the formula is invalidated. At last, we conclude and give some perspectives to this work.

2 The Observation Graph

2.1 Model Checking of an Evenemential Linear Formula

We introduce here the context of our model-checking problem. The system we want to study is given by:

- A state description represented by a fixed vector of boolean variables. An initial state is associated to the system.
- A finite set of events T. To each event is associated an identifier, an enabling predicate on states and a tranformation function.

We suppose that the dynamics of the system can be symbolically evaluated by the following operations: $Img(S,t)$ which returns the set of immediate successors of the states of S by the occurrence of the event t and $Preimg(S,t)$ which returns the set of immediate predecessors of the states of S by the occurrence of the event t. We derive from these operations and the boolean ones, other useful operations with straightforward interpretations:

$$Img(S,T') = \bigcup_{t\in T'} Img(S,t)$$
$$Preimg(S,T') = \bigcup_{t\in T'} Preimg(S,t)$$
$$Enab(S,T') = S \cap Preimg(Img(S,T'),T')$$

$Enab(S,T')$ selects all states in S which have (at least) one successor by occurrence of some event in T'.

Our method does not rely on the particular syntax and semantic of a temporal logic over events but we make two assumptions. The first hypothesis is that given the events T' occurring in the formula ϕ, the satisfaction of ϕ by a sequence of events σ depends only on the observed sequence $Proj(\sigma,T')$, the projection of σ on the events of T' (the sequence obtained by removing, from σ, all events not in T').

So our method builds a finite automaton called the observation graph which includes enough information to express three kinds of sequences of the system (i.e. three languages).

- $L_{inf}^{T'} = \{\sigma' = Proj(\sigma,T') \in T'^{\infty} \,|\, \sigma \text{ is an execution sequence}\}$ the infinite observable sequences,
- $L_{max}^{T'} = \{\sigma' = Proj(\sigma,T') \,|\, \sigma \in T^* \text{ is a maximal execution sequence}\}$ the finite maximal sequences,
- $L_{div}^{T'} = \{\sigma' = Proj(\sigma,T') \in T'^{*} \,|\, \sigma \in T^{\infty} \text{ is an execution sequence}\}$ the divergent sequences.

Then the second hypothesis is that such a formula could be checked by the search for particular paths in some appropriate synchronized product between the observation graph and an automaton related to the formula. Such an hypothesis is very standard and covers the case of formula languages like LTL or the linear time μ-calculus.

In conclusion, by preserving the above three kinds of sequences, the obtained observation graph preserves the validity of formulas written in classical Manna-Pnueli linear time logic [9] (LTL) from which the "next operator" (X) has been removed (see [10,11]). This logic is extremely important in verification of concurrent systems. Even if Manna-Pnueli linear time logic is state-based logic, interpretation of this logic in an event-based (or action-based) setting is possible. This can be done in more than one way. An alternative interpretation that is perhaps more relevant for practical verification than the original one was given in [12] (more easily found in [13] pp. 498 – 499).

2.2 Algorithms

The algorithm **BuildOG** builds the (deterministic) observation graph related to an initial state s_0 and a set of observable events $Obs \subseteq T$ (T is the set of all events of the system). The data structures manipulated by the algorithm are the following ones:

- a shared OBDD which contains symbolic representations of subsets of reachables sets,

Algorithm 2.1 Building of the observation graph

```
 1: BuildOG (state s_0 , Events Obs)
 2: set S'; vertex v,v';
 3: Vertices V; Edges E;
 4: Events Unobs = T \ Obs; stack st;
 5: S' = Saturate({s_0}, Unobs);
 6: v.dead = DetectDead(S');
 7: v.loop = DetectLoop(S', Unobs);
 8: v.set = Reduce(S', Unobs);
 9: V = {v}; E = ∅;
10: st.Push(⟨v, S'⟩);
11: repeat
12:    st.Pop(⟨v, S⟩);
13:    for t ∈ Obs do
14:       S' = Img(S, t);
15:       if (S' ≠ ∅) then
16:          S' = Saturate(S', Unobs);
17:          v'.dead = DetectDead(S');
18:          v'.loop = DetectLoop(S', Unobs);
19:          v'.set = Reduce(S', Unobs);
20:          if (∃w ∈ Vs.t.w == v') then
21:             E = E ∪ {v -t-> w};
22:          else
23:             V = V ∪ {v'};
24:             E = E ∪ {v -t-> v'};
25:             st.Push(⟨v', S'⟩);
26:          end if
27:       end if
28:    end for
29: until st == ∅;
```

```
 1: Saturate(set S, Events Unobs)
 2: set From, Reach, To;
 3: From = S; Reach = S;
 4: repeat
 5:    To = Img(From, Unobs);
 6:    From = To \ Reach;
 7:    Reach = Reach ∪ To;
 8: until From == ∅;
 9: return Reach;
```

```
 1: DetectDead(set S)
 2: return Enab(S, T) ≠ S;
```

```
 1: DetectLoop(set S, Events Unobs)
 2: set From, Reach;
 3: From = S;
 4: repeat
 5:    Reach = Img(From, Unobs);
 6:    if Reach == From then
 7:       return TRUE;
 8:    end if
 9:    From = Reach;
10: until Reach == ∅;
11: return FALSE;
```

– a standard graph representation with a set of vertices (V) and a set of edges (E). Three attributes are associated to a node v: a symbolic subset of states $v.set$ characterizing the behaviour of the system starting from this node, $v.loop$ a boolean indicating that any sequence of observable events leading to this node is the projection of a divergent sequence and $v.dead$ a boolean indicating that any sequence of observable events leading to this node is the projection of a finite maximal sequence,
– a stack whose items are tuples composed by a node of the graph and a symbolic subset of states (the interpretation of this set is given below).

The initialization step of the algorithm (lines 5 – 9) allows to compute the first (initial) node of the observation graph (lines 5 – 8) and to initialize the graph structure (line 9). An iteration of the main loop consists in picking and processing an item $\langle v, S\rangle$ of the stack until it is empty. The goal of the iteration is to generate the successors of the current node in the observation graph. The set of

states S corresponds to the states reached by any sequence of observed events leading from the initial node of the observation graph to v. Thus one successively computes the image S' of S by each observed event. If S' is not empty, it generates a new edge of the observation graph labelled by the event.

Now one must check whether the node reached by this edge is a new one. So we compute the different attributes of this node. At first, we compute the closure of S' under the action of the unobserved events (via **Saturate**). Then we check the existence of dead states with the help of the symbolic operation *Enab* (via **DetectDead**) and the existence of a loop of unobserved events by applying a kind of topologic sort of the underlying graph of S' (via **DetectLoop**). At last, we compute from S' a subset of states characterizing the observable behaviour starting from S' (via **Reduce**).

Finally, we look for an identical node in the graph. If such a node is not present we add a new node in the graph and push it on the stack with S'. In fact, we could avoid to push S' and retrieve the significant information from $v.set$ but this would complicate the presentation.

We devote the next subsection to the presentation of **Reduce** since its applications lead to important memory savings but may involve a large additional computation time.

2.3 Canonization

In this section, we detail the *Reduce* function which extracts from a set of states S, a subset sufficient to characterize the observed behaviour starting from S. At first, since S is closed under the action of the unobserved events, we can restrict S to the subset of states enabling any observed event. In practice, this first reduction does not lead to memory savings perhaps because such a set has a more irregular structure than the original one w.r.t. the OBDD representation. So we will not present evaluations of this variant.

Taking the unobserved events as edges, S may be viewed as a graph and it is sufficient to extract one representant per initial SCC of this graph in order to preserve the observed behaviour starting from S.

Whereas a symbolic search for all the SCCs of a graph is a theoretical issue (see [8,6,7]), the search of initial SCCs can easily be done in a number of operations proportional to the number of states as shown by the algorithm **SCAN**. Each iteration of the external loop, starting from a single state Max, computes its forward closure F and then begins to compute its backward closure B. As soon as B is no more included in F we know that Max does not belong to an initial SCC, so we prune F from the current set R and we start a new iteration with a state which is a predecessor of Max ($Maxpick$ extracts a singleton set reduced to the maximal state of a set w.r.t. the lexicographic order induced by the variables of the OBDD). In the other case, B is an initial SCC including Max, so we add its representant to the set of representatives, we prune F and we start a new iteration with an arbitrary remaining state.

Unfortunately, this algorithm has a bad empirical time complexity much greater than the complexity of the saturation. Thus we have designed a new

Algorithm 2.2 Standard and dichotomic canonization

```
1: SCAN (set S, Events Un)
2: set R, F, B, Frt, Max, Repr, Pred;
3: R = S; Repr = ∅; Max = MaxPick(R);
4: repeat
5:    F = Max;
6:    Frt = F;
7:    repeat
8:       Frt = (Img(Frt, Un) ∩ R) \ F;
9:       F = Frt ∪ F;
10:   until Frt == ∅;
11:   B = Max;
12:   Frt = B;
13:   repeat
14:      Frt = (Preimg(Frt, Un) ∩ R) \ B;
15:      B = Frt ∪ B;
16:      Pred = Frt \ F;
17:   until (Frt == ∅) or (Pred ≠ ∅);
18:   R = R \ F;
19:   if (Frt == ∅) then
20:      Repr = Repr ∪ MaxPick(B);
21:      Max = MaxPick(R);
22:   else
23:      Max = MaxPick(Pred);
24:   end if
25: until (R = ∅);
26: return Repr;
```

```
1: DCAN (set S, Events Un, int i)
2: set S[0..1], Front, Reach, Repr;
3: S[1] = S ∩ ite(x_i, 1, 0);
4: S[0] = S \ S[1];
5: if (S[0] ≠ ∅) and (S[1] ≠ ∅) then
6:    Front = S[1]; Reach = S[1];
7:    repeat
8:       Front = Img(Front, Un) \ Reach;
9:       Reach = Reach ∪ Front;
10:      S[0] = S[0] \ Front;
11:   until (Front == ∅) or (S[0] == ∅);
12: end if
13: if (S[0] ≠ ∅) and (S[1] ≠ ∅) then
14:   Front = S[0]; Reach = S[0];
15:   repeat
16:      Front = Img(Front, Un) \ Reach;
17:      Reach = Reach ∪ Front;
18:      S[1] = S[1] \ Front;
19:   until (Front == ∅) or (S[1] == ∅);
20: end if
21: i++; Repr = ∅;
22: for j from 0 to 1 do
23:   if size(S[j]) ≤ 1 then
24:      Repr = Repr ∪ S[j];
25:   else
26:      Repr = Repr ∪ DCAN(S[j], Un, i);
27:   end if
28: end for
29: return Repr;
```

algorithm **DCAN** adapted to the parallel systems. This recursive algorithm splits the set of states into two subsets ($S[1]$ and $S[0]$) w.r.t. the value of the current variable (x_i, with i initially equal to 1). It prunes from the second subset all the states which are in the forward closure of the first subset. Such a deleted state either does not belong to an initial SCC or the representant of its SCC is in the first subset. Now $S[0]$ contains states not reachable from $S[1]$. We now eliminate the states of $S[1]$ reachable from $S[0]$ since they do not belong to an initial SCC. After this double reduction, both these subsets may be independently analyzed in order to find the representatives of the initial SCCs. This leads to at most two recursive calls. Of course when a set is a singleton, we have found a representant.

A last improvement is possible: an initial SCC obviously contains a state of S' in line 14 in the main algorithm before its saturation in line 16. Thus we can

restrict our search to the intersection of the backward and the forward closure (by the unobserved events) of S' before its saturation.

Even if **DCAN** has a worse theoretical complexity than **SCAN** (quadratic versus linear), it is more convenient to parallel systems. To explain the former assertion, let us illustrate the time complexity (number of *img* and *preimg* executions) of these two algorithms on the toy parallel program $x_1 = true | x_2 = true | \ldots | x_m = true$ with all the variables initialized to false. Depending on the number of variables, the state space of such a program can be illustrated by an hypercube. When $m = 3$, the state space is given by one of the two cubes of the figure depending on the encoding of *true* by 1 or 0. This would modify the initial choice of Max in the *SCAN* algorithm which has a time complexity either linear $(m + 2)$ or exponential (2^{m+1}) w.r.t. m, whereas *DCAN* has in both cases a linear complexity. We give below an informal proof of these claims.

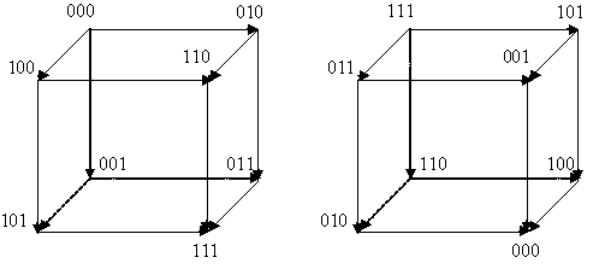

Fig. 1. Possible State space graphs of a parallel program

Let first consider the *SCAN* algorithm.

- $true = 1$: The canonization is performed in as many iterations as the number of the cube nodes (2^m). At each step a maximum node is picked (111, 110, 101, ..., 000 successively). By performing once *img*, leading to now successor, followed by one *preimg* operation, leading to some predecessors which are not successors, the picked node is removed from the whole set. The total number of *img* and *preimg* operations is hence $2 * n$ where $n = 2^m$ is the number of the cube nodes.
- $true = 0$: The canonization is here performed in the first iteration. By taking the maximal node 111, we perform three *img* steps to reach all cube nodes (plus one saturation execution), then one *preimg* is sufficient to know that this node has no predecessor. All successors are so moved and 111 is selected as the representant of this elementary SCC.

We consider now the *DCAN* algorithm:

- In both cases ($true = 1$ and $true = 0$): The canonization is performed in $m = 3$ iterations. At each step the current state space is split into two equal subsets. At the first iteration, taking states where v_1 has true value (i.e. nodes belonging to one face of the cube) leads to move in one img step all remaining nodes (i.e. all nodes belonging to the opposite face). The same thing is done in the two next iterations by removing in one step the $n/2$ nodes from the n existing ones. In conclusion, if n is the number of the cube nodes, then we need to accomplish $log_2(n) = m$ img steps in order to canonize the hole space state.

Note that when $m > 3$ the proof is identical.

3 Evaluation of the Observation Method

In this section, we report the results obtained with the observation graph compared to those obtained with a symbolic approach both for the building and the analysis of the state space. We also compare the two algorithms of canonization described in section 2. In general, time and memory consumption is very sensitive to the implementation details of the OBDD tool. Here, our goal is not to reach performances of existent tools [14,15], but we want to demonstrate that, given some software, the construction of the observation graph decreases time and space consumption with respect to an OBDD-based construction of some state space followed by a symbolic search of fair cycles. Therefore, we have developed our algorithms with the free package BuDDy [16] for both the symbolic observation graph construction and the whole state space construction. The implementation of the Img and $Preimg$ operations is similar to the one of [17]. All the tested examples are parameterized and the size of the reachable states space is exponential with respect to the parameter value. For each example, there are two observed events occurring in a linear time formula expressing a fairness property.
The following three examples were used:
1. Dining philosophers: The system consists of a ring of n dining philosophers. Each philosopher has two variables that model the state of his left and right forks (up or down). A philosopher first picks up his left fork, then his right, then, after he finishes eating, puts down his left, and finally his right, returning to his initial state. A fork can only be picked up if the neighbor that shares the fork is not using it.
2. Distributed database: The system consists of a database distributed among n sites (each site has a copy of the database). Its modifications are done in mutual exclusion. After such an operation, the site broadcasts its transaction. Upon reception, the other sites update their local copy and send back a grant message. When all the grants are received, the database is released.
3. Slotted ring: The system models a protocol for Local Area Networks called *slotted ring* [17]. The network is composed of n vertices. Each node shares two events with his neighbor: $Free_{(i+1) \bmod n}$ and $Used_{(i+1) \bmod n}$.

Table 1. Experimental results

	n	# state	OBDD			SOG								
			# vert	# op	CPU time	# MS	# edges	# rep	# op	# vert	Std. Canonisation		Dich. Canonisation	
											Const (sec)	Can (sec)	Const (sec)	Can (sec)
Philo	2	22	55	6	0	2	2	2	13	18	0	0	0	0
	5	$2\ 10^3$	175	13	0	2	2	2	21	39	0	0	0	0
	10	$4\ 10^6$	375	26	2	2	2	2	34	74	12	10	1	0
	20	$2\ 10^{13}$	775	51	51	2	2	2	59	144	349	326	30	7
	30	$1\ 10^{20}$	1175	76	390	2	2	2	84	214	2578	2425	186	33
BD	2	7	32	4	0	2	2	2	11	21	0	0	0	0
	5	406	104	10	0	2	2	2	29	51	0	0	0	0
	10	$1\ 10^5$	224	20	0	2	2	2	59	101	1	0	0	0
	20	$2\ 10^{10}$	464	40	6	2	2	2	119	201	46	33	24	11
	30	$2\ 10^{15}$	704	60	35	2	2	2	179	301	278	198	150	68
Slotted Ring	2	52	72	11	0	2	3	2	28	26	0	0	0	0
	5	$5\ 10^4$	349	39	1	2	3	2	83	74	7	6	4	3
	10	$8\ 10^9$	1280	115	377	2	3	2	215	154	-	-	888.58	759

The table 1 includes performance results obtained for both the symbolic reachability space and the observation graph constructions. Its first column specifies the system parameter (i.e. the number of components). The second one lists the size of the reachability space. The remaining columns are divided into subsets of columns corresponding to measurements for the standard OBDD algorithm and the observation graph. The comparison criteria are the number of symbolic operations, the CPU time and the number of OBDD vertices. In addition, we give for each observation graph the number of its vertices and edges and the sum of the sizes of the sets associated to each vertex (i.e. $\Sigma_v size(v.set)$). The last four columns compare the CPU time of the construction depending on the used canonization algorithm.
The analysis of this table brings out the following statements:

- The size of the OBDD associated to the observation graph is significantly smaller than the size of the OBDD associated to the reachability space (e.g. 18% for 30 philosophers, 4% for 10 nodes in the slotted ring, 25% for 30 copies of the database).
- For the three examples, the size of the observation graph is independent of the value of the parameters. More generally, other experimentations have confirmed that the size of this graph is neglictible w.r.t. the OBDD size and grows slowly w.r.t. the parameter.
- The computation of the initial SCCs leads to drastic differences depending on the use of **SCAN** or **DCAN**. The latter has a smaller time consumption

than the former (1% for 30 philosophers, 34% for 30 copies of the database). Its time consumption has the same order as the symbolic building of the reachability space.

- There are more symbolic operations for the observation graph than for the reachability space. Combined with the previous statement, one concludes that the mean size of the manipulated OBBDs is smaller for the first one.

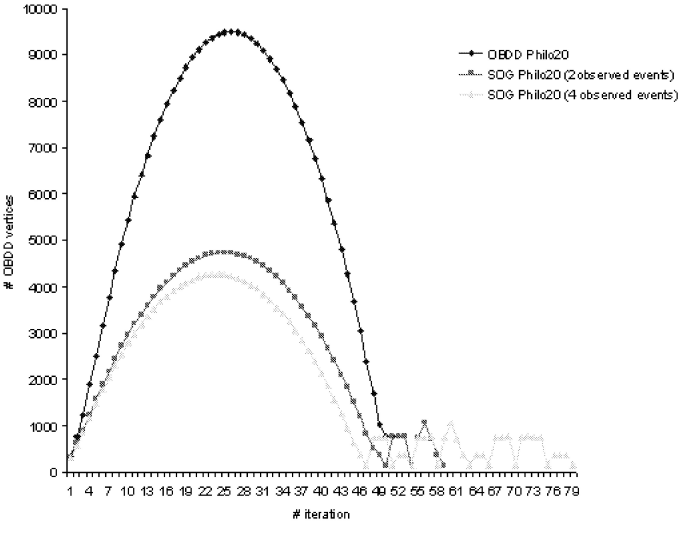

Fig. 2. Evolution of intermediary OBDDs sizes for 20 dining philosophers

It is well known that the ratio between the maximal size of the OBDD during the computation and its final size may be important. So we have analyzed the size of these intermediary OBDDs. The figure 2 depicts the evolution of the intermediary OBDD sizes for the dining philosophers example. The first curve is related to the symbolic reachability space building while the two other ones correspond to the observation graph (one with two observed events and the other with four events). We point out that the size of the intermediary OBDDs are also reduced by our algorithm and depends on the number of observed events. Generally increasing the number of observed events leads to smaller intermediary OBDDs.

Finally, we compare the complete verification process using the observation graph with a whole symbolic model-checking. We have chosen two robust and efficient symbolic algorithms: the Emerson-Lei and OWCTY algorithms [3]. The considered formula expresses that a philosopher will never indefinitely wait to eat.

Table 2. Model checking : SOG *vs* Emerson-Lei and OWCTY algorithms

n	#states	Symbolic observation graph						OBDD			Cycle detection	
		#meta states	#edges	# rep	#BDD vert	Max bdd size	CPU time	# vert	Max bdd	CPU time	EL (CPU Time)	OWCTY (CPU Time)
2	36	2	1	2	32	66	0.00	90	90	0.01	0.00	0.00
5	$3\,10^3$	2	1	2	74	509	0.07	326	804	0.06	0.23	0.11
10	$7.7\,10^6$	2	1	2	144	2252	2.39	726	4072	3.47	13.72	5.48
20	$3.6\,10^{13}$	2	1	2	284	9499	58.70	1526	18119	100.12	228.39	88.03
30	$1.6\,10^{20}$	2	1	2	424	21742	421.57	2326	42152	759.50	1082.33	427.91

The table 2 lists the measurements. Since the complete symbolic approach builds an OBDD representation of a synchronized product (thus different from the reachability space of the system), we have taken into account space and time comparison criteria.

First the construction of the observation graph consumes less CPU time and less memory than the construction of the symbolic synchronized product. Moreover, the CPU time consumed by the OWCTY algorithm (resp. Emerson-Lei) may be as large as (resp. two times larger than) the one of the whole state space construction.

4 Search of a Counter Example

Once a given property has been proven not to hold on a system, the modeller needs to modify its design. In order to help this process, providing a counter example is useful. It should be clear that the verification process based on the observation graph exhibits a sequence of observed events invalidating the formula. Such a sequence σ corresponds to a path in the observation graph. Sometimes, this is sufficient for the engineer to correct the design of its system. However, when the abstraction induced by the unobserved events is too strong, the modeller needs a sequence of the system whose projection on the observed events is σ.

We first handle the case of a finite maximal sequence which corresponds to a path $v_0 \xrightarrow{o_1} v_1 \xrightarrow{o_2} \dots v_n$ with $v_n.deadlock = true$. At first, we develop again the sets of states S_i corresponding to the nodes v_i and we also memorize S'_i the subset of S_i corresponding to the set S' in line 14 of the main algorithm. We call S'_i the entry points of S_i.

Using **DetectDead**, one determines the set of dead states S_{dead} included in v_n. We compute the backward closure of S_{dead} stacking the different fronts until we reach at least one entry point of S_n, say s_{in}. Then we compute an explicit subsequence from s_{in} leading to a dead state through the fronts which are popped from the stack. With the $Preimg(\{s_{in}\}, o_n) \cap S_{n-1}$ operation, we obtain a transition $s_{n-1} \xrightarrow{o_n} s_{in}$ with $s_{n-1} \in S_{n-1}$. Iterating this backward process starting from s_{n-1} we eventually reach s_0 and build the searched sequence.

We now handle the case of an infinite observable sequence which corresponds to a path $v_0 \xrightarrow{o_1} v_1 \xrightarrow{o_2} \dots v_m \dots v_n$ with $v_m = v_n$. We detail the search of a circuit which passes a finite number of times (not necessarily 1) through $v_m, v_{m+1}, \dots, v_n$. As shown on the figure 3, to the circuit $S_l \xrightarrow{o_1} S_m \xrightarrow{o_2} S_l$ in the observation graph corresponds for instance the circuit $s_1, s_5, s_8, s_{11}, s_2, s_6, s_9, s_{12}, s_3, s_1$ in the state graph.

We explain our algorithm on this example. We encode sets of couples of states $C = \{\langle s, s' \rangle\}$ with an OBDD representation where each variable is duplicated. We transform the image operation as follows: $Img(C, t) = \{\langle s, s" \rangle | \exists \langle s, s' \rangle \in C$ **and** $s' \xrightarrow{t} s"\}$. In other words, the image operates on the right item. We handle a sequence of sets $C_0, C'_0, C_1, C'_1, \dots$ where $\langle s, s' \rangle \in C_i$ if $s' \in S_l$ is reachable from $s \in S_l$ by a sequence whose projection on the observed events is $(o_1 o_2)^i$ and where $\langle s, s' \rangle \in C'_i$ if $s' \in S_m$ is reachable from $s \in S_l$ by a sequence whose projection on the observed events is $(o_1 o_2)^i o_1$.

We start with $C_0 = \{\langle s, s \rangle | s \in S_l\}$. Then we iterate simultaneously on the C_i and the C'_i as follows.

$$C_0 = C_0 \cup Img(C_0, Unobs) \;\; C_{i+1} = C_{i+1} \cup Img(C_{i+1}, Unobs) \cup Img(C'_i, o_2)$$
$$C'_i = C'_i \cup Img(C'_i, Unobs) \cup Img(C_i, o_1)$$

We stop as soon as some C_i for $i > 0$ contains a couple $\langle s_{loop}, s_{loop} \rangle$. Now we restrict the sets H_j, H'_j to items $\langle s_{loop}, s \rangle$ and we project them on their second component obtaining sets that we denote H_j, H'_j. Then we apply a construction for *sloop* similar to the one for s_{dead} through the sets $H_0, H'_0, \dots, H'_{i-1}$.

At last, the construction of the prefix which leads from s_0 to s_{loop} is identical to the construction of the sequence from s_0 to s_{dead}.

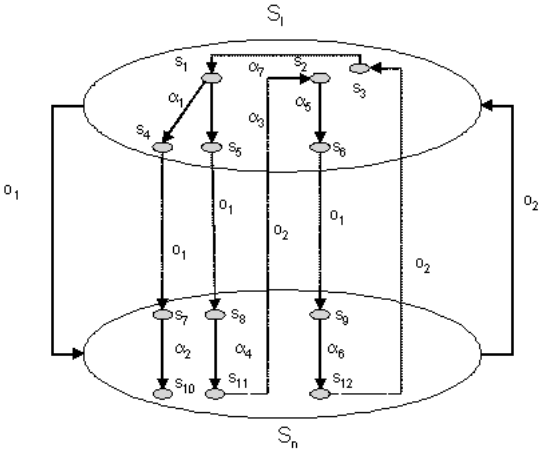

Fig. 3. Example of cyclic path in the symbolic observed graph

The case of a divergent sequence is similarly handled except that we remain inside the set of states associated to the final node of the path of the observation graph.

Our algorithms are currently evaluated and the results will be presented in a forthcoming paper. However we anticipate that the critical factor will be the management of the sets of couples.

5 Conclusion

In this work, we have presented a new method for the symbolic model checking problem. This method builds an observation graph represented in a non symbolic way but where the nodes are essentially symbolic sets of states. Then a standard model checking is applied on this graph which usually has a very moderate size. Thus we have focused our work on efficient symbolic algorithms for subproblems involved in the construction of this graph. The evaluations we have done on standard examples have shown that our method outperforms the pure symbolic methods which makes it attractive. We pursue this work on different directions. At first, it is straightforward to adapt our method for a stuttering invariant propositional linear time logic. From a software point of view, we want to transform our prototype in a intermediate library which can be bound with different OBDD software packages and called by a verification framework. We have noticed that the performance of our method depends on the events to be observed much more than on the number of these events. Since the choice of a superset of these events is still possible, we want to investigate heuristics for this choice based on the structure of the model and more particularly on a Petri net model. At last, we are looking for some characterization of the parallel systems for which our dichotomic symbolic search of initial SCCs of the state graph outperforms a standard symbolic search.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35** (1986) 677–691
2. Wegener, I.: Branching Programs and Binary Decision Diagrams: Theory and Application. SIAM Monographs on Discrete Mathematics and Applications (2000)
3. Fisler, K., Fraer, R., Khami, G., Vardi, M., Yang, Z.: Is there a best symbolic cycle-detection algorithm? Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 **2031** (2001) 420–434
4. Emerson, E., Lei, C.: Efficient model-checking in fragments of propositional model mu-calculus. Proceedings of LICS86 (1986) 267–278
5. Ravi, K., Bloem, R., Somenzi, F.: A comparative study of symbolic algorithms for the computation of fair cycles. In: International Conference on Formal Methods for Computer-Aided Verification. L.N.C.S., Springer-Verlag (2000) 143–160
6. Bloem, R., Gabow, H., Somenzi, F.: An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In: International Conference on Formal Methods for Computer-Aided Verification. L.N.C.S., Springer-Verlag (2000) 37–54
7. Gentilini, R., Piazza, C., Policriti, A.: Computing strongly connected components in a linear number of symbolic steps. In: Proceedings of International Symposium on Discrete Algorithms (SODA'03), ACM/SIAM (2003) 573–582

8. Xie, A., Beerel, P.: Implicit enumeration of strongly connected components and an application to formal verification. IEEETCAD: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **19** (2000)
9. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Volume 1. Springer-Verlag New York, Inc. (1992)
10. Kaivola, R., Valmari, A.: The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In: International Conference on Concurrency Theory. (1992) 207–221
11. Puhakka, A., Valmari, A.: Weakest-congruence results for livelock-preserving equivalences. In: Proceedings of the 10th International Conference on Concurrency Theory, Springer-Verlag (1999) 510–524
12. Valmari, A.: Failure-based equivalences are faster than many believe. In Desel, J., ed.: Structures in Concurrency Theory, Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT), Berlin, 11-13 May 1995. (1995) 326–340
13. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, Springer-Verlag (1998) 429–528
14. Miner, A., Ciardo, G.: Efficient reachability set generation and storage using decision diagrams. In: International Conference on Application and Theory of Petri Nets. LNCS, Springer-Verlag (1999) 388–393
15. Geldenhuys, J., Valmari, A.: Techniques for smaller intermediary bdds. In: CONCUR 2001 - Concurrency Theory, 12th International Conference. Volume 2154 of Lecture Notes in Computer Science., Springer (2001) 233–247
16. Lind-Nielsen, J.: Buddy, a binary decision diagram package. Technical Report IT-TR 1999-028, Institute of Information Technology Technical University of Denmark (1999) http://cs.it.dtu.dk/buddy.
17. Pastor, E., Roig, O., Cortadella, J., Badia, R.M.: Petri net analysis using boolean manipulation. In Valette, R., ed.: Proc. of the 15th Int. Conf. on Application and Theory of Petri Nets (PNPM'94), Zaragosa, Spain. LNCS 815, Springer (1994) 416–435