Symbolic Bisimulation Minimisation

Amar Bouali[1] and Robert de Simone[2]

[1] ENSMP Centre de Mathématiques Appliquées
B.P. 207 F-06904 Sophia-Antipolis
FRANCE
amar@cma.cma.fr
[2] I.N.R.I.A.
2004 route des Lucioles
F-06904 Sophia-Antipolis
FRANCE
rs@cma.cma.fr

Abstract. We describe a set of algorithmic methods, based on symbolic representation of state space, for minimisation of networks of parallel processes according to bisimulation equivalence. We compute this with the Coarsest Partition Refinement algorithm, using the Binary Decision Diagram structures. The method applies to labelled synchronised vectors of finite automata as the description of systems. We report performances on a couple of examples of a tool being implemented.

1 Introduction

Bisimulation is a central notion in the domain of verification for parallel communicating systems [17]. It was defined as an equivalence in the abstract formal setting of process algebras [17,2], through interpretation by labelled transition systems.

Bisimulation's algorithmic properties in the finite automata case have been widely studied [15,19,10], leading to a large body of automata-theoretic methods, complexity theory results and experimental comparisons based on verification tools [24,6,10,11]. Studies were also pursued with the additional concern of *observational* bisimulations, with a particular hidden action. Various treatments of this action were considered (*weak, branching ...*) [17,23].

The usual drawback of bisimulation is that, being defined on underlying automata, its computation requires all the informations (on states and transitions) collected in this global structure. Building the full automaton can lead to combinatorial explosion, while recomputing information dynamically is a time penalty. Several methods have nevertheless be proposed in the latter direction, either in the general case or in specific subcases (bisimulation comparaison with a determinate process for instance), with some success [12,3,20,18]. Anyhow most of these methods require to keep track at least of the reachable state space –if not of the transitions–, even though the problem of efficient representation or approximation of this state space has also been tackled [13,14].

In this paper we study one such efficient representation of the state space, using the symbolic Binary Decision Diagrams [4]. These data structures have been

now widely recognised for their ability to concisely represent sets and relations in practice, and applied to generate state spaces in several frameworks [7,22,16]. They have also been already applied to the problem of bisimulation, but inside a quite general setting, through formulation of the bisimulation property in general logical terms, based on the μ-calculus [5,9]. The solution here is more closely algorithmic in a sense directly related to the early methods in bisimulation checking. We first introduce a "concrete" formalism to represent Networks of Process Automata and shortly discuss its relation to process algebraic constructions in the finitary case. Then we recall the computational definition of bisimulation and rephrase it in a way that anticipates on the functions and techniques available with the particular BDD data structures. We then describe some of these functions on BDD, that will be used in our approach. In the two following sections we actually describe the algorithms for (symbolic) state space construction and bisimulation refinement; we do this on the model of Process Automata Networks, and in terms of the BDD functions just introduced. We end with a discussion on efficiency and implementation, and experimental results.

2 The Model and the Verification Method

We now introduce our description model formally. Then we sketch its relations with process algebraic static networks and its way of compilation into global automata. Last in this section we introduce bisimulation and characterize it in set-theoretic fashion.

We recall briefly the celebrated definition of automata:

Notation 2.1 *A* finite labelled transition system (lts) $A =< S, Sort(A), s^0, T >$ *-or* automaton- *is a 4-uple with* S *and* $Sort(A)$ *finite sets of respectively* states *and* labels, $s^0 \in S$ *an* initial *state and* $T \in (S \times Sort(A) \times S)$ *a set of (labelled)* transitions.

T *can be sorted by labels into relations* $T_a \subset S \times S, a \in Sort(A)$. *As usual we note* $s \xrightarrow{a} s'$ *for* $(s,a,s') \in T$ *(or* $(s,s') \in T_a$*). We shall assume w.l.o.g. all* T_a *to be nonempty.*

Also $T_a^{-1} = \{(s',s) \in S \times S \mid (s,s') \in T_a\}$. *For* $C \subseteq S$, *we note* $s \xrightarrow{a} C$ *if* $\exists s' \in C$ *such that* $s \xrightarrow{a} s'$.

2.1 Networks of Processes

A *Labelled Synchronised Automata Vector* consists in a finite vector of automata components (individual processes), together with a set of *Labelled Synchronisation Vectors* to constrain their relative behaviors. This model has been introduced by Arnold and Nivat [1]. The only difference here is that our synchronisation vectors are themselves labelled, introducing compositionality: a network could itself be expanded into an automaton, and act as component to a larger system. Another way to go is by flattening a structured description, with subnetworks used as components of larger networks, into a single vector, with only "leaf" individual processes remaining.

Still in this paper we shall concentrate on flat networks, so that each rational process component is a finite labelled transition system.

Definition 1. A *Labelled Synchronised Automata Vector (LSAV) N* (of size n) is a 4-uple $< \mathbf{A}, V, Sort(N), label >$ consisting of:

- a n-ary Automata Vector $\mathbf{A}$ of $A_i =< S_i, Sort(A_i), s_i^0, T_i >$ as automata components,
- a finite set V of n-ary Labelled Synchronisation Vectors, so that, for each $sv \in V$, $sv_i \in Sort(A_i)$ or $sv_i = \star$. The particular symbol $\star$ indicates inaction: the corresponding component does not engage in this global behavior.
- a finite set of labels $Sort(N)$.
- a labelling surjective function $label : V \rightarrow Sort(N)$

Notation 2.2 *We note n] the integer interval* $[1..n]$. *For sv a given synchronisation vector we note support(sv) the set* $\{i \in n], sv_i \neq \star\}$.

Definition 2. Let N be a LSAV. The *global automaton* A_N *associated to* N is given by:

- $Sort(A_N) = Sort(N)$ as (global) label space,
- $S = S_1 \times S_2 \times \ldots \times S_n$, as (global) state space,
- $s^0 = (s_1^0, s_2^0, \ldots, s_n^0)$ as initial state,
- $T = \{((s_1, s_2, \ldots, s_n), a, (s'_1, s'_2, \ldots, s'_n)), \exists sv \in V, label(sv) = a \wedge ((s_i, sv_i, s'_i) \in T_i \vee (s_i = s'_i \wedge sv_i = \star))\}$ as transition relation.

Later we shall of course only be interested in the global states that are reachable from s^0. This subset will be computed symbolically, using BDDs.

We end this section with informal motivations of our description formalism in the light of process algebraic constructors.

Because of closure by composition the LSAV model allows to represent a large body of process calculi expressions, namely the non-recursive ones built only from so-called static operators in [17]. Recursion is then only used for the creation of individual automata components with dynamic operators. This was shown in [8]. Shortly, such operators have SOS semantic rules such that the residual expression keeps unchanged the shape of the term (parallel rewrites into parallel, etc...). In this sense the various *parallel / scoping / relabelling* operators describing the network can be thought of as an elegant syntax for shaping the description of the network.

Note that, in addition to the static operators, production of actual synchronisation vectors also require the *Sorts* of (uninstanciated) components.

While simplifying the construction of global state spaces and automata, since a number of successive static constructions may here be combined at once, our concrete formalism of *LSAVs* do not favour compositional reduction (minimisation applied on subsystems). This latter method is dramatically used in AUTO for example. This is not a penalty in the symbolic approach, where intermediate automata are not built anyway. Of course the two approaches could be juxtaposed independantly, wherever more beneficial.

2.2 Bisimulation Equivalence

The present section recalls the notion of bisimulation, a "behavioral" equivalence defined on transition systems. We derive from this definition a set-theoretic formulation to compute it as a fix-point on finite transition systems.

We shall not motivate the philosophy behind bisimulation equivalence, see [17]. Its main characteristic is that equivalent states all have the same behavior ability, so that there exists a canonical minimal form, based on classes as states, even for nondeterministic transition systems.

Bisimulation equivalence is used in the setting of verification both to reduce an underlying automaton and to compare two distinct automata. We are here interested in the former role, and we want to extract the equivalence classes, from the LSAV description and a representation of the reachable state space. Actually constructing the minimal automaton is then straightforward (and possibly useless).

Notation 2.3 *Given an equivalence relation $\mathcal{R}$ on a set E we note $[e]_{\mathcal{R}}$ the equivalence class of $e \in E$.*

Definition 3. Let $A =< S, Sort(A), s^0, T >$ be a lts. A binary symmetric relation $\mathcal{R} \in S \times S$ is a bisimulation if:

$$\forall s, s' \in S, (s \mathcal{R} s') \Rightarrow \left(\forall a \in Sort(A), \forall t \in S, (s \xrightarrow{a} t \Rightarrow \exists t', (s' \xrightarrow{a} t') \wedge (t R t'))\right)$$

We note $\sim$ the largest bisimulation relation $\mathcal{R}$ verifying the previous definition.

The bisimulation property above is defined recursively. Now consider the following chain of equivalence relations:

$$\begin{aligned}
\mathcal{R}_0 &= S \times S \\
\mathcal{R}_{n+1} &= \{(s, s') \in \mathcal{R}_n \mid \forall a \in Sort(A), \\
&\quad ((s \xrightarrow{a} s_1) \Rightarrow (\exists s'_1, s' \xrightarrow{a} s'_1 \wedge (s1, s'_1) \in \mathcal{R}_n)) \wedge \\
&\quad ((s' \xrightarrow{a} s'_1) \Rightarrow (\exists s_1, s \xrightarrow{a} s1 \wedge (s_1, s'_1) \in \mathcal{R}_n))\}
\end{aligned}$$

It is folklore that for finite automata $\mathcal{R}_\infty = \bigcap \mathcal{R}_n$ corresponds to $\sim$, and is obtained through finite iteration: it exists n_0 such that $\mathcal{R}_{n_0} = \mathcal{R}_\infty$. Each $\mathcal{R}_j$ belonging to the sequence of relations can be seen as a union of n_j equivalence classes, that is

$$\mathcal{R}_j = \bigcup_{i=1}^{n_j} C_{i,j} \times C_{i,j}$$

with the $C_{i,j}$ ranging over the equivalence classes of R_j. To each $\mathcal{R}_j$, we associate the related partition $P_j = \{C_{i,j} | i \in n_j]\}$

For convenience we introduce the auxiliary values:

$$E_{j,a} = \bigcup_{i=1}^{n_j} C_{i,j} \times T_a^{-1}(C_{i,j})$$

Now we can rewrite the definition of $\mathcal{R}_{j+1}$ from $\mathcal{R}_j$ as

$$\mathcal{R}_{j+1} = \mathcal{R}_j \cap \left(\bigcap_{a \in Sort(A)} \{(s_1, s_2) \mid \forall s, (((s, s_1) \in E_{j,a} \iff (s, s_2) \in E_{j,a})\} \right)$$

$$= \mathcal{R}_j \cap \left(\bigcap_{a \in Sort(A)} \{(s_1, s_2) \mid \neg\exists s, (((s, s_1) \in E_{j,a} \iff (s, s_2) \notin E_{j,a})\} \right)$$

This set-theoretic identity will base an algorithm to compute $\sim$ by refinement in the second part of the paper. Note already that each iteration requires **a single application** of T_a^{-1}, in the computation of $E_{j,a}$.

3 Symbolic Representation of LSAV and Bisimulation

We now shortly discuss the symbolic encoding of state spaces into Binary Decision Diagrams. We describe the simple computation of their reachable parts, and then the symbolic computation of bisimulation on (the encoding of) LSAVs.

3.1 Sets and BDD Encodings

A BDD is an acyclic graph representation of (the truth table of) a Boolean propositional formula, on a finite predefined support of predicate variables. It is based on decision trees, where branching splits according to the alternative values of the variables, encountered in some fixed order. BDDs are then obtained by sharing subtrees as much as possible. BDDs are canonical, relative to a given ordering of the variables, in the sense that different syntaxic formulas with the same truth values have identical (minimal) representations. See [4] for full details.

As usual in the BDD litterature we shall identify *finite* sets and relations with their characteristic functions, and further with an encoding using boolean variables of this characteristic functions. We shall now be more specific on our notion of encoding.

Definition 4. Let E be a finite set. An *encoding* for E consists of: a finite, totally ordered *encoding array* of size m (say $x = [x_1 < x_2 < \ldots < x_m]$) of boolean variables together with a surjective partial function $\xi : Bool^m \longrightarrow E$.

Notation 3.1 *We note $\underline{x}_m$ the ordered (encoding) array $[x_1 < x_2 < \ldots < x_m]$ of propositional variables. x is then called the* basename, *and m the size. We omit the size subscript and the underline when clear from context.*

With our definition an element of E needs not be represented by a *unique* boolean valuation, nor does *any* boolean valuation represent an element of E. Just a boolean valuation may not be associated with two elements of E ambiguously. As previously mentioned we will abusively call E for its encoding syntactic BDD also (this makes an implicit reference to a specific encoding). We extend this and note $\overline{E}$ for the BDD representing $Bool^m \setminus \xi^{-1}(E)$ on the proper encoding array.

The value m in the previous definition is not fixed *a priori*. Encodings of sets usually range from those minimal variables numbers ($m = \lceil log(\#E) \rceil$ to *One-hot* encodings where a variable is assigned to each element ($m = \#E$). Since a key issue in BDDs lies not so much in the symbolic representation of states as in this of transition relations, our choice of encoding should try and make the encoding of transitions as small as possible. This was one reason why we gave such a broad definition of encodings.

Encoding arrays are needed in the definition since boolean variables in BDDs are syntactic elements. Later on we will need at places to introduce disjoint samples of encoding sets for the same E and the same *coding* function, so they will differ only by their explicit range of variables.

Coming down to our precise needs for symbolic modelisation, we shall have to represent: local states, global states (as vectors of local states), T_a transition relations and $\mathcal{R}_i$ equivalence relations (as couples of global states). So local states will require basic encoding, from which we deduce the other encodings by taking product encodings, on disjoint unions of encoding arrays.

The "good encoding" issue now splits into: first, find appropriate local encodings; second, find interesting order extensions when putting different (disjoint) encodings together in disjoint union.

We now introduce two natural ordering extensions of encodings in case of cartesian products E^k.

Definition 5. Let $\{\underline{x}^1, \ldots, \underline{x}^k\}$, compose k pairwise disjoint encoding arrays of size m for a common set E (with the same encoding function), such that $\forall i \in k], \underline{x}^i = [x^i_1, \ldots, x^i_m]$. We introduce two extensions $\ll$ and $\lhd$ of the total orders on the $\underline{x}_i$'s by

$$\begin{aligned} \ll &: \forall i \in (k-1)] x^i_m \ll x^{i+1}_1 \\ \lhd &: \ \forall i \in (k-1)], \forall j \in m] \ \ x^i_j \lhd x^{i+1}_j \\ &\quad \forall j \in (k-1)] x^m_j \lhd x^1_{j+1} \end{aligned}$$

We call $\ll$ the concatenated extension, for it will not mix (variables from) different coding sets. On the contrary $\lhd$ the shuffled extension will put as close as possible corresponding variables from various coding sets.

The ordering extension to (vector) global states here should also be discussed. A brute force solution consists in concatenating these orders without mixing. A more insighful solution should bring as close as possible those variables which are correlated, no matter which local component they encode. This corresponds to an attempt at bringing together (when feasible) the different local aspects of a same global event. Still, it will be the case that, in the encoding of a global state, each variable helps in encoding exactly one local component. We note $\underline{x}_i$ the subarray obtained by collecting in increasing order all boolean variables that deal with automata component A_i, and $\underline{x}_{sv}$ for the concatenation (still in increasing order) of the $\underline{x}_i, i \in support(sv)$.

We end this section by recalling some less familiar boolean operators to be used later in algorithms.

Definition 6. Let f be a boolean function. The smoothing of f by boolean variables $X = (x_{i_1}, \ldots, x_{i_p})$ is defined as

$$S_{(x_{i_1}, \ldots, x_{i_p})}(f) = S_{x_{i_1}} \circ \ldots \circ S_{x_{i_p}}(f)$$
$$S_{x_{i_j}}(f) = f_{x_{i_p}} + f_{\overline{x_{i_p}}} \text{ where}$$
$$f_{x_i}(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_r) = (x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_r) \text{ and}$$
$$f_{\overline{x_i}}(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_r) = (x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_r)$$

Logically, the smoothing operator performs existential quantification on the smoothed variables: $S_{(x_{i_1}, \ldots, x_{i_p})}(f) = \exists x_{i_1} \ldots \exists x_{i_p} f$.

Definition 7. Let f be a boolean function. Substitution of array of boolean variables $X = [x_1, x_2, \ldots, x_p]$ by array $Y = [y_1, y_2, \ldots, y_p]$ in f, noted $[Y \leftarrow X]f$, consists of the simultaneous replacement of (boolean variables) array items x_i by corresponding y_i. Formally

$$[Y \leftarrow X]f \equiv_{def} S_X \left(\bigwedge_{i \in p]} (y_i \iff x_i) \wedge f \right)$$

The resulting BDD may be very different from f due to different respective ordering of x's and y's and other variables.

A third boolean operator we shall use on BDD is the *cofactor* operation. For lack of room we shall not describe it here. Intuitively $cofactor(B, C)$ represents *"B provided C"*, or what is strictly needed to recover B under the assumption of C being true.

3.2 The Reachable States Algorithm

The space of all reachable states in a *LSAV* will be attained in a breadth-first search iterative manner, where at each iteration all states directly following the ones just obtained are reached. To do this, one should just apply the transition relation once. The main problem here is that encoding the full transition relation proves in practice to be much more space consuming than the state space itself. We solve this problem by splitting the application of the transition relation according to the synchronisation vectors. We now discuss this issue in relation with our model.

When building a global (unlabelled) transition relation $T(x, y)$ from the synchronisation vectors and the local transitions of individual automata, one needs to encode the identity where $\star$ (inaction) is indicated (leading to $stable_i$ BDDs in [9]). This completion does not seem too dramatic when considering a single synchronisation vector, but when building the union of all transitions it does not behave friendly according to sharing of subtrees. The resulting BDD is thus frequently very large.

On the other hand, if one considers building T_{sv} the part of the transition relation corresponding to an only synchronisation vector sv, nice features show up.

Then the full transition is then applied in steps by iteration on synchronisation vectors. The algorithm is shown in figure 1.

Nice features just mentioned are:

- first, the synchronisation vector allows a priori any combination of local transitions properly labelled. Thus the union of equaly labelled transitions can be formed locally, and this indicates natural sharing,
- second, T_{sv} can be defined and applied on *support(sv)* only, disregarding the other components, which will therefore be left unchanged. This drops the need for the stable completion.

Definition 8. $T_{sv}(\underline{x}_{sv}, \underline{y}_{sv}) \equiv_{def} \bigwedge_{i \in support(sv)} \left(\bigvee_{t_i \in T_{i_{label(sv_i)}}} T_{t_i}(\underline{x}_{\{i\}}, \underline{y}_{\{i\}}) \right)$

where $T_t(\underline{x}, \underline{y})$ encodes transition t's local source/target states with respective encoding arrays $\underline{x}$ and $\underline{y}$.

Application of transitions corresponding to sv to a state space St is expressed as:

$$Apply(T_{sv}(\underline{x}_{sv}, \underline{y}_{sv}), St(\underline{x})) = [\underline{x}_{sv} \leftarrow \underline{y}_{sv}] S_{\underline{x}_{sv}}(St(\underline{x}) \wedge T_{sv}(\underline{x}_{sv}, \underline{y}_{sv}))$$

```
(1)   States(x) = S^0(x)
(2)   New(x) = States(x)
(3)   while New ≠ 0
(4)   begin
(5)      Temp(x) = States(x)
(6)      for each sv ∈ V do
(7)      begin
(8)         States(x) = States(x) ∨ Apply(T_sv(x, y), States(x))
(9)      end
(10)     New(x) = States(x) ∧ ¬Temp(x)
(11)  end
```

Fig. 1. Reachable states with vector transition relation

3.3 The Real Transition Relation

No matter whether it is fully constructed or applied by chunks along the synchronisation vectors, the transition relation above is actually syntactically derived as a superset of the real one: not everything needs to be applicable. Certain conjunctions of local transitions, while legible from the synchronisation point of view, may leave from non-reachable states and therefore not be fireable. More generally, the transition relation may pay attention to behaviors outside the state space.

We define below the (very simple) way to restrict this relation to its useful part. We shall need this in the bisimulation partitioning algorithm.

Definition 9.

$$\tilde{T}_{sv}(\underline{x},\underline{y}) \equiv_{def} T_{sv}(\underline{x}_{sv},\underline{y}_{sv}) \wedge States(\underline{x}) \wedge \left(\bigwedge_{i \notin support(sv)} Stable_i \right)$$

$$\tilde{T}_a(\underline{x},\underline{y}) \equiv_{def} \bigvee_{label(sv)=a} \tilde{T}_{sv}(\underline{x},\underline{y})$$

3.4 BDD Bisimulation

With explicit reference to variables encoding, the relation between R_j and R_{j+1} can now be expressed as:

$$E_{j,a}(x,z) = S_y(R_j(x,y) \wedge \tilde{T}_a(y,z))$$

$$Bad_j(x,y) = [x \leftarrow z] \left(\bigvee_{a \in sort(N)} S_x \left([y \leftarrow z] E_{j,a}(x,z) \Leftrightarrow \overline{E_{j,a}(x,z)} \right) \right)$$

$$R_{j+1}(x,y) = R_j(x,y) \wedge \overline{Bad_j(x,y)}$$

One starts with $R_0 = S_N \times S_N$, or taking coding sets into account: $R_0(x,y) = S_N(x) \wedge S_N(y)$. The algorithm of figure 2 implements this: lines (8-9) compute $E_{j,a}$, while lines (10-14) compute bad couples on a label, not with a XOR operation but as a two-fold union, this due to the fact that our implementation package allowed to perform simultaneously the conjunction and smoothing in lines 13 or 14, which is an interesting save-up in complexity. The loop at lines (5-17) accumulates bad couples for all actions. Lines 18 computes the termination test.

Figure 2 refines the computation at lines 1 and 8, where the cofactor function is introduced, so that all sets are given "provided" $S_N \times S_N$, into which they are all included.

4 Observational Bisimulation Equivalences

Two main variations on bisimulation have been proposed in order to deal with a specific actions (τ) as invisible: *weak* bisimulation and *branching* bisimulation. We shall not detail them here. We briefly indicate how to modify our previous algorithms to cope with these two extensions.

Weak bisimulation equivalence simply uses a new transition relation, deduced straightforwardly from the original one.

Definition 10. The weak transition relation $\Rightarrow$ of a lts A is:

$$s \stackrel{\tau}{\Longrightarrow} s' \text{ iff } \exists s_0, s_1, \ldots, s_n, n \geq 0, s = s_0 \stackrel{\tau}{\longrightarrow} s_1 \stackrel{\tau}{\longrightarrow} \ldots \stackrel{\tau}{\longrightarrow} s_n = s'$$

$$s \stackrel{a}{\Longrightarrow} s' \text{ iff } \exists s_1, s_2, s \stackrel{\tau}{\Longrightarrow} s_1 \stackrel{a}{\longrightarrow} s_2 \stackrel{\tau}{\Longrightarrow} s'$$

(1) $NewR(x,y) = cofactor(R_0(x,y), R_0(x,y))$
(2) $i = 0$
(3) **while** $NewR(x,y) \neq 0$
(4) **begin**
(5) $Bad_j(x,y) = \emptyset$
(6) **For all** $a \in Sort(N)$
(7) **begin**
(8) $trans(z,y) = [z \leftarrow x]cofactor(T_a(x,y), R_0(x,))$
(9) $E^1_{j,a}(x,z) = S_y(R_i(x,y) \wedge trans(z,y))$
(10) $E^2_{j,a}(x,y) = [y \leftarrow z]E^1_{j,a}(x,z)$
(11) $T_1(x,z) = \overline{E^1_{j,a}(x,z)}$
(12) $T_2(x,y) = \overline{E^2_{j,a}(x,y)}$
(13) $B_1(z,y) = S_x(E^1_{j,a}(x,z) \wedge T_2(x,y))$
(14) $B_2(z,y) = S_x(T_1(x,z) \wedge E^2_{j,a}(x,y))$
(15) $Bad_j(x,y) = Bad_j(x,y) \vee [x \leftarrow z](B_1(z,y) \vee B_2(z,y))$
(17) **end**
(16) $R_{i+1}(x,y) = R_i(x,y) \wedge \overline{Bad_j(x,y)}$
(18) $NewR(x,y) = R_i(x,y) \wedge R_{i+1}(x,y)$
(19) $i = i + 1$
(20) **end**

Fig. 2. Bisimulation with BDD

One may either computes these relations symbolically, or composes dynamically the application of three relations ($T_{\tau^\star}^{-1}$, then T_a^{-1} and $T_{\tau^\star}^{-1}$ again). We use the iterative squaring method (see [22]) to compute $\stackrel{\tau}{\Longrightarrow}$ as the fixpoint of

$$F(x,y) = (x \stackrel{\tau}{\Longrightarrow} y) \vee \exists z(F(x,z) \wedge F(z,y))$$

Branching bisimulation instead restricts the above application of surrounding $\tau^\star$ to state inverse images that preserve equivalence (so far). It can be proven in this case that $\tau^\star$ need only be applied once (either before or after the "real action"), without affecting the resulting equivalence. Also, when τ is used as the discriminating action, only τ transitions joining (already) non-equivalent states need be applied. Thus, we define a set of transition relations Tb^j at each step in the equivalence partitioning. Noticeably these sets are very easy to recompute dynamically, as all they do is gather two informations already present: $\mathcal{R}_j$ equivalence, and the fixed global $\tau^\star$ transition relation.

$$\begin{aligned} Inert^j_{\tau^\star} &= T_{\tau^\star} \cap \mathcal{R}_j \\ Tb^j_a &= Inert^j_{\tau^\star} \circ T_a \\ Tb^j_\tau &= Inert^j_{\tau^\star} \circ (T_\tau \cap \overline{\mathcal{R}_j}) \end{aligned}$$

5 Applications

5.1 Encoding and Variable Ordering

It is shown in [9] that the interleaved order extension was theoretically best for the representation of transitions and states, during the Reachable State Space construction. Now the question is: would the same considerations apply for the representation of equivalent state couples during the Partition Refinement step?

As a rule of thumb it seems that, while the interleaved order gets better as the final relation is more discriminating and states are less identified, the concatenated order takes advantage of coarser cases where classes are fewer and more state identifications take place. This is understandable: in this order, the description of either instance of the class is factorised independantly (as first or second component of couples). In the former case use of BDDs may be questioned, for its interest is based on gain of sharing for sets representation. As more elements are isolated, they each tend to consume space individually in the BDD. Of course this is rough estimation. Weak bisimulation often allows more identifications to take place, specially when not all actions are kept visible. This case is certainly the most beneficial for the symbolic approach to bisimulation minimisation.

Sensibility to the alternative ($\ll$ /$\lhd$) variable orderings is illustrated and contrasted below on two examples with opposite teachings.

5.2 Test Examples

Two examples are exposed in this section. The first one is the LSAV specification of the scheduler with different number of cyclers. The second one is the Dekker Mutual Exclusion algorithm. For a precise presentation of these examples, the reader should refer to [17] for the former example, and [21] for the latter.

Table displayed in figure 3 shows the results obtained on the different instanciations of the scheduler specification problem, namely $4, 8, 16, 18$, and 20 cyclers for which a specific row is attributed in the table, and for the Dekker algorithm whose results row begins with a "D". All results have been obtained on a SPARC Station 2 with 24 MBytes of memory. Each row contains from letf to right: the concerned example, the number of reachable states followed by the size of the corresponding BDD and the time taken by the tool to compute it; for each type of variable order in $(\lhd, \ll)$, the related time taken to compute the bisimulation equivalence on the reachable state space and the size of the corresponding BDD; finally, the number of equivalent pairs under bisimulation. The symbol "-" means that the tool was oborted during execution, due to lack of memory. In the case of the scheduler example, the $\lhd$ order is better than $\ll$,unlike in the other examples.

Conclusion

We have described an implementation of Bisimulation Minimisation based on symbolic representation using Binary Decision Diagrams. This implementation

	States	BDD	Time	Var	~ Time	~ BDD	Pairs
4	128	16	0.29	$\lhd$	4.88	59	256
				$\ll$	24.83	859	
8	4096	16	0.29	$\lhd$	4.88	59	256
				$\ll$	24.83	859	
16	2097152	64	11.99	$\lhd$	131.11	251	4194304
				$\ll$	-	-	
18	9437184	72	16.26	$\lhd$	165.42	283	18874368
				$\ll$	-	-	
20	41943040	80	22.47	$\lhd$	213.90	315	83886080
				$\ll$	-	-	
D	126	74	1.36	$\lhd$	104.20	962	450
				$\ll$	20.50	686	450

Fig. 3. Performances of the tool

tried to optimize computations and BDD sizes wherever possible, and in particular:

- works on LSAV as a compound version of process algebraic networks, so that several operators may be applied at once, and impossible behaviors detected as soon as possible,
- applies transitions as events individually, eliminating the need for storing the full transitions, with its out-of-support intricacies. Event-parted transitions allow well BDD sharing,
- applies backward behavior image only once for the coarsest partition fixpoint algo- rithm (instead of on both components of equivalent couples),
- performs simultaneous splitting of all classes by all classes.

References

1. A. Arnold and M. Nivat. Comportements de processus. In *Les Mathématiques de l'Informatique*, pages 35–68. Colloque AFCET, 1982.
2. D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theorical Computer Sciences*, 1(30), 1984.
3. A. Bouali. Weak and branching bisimulation in fctool. Technical Report 1575, INRIA, 1991.
4. Randal E. Bryant. Graph-based algorithms for boolean function manipulaiton. *Transactions on Computers*, C-35(8):677–691, August 1986.
5. J.R. Burch, E.M. Clarke, L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} and beyond. In *5^{th} IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, 1990.
6. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In *Automatic Verification Methods for Finite State Systems*, pages 24–37. LNCS, 1989.

7. O. Coudert, C.. Berthet, and J.C. Madre. Verification of sequential machines using boolean vectors. In *IFIP Internationnal Workshop, Applied Formal Methods for Correct VLSI design*, Leuven, November 1990.
8. R. de Simone and A. Bouali. Causal models for rationnal algebraic processes. In J.C.M. Baeten and J.F. Groote, editors, *2nd internationnal Conference on Concurrency Theory*, Amsterdam, August 1991. CONCUR'91, Springer-Verlag.
9. R. Enders, T. Filkorn, and D. Taubner. Generating bdds for symbolic model checking. In *Third Workshop on Computer Aided Verification*, volume 1, pages 263–278. University of Aalborg, July 1991.
10. J.C. Fernandez. *Aldébaran: un système de vérificatoin par réduction de processus communiquants.* PhD thesis, Grenoble, 1989.
11. J.C. Godskesen, K.G. Larsen, and M. Zeeberg. Tav, tools for automatic verification. In *Automatic Verification Methods For Finite State Systems*, pages 232–246, Grenoble, France, 1989. LNCS, Springer-Verlag.
12. J.F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. *ICALP '90*, 1990.
13. G.J. Holzmann. Algorithms for automated protocol validation. In *International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.
14. C. Jard and T. Jéron. Bounded-memory algorithm for verification on-the-fly. In Larsen.K.G. and A. Skou, editors, *Third Workshop on Computer Aided Verification*, volume 1, pages 251–262, July 1991.
15. P.C. Kanellakis and S.A. Smolka. Ccs expressions, finite state processes, and three problems of equivalence. In *ACM Symposium on Principles of Distributed Computing*, pages 228–240, 1983.
16. B. Lin and A.R. Newton. Efficient manipulation of equivalence relations and classes. In *ACM International Workshop on Formal Methods in VLSI design*, Miami, January 1991.
17. Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.
18. Laurent Mounier. *Méthodes de vérification de spécifications comportementales: étude et mise en oeuvre.* PhD thesis, LGI Grenoble, 1991.
19. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM*, 16(6), 1987.
20. Huajun Qin. Efficient verification of determinate processes. Technical report, Dep. of Comp. Sc., SUNY, Stony Brook, 1991.
21. M. Raynal. *Algorithmes du Parallélisme: le Problème de l'Exclusion Mutuelle.* Dunod Informatique, 1984.
22. H.J. Touati, H. Savoj, B. Lin, and Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *Internationnal Conference on Computer Aided Design*, 1990.
23. R.J. Van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). *Information processing '89 (G.X. Ritter, ed.) Elsevier Science*, pages 613–618, 1984.
24. D. Vergamini. *Vérification de réseau d'automates finis par equivalence observationnelle: le système AUTO.* PhD thesis, Université de Nice, 1987.