Toupie = μ-calculus + constraints

Antoine Rauzy

LaBRI - CNRS URA 1304 - Université Bordeaux I
51, cours de la Libération, F-33405 Talence (France)
email: rauzy@labri.u-bordeaux.fr

Abstract. This paper presents some of the features of the constraint language Toupie (version 0.26). Toupie is basically a μ-calculus intepreter. Variables takes their values in finite domains, i.e. finite sets of symbolic or numerical constants. Toupie integrates a solver for systems of linear inequations over finite domains and uses an extension of Bryant's binary decision diagrams to encode relations. Combination of μ-calculus expressiveness, efficient coding and manipulation of relations through the use of n-ary decision diagrams and constraint solving technics make Toupie a powerfull tool to perform system of communicating processes analyses.

1 Introduction

Constraint logic programming (CLP) [15, 14] has demonstrated its ability to handle problems coming from operation research. We think that basic ideas of this paradigm — simplicity of problem expressions, fast prototyping, flexibility, versatility of the tools — could be useful in other areas, especially for what concerns the analysis of systems of concurrent processes. However, program analysis requires often solvers for second order constraints, i.e. mainly fixpoint equations, that are not available in CLP languages.

We present here the language Toupie that handles such second order constraints. More precisely, Toupie implements an extension of the propositional μ-calculus to finite domain constraints. In addition to classical functionalities of finite domain constraint solvers, Toupie allows a full universal quantification and the definition of relations as least or greatest fixpoints of monotone functions. These definitions can be seen as a kind of quantification over relations.

The main implementation problem with second order constraints is to store efficiently tuples belonging to relations. Analyses of systems of concurrent processes — which is the main goal of Toupie — require to handle relations with huge numbers of elements because, even if each individual process can be described by means of a small finite state automaton, the whole system often goes through thousands and thousands of different states. This combinatorial explosion is due to the various possible interleaving of individual process actions. Toupie achieves this goal by encoding relations by means of decision diagrams, an extension to finite domains of Bryant's binary decision diagrams [5].

The idea of using BDDs to encode large finite state automata is not new. Mac Millan & als on the one hand [8], Madre and Coudert on the other hand [10] have shown very impressive examples of its power. The contribution of Toupie is twofolds : At a technical level, it integrates decision diagrams features and constraint solving. At a functional level, it integrates already mentioned basic ideas of the CLP paradigm into a model checking tool.

The remaining of this paper is organized as follows : Toupie are presented section 2. Some technical implementation details are given section 3. The use of Toupie for symbolic model checking is described section 4.

2 Syntax and Semantics of Toupie Programs

In order to present syntax and semantics of Toupie in an informal way, we deal, through this section, with the well-known two players Nim's game (the reader interested in a more formal presentation could see [17]).

Nim's game : The game starts with N lines numbered from 1 to N and containing $2 \times i - 1$ matches (where i is the number of the line). At each step, the player who has the turn takes as many matches as he wants in one of the line. Then the turn changes. The winner is the player who takes the last matches.

Variables, Constants, Domains, Composite Variables : A position in the Nim's game is characterized by the player who has the turn and the number of matches in each line. In Toupie, such a position is described by means of a composite variable that groups several individual variables. Before using a composite variable, one must declare its type. For the Nim's game with three lines of matches, the declaration is as follows.

```
let position = tuple (
       P  : {a,b},
       L1 : 0..1,
       L2 : 0..3,
       L3 : 0..5)
```

A variable of type `position` groups four individual variables : `P` (for Player) that takes its value in the set of symbolic constants `{a,b}` and `L1`, `L2` and `L3` (for Line 1, 2 and 3) that take there values in ranges of integers. `{a,b}`, `0..1`, `0..3` and `0..5` are the domains of the variables `P`, `L1`, `L2` and `L3`. Variable identifiers begin with upper case letters and symbolic constant identifiers begin with lower case letters.

Formulae, Predicates : Assume declared a variable `Pos` of type position. In order to constrain `Pos` to describe the initial state of the game, one uses a formula :

```
((Pos.P=a) & {Pos.L1=1, Pos.L2=3, Pos.L3=5})
```

The above formula is a conjunction — `&` stands for $\wedge$ — of two atomic constraints : an equality and a system of linear inequations. The "field" `F` of a composite variable `C` is denoted by `C.F`. When a composite variable is manipulated per se its identifier is prefixed with a caret : `^C`.

A move is described by means of binary predicate, i.e. a relation, whose first member (`^S` for Source) is the position before the move and the second member (`^T` for Target) is the position after the move :

```
move(^S:position,^T:position) += (
     (S.P#T.P)
   & (
         {S.L1>T.L1, S.L2=T.L2, S.L3=T.L3}
       | {S.L1=T.L1, S.L2>T.L2, S.L3=T.L3}
       | {S.L1=T.L1, S.L2=T.L2, S.L3>T.L3}
     )
   )
```

The relation `move` is defined as a conjunction of the difference `S.P#T.P` meaning that the turn changes and the disjunction of three systems of linear inequations representing the different ways the player who has the turn can take matches in a line.

A Toupie program is a set of n-ary predicate definitions.

Quantifiers, Queries : Toupie is an interpreter. Once entered the definition of `position` and `move`, it is possible to ask queries. For instance, the positions where no move is playable are obtained by means of the following query.

```
lambda (^S:position) forall ^T:position ~move(^S,^T) ?
```

The form `lambda` is just a way to declare the type of the variable(s) of the query (here `^S` is of type `position`). The quantifier `forall` has its intuitive meaning and `~` stands for $\neg$. In response to the above query, one obtains :

```
{S.L1=0,S.L2=0,S.L3=0}
```

This encodes the two final positions (player `a` wins or player `b` wins). This example illustrates the fact that Toupie is a deterministic language. In response to a query it computes the decision diagram associated with this query and then goes through this data structure to display the tuples belonging to the relation. The result of a computation is thus an unique relation eventually containing several tuples. Decision diagrams perform some factoring of tuples which explains that only one tuple is displayed.

Fixpoints : All the positions are not reachable from the initial one (for instance, `<a,0,3,5>` is not). A position `^T` is reachable either if it is the initial one or if there exists a reachable position `^S` and a move from `^S` to `^T`. This characterization of reachable positions is recursive. It means that it is not possible to express it just with first order constraints. One needs a kind of quantification over relations. As a matter of fact, our characterization is typically a least fixpoint definition.

In Toupie, predicates are actually defined as least or greatest fixpoints of equations for the inclusion in the powerset $2^{\mathcal{D}}$ of the cartesian product $\mathcal{D} =$

$D_1 \times \ldots \times D_n$ of the domains D_i's of their formal parameters. $2^{\mathcal{D}}$ equipped with the set inclusion forms a complete lattice. The Tarksi's theorem asserts that given a monotone function f from $2^{\mathcal{D}}$ to $2^{\mathcal{D}}$,

1. f is continuous.
2. The equation $R = f(R)$ $(R \in \mathcal{P}(\mathcal{D}))$ admits a least and a greatest solutions, denoted with $\mu R.f(R)$ and $\nu R.f(R)$, that are respectively equal to $\bigcap\{R | f(R) \subseteq R\}$ and $\bigcup\{R | f(R) \supseteq R\}$.
3. There exist two integers m and n such that $\mu R.f(R) = f^m(\emptyset)$ and $\nu R.f(R) = f^n(\mathcal{D})$ where $f^k(R)$ denotes the k-nth application of f to R.

We use only a very restricted version of this theorem, since we are only concerned with finite Boolean lattices. It is easy to generalize this result to the case of systems of fixpoint equations. It provides the computation principle of Toupie predicates.

Syntactically, least and greatest fixpoint definitions are denoted by equations respectively in the form p += f and p -= f.

The predicate **move** is thus defined as a least fixpoint, but, since there is no recursive call in its equation, it could be defined as a greatest fixpoint as well.

The predicate encoding reachable positions is as follows.

```
reachable(^T:position) += (
     initial(^T)
   | exist ^S:position (reachable(^S) & move(^S,^T)))
```

Winning Positions : A position is winning if there exists a move leading to a losing position and conversely a position is losing if any playable move leads to a winning position. This is simply what express the two following predicates.

```
winning(^S:position) +=
    exist ^T:position (move(^S,^T) & losing(^T))

losing(^S:state) +=
    forall ^T:position (move(^S,^T) => winning(^T))
```

The following table gives some running times necessary to compute reachable and winning positions on a SUN IPX Sparc Station 48 MgB.

Number of lines	4	5	6	7	8
Number of reachable positions	763	7,674	92,153	1,290,232	20,643,831
Times reachable positions	0s21	0s43	0s75	1s25	1s93
Times winning positions	0s50	1s73	6s23	25s18	141s60

Nested fixpoints : Nested fixpoints are also allowed in Toupie version 0.26. In the literature, nested fixpoint definitions are used mainly to express infinite path properties. Let us consider the automaton pictured Fig.1, and let us characterize states that are source of paths going infinitely often through odd states.

This is achieved by means of a $\nu\mu$ term, i.e. a greatest fixpoint of a least fixpoint.

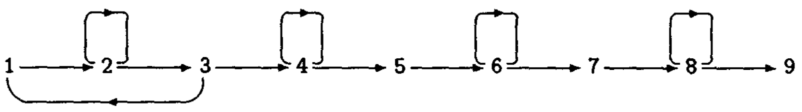

Fig. 1. An automaton

```
let vertex = domain 1..9
g(S:vertex,T:vertex) += /* description of the automaton */

odd(S:vertex) += exist N:0..4 {S=2*N+1}

tau(U:vertex) -=
  let aux(V:vertex) += (   /* local definition */
          exist W:vertex (g(V,W) & aux(W))
        | exist W:vertex (g(V,W) & odd(W) & tau(W))
        )
  in  aux(U)               /* body of the definition */
```

The idea of this definition is to compute the set of predecessors of odd states, then to intersect this set with the set of odd states, then to restart the computation of predecessors from this last set and so on. The remaining states are those belonging to a loop going through at least an odd state.

Toupie versus propositional μ-calculus : There exist several different presentations of the μ-calculus in the literature. This formalism allows the expression of state properties of automata. The differences between the presentations stand mainly in the type of the considered automata. The key point being to know whether transitions are labeled or not. Authors working with labeled transitions add to the formalism connectives allowing to characterize transition labels and coming typically from the Henessy & Milner's logic [13].

Toupie is closer to the Park's original presentation [16] (used for instance in the already cited paper [8]). We prefer this version because it is as expressive as the former but does not impose interpretations in terms of automata and thus is far more versatile.

Toupie is different from this theoretical formalism for essentially two reasons.
– Atoms of the propositional μ-calculus are in the form $[X = Y]$, where X and Y are individual variables. If individual variables are interpreted in a finite domain, there is no substantial difference with Toupie. In general, the authors consider only individual variables belonging to $\{0, 1\}$. The extension to finite domain variables improves the efficiency, especially when dealing with arithmetical constraints (of course, it does not provide any improvement for what concerns expressiveness). Infinite interpretation domains would raise some effectiveness problems ...
– The μ-calculus does not allow to name relations and thus it does not consider systems of fixpoint equations. However, this extension is easy and useful.

3 DDs versus BDDs

Decision Diagrams : Decision Diagrams used in Toupie to encode relations are an extension for finite domains of the Bryant's Binary Decision Diagrams [4]. The BDD associated with a Boolean formula is a compact encoding of the decision tree describing the set of solutions of this formula. This encoding is canonical up to a variable ordering. Even if the worst case size of a BDD is exponential w.r.t. the number of variables, in many practical cases this size is quite small. Classical Boolean operations can be performed directly on BDDs. One of the most interesting features of BDDs is that, once built the BDDs associated with formulae, testing whether a formula is satisfiable or a tautology, testing the equivalence of two formulae becomes trivial. Due to space limitations, we cannot present BDDs here, and so we limit ourselves to innovations introduced in Toupie. The interested reader should see the paper by Bryant [6].

The main difference between BDDs and DDs is that a DD node is n-ary (and not binary), where n is the cardinality of the domain of the variable it is labeled with. Such a node encodes a *case* connective :

Let X be a variable, $\{k_1, \ldots, k_r\}$ be its domain, and $f_1, \ldots, f_r$ be formulae.

$$case(X, f_1, \ldots, f_r) = ((X = k_1) \wedge f_1) \vee \ldots \vee ((X = k_r) \wedge f_r)$$

The case connective keeps all good properties of the If-Then-Else (ITE) connective that labels the BDD nodes : It is orthogonal both with the connective $\neg$ — which makes a negation in constant time possible by putting a flag on negated edges, the leftmost outedge of a node being always positive to ensure the canonicity of the representation — and with the *ITE* connective — which makes possible to use exactly the same computation principle based on a unique connective (ITE) for DDs than for BDDs.

Let `X` and `Y` be two variables and `0..2` be their domain. The DD encoding the constraint `(X + Y <=1)` is pictured Fig. 2(a) (complemented edges are marked with a black dot).

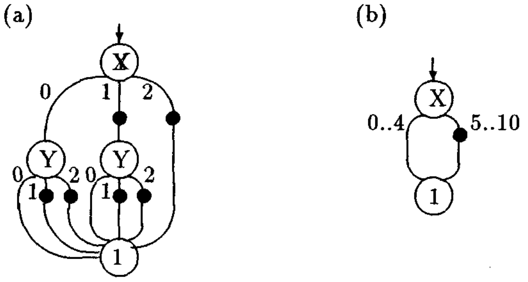

Fig. 2. The DD associated with $X + Y \leq 1, X, Y \in 0..2$ (a) and the compacted DD associated with $X < 5, X \in 0..10$ (b)

Compacted representation : When dealing with variables having rather large domains (it could be the case especially for numerical variables) many consecutive outedges of a node may point to the same son. In this case, one can compact the representation by labeling outedges with ranges of constants rather than with individual constant (such a DD is pictured Fig. 2(b)).

The "good" properties of DDs are preserved: negation in constant time, canonicity. The representation is canonical if any two adjacent ranges that point the same DD are merged. Logical operations can be accelerated by this coding since one can treat several values in one step when applying the recursive computation principle. E.g.

$$\begin{aligned}&ITE(case(X, 0..2 : F_1, 3..6 : F_2), case(X, 0..3 : G_1, 4..6 : G_2), 1)\\ &= case(X, 0..2 : ITE(F_1, G_1, 1), 3..3 : ITE(F_2, G_1, 1), 4..6 : ITE(F_2, G_2, 1))\end{aligned}$$

Note, finally, that such a representation allows the coding of approximations of relations between variables taking their values in dense domains : there is no need for X to be a natural number in the DD pictured Fig. 2. It could be a real variable as well.

Constraint solving : In order to solve systems of linear equations, we use the classical implicit enumeration/propagation technique (see [14]). Note that solving a system means here computing a DD that encodes all the solutions of the system. We had restricted allowed constraints to linear sets of linear inequations for sake of efficiency only.

The principle of the resolution is to going through a search tree and to build the DD in a bottom-up way. Each time a value is assigned to a variable (when descending along a branch of the tree) a filtering mechanism is applied to remove from the domain of the remaining variables the values that are not consistent with the current partial assignment. The propagation we adopt — the Waltz's filtering [19] — consists in maintaining, for each variable a minimum value and maximum value. The values between these two extrema are considered as allowed.

Let $\alpha_1 X_1 + \ldots + \alpha_n X_n \leq b$ be a linear inequation ($\alpha_i \neq 0$, $\forall i$). The maximum value of the variable X_i, denoted by $max(X_i)$, must verify the following inequality.

$$max(X_i) \leq max\left(\frac{b - \Sigma_{j \neq i} \alpha_j X_j}{\alpha_i}\right)$$

Propagating a modification in the domain of one of the X_j consists in updating each $max(X_i)$ according to the above inequality. Each time a variable domain is modified, this modification is propagated on the whole system until a fixpoint is reached.

Note that the compacted representation of DDs shown above is well adapted to this kind of propagation.

Variable ordering : Since the original paper by R. Bryant [5], it is well known that the size of a decision diagram (binary or not) crucially depends on the indices chosen for the variables.

By default, in Toupie, the variables are indexed with a very simple heuristic, known for its rather good accuracy. It consists in traversing the formula considered as a syntactic tree with a depth-first left-most procedure and to number variables in the induced order.

Nevertheless, this heuristic can produce very poor performances. The user is allowed to define its own indices. A variable declaration in the form `X@i` indicates that the variable `X` has the index `i`. For composite variables, declarations with fixed indices are in the form `^X@i!j`, meaning that the first field of `X` is numbered `i`, the second one `i+j`, the third one `i+2j` and so on.

What's new ? The idea that the case connective allows a canonical encoding of discrete functions is rather old and due, as far as we know, to J.P. Billon [2]. Buettner, in [7], used this encoding for multivaluated functions. However, none of these works fully implements BDDs technics as they are described in [4]. The extension from BDDs to DDs has been proposed in [18]. We proposed it independently in 1992. The compacted representation proposed here is original (at least in our knowledge). The integration of constraint solving technics and DDs is also new.

DDs versus Constraints A simple illustration of the power of constraints is as follows. Consider n variables $X_1, \ldots, X_n$ belonging to the range $[1, 10]$. The problem is to determine the tuples such that:

- $X_i \leq X_{i+1}$, for $i = 1, .., n-1$.
- $\sum_{i=1,..n} X_i = (n \times 10)/2$.

With an almost pure DD formulation, it would give something like:

```
((X1<=X2) & (X2<=X3) & (X3<=X4) & {X1+X2+X3+X4=20})
```

With a pure constraint formulation, it gives something like:

```
{X1<=X2, X2<=X3, X3<=X4, X1+X2+X3+X4=20}
```

The difference between the two formulations is that in the former case, one cannot use the inequalities to solve the equality. The cost is thus more or less proportional to an exploration of the whole cartesian product of variable domains. On the converse, with the latter formulation, each constraint is used to prune this exploration. Running times are as follows.

n	4	5	6	7	8	9	10
DD formulation	0s26	3s81	50s48	562s06	?	?	?
Constraint formulation	0s05	0s15	0s51	1s35	3s33	7s70	16s70

A comparison between the various possible formulations of a number of classical artificial intelligence puzzles and their relative efficiencies, including a comparison DD versus BDD, could be found in [17].

4 Symbolic Model Checking within Toupie

In this section we study, by means of an example, how Toupie is used to perform system of concurrent process analyses.

The notion of *transition system* plays an important role for describing processes and systems of communicating processes. A simple way to represent processes widely used in many works on verification, is to consider that a process is a set of *states* and that an *action* or an *event* changes the current state of the process and can thus be represented as a transition between states. Transition systems are also used to describe systems of communicating processes: states of the system are tuples of states of its components and transitions are tuples of allowed transitions. The resulting automaton is called by Arnold and Nivat the *synchronized product* [1]. This model is basically synchronous, even if it allows the description of non-synchronous phenomena.

Description. Let us consider the problem of designing a protocol between a resources dispatcher and a number of buffers. At the beginning, the dispatcher owns a given number of resources and the buffers are empty. During the process, each buffer tries to get one by one a number of resources from the dispatcher. When it has obtained them, it performs an action (no matter what this action actually is), then starts to give them back to the dispatcher (still one by one). When it is empty, it performs another action and starts to get resources again. And so on infinitely. The protocol must ensures some properties such as deadlock freeness, safety, fairness ...

Let us examine first a very simple version in which the dispatcher non deterministically gives a resource to a buffer or get a resource from a buffer without any additional control. In order to make the Toupie code sufficiently small to be easily readable we consider the case where there are only two buffers.

Individual Processes. The behavior is thus modeled by means of labeled transition system. For the dispatcher the states are the possible numbers of resources the dispatcher has, and the transitions model its actions, i.e. **get** (a resource from a buffer), **put** (a resource in a buffer) and **e** (when it remains idle). This transition system is described by using a ternary predicate **dispatcher(S,L,T)**, where the variables S, L, T are respectively the sources, labels and targets of transitions.

```
let resources = 5  /* number of resources */
let dispatcher_state = domain 0..resources
let dispatcher_label = domain {e,get,put}

dispatcher(S:dispatcher_state,L:dispatcher_label,T:dispatcher_state) += (
     ((L=e)   & (T=S))
   | ((L=get) & (T=S+1))
   | ((L=put) & (T=S-1)) )
```

The behavior of the two buffers is modeled in the same way. Here, states are pairs (section, current number of resources), where section is a Boolean variable indicating whether the buffer tries to get (up) or to put back (**down**) a resource. The actions performed by the buffer when it is full or empty are modeled by means of the same transition label **tau**.

```
let maxsize = 5  /* maximum size of buffers */
let buffer_size    = domain 0..maxsize
let buffer_section = domain {up,down}
let buffer_label   = domain {e,get,put,tau}
let buffer_state   = tuple (Size:buffer_size, Section:buffer_section)

buffer(^S@1!1:buffer_state,L@3:buffer_label,^T@4!1:buffer_state) += (
     ((L=e)   & (T.Size=S.Size)  & (T.Section=S.Section))
   | ((L=get) & (S.Section=up) & (T.Section=up) &
      {S.Size<maxsize, T.Size=S.Size+1})
   | ((L=put) & (S.Section=down) & (T.Section=down) &
      {S.Size>0, T.Size=S.Size-1})
   | ((L=tau) & ((T.Section=down)  & (S.Section=up)   &
      {S.Size=maxsize,T.Size=S.Size})
   | ((T.Section=up) & (S.Section=down) & {S.Size=0, T.Size=S.Size}))))
```

Variable indices are fixed in the above predicate. **S.Size**, **S.Section**, **L**, **T.Size**, **T.Section** receive respectively indices 1 to 5. In the remaining, we keep index declarations in order to provide a faithful Toupie session, but we don't discuss these choices. Such a discussion can be found in [11] and [9].

Synchronized Product. Now, one must synchronize the three processes, that is to constrain, for instance, the dispatcher to give a resource (**put**) when the first buffer gets it (**get**) while the second one remains idle (**e**) :

```
let label = tuple (D:dispatcher_label, B1:buffer_label, B2:buffer_label)

synchronizator(^L@3!5:label) += (
      ((L.D=e)   & (L.B1=tau) & (L.B2=e))
    | ((L.D=e)   & (L.B1=e)   & (L.B2=tau))
    | ((L.D=get) & (L.B1=put) & (L.B2=e))
    | ((L.D=get) & (L.B1=e)   & (L.B2=put))
    | ((L.D=put) & (L.B1=get) & (L.B2=e))
    | ((L.D=put) & (L.B1=e)   & (L.B2=get)))
```

Initial state and edges of the synchronized product are described as follows:

```
let state = tuple (
      D:dispatcher_state, ^B1@0!1:buffer_state, ^B2@0!1:buffer_state)

initial(^S@1!5:state) += ((S.D=resources) & (S.B1.Size=0) & (S.B2.Size=0))

edge(^S@1!5:state, ^T@4!5:state) += exist ^L@3!5:label (
     dispatcher(S.D,L.D,T.D)
   & buffer(S.^B1, L.B1, T.^B1)
   & buffer(S.^B2, L.B2, T.^B2)
   & synchronizator(^L) )
```

There are tuples of individual states that do not correspond to reachable states of the synchronized product. The set of reachable states is computed by means of a least fixpoint, starting from the initial state and traversing the automaton:

```
reachable(^T@4!5:state) += (
      initial(^T)
    | exist ^S@1!5: state (reachable(^S) & edge(^S,^T)))
```

Deadlocks. The predicates above (**reachable** and **edge**) allow the verification of properties of the system. Let us recall that a deadlock (in a weak sense) is a reachable state from which no transition is possible or only a transition leading in a deadlockstate. The Toupie program to detect deadlocks is as follows:

```
deadlock(^S@1!5:state) += (
    reachable(^S)
  & forall ^T@4!5:state (transition(^S,^T) => deadlock(^T)))
```

There are 10 deadlock states (in this toy example). A quick analysis shows that the problem arises when both buffers try to get (**up**) resources while the dispatcher has not enough resources to satisfy at least one of them.

The protocol must be modified to avoid this situation. A simple way to do this is to add the constraint that the dispatcher never gives a resource to a buffer if it has not enough resources to satisfy its request. This is done by modifying the synchronization constraints in the following way:

```
synchronizator(^S@1!5:state, ^L@3!5:label) += (
      ((L.D=e)   & (L.B1=tau) & (L.B2=e))
    | ((L.D=e)   & (L.B1=e)   & (L.B2=tau))
    | ((L.D=get) & (L.B1=put) & (L.B2=e))
    | ((L.D=get) & (L.B1=e)   & (L.B2=put))
    | ((L.D=put) & (L.B1=get) & (L.B2=e)   & (S.D>=maxsize-S.B1.Size))
    | ((L.D=put) & (L.B1=e)   & (L.B2=get) & (S.D>=maxsize-S.B2.Size)) )
```

The protocol shows now to be deadlock free. Note that the same kind of synchronizator could be used in order to break the symetries between processes. For instance, by forcing if all of the buffers are in their up sections, the buffer 1 to have more resources than the buffer 2 that must have itself more resources than buffer 3 and so on.

Fairness. An important property to be verified by the protocol is that a buffer that asks resources will always obtain these resources. Such a fairness property can be expressed as the greatest fixpoint of a least fixpoint. With the least one, we compute the set of states S such that every path leaving S goes to a state in which the first buffer is full (from symmetry arguments it suffices to consider only the first buffer). With the greatest one, we remove from the set the states that are not on infinite loops composed by states of the set.

```
transition(^S@1!5:state,^T@4!5:state) += (reachable(^S) & edge(^S,^T))

live(^S@1!5:state) += (reachable(^S) & ~deadlock(^S))

to_full_buffer1(^S@1!5:state) += (
    live(^S)
  & forall ^T@4!5:state
      (transition(^S,^T) => ((T.B1.Size=maxsize) | to_full_buffer1(^T))))

fair_state(^S@1!5:state) -= (
    to_full_buffer1(^S)
  & forall ^T@4!5:state (transition(^S,^T) => fair(^T)))
```

The computation reveals that the protocol is not fair. Actually, it is possible to make it fair, but it would be too long to present the new protocol here. Anyhow, the fairness property is interesting both from practical and theoretical points of view since it requires a greatest fixpoint computation.

Performances. The tables below indicate running times for various number of buffers, buffer sizes and initial number of resources respectively for the non-compacted representation (left), and the compacted one (right).

	5/5/5	5/5/10	5/5/15	5/5/25
states	832	58944	189696	248832
reachable	0s85	10s63	22s28	28s26
fairness	3s93	17s23	23s06	30s05

	5/5/5	5/5/10	5/5/15	5/5/25
states	832	58944	189696	248832
reachable	0s70	10s28	20s01	20s55
fairness	2s88	12s00	11s01	11s03

These examples show that Toupie can handle rather large examples. It is not surprising that limitations are due to lack of memory and not to excessive running times: it is in general the case with BDDs. It is also interesting to point out that the compacted representation really improves the performances.

5 Conclusion

Experiments reported in [9] show that Toupie is more efficient, on classical examples such as the Milner's scheduler or the dinning philosophers, than specialised tools based on BDDs such those described in [3, 11]. It shows that its expressiveness can be coupled a good efficiency, thanks to DDs.

Nevertheless, Toupie can be improved is several ways : DD management, introduction of arithmetic builtins, heuristics for variable indexing, ...

A very interesting way to extend Toupie is to handle constraints over dense domains (real or rational) in order to modelize real time systems. The union of two relations being approximated, as proposed by Halbwachs [12], by computing the convex hull of the union of each relation.

References

1. A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET "Les Mathématiques de l'informatique"*, 1989.

2. J.P. Billon. Perfect Normal Forms for Discrete Functions. Technical Report DSG/CRG/87014, Centre de Recherche, BULL, 1987.
3. A. Bouali. *Études et mises en œ uvre d'outils de vérification basée sur la bisimulation.* PhD thesis, Université Paris VII, 03 1993. in french.
4. K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference.* IEEE 0738, 1990.
5. R. Bryant. Graph Based Algorithms for Boolean Fonction Manipulation. *IEEE Transactions on Computers*, 35:677–691, 8 1986.
6. R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 1992.
7. W. Buettner. Unification in Finite Algebras is Unitary (?). In 9^{th} *Conference on Automatic Demonstration*, volume 310. LNCS, 1988.
8. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *IEEE transactions on computers*, 1990.
9. M-M. Corsini and A. Rauzy. Symbolic Model Checking and Constraint Logic Programming: a Cross-Fertilization. In Don Sannella, editor, *Proceedings of the European Symposium on Programming ESOP'94*, volume 788. LNCS, 1994.
10. O. Coudert, C. Berthet, and J-C. Madre. Verification of Synchronous Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407. LNCS, 1989.
11. R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. *Journal of Distributed Computing*, 6:155–164, 6 1993.
12. N. Halbwachs. Delay Analysis in Synchronous Programs. In *Proceedings of the 5th international conference on Computer Aided Verification CAV'93*, volume 697 of *LNCS*. Springer Verlag, June 1993.
13. M. Henessy and R. Milner. Algebraic laws for non-determinism and concurrency. *J. Assoc. Comput. Mach.*, 32:137–161, 1985.
14. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. MIT Press, 1989.
15. J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *Proceedings of Principle of Programming Languages (POPL'87)*, january 1987.
16. D. Park. Fixpoint Induction and Proofs of Program Properties. *Machine Intelligence*, 5, 1970.
17. A. Rauzy. Toupie Version 0.25 : User's Manual. Technical Report 959-94, LaBRI – URA CNRS 1304 – Université Bordeaux I, 1994.
18. A. Srinivasan, T. Kam, S. Malik, and R.K. Brayton. Algorithms for Discrete Function Manipulation. In *Proceedings of International Conference on Computer Aided Design, ICCAD'90*, pages 92–95. IEEE, 1990.
19. D.L. Waltz. Generating semantic descriptions for drawings of scenes with shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. Mc Graw Hill, New York, 1975.