Toupie: the μ-calculus over Finite Domains as a Constraint Language

Marc-Michel Corsini and Antoine Rauzy
LaBRI, URA CNRS 1304 – Université Bordeaux I
351, cours de la Libération,
33405 Talence Cedex FRANCE
e-mail: {corsini, rauzy}@labri.u-bordeaux.fr

Abstract. In this paper, we report experiments we did with the constraint language Toupie. Toupie is a finite domain μ-calculus interpreter. In addition to classical functionalities of a finite domain constraint solver, it provides a full universal quantification and relations (predicates/constraints) can be defined as least or greatest fixpoints of equations. This expressiveness is coupled with a practical efficiency that comes from the management of relations via Decision Diagrams. We advocate the use of this paradigm to model and solve efficiently difficult constraint problems such as the computation of properties of finite state machines and the implementation of abstract interpretation algorithms for logic languages.

Key words: Automatic Deduction, Constraint Languages, μ-calculus

1. Introduction

Constraint Logic Programming (CLP) has shown to be a very attractive field of research over recent years, and languages such as CLP($\mathcal{R}$) [40], CHIP [54] and PrologIII [21] have proved that this approach opens Logic Programming to a wide range of real life problems. Advantages of CLP are well known: declarativity, versatility, fast prototyping and last but not least, efficiency. A large part of the CLP success is due to finite domain constraint solvers. Languages of the family CLP($\mathcal{FD}$) rely on the paradigm enumeration/propagation. It follows that they are mainly designed to find one solution to a given problem, eventually an optimal one according to a given criterion (objective function). Problems handled with CLP($\mathcal{FD}$) come mainly from operation research (see [48] where several applications are presented).

In this paper, we are interested in problems whose solutions are not individual assignments of variables, but sets of such assignments. These problems come typically from model checking where one is interested in determining sets of states of a given transition system that verify a given property. In general, they cannot be handled directly within available CLP($\mathcal{FD}$) systems because their resolution requires universal quantifications and computations of fixpoints. The motivation of our work was to study whether the CLP approach can be applied to this kind of problems. In other words, we wanted to investigate the ways constraint logic programming and model checking can be cross-fertilized.

To this aim, we designed the constraint system Toupie. In addition to classical functionalities of finite domain constraint solvers, it provides a full universal quantification and definitions of relations as least or greatest fixpoints of equations. These definitions can be seen as a kind of quantification over relations or, in other words, as second order constraints. Namely, Toupie is a finite domain μ-calculus interpreter.

The propositional μ-calculus is a logic that permits the description of properties of finite state machines and that can be seen as an assembly language for temporal logics (see for instance [50, 13]). From this point of view, Toupie is close to model checkers such as the Concurrency Workbench [18] or SMV [45], or to programs computing the fixpoint of the Loyd's T_P operator [43] of Datalog programs. However, Toupie has been designed with the will to obtain a language, i.e. a versatile tool, conversely to model-checkers that are specific purpose tools.

In this paper, we advocate the use of the μ-calculus over finite domain variables to model and to solve efficiently interesting and difficult problems such as the analysis of two players games [24], the implementation of abstract interpretation algorithms for logic languages [22] and the computation of properties of finite state machines [23, 24, 7].

We show that this expressiveness can be coupled with a practical efficiency thanks to the management of relations via Decision Diagrams, an extension to finite domain of Bryant's Binary Decision Diagrams (BDD's) [8, 6, 9]. We report experiments showing that Toupie performances are comparable to those of classical model-checkers.

A tool such as Toupie can be considered from different points of view:

As a new solver for CLP($\mathcal{FD}$) allowing a kind of relational calculus within this framework. This solver could come in addition to the already embedded ones.

As a new paradigm for constraint logic languages. In this case, the μ-calculus should be extended in order to be a full programming language. We give some indications on this question in the conclusion.

As a versatile model checker that can be used for prototyping and pedagogical purposes and from which can be derived customized tools.

The remaining of the paper is organized as follows: Section 2 is devoted to a presentation of the Toupie language. A formal semantics of Toupie program is given section 3. In Section 4, we give a short presentation of the Decision Diagrams. Sections 5, 6 and 7 are devoted to applications: problem solving, abstract interpretation of logic programs and computation of finite states machines properties.

2. Informal Presentation of Toupie

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.

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 lines. Then the turn changes. The winner is the player who takes the last matches.

Variables, Constants, Domains, Variable Tuples: 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 variable tuple that groups several individual variables. Before using a variable tuple, one must declare its type. For the Nim's game with three lines of matches, this is done 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`. As in Prolog, variable and constant identifiers begin respectively with upper and 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 variable tuple `T` is denoted by `T.F`. When a variable tuple is manipulated *per se* its identifier is prefixed with a caret: `^T`. All of the usual connectives are available, including $\neg$, $\vee$, $\wedge$, $\Rightarrow$, … (denoted respectively by `~`, `|`, `&`, `=>`), as well as all of the linear inequation relations $=$, $\neq$, $\leq$, … (denoted respectively by `=`, `#`, `<=`).

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` is thus 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:

```
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}
      ))
```

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, 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. The quantifier `forall` has its intuitive meaning. In response to the above query, one obtains:

```
{S.P=a,S.L1=0,S.L2=0,S.L3=0}
{S.P=b,S.L1=0,S.L2=0,S.L3=0}
```

This encodes the two final positions (player `a` wins or player `b` wins). 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 tuples belonging to the relation. The result of a computation is thus an unique relation eventually containing several tuples as it could be obtained in Prolog by means of the meta-predicate `bagof`.

Fixpoints: All of the possible 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 natural characterization of reachable positions is recursive. It is a typical least fixpoint definition. Actually, it is not possible to remove recursivity since reachability is not first order expressible (see for instance [49]).

Toupie predicates are always 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. 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))
```

At this point, we have presented almost all of the fundamental constructions of the μ-calculus over finite domain variables (as it is implemented in Toupie): constraints, universal and existential quantifications, systems of fixpoint equations. It should be point out that in order to find winning positions of the Nim game, we maked an extensive use of universal quantification and fixpoint definitions. Indeed, both can be programmed in classical theorem provers such as Prolog by using auxiliary lists in which already encountered positions are stored. However, such a meta-programmation introduces procedural aspects in the description of the problem, breaking in this way the declarivity of the language. In addition, it would be very inefficient as soon as the number of positions becomes large. The table I gives some running times necessary to compute reachable and winning positions on a SUN IPX Spark Station with 48 MgB memory.

TABLE I
Running times for the Nim's game.

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

3. The μ-calculus over finite domain variables

This section is devoted to the denotational semantics of Toupie programs and formulae, i.e. the μ-calculus over finite domain variables. For sake of clarity, we present it for an abstract syntax (the translation from the concrete syntax, sketched in the previous section, to this abstract one is straightforward). In addition, we consider atomic constraints in a very abstract way. Taking into account actual atomic constraints, especially systems of linear inequations, is easy but tedious and adds nothing to our purpose. Finally, we consider a restricted version of program in which there is only local fixpoint definition. A full denotational semantics of Toupie would be too complicate to be interesting here.

3.1. Abstract Syntax

Formulae are built over four denumerable (distinct) alphabets: a set $\mathcal{X} = \{X_1, \ldots\}$ of *individual variables*, a set $\mathcal{K} = \{k_1, k_2, \ldots\}$ of *constants*, a set $\mathcal{C} = \{c_1, c_2, \ldots\}$

of *atomic constraints*, and a set $\mathcal{P} = \{p_1, p_2, \ldots\}$ of *predicate variables*. Each individual variable X_i is associated with its *domain*, i.e. a finite subset of $\mathcal{K}$. In the sequel, the domain of X_i is denoted by $dom(X_i)$ or by D_i when i is clear from the context. An arity, i.e. a positive integer, is associated with each atomic constraint and each predicate variable.

The set of formulae is smallest set such that:

- If $X_1, \ldots, X_n$ are individual variables or constants and r is an atomic constraint or a predicate variable of arity n then $r(X_1, \ldots, X_n)$ is a formula.
- If f and g are two formulae then so are $\neg f$, $f \wedge g$, $f \vee g$, $f \Rightarrow g \ldots$
- If f is a formula and X is an individual variable then $\exists X f$ and $\forall X f$ are formulae.
- If $X_1, \ldots, X_n$ are individual variables, p is a predicate variable and f is a formula monotone in p (i.e. in which p does not occur under an odd number of negation) then $\mu p(X_1, \ldots, X_n).f$ and $\nu p(X_1, \ldots, X_n).f$ are formulae.

Note that concrete equations do not appear in this abstract syntax. It means that some work must be performed to handle not only individual fixpoints but systems of fixpoint equations. This has been already done in the literature, (see for instance [17]). Intuitively, a system of fixpoint equations is translated into a tuple of μ-calculus formulae, one for each equation. Then, the semantic function is applied pointwisely.

Note also that usual duality rules apply here too.

$$\begin{aligned}
\neg(f \vee g) &= (\neg f \wedge \neg g) \\
\neg \exists X f &= \forall X \neg f \\
\nu p.f &= \neg \mu p.\neg f[p \leftarrow \neg p]
\end{aligned}$$

Where $f[p \leftarrow g]$ denotes the formula f in which g has been substituted simultaneoulsy for all of the occurrences of p.

Finally, we consider in what follows only *closed* formulae, i.e. formulae in which each individual variable appear within the scope of a quantifier or as the parameter of a fixpoint formula and each predicate variable appear within the scope of a fixpoint formula.

3.2. Denotational Semantics

As said in the previous section, a formula in which occur the variables $X_1, \ldots, X_n$ whose domains are respectively $D_1, \ldots, D_n$ is interpreted into the powerset $2^{\mathcal{D}}$ of the cartesian product $\mathcal{D} = D_1 \times \ldots \times D_n$, with respect to an environment ρ. An environment is a function that maps each atomic constraint and predicate variable of arity n into a subset of the cartesian product $\mathcal{K}^n$.

We denote by $Var(f)$ the set of individual variables occurring in a formula f.

We denote by $\rho[p \mapsto R]$ the environment that results by updating the binding of the predicate variable p to R in ρ.

Let $X_1, \ldots, X_n$ be variables whose domains are respectively $D_1, \ldots, D_n$.

Let R be a relation over the $X_1, \ldots, X_n$ and $X_{i_1}, \ldots, X_{i_k}$ be a subset of $X_1, \ldots, X_n$. We denote by $\pi_{\{X_{i_1}, \ldots, X_{i_k}\}}(R)$ the projection of R on the indices $i_1, \ldots, i_k$.

Now, let P be a relation over $X_{i_1}, \ldots, X_{i_k}$, i.e. a subset of $D_{i_1} \times \ldots \times D_{i_k}$. We denote by $\sigma_{\{X_1, \ldots, X_n\}}(P)$ the maximal subset Q of $D_1 \times \ldots \times D_n$ such that $\pi_{\{X_{i_1}, \ldots, X_{i_k}\}}(Q) = P$.

By abuse, we denote by 1 the full relation, when the underlying powerset is clear from the context.

Finally, we denote by $\backslash$ the set difference.

Semantics of formulae is as follows. Let $X_1, \ldots, X_n$ be individual variables or constants and $D_1, \ldots, D_n$ be their domains (by extension, the domain of a constant k is $\{k\}$). Let f and g be formulae, and let c and p be respectively an atomic constraint and a predicate variable of arity n.

$$\begin{aligned}
[\![c(X_1, \ldots, X_n)]\!]\rho &\overset{\text{def}}{=} \rho(c) \cap D_1 \times \ldots \times D_n \\
[\![\neg f]\!]\rho &\overset{\text{def}}{=} 1 \backslash [\![f]\!]\rho \\
[\![f \vee g]\!]\rho &\overset{\text{def}}{=} \sigma_{Var(f \vee g)}([\![f]\!]\rho) \cup \sigma_{Var(f \vee g)}([\![g]\!]\rho) \\
[\![f \wedge g]\!]\rho &\overset{\text{def}}{=} \sigma_{Var(f \wedge g)}([\![f]\!]\rho) \cap \sigma_{Var(f \wedge g)}([\![g]\!]\rho) \\
[\![\exists X f]\!]\rho &\overset{\text{def}}{=} \bigcup_{k \in dom(X)} [\![f[X \leftarrow k]]\!]\rho \\
[\![\forall X f]\!]\rho &\overset{\text{def}}{=} \bigcap_{k \in dom(X)} [\![f[X \leftarrow k]]\!]\rho \\
[\![p(X_1, \ldots, X_n)]\!]\rho &\overset{\text{def}}{=} \rho(p) \cap D_1 \times \ldots \times D_n \\
[\![\mu p(X_1, \ldots, X_n).f]\!]\rho &\overset{\text{def}}{=} \bigcap \{R \subseteq D_1 \times \ldots \times D_n | R \supseteq [\![f]\!]\rho[p \mapsto R]\} \\
[\![\nu p(X_1, \ldots, X_n).f]\!]\rho &\overset{\text{def}}{=} \bigcup \{R \subseteq D_1 \times \ldots \times D_n | R \subseteq [\![f]\!]\rho[p \mapsto R]\}
\end{aligned}$$

The syntactic restriction on the occurrences of p in formulae of the form $\mu p.f$ and $\nu p.f$ ensures that semantically, the bodies (f) rise to monotone functions in the powerset $\mathcal{K}^n$. In addition, the semantics of a formula is always a finite subset of $\mathcal{K}^n$. Let $\mathcal{D} = D_1 \times \ldots \times D_n$ be a finite subset of $\mathcal{K}^n$. $2^{\mathcal{D}}$ equipped with the set inclusion forms a complete lattice. The Knaster-Tarksi's theorem [53] asserts that given a monotone function f from $2^{\mathcal{D}}$ to $2^{\mathcal{D}}$,

1. f is continuous.
2. The equation $R = f(R)$ ($R \in 2^{\mathcal{D}}$) 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.

Finally, the following lemma holds, that asserts that the semantics of a formula does not depend of its environment.

LEMMA 1. *Let f be a formula and ρ and ρ' be two environments that interpret in the same way atomic constraints. Then $[\![f]\!]\rho = [\![f]\!]\rho'$.*

3.3. 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 [36]. Toupie is closer to the Park's original presentation [50] (used for instance in [13]). 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 [17].

In our informal presentation, we omit nested fixpoints. In the literature, nested fixpoint definitions are used mainly to express infinite path properties on graphs. For instance, states of a graph `g` that are source of paths going infinitely often through states having a property `p` are characterized by means of a $\nu\mu$ term, i.e. a greatest fixpoint of a least fixpoint, as follows.

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

Nested fixpoints are not strictly necessary (except for sake of efficiency) since any expression with nested fixpoint can be rewritten into a expression that is just a fixpoint hierarchy. This result is due to Immerman [38]. For instance, states characterized above can be computed in two steps. First, one computes the transitive closure of the automaton, second one selects states s such that there exists a path from s to a state t verifying p and conversely a path from t to s. Indeed, this requires to compute a binary relation which is more costly than the presented program. Moreover, we don't know any "natural" property requiring more than two nested fixpoints.

4. Decision Diagrams

The interest of a CLP($\mathcal{X}$) language is not only a matter of expressiveness: it stands also in the practical efficiency of the embedded solver for $\mathcal{X}$. CLP($\mathcal{FD}$) systems solves first order constraints, i.e. decision or optimization problems that are in general NP-complete. The resolution of second order constraints, such as those expressible within the μ-calculus over finite domain variables, is far more difficult because the size of solutions (that are sets of tuples) can be exponential w.r.t. the number of variables. Therefore, the design an efficient method to solve such constraints consists mainly in chosing a data structure as compact as possible to store relations. Fortunately, Bryant's Binary Decision Diagrams (BDD for short) [8, 6, 9] provide such a data structure. 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 BDD's. One of the most interesting features of BDD's is that, once built the BDD's associated with formulae, testing whether a formula is satisfiable or is a tautology, testing the equivalence of two formulae becomes trivial.

In Toupie, we extend BDD's techniques to Decision Diagrams, i.e. to the case where variables are not Boolean but take their values into finite domains. The aim of this paper is not to present BDD's and DD's, so we limit ourselves to what is necessary to discuss innovations introduced in Toupie. For a comprehensive survey on BDD's, the interested reader should see the paper by Bryant [9].

Reduced Ordered Decision Diagrams: The main difference between BDD's and DD's is that nodes of DD's are of various arities (and not always binary).

Let X_1, $\ldots$, X_n be variables whose domains are D_1, $\ldots$, D_n and let $\mathcal{D} = D_1 \times \ldots \times D_n$. A *Decision Diagram* α for $X_1, \ldots, X_n$ is a directed acyclic graph with an unique root node and verifiying conditions (i) and (ii).

(i) α has two sink nodes labeled respectively with 0 and 1.

(ii) Each internal node of α is labeled with a variable X_i whose domain is $D_i = \{k_1, \ldots, k_r\}$ and has r outedges labeled respectively with $k_1, \ldots, k_r$.

The denotation $[\![\alpha]\!]$ of the root node α of a DD is a formula defined as follows. If α is a leaf labeled with 0 or 1 then $[\![\alpha]\!]$ is the corresponding Boolean constant. If α is an internal node labeled with X_i and whose outedges point respectively to $\alpha_1, \ldots, \alpha_r$, then:

$$[\![\alpha]\!] \stackrel{\text{def}}{=} \mathit{case}(X_i, [\![\alpha_1]\!], \ldots, [\![\alpha_r]\!])$$

Where, the *case* connective is defined as follows. Let X be a variable and $\{k_1, \ldots, k_r\}$ be its domain, let $f_1, \ldots, f_r$ be formulae. Then,

$$\mathit{case}(X, f_1, \ldots, f_r) \stackrel{\text{def}}{=} ((X = k_1) \wedge f_1) \vee \ldots \vee ((X = k_r) \wedge f_r)$$

Let $<$ be a total order on the variables $X_1, \ldots, X_n$. A DD is said *ordered* if condition (iii) holds.

(iii) For any pair of nodes (α,β) labeled respectively with the variables X_i and X_j, if β is reachable from α then $X_i < X_j$.

A DD is said *reduced* if conditions (iv) and (v) hold.

(iv) It contains no internal node whose outedges are all pointing to the same sub-graph.

(v) It contains no isomorphic sub-graphs.

Condition (iv) follows from the equivalence: $\mathit{case}(X, f, \ldots, f) \equiv f$.

It is clear that for any relation R in $2^{\mathcal{D}}$ it exists at least one reduced ordered decision diagram (RODD) whose denotation is a formula encoding R. Moreover, the following property holds.

LEMMA 2 (Canonicity of reduced ordered decision diagrams). *For given a variable order, the RODD associated R is unique up to an isomorphism.*

In the sequel, we will write simply DD for RODD.

Negation: The *case* connective keeps another interesting property of BDD's: it is orthogonal with the negation:

$$\neg \mathit{case}(X, f_1, \ldots, f_r) = \mathit{case}(X, \neg f_1, \ldots, \neg f_r)$$

The DD encoding the complementary of a relation R is thus obtained from the DD encoding R by exchanging its leaves. This makes a negation in constant time possible by putting flags on edges that indicate whether the pointed DD's must be considered positively or negatively. In order to preserve the canonicity of the representation, one uses orthogonality to encode only nodes with a positive leftmost outedge. As a side effect, only one leaf remains necessary (the other one being its negation).

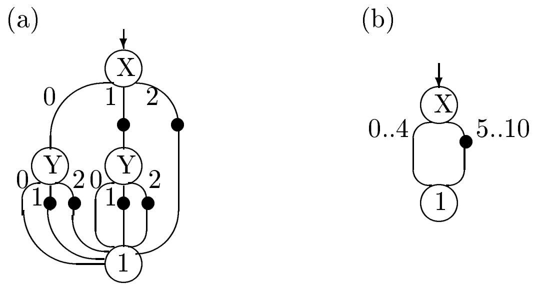

Fig. 1. The DD associated with $X + Y \leq 1, X, Y \in 0..2$ (a), and the compacted DD associated with $X < 5, X \in 0..10$ (b) (complemented edges are marked with a black dot).

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

Memory Management for DD's: DD's are a very compact representation thanks to the sharing of isomorphic subtrees. This sharing is automatically performed by keeping nodes into a hashtable: each time a node $case(X, \alpha_1, \dots, \alpha_r)$ is required, one first looks up the table and the node is created only if it is not already in the table (as we will see DD's are always created in a bottom-up way, which makes the above principle possible).

Logical Operations on DD's: In order to compute the DD associated with a formula $f = g \odot h$, where $\odot$ is any usual connective, one first computes the DD's associated with g and h, then one performs $\odot$ on these DD's. Usual logical operations ($\vee$, $\wedge$, $\otimes$, ...) are performed by means of a single connective, so-called *ite* for If-Then-Else. *ite* is defined as follows:

$$ite(f, g, h) \stackrel{\text{def}}{=} (f \wedge g) \vee (\neg f \wedge h)$$

Usual connectives can be rewritten using an *ite* and possibly a negation:

$$\begin{array}{lcl@{\qquad\qquad}lcl} f \vee g &=& ite(f, 1, g) & f \wedge g &=& ite(f, g, 0) \\ f \Leftrightarrow g &=& ite(f, g, \neg g) & f \otimes g &=& ite(f, \neg g, g) \end{array}$$

Let α, β, γ be three DD's. The operation $ite(\alpha, \beta, \gamma)$ is performed by means of the following inductive principle:

$$ite(\alpha, \beta, \gamma) \;=\; case(X, ite(\alpha_{X \leftarrow k_1}, \beta_{X \leftarrow k_1}, \gamma_{X \leftarrow k_1}), \dots, ite(\alpha_{X \leftarrow k_r}, \beta_{X \leftarrow k_r}, \gamma_{X \leftarrow k_r}))$$

Where X is the least variable labeling the root of the DD's α, β and γ, $\{k_1, \dots, k_r\}$ is its domain and $\sigma_{X \leftarrow k}$ denotes the son of the DD σ pointed by the outedge labeled by k if the root node of σ is labeled by X, and σ itself otherwise.

It is easy to induce an effective procedure from this principle.

Another very important point that makes BDD's and DD's efficient in practice is that the computation procedure uses a learning mechanism: a second hast-table is used to store 4-tuples $< \alpha, \beta, \gamma, \rho >$ such that $\rho = ite(\alpha, \beta, \gamma)$. Each time a computation $ite(\alpha, \beta, \gamma)$ must be performed, one first looks up this table to see whether the result has not been already computed. If it not the case, the computation is actually performed and its result stored. This ensures that a computation is almots never performed twice which achieves a substantial improvement of efficiency.

Table II summarizes the main computational costs of operations on DD's.

TABLE II
Computational costs of operations on DD's

Creation of a new node	$\mathcal{O}(1)$
$ite(\alpha, \beta, \gamma)$	$\mathcal{O}(\lvert\alpha\rvert \times \lvert\beta\rvert \times \lvert\gamma\rvert)$
$\exists X\alpha$, $\forall X\alpha$	$\mathcal{O}(\lvert\alpha\rvert^{\lvert dom(X)\rvert})$

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. 1(b)).

"Good" properties of DD's are preserved by this new compaction. The representation is canonical if any two adjacent ranges that point to the same DD are merged. Logical operations may be even accelerated since one may treat several values at once.

Note finally that such a representation could be used to encode approximations of relations between variables taking their values in dense domains.

Constraint solving: In order to solve systems of linear equations, we use the classical implicit enumeration/propagation technique (similar, for instance, to one embedded in CHIP [54]). Note that solving a system means here computing a DD that encodes all the solutions of the system.

The principle is to enumerate variables domains in the order of their indices, and to build the DD in a bottom-up way. The propagation we adopt — the Waltz's filtering [55] — consists in maintaining, for each variable a minimum and maximum value. Each time a variable domain is modified, this modification is propagated until a fixpoint is reached (see [42] for a discussion about these techniques).

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

Variable ordering: Since the seminal paper by R. Bryant [8], it is well known that the size of a (binary) decision diagram depends dramatically on the chosen order over variables. By default, Toupie orders variables with a very simple heuristic, known for its rather good accuracy. It consists in traversing the formula considered as a syntactic tree in a depth-first left-most way and to number variables in the induced order. Nevertheless, this heuristic sometimes produces very poor performances. The user is thus allowed to define its own indices.

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 [4]. Buettner, in [10], used this encoding for multivaluated functions. However, none of these works fully implements BDD's techniques as they are described in [6]. The extension from BDD's to DD's has been proposed in [52]. We proposed it independently in 1992. The compacted representation proposed here is original (at least in our knowledge). The integration of constraint solving techniques and DD's is also new.

5. Problem Solving

As shown at section 2, one can model mathematical games within Toupie thanks to universal quantification and fixpoints. In this section, we examine several very simple problems that permit a comparison of performances of the various ways constraint solving techniques and decision diagrams can be integrated.

Classical Artificial Intelligence Puzzles: We start with classical artificial intelligence puzzles that can be found in [41, 54]. These problems do not require second order constraints but they are of a special interest for our comparison purpose.

Let us first consider the pigeon-hole problem (recall that the problem is to put M pigeons into N holes in such way that there is at most one pigeon per hole).

The table III and IV give running times obtained with:

1. Toupie using N-ary DD's, the outedges being labeled with constants.
2. Toupie using N-ary DD's, the outedges being labeled with ranges.
3. Toupie using binary DD's. In order to encode that a variable takes its value in a domains of size N, one uses $\lceil log_2(N) \rceil$ Boolean variables. These variables being ordered consecutively.
4. Same as 3, but with an interleaved order on variables.
5. Toupie using binary DD's. In order to encode that a variable takes its value in a domains of size N, one uses N Boolean variables. These variables being ordered consecutively.

TABLE III
Running times for the Pigeon-Hole problem N/N

N	6	7	8	9	10	11	12	13	14	15
1	0s10	0s18	0s43	1s03	2s56	6s56	16s91	46s00	127s35	336s00
2	0s11	0s26	0s65	1s63	4s21	10s65	27s16	69s45	174s58	449s15
3	0s23	0s53	1s53	6s51	29s40	114s28	?	?	?	?
4	0s20	1s13	2s16	38s58	265s83	?	?	?	?	?
5	0s91	5s16	41s25	325s98	?	?	?	?	?	?
6	0s11	0s25	0s63	2s18	5s80	15s11	39s28	102s33	279s03	?
7				82s96	848s80	?	?	?	?	?

6. Same as 3, using the BDD package Aulne [51]. Aulne is implemented following the Bryant's paper [6] which makes it a good comparison tool.
7. The constraint logic language CHIP [54] using finite domain constraints[1].

Running times for instances with N pigeons and N holes ($N!$ solutions) are presented Table III. Those for instances with $N+1$ pigeons and N holes (no solution) are presented Table IV.

As one can see, DD's are slightly more efficient than BDD's one the Pigeon-Hole problem (moreover, Aulne is more optmized than Toupie as it appears clearly when comparing binary DD's with BDD's. The compacted representation has no interest on this example. However, it is not too bad too. The logarithmic coding of finite domains is more efficient than the linear one. Notice that is not true for enumerative methods such as the Davis and Putnam's procedure [28]. Finally, DD's give better results than enumerative methods on this problem as shown by the comparison with CHIP running times. However, this is not true in general since DD's are efficient only for problems having many regularities (symmetries).

Cryptogram: Let us consider now the following cryptogram.

```
  D O N A L D
+ G E R A L D
-------------
= R O B E R T
```

[1] Daniel Diaz communicated us these times, we would like to thanks him here.

TABLE IV
Running times for the Pigeon-Hole problem $N+1/N$

N	5	6	7	8	9	10	11	12	13	14
1	0s06	0s11	0s25	0s51	1s21	2s98	7s56	20s25	50s58	138s50
2	0s08	0s16	0s35	0s80	1s98	4s96	12s33	32s15	78s93	197s60
3	0s11	0s21	0s50	1s51	6s56	29s40	111s55	?	?	?
4	0s15	0s28	1s33	2s70	39s98	?	?	?	?	?
5	0s28	1s28	7s95	61s86	467s56	?	?	?	?	?
6	0s06	0s25	0s31	0s75	2s61	6s76	17s38	44s00	115s46	?
7					92s80	?	?	?	?	?

The problem is to assign a number in 0..9 to each letter in such a way that no number is assigned to more than one letter and that the addition is verified. This problem can be solved in several ways within Toupie, thanks to constraints and logical formulae.

The first way to solve the problem is to write a set of finite domain constraints (the $R_i's$ take their values in $\{0,1\}$).

```
donald(D,O,N,A,L,G,E,R,B,T) +=
  exist R1, R2, R3, R4, R5, R6 {
       2*D=10*R1+T,
    R1+2*L=10*R2+R,
    R2+2*A=10*R3+E,
    R3+N+R=10*R4+B,
    R4+O+E=10*R5+O,
    R5+D+G=10*R6+R,
    R6=0,
    D#O, D#N, D#A, D#L, D#G, D#E, D#R, D#B, D#T,
         O#N, O#A, O#L, O#G, O#E, O#R, O#B, O#T,
              N#A, N#L, N#G, N#E, N#R, N#B, N#T,
                   A#L, A#G, A#E, A#R, A#B, A#T,
                        L#G, L#E, L#R, L#B, L#T,
                             G#E, G#R, G#B, G#T,
                                  E#R, E#B, E#T,
                                       R#B, R#T,
                                            B#T,
    D#0, G#0, R#0
    }
```

A general heuristic in constraint problems is that variables with smallest domains must be assigned first. This is called the first fail principle in the literature. In the above formulation, with the a variable ordering that follows this principle, the unique solution computed in 1s75.

An alternative to the previous formulation is to use as few constraints as possible, leaving to decision diagram mechanisms the main part of the computation.

The equality is modeled by means of the following predicates:

```
equality(D,O,N,A,L,G,E,R,B,T) +=
  exist R1, R2, R3, R4, R5 (
      add(0, D,D,T,R1)
    & add(R1,L,L,R,R2)
    & add(R2,A,A,E,R3)
    & add(R3,N,R,B,R4)
    & add(R4,O,E,O,R5)
    & add(R5,D,G,R,0))

add(R1,I1,I2,O,R2) += (
    ((R1+I1+I2 < 10)    & (R2=0) & (O=R1+I1+I2))
  | ((R1+I1+I2+C >= 10) & (R2=1) & (O=R1+I1+I2-10)))
```

The solutions of `equality` are computed in 2s35. The mutual exclusion is just a pigeon-hole problem with ten pigeons (the letters) and ten holes (in 0..9). By combining both, one obtains the unique solution of `donald` in 4s95.

This example shows that even on typical CLP($\mathcal{FD}$) examples Toupie has good performances and that performances of DD's, if well used, can often be compared with performances of forward-checking algorithms.

Advantages of the Compacted Representation: The compacted representation could be far more efficient than the standard one. The following example, coming from data base literature, gives an illustration. The problem is to compute the relation "cousin" in a complete binary tree of height H (two vertices are cousins if they are at the same deep in the tree).

```
father(P,F) += ((F=2*P) | (F=2*P+1))

cousin(C1,C2) += (
    exist P (father(P,C1) & father(P,C2))
  | exist P1, P2 (father(P1,C1) & father(P2,C2) & cousin(P1,P2))
  )
```

The table V gives running times following the same numbering than in tables of results for the pigeon-hole problem.

Conclusion: This small set of problems is sufficient to show that from pure constraint solving to pure DD solving (with exhaustive or compacted representations)

TABLE V
Running times for Cousin

H	5	6	7	8	9	10
1	0s65	4s10	28s58	219s06	?	?
2	0s10	0s35	1s28	5s38	27s86	173s26
3	0s20	0s53	1s91	9s86	105s31	?

each mechanism has its advantages and its drawbacks. This is a strong argument in favour of embedding and integrating these mechanisms into a versatile tool.

6. Abstract Interpretation of Logic Programs

In this section, we give some arguments to show that the μ-calculus over finite domain variables is a natural framework to define abstract semantics of programming languages. We examplify that by using Toupie to perform abstract interpretations of logic programs. Abstract Interpretation of logic languages (and Prolog in particular) has been very active in the last few years. The motivation of these researches is the need of optimizations in compilers. Roughly speaking an abstract interpretation consists in executing a program over a finite abstract domain — which is a partition of the concrete Herbrand domain — in order to get information about the concrete executions of the program. The idea of using constraint languages for this purpose has been proposed in [20, 34] (but only from a theoretical point of view). Here we translate almost directly abstractions of definite logic programs into Toupie programs whose variables take their values in the chosen abstract domains. We only give an informal presentation of the method, for a detailed discussion see [22]. This approach can be related to works on abstract compilation, e.g. [37], but here we abstract the syntax of the programs as in [33, 34].

Semantics of definite logic programs: A *definite clause* is a clause of the form $H \leftarrow B_1, \ldots, B_n$, where H, B_1, ..., B_n are first order atoms. A *definite program* is a set of definite clauses. Lloyd in [43] showed that the semantics of a definite program P can be defined as the least fixpoint in the lattice 2^{B_P}, where B_P is the Herbrand's base of P, of the mapping $T_P : 2^{B_P} \rightarrow 2^{B_P}$, defined as follows. Let I be an Herbrand interpretation. Then

$T_P(I) \overset{\text{def}}{=} \{H \in B_P : H \leftarrow B_1, \ldots, B_n$ is a ground instance of a clause in P and

```
qsort(X1 , X2 ) :-
     X3 = [], qsort( X1 , X2 , X3 ).

partition(X1 , X2 , X3 , X4 ) :-
     X1 = [], X3 = [], X4 = [].
partition(X1 , X2 , X3 , X4 ) :-
     X1 =  [ X5 | X6 ] , X3 =  [ X5 | X7 ] , X5 <= X2,
     partition( X6 , X2 , X7 , X4 ).
partition(X1 , X2 , X3 , X4 ) :-
     X1 =  [ X5 | X6 ] , X4 =  [ X5 | X7 ] , X5 > X2,
     partition( X6 , X2 , X3 , X7 ).

qsort(X1 , X2 , X3 ) :-
     X1 = [], X3 = X2.
qsort(X1 , X2 , X3 ) :-
     X1 =  [ X4 | X5 ] ,
     partition( X5 , X4 , X6 , X7 ),
     qsort( X6 , X2 , X8 ),
     X8 =  [ X4 | X9 ] ,
     qsort( X7 , X9 , X3 ).
```

Fig. 2. Flat version of Quicksort in Prolog

$$\{B_1, \ldots, B_n\} \subseteq I\} \tag{1}$$

Abstract Domains: A typical example of abstract domains is `Prop` [44, 25]. This domain uses two constants `g` (for ground) and `ng` (for nonground). The intuition behind this is that a substitution θ is abstracted by a function f such that $f(x_i) = \mathtt{g}$ if and only if for all instances θ' of θ, θ' grounds x_i.

Flat Programs: For technical reasons, Toupie works on flat version of logic programs. Any pure logic programs can be translated into flat forms in a straightforward manner, thus it is not a restriction to only consider flat programs. The basic idea is to write explicitely all the unifications with two builtins $X = Y$ where X and Y are variables or $X = t$ where X is a variable and t a flat term *i.e.* a term with a functor and only variables as arguments. As an example, the flat version of the quicksort program using difference lists is given figure 2.

The Toupie translation: The translation of a (flat) Prolog program into a Toupie program is rather simple. To each Prolog literal corresponds a Toupie least fixpoint equation. Each sequence of Prolog literals is translated into a conjunction of predicate calls. Finally each set of clauses with same head is viewed as a disjunction of conjunction of relations. The figure 3 will enlight the process. It is

```
set domain {g, ng}

qsort(X1,X2) += (nil(X3) & qsort(X1,X2,X3))

partition(X1,X2,X3,X4) += (
    (nil(X1) & nil(X3) & nil(X4))
  | (cons(X1,X5,X6) & cons(X3,X5,X7) & ari(X5,X2) & partition(X6,X2,X7,X4))
  | (cons(X1,X5,X6) & cons(X4,X5,X7) & ari(X5,X2) & partition(X6,X2,X3,X7))
  )

qsort(X1,X2,X3) +=  (
    (nil(X1) & (X3=X2))
  | (
        cons(X1,X4,X5) & partition(X5,X4,X6,X7)
      & qsort(X6,X2,X8) & cons(X8,X4,X9) & qsort(X7,X9,X3)
    )
  )

nil(X1) += (X1=g)

ari(X1,X2) += ((X1=g) & (X2=g))

cons(X1,X2,X3) += (
    (X1=g) & (X2=g) & (X3=g)
  | (X1=ng & ((X2=ng) | (X3=ng)))
  )
```

Fig. 3. Toupie translation of Quicksort

rather clear, even without entering into technical details, that abstract semantics "à la Lloyd" of definite logic programs and denotational semantics of the translated Toupie programs coincide. We proved formally the correctness of the translation in in [22].

The remaining difficulty is to deal with the explicit unification of the form $X = Y$ and $X = t$ where X, Y are variables and t is any (flat) prolog term. The translation of the builtin $X = Y$ is straightforward, whereas the translation of the builtin $X = t$ heavily depends on the domain under interest, as an example figure 3 depicts the automated translation of the quicksort program for the domain `Prop` wherein `nil`, `cons` and `ari` are domain-dependent translation of built-ins.

As a response to the query `qsort(L1,L2)`, one obtains `{{L1=g, L2=g}`, `{L1=ng, L2=ng}}`, which means that `L1=L2` in the interpretation domain `Prop`.

Performances: We use the benchmark programs proposed by B. Le Charlier and P. Van Hentenryck in [14]. In the table VI, the running times (in seconds) for the domain `Prop` are presented as well as those for the domain `Types`. `Types`

TABLE VI
Running times for abstract interpretations

name	cs	disj	gabriel	kalah	peep	pg	plan	press1	press2	read
Prop	0s53	0s48	0s18	0s50	0s83	0s11	0s10	0s76	0s75	0s83
Types	0s20	0s30	0s18	0s35	0s55	0s11	0s06	0s68	0s70	2s40

contains four elements: `int` for integer, `lst` for lists, `cst` for symbolic constants and `fct` for the other terms. Note that only the definitions of the built-ins have to be changed according to the domain.

The results show that Toupie is extremely fast on the benchmark programs and much faster that any system we are aware of. Similar experiments for the `Prop` domain and the same programs have been reported in [15] for the generic abstract interpretation algorithm *GAIA*. Toupie is at least five time faster than *GAIA*, on the average. Interestingly, the implementation of [15] also uses decision diagrams for the domain and caching techniques. However, caching is used to a lesser extent. Finally, it is fair noticing that *GAIA* performs a top-down analysis implying that input patterns have to be recorded, not only success patterns. This also complicates the fixpoint algorithm. In [19], M. Codish & al use a similar approach to abstract interpretation over the domain `Prop`. Instead of translating logic programs to Toupie, Codish et al transform them to Datalog, and apply a simple *Tp* evaluation. Their approach is simpler since they do not use a constraint language. But, we believe that our approach is more powerful, since Extended DDs encoding permit larger interpretation domains without loss of efficiency, see the benchmarks in [22].

It is clear that the same kind of techniques could be applied to any languages with a well defined fixpoint semantics.

7. Symbolic Model Checking

In this section we show, by means of an example, how the μ-calculus over finite domain variables can be used to perform analyses of systems of concurrent processes. As an illustration, we detail the implementation in Toupie of a very simple and powerful model of concurrency that was proposed by A. Arnold and M. Nivat in [3] (see [2] for a comprehensive survey). This model consists in describing individual processes by means of labeled transition systems and in combining these transition systems in a convenient way to describe interactions between processes.

Formally, a labeled transition system is a tuple $\mathcal{A} =< S, L, T >$, where S is a set of states, L, is as set of labels, and T is a set of transitions, i.e. a subset of the cartesian product $S \times L \times S$. In a transition $< s, l, t >$, s is called the source state, l the label and t the target state.

A process is thus considered as a set of *states*. An *action* or an *event* changes the current state of the process. It is represented by a transition whose label is the name of the action or the event.

A system $\mathcal{P}$ of n communicating processes $P_1, \ldots, P_n$ is also described by means of a transition system $\mathcal{A}$, called the *synchronized product* of the transition systems $\mathcal{A}_1, \ldots, \mathcal{A}_n$ describing $P_1, \ldots, P_n$. States (resp. labels) of $\mathcal{A}$ are n-tuples of states (resp. labels) of the $\mathcal{A}_i$'s. Some constraints are put on the tuples of labels in order to describe the way processes communicate. The allowed tuples of labels are called *synchronization vectors.*

Formally, the synchronized product of n labeled transition systems $\mathcal{A}_1 =< S_1, L_1, T_1 >, \ldots, \mathcal{A}_n =< S_n, L_n, T_n >$ for the set V of synchronization vectors ($V \subseteq L_1 \times \ldots \times L_n$), is a labeled transition system $\mathcal{A} =< S, L, T >$ such that:

$$\begin{aligned} S &= S_1 \times \ldots \times S_n \\ L &= V \\ T &= \{< \vec{s}, \vec{l}, \vec{t} > \in S \times L \times S \mid \forall i = 1..n, < \vec{s}_i, \vec{l}_i, \vec{t}_i > \in T_i\} \end{aligned}$$

Where $\vec{x}$ denotes a n-tuple and $\vec{x}_i$ denotes its i-th component.

Once the labeled transition system describing the behavior of the system obtained, one can test whether the system verify some desired properties (such as deadlock freeness, safety, fairness and so on) by computing graph properties.

Note that labeled transition systems are actually the underlying model of most of other formalisms used to describe processes. This is the case, for instance, of the tractable restrictions of process algebrae such as CCS [46], as shown in [2]. In some sense, the way processes are described is just a matter of interface, and each tool has its own one: process algebrae for CWB [18], small languages to enter automata for Mec [1], Auto [29] or Aldébaran [32], or even a small imperative language for SMV [45]. Another difference between model checkers, is the way properties are expressed. A number of temporal logics are proposed in the literature, such as CTL [16] that is used in a several tools. As shown in [2], these logics are less expressive than the propositional μ-calculus (and for some of them strictly less expressive, see [30] for a survey on that question). The choice of the interface language is thus essentially a matter of taste. The advantage of an open tool such as a CLP system is precisely that th user is able to design its own interface.

Much more important is the way in which transition systems are encoded. The main drawback of the model is that their sizes become quickly huge, even for very simple systems. With an explicit representation transition systems with more than say 100,000 states cannot be handled (such sizes are not uncommon at all). In section 2, we showed how transition systems can be represented in a symbolic way by means of DD's. This idea is due to Madre and Coudert [27] and Clarke & als [13]. As examplified by the Nim's game, it allows to encode very large systems.

Example: In order to illustrate the approach, let us consider the problem of designing a protocol between a dispatcher of resources 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 it starts to give them back to the dispatcher (still one by one). When it is empty, it performs another action and it starts to get resources again. And so on infinitely.

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:buffer_state,L:buffer_label,^T: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}))))
```

For sake of simplicity, we do not discuss here the problem of the choice of a good order over the variables. Such a discussion can be found in [31, 5, 24].

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: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:buffer_state, ^B2:buffer_state)

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

edge(^S:state, ^T:state) +=
  exist ^L:label (
      dispatcher(S.D,L.D,T.D)
    & buffer(S.^B1, L.B1, T.^B1)
    & buffer(S.^B2, L.B2, T.^B2)
    & synchronizator(^L)
    )
```

The Arnold-Nivat model is basically synchronous, since all of the processes are assumed to perform an action at time. The trick of adding idle transitions `e` on each state allows, as shown above, to handle asynchronous phenomena.

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:state) += (
      initial(^T)
    | exist ^S: state (reachable(^S) & edge(^S,^T)))
```

Safety Properties: The predicates above (`reachable` and `edge`) allow the verification of properties of the system. For instance, one may want to check whether it fulfils safety properties such as deadlock freeness. 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 deadlock state. The Toupie program to detect deadlocks is as follows:

```
deadlock(^S:state) += (
    reachable(^S)
  & forall ^T: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:state, ^L: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 symmetries 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 Properties: 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:state,^T:state) += (reachable(^S) & edge(^S,^T))

live(^S:state) += (reachable(^S) & ~deadlock(^S))

to_full_buffer1(^S:state) += (
```

TABLE VII
Running times for the exhasutive representation

	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

TABLE VIII
Running times for the compacted representation

	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

```
    live(^S)
  & forall ^T:state
      (transition(^S,^T) => ((T.B1.Size=maxsize) | to_full_buffer1(^T))))

fair_state(^S:state) -= (
    to_full_buffer1(^S)
  & forall ^T: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 VII and VIII indicate running times for various number of buffers, buffer sizes and initial number of resources respectively for the non-compacted representation and the compacted one.

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.

TABLE IX
Running times to solve constraints $X_1 \leq \ldots \leq X_n$

sizes of domains	numbers of variables					
	5	6	7	8	9	10
5	0s01	0s02	0s04	0s08	0s14	0s21
10	0s11	0s36	1s02	2s64	6s22	13s63

Constraints: In the previous example, constraints have been used in rather poor way, i.e. just to model resource transfers. However, one could use them in a far more powerfull way. For instance, on could remark that buffers play a symmetrical role. Thus, in order to limit the combinatorial explosion we may constrain the first buffer to contain more resources than second one that itself must contain more resources than third one and so on (such approach has been proposed in [39]). This kind of constraints is expressed by a system in the form $X_1 \leq \ldots \leq X_n$. In tools such as SMV [45], tuples verifying this system are obtained by exploring the whole cartesian product of variable domains, which becomes quickly intractable. On the converse, constraint resolution permits to compute the DD associated with such a system rather efficiently as shown by the table IX.

CTL**:* It is interesting to show how formulae of temporal logics could be translated into Toupie formulae. As an example, let us consider the main connectives of the well known CTL* [30]: $\mathbf{E}p$, $\mathbf{A}p$ and $p\mathbf{U}q$ where p and q are path formulae.

$\mathbf{E}p$ characterizes the states s of the underlying graph such that there exists a infinite path, starting from s, on which the property p is always verified. If the graph is encoded by the predicate **g**, **E** is translated as follows.

```
ep(^S:state) -= (p(^S) & exist ^T:state (g(^S,^T) & ep(^T)))
```

$\mathbf{A}p$ is the same than $\mathbf{E}p$ excepted that all of the infinite paths starting from s must verify the property p.

```
ap(^S:state) -= (p(^S) & forall ^T:state (g(^S,^T) => ap(^T)))
```

Finally, $p\mathbf{U}q$ characterizes the states from which starts a path on which the property p is verified until a state verifying the property q is encountered.

```
pUq(^S:state) -= (q(^S) | (p(^S) & exist ^T:state (g(^S,^T) & pUq(^T))))
```

Bisimulations: The reduction by bisimulation is a very important tool for model checking [47]. A bisimulation is an equivalence relation between transition systems or different states of the same transition system (see the literature for a formal definition). The generic pattern of predicates that compute such bisimulations is as follows.

```
equivalent(^S:state,^T:state) -= (
    reachable(^S) & reachable(^T)
  & forall ^L:label (
        forall ^U:state
          (tau_path(^S,^L,^U)
           => exist ^V:state
                (tau_path(^T,^L,^V) & equivalent(^U,^V)))
      & forall ^V:state
          (tau_path(^T,^L,^V)
           => exist ^U:state
                (tau_path(^S,^L,^U) & equivalent(^U,^V)))

      )
  )
```

The reader familiar with bisimulations can easily see that the predicate `equivalent` mimics exactly the formal definition of bisimulations. In order to obtain different bismulations, it suffices to modify the definition of the predicate `tau_path`. For instance, the well known "observational equivalence" is obtained by encoding paths in the form $\tau^\star t \tau^\star$ in `tau_path`.

In [24], we have reported performances of two BDD based model-checkers [5, 31] and Toupie on the Milner's scheduler — which is a very common benchmark. These comparative results show that Toupie is at least as efficient as these tools.

8. Conclusion and Future Works

In this paper, we have presented several nontrivial applications of Toupie. These applications show that μ-calculus over finite domain variables has a great expressive power and that this expressiveness can be coupled with a good practical efficiency thanks to the use of Decision Diagrams.

The μ-calculus over finite domain variables is not however a complete language. As mentioned in the introduction, it can be considered from different points of view:

– As a new domain for CLP systems. Binary Decision Diagrams have been used in order to implement the Boolean solver of CHIP [12]. Buettner in [11] used an extension of BDD's to encode complex domain constraints. Toupie can be seen as an extension of this work in several ways: extension of BDD's to finite domains, smooth integration of finite domain constraints solving and DD's, and finally extension of the constraint expressiveness to the μ-calculus. There are strong motivations (thanks to applications) to introduce a Toupie-like solver in a CLP($\mathcal{FD}$) language in complement to to currently embedded ones. The introduction of universal quantification is rather simple. The introduction of a least fixpoint mechanism should

not be too difficult by using tabulation mechanism, while the introduction of greatest fixpoints is more tedious.
– As a new paradigm for constraint logic languages. In this case, it should be adapted in order to be a full programming language. This could be done in two ways: first restrict the language (for the constraint on the Herbrand universe) in order to make the relations computable by means of a tabulation mechanism. This implies to forbid general universal quantification and greatest fixpoints on this domain. Second, by using widening operators as proposed by the Cousot in [26]. This approach could be of a particular interest to analyze higher order functional languages. A very interesting way to extend Toupie is to handle constraints over dense domains (real or rational) in order to model real time systems. The union of two relations being approximated, as proposed by Halbwachs [35], by computing the convex hull of the union of each relation.

At this point, we think Toupie as a versatile model checker that can be used for prototyping and pedagogical purposes and from which can be derived tailored tools. In our experience, the μ-calculus, as most of temporal logics, seems too hard to be manipulated by non specialists. However, this drawback could be removed by providing the user with a set of predefined functions. On the other side, several translators from automata description languages to Toupie are already available, making writing of programs easier.

References

1. A. Arnold. MEC: a System for Constructing and Analysing Transition Systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer Verlag, June 1989.
2. A. Arnold. *Finite Transition Systems.* C.A.R Hoare. Prentice Hall, 1994. ISBN 0-13-092990-5.
3. A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET "Les Mathématiques de l'informatique"*, 1982.
4. J.P. Billon. Perfect Normal Forms for Discrete Functions. Technical Report DSG/CRG/87014, Centre de Recherche, BULL, 1987.
5. A. Bouali. *Études et mises en œuvre d'outils de vérification basée sur la bisimulation.* PhD thesis, Université Paris VII, 03 1993. in french.
6. K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45. IEEE 0738, 1990.
7. S. Brlek and A. Rauzy. Synchronization of Constrained Transition Systems. In H. Hong, editor, *Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO'94)*, pages 54–62, Linz, Ostreich, 1994. World Scientific Publishing.
8. R. Bryant. Graph Based Algorithms for Boolean Fonction Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
9. R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24:293–318, September 1992.
10. W. Buettner. Unification in Finite Algebras is Unitary (?). In *Proceedings of* 9^{th} *International Conference on Automated Deduction, CADE'9*, volume 310, pages 368–377. LNCS, 1988.

11. W. Buettner. Implementing Complex Domains of Application in an Extended Prolog System. *Journal of General System*, 15:129–139, 1989.
12. W. Buettner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, 1987.
13. 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.
14. V. Chandru and J.N. Hooker. Detecting embedded Horn structure in propositional logic. *Information Processing Letters*, 42:109–111, 1992.
15. B. Le Charlier and P. van Hentenryck. Groundness Analysis for Prolog: Implementation and Evaluation of the Domain prop. In *Proceedings of the 1993 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (*PEPM'93*)*, 1993.
16. E.M. Clarke, E.A. Emmerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. on Prog. Lang Syst*, 8:244–263, 1986.
17. R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal mu-calculus. In G.C. Bochmann and D.K. Probst, editors, *Proceedings of the Fourth international workshop on Computer Aided Verification, CAV'92*, volume 663 of *LNCS*, pages 410–422. Springer Verlag, 1992.
18. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.
19. M. Codish and B. Demoen. Analysing Logic Programs using prop-ositional Logic Programs and a Magic Wand. In *Proceedings of ILPS'93*, 1993.
20. P. Codognet and G. Filè. Computations, Abstractions and Constraints in Logic Programs. In *Proceedings of the IEEE International Conference on Computer Languages, ICCL'92*. IEEE Press, 1992.
21. A. Colmerauer. An Introduction to PrologIII. *Communications of the ACM*, 28(4), July 1990.
22. M-M. Corsini, B. Le Charlier, K. Musumbu, and A. Rauzy. Efficient Abstract Interpretation of Prolog Programs by means of Constraint Solving over Finite Domains (extended abstract). In *Proceedings of the 5th Int. Symposium on Programming Language Implementation and Logic Programming, PLILP'93*, volume 714, Tallin, Estonia, 1993. LNCS.
23. M-M. Corsini, A. Griffault, and A. Rauzy. Yet another Application for Toupie: Verification of Mutual Exclusion Algorithms. In A. Voronkov, editor, *Proceedings of Logic Programming and Automated Reasonning, LPAR'93*, volume 698, pages 86–97. LNAI, 1993.
24. 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 of *LNCS*, pages 180–194. Springer Verlag, 1994.
25. A. Cortesi, G. Filè, and W. Winsborough. Prop Revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, 1991.
26. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3), 1992.
27. C. Cubadda and M-D. Mousseigne. *Variantes de l'algorithme de SL-Resolution avec retenue d'informations.* PhD thesis, Université d'Aix-Marseille II, Faculté de Luminy, Laboratoire A.P.I., 1989.
28. M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *CACM*, 5:394–397, 1962.
29. R. de Simone and D. Vergamini. Aboard Auto. Technical Report RT 111, INRIA Sophia Antipolis, 1989.
30. E.A. Emerson. Temporal and Modal Logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1071. Elsevier, 1990.
31. R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. *Journal of Distributed Computing*, 6:155–164, 6 1993.

32. J.C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1989.
33. J.P. Gallagher and D.A. de Waal. Regular Approximations of Logic Programs and their uses. Technical Report CSTR-92-06, Computer Science Department of Bristol University, March 1992.
34. R. Giacobazzi, S. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proceedings of FGCS'92*, pages 581–591, 1992.
35. N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publisher, 1993. ISBN 0-7923-9311-2.
36. M. Hennessy and R. Milner. Algebraic laws for non-determinism and concurrency. *J. Assoc. Comput. Mach.*, 32:137–161, 1985.
37. M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, 1992.
38. N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Control*, 68:86–104, 1986.
39. C-W.N. Ip and D.L. Dill. Better verification through symmetry. In *Proceedings of the* 13^{th} *International Conference on Computer Hardware Description Language*, pages 87–100, 1993.
40. J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *Proceedings of the* 14^{th} *ACM Symposium on Principles of Programming Languages, POPL'87*, pages 111–119, January 1987.
41. J.L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10:29–127, 1 1978.
42. O. Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of the* 13^{th} *International Conference on Artificial Intelligence, IJCAI'93*, pages 232–238, 1993.
43. J.W. Loyd. *Foundations of Logic Programming.* Symbolic Computation. Springer Verlag, 1984.
44. K. Marriott and H. Søndergaard. Abstract Interpretation of Logic Programs : the Denotational Approach. In *Proceedings of the Quarto Convegno sulla Programmazione Logica*, 1990.
45. K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publisher, 1993. ISBN 0-7923-9380-5.
46. R. Milner. *A calculus of communicating systems.* LNCS. Springer Verlag, Berlin, 1980.
47. R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.
48. U. Montanari and F. Rossi, editors. *Proceedings of the First International Conference on Principles and Practice of Constraint Programming, CP'95*, volume 976 of *LCNS*. Springer Verlag, 1995.
49. C.H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994. ISBN 0-201-53082-1.
50. D. Park. Fixpoint Induction and Proofs of Program Properties. *Machine Intelligence*, 5, 1970.
51. A. Rauzy. Aulne Version 0.2 : User's Guide. Technical Report 834-94, LaBRI – URA CNRS 1304 – Université Bordeaux I, 1994.
52. A. Srinivasan, T. Kam, S. Malik, and R.K. Brayton. Algorithms for Discrete Function Manipulation. In *Proceedings of IEEE International Conference on Computer Aided Design, ICCAD'90*, pages 92–95. IEEE, 1990.
53. A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific. J. Math.*, 5:285–309, 1955.
54. P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. MIT Press, 1989. ISBN 0-262-08181-4.
55. 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.