Symbolic Model Checking and Constraint Logic Programming: a Cross–Fertilization

M.-M. Corsini, A. 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 present the constraint language **Toupie** which is a finite domain μ-calculus interpreter that uses extended decision diagrams to represent relations and formulae. "Classical" constraint logic programming languages over finite domains (CLP($\mathcal{FD}$)) are designed to find one solution to a constraint problem, eventually the best one according to a given criterion. In **Toupie**, constraints are used to characterize existing relationships between variables. We advocate the use of this paradigm to model and solve efficiently difficult constraint problems that are not tractable with CLP($\mathcal{FD}$) languages.
Keywords: Symbolic Model Checking, Constraint Languages

1 Introduction

Constraint Logic Programming (CLP) has shown to be a very attractive field of research over recent years, and languages such as CLP($\mathcal{R}$) [JL87], CHIP [Hen90] and PrologIII [Col90] have proved that this approach opens Logic Programming to a wide range of real life problems.

Languages of the family CLP($\mathcal{FD}$), with constraints over finite domains, are based on the paradigm enumeration/propagation. They are mainly designed to find one solution to a given problem, eventually the best one according to some criterion (objective function). They use widely algebraic properties of the underlying domain, i.e. the set of relative numbers.

In this paper, we present the constraint language **Toupie** which is based on a different paradigm: constraints are mainly symbolic and are used to characterize relationships existing between variables. Namely, **Toupie** implements an extention of the propositional μ-calculus to finite domains. The propositional μ-calculus is a language designed to model the behavior of systems of concurrent processes, where μ denotes a least fixpoint operator used to describe properties of finite state machines.

In addition to the classical functionalities of symbolic finite domain constraint languages, a full universal quantification is available in **Toupie** and one can define relations (predicates) as fixpoints of equations.

This gain in expressiveness is coupled with a practical efficiency that comes from the management of the relations via decision diagrams:
- Decision diagrams encode relations in a very compact manner (thanks to the sharing of the subtrees).
- The algorithm that computes logical operations between two decision diagrams uses a learning mechanism: the more computations it has performed, the more it is efficient.

The idea of using Boolean functions encoded by means of binary decision diagrams (BDDs for short [Bry92]) to manipalute relations is due to Mac Millan & al (see for instance [BMDH90]). Since this pioneering paper, many works have been done on symbolic model checking, where transition systems are encoded by means of BDDs. Very impressive examples have been shown, demonstrating how powerful this approach is. BDDs have been used also to implement Boolean solvers of CLP languages [BS87].

With **Toupie**, we extend these ideas to obtain a full constraint language and thus we open the μ-calculus to a large spectrum of applications. Of course, problems that can be handled in this paradigm are of a different nature than those handled in CLP($\mathcal{FD}$) (that come mainly from Operation Research). For instance, **Toupie** has been used to perform very efficient abstract interpretation of Prolog programs [CCMR93] and to verify mutual exclusion algorithms [CGR93].

In this paper, we show that **Toupie** is actually an efficient model-checker, or more precisely that the use of (extended) decision diagrams instead of binary ones (as done, for instance in [BMDH90, Bou93, EFT93]) improve the efficiency of symbolic model checking. We demonstrate "en passant" that the iterative squaring technique [Bry92], that seems a so pretty idea, is very doubtful in practice. We also show some funny issues in the computation of winning strategies in mathematical games.

It must be clear that these problems cannot be handled directly with the implemented solvers of CLP($\mathcal{FD}$), due to the need of universal quantification and fixpoints.

The remaining of the paper is organized as follows: Section 2 is devoted to a presentation of the **Toupie** language. Sections 3 and 4 are devoted to applications. Finally, we examine the relation with other works in section 5.

2 The Constraint Language Toupie

2.1 Syntax and Semantics of Toupie Programs

A number of different versions of the propositional μ-calculus have been proposed in the literature. Hereafter, we summarize the syntax of **Toupie** programs, which departs, for many (technical) reasons, from the usual approaches.

There are two syntactic categories in **Toupie**: *formulae* and *predicate definitions*. A **Toupie** program is a set of predicate definitions, having different head predicate symbols. A **Toupie** query is a formula. Formulae have the following form:

- The two Boolean constants `0` and `1`.
- `(X1=X2)` or `(X1=k)` or `(X1#X2)` or `(X1#k)` where `X1` and `X2` are variables and `k` is a constant symbol (`#` stands for disequality).
- `p(X1,..., Xn)` where `p` is an n-ary predicate variable and `X1,...,Xn` are individual variables.
- `~ f, f & g, f | g, f <=> g, ...` where `f` and `g` are formulae and `~`, `&`, `|`, `<=>` denote the logical connectives $\neg, \wedge, \vee, \Longleftrightarrow$.
- `forall X1,..., Xn f` or `exist X1,..., Xn f` where `X1,...,Xn` are variables and `f` is a formula.

Predicate definitions are as follows:

`p(X1,...,Xn) += f` or `p(X1,...,Xn) -= f` where `p` is an n-ary predicate variable, `X1,...,Xn` are individual variables, and `f` is a formula. The tokens `+=` and `-=` denote respectively least and greatest fixpoint definition of the equation `P(X1,...,Xn) = f`.

Each variable occurring in a fixpoint definition or request must have an interpretation domain. This domain must be declared with the first occurrence of the variable. A domain declaration is in the form: `X : { k1, ..., kn}` or `X : i..j` where `X` is a variable the `ki` are constant symbols and `i` and `j` are integers (and thus `i..j` denotes the corresponding range). It is possible to declare a default interpretation domain, and to name domains. In the following we denote by $dom(X)$ the interpretation domain of a variable X.

The semantics of `Toupie` programs is the attended one. That is that the fixpoint of an equation `p(X1,...,Xn) = f` is computed for the inclusion order in the powerset of $dom($`X1`$) \times \ldots \times dom($`Xn`$)$. The interested reader could refer to the appendix A for a precise denotational semantics. Note that the fixpoint definitions must be monotonic in order to ensure the existency of fixpoints and that this condition could be easily checked syntactically.

2.2 Decision Diagrams

Decision diagrams used in `Toupie` to encode relations, are an extension for symbolic finite domains of the binary decision diagrams [Bry92].

Shannon Decomposition of Relations

Definition 1. case connective
Let X be a variable, $dom(X) = \{k_1, \ldots, k_r\}$ be its interpretation domain, and $f_1, \ldots, f_r$ be formulae. Then:

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

Definition 2. Shannon Normal Form
A formula f is in Shannon normal form (SNF for short) if one of the following points holds:
- $f = 0$ or $f = 1$,
- $f = case(X, f_1, \ldots, f_r)$, where X is a variable and $f_1 \ldots f_r$ are formulae in SNF wherein X does not occur.

Property 3. Shannon Decomposition
Let $V = \{X_1, \ldots X_n\}$ be a set of variables, and *Const* be a set of constants. Then, for any n-ary relation $R : (V \rightarrow Const) \rightarrow \mathcal{B}$ there exists a formula in SNF encoding R.

Reduced Ordered Decision Diagrams We first define decision diagrams:

Definition 4. Decision Diagrams
Let $V = \{X_1, \ldots, X_n\}$ be a set of variables. A *Decision Diagrams* F is a directed acyclic graph such that:

- F has two leaves 0 and 1.
- Each internal node of F is labelled with a variable X belonging to V and if $dom(X) = \{k_1, \ldots, k_r\}$ then the node has r outedges labelled with $k_1, \ldots, k_r$.
- If a node labelled with the variable X is reachable from a node labelled with the variable Y then $X \neq Y$.

Now, it is clear that a decision diagram encodes a formula in SNF: the leaves encode the corresponding Boolean constants and each internal node encodes a *case* connective.

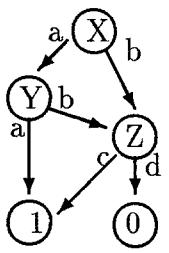
Now, we define a specific class of decision diagrams: reduced ordered decision diagrams.

Definition 5. Reduced Ordered Decision Diagrams
Let $<$ be a total order over the variables $X_1, \ldots, X_n$. A *Reduced Ordered Decision Diagram* F is a decision diagram such that:

- If a node labelled with the variable X is reachable from a node labelled with the variable Y then $X > Y$.
- Any node has at least two distinct sons ($case(X, f, \ldots, f) \equiv f$).
- Two distinct nodes F and G are syntactically distinct, i.e. either they are labelled with different variables or there exists an index i such that the i-nth son of F is distinct of the i-nth son of G (reduction by means of maximum sharing of the sub-graphs)

In the remaining, we will consider only Reduced Ordered Decision Diagrams and call them Decision Diagrams (or DD for short).

Example 1. Let X, Y, Z be variables and $dom(X) = dom(Y) = \{a, b\}$ and $dom(Z) = \{c, d\}$. Let $p(X, Y, Z) = \{\langle a, a, c\rangle, \langle a, a, d\rangle, \langle a, b, c\rangle, \langle b, a, c\rangle, \langle b, b, c\rangle\}$; then the DD associated with p for the order $X < Y < Z$ is pictured beside. It encodes the formula: $case(X, case(Y, 1, case(Z, 1, 0)), case(Z, 1, 0))$ which is equivalent to p.



Property 6. Canonicity
Let R be a n-ary relation on the variables $X_1, \ldots, X_n$ and let $<$ be a total order over these variables. Then, there exists *one and only one* DD encoding R.

It follows that the test of equality between two relations encoded by means of two DDs is reduced to a test between the addresses of the DDs.

Logical Operations on DDs Decision Diagrams are also very efficient for performing logical operations on relations. The following property holds:

Property 7. Induction Principle
Let $\odot$ be any binary logical operation and let $p = case(X, p_1, \ldots, p_r)$ and $q = case(X, q_1, \ldots, q_r)$ be two formulae in SNF. Then, the following equality holds:

$$case(X, p_1, \ldots, p_r) \odot case(X, q_1, \ldots, q_r) = case(X, p_1 \odot q_1, \ldots, p_r \odot q_r)$$

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

Memory Management for DDs Decision Diagrams encode relations over finite domains in a very compact way by means of the sharing of the subtrees. This sharing is automatically performed by storing the nodes in an hashtable: each time a node $case(X, p_1, \ldots, p_r)$ is required, one first looks up the table and the node is created only if the node does not belong to it.

Another very important point that makes DDs efficient in practice is that the computation procedure uses a learning mechanism: each time a computation $p \odot q$ is performed, the result is memorized in an hashtable. Thus, this computation is never performed twice. Since the time required to an access in the hashtable is quasi-linear, the overhead due to this memorization is negligible. Moreover, the improvement obtained is often very big in practice, and becomes more and more important as the size of the problem grows up.

Variable Ordering Since the original paper by R. Bryant, it is well known that the size of a decision diagram (binary or not) crucially relies on the indices chosen for the variables. In his paper, R. Bryant gives an example where the BDD can be either linear or exponential w.r.t. the number of variables following the variable indexing.

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 due to the projection operation. This operation is used each time a predicate `p(X1,...,Xn)` is called since the result of the computation of the corresponding fixpoint must be projected on the arguments of the call, here `X1,...,Xn`. Projection can be dramatically unefficient if arguments are not ordered as the formal parameters.

This is the reason why, the user is allowed to define its own indices by `X@i`, where `X` is the first occurrence of a variable and `i` is any integer.

Advanced Features The effective implementation of fixpoint computations uses some tricky algorithms (projection by renaming and tabulation, dependency graphs) that avoid useless works and increase dramatically the performances. The interested reader could see [CR93] for a detailed presentation.

3 Symbolic Model Checking within Toupie

3.1 The Arnold-Nivat Model of Concurrency

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 semantics and verification (*model checking*), 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 the two states. Transition systems are also used to describe systems of communicating processes: the states of the system are tuples of states of its components and the transitions are tuples of allowed transitions. The resulting automaton is called by Arnold and Nivat the synchronized product. This model of concurrency is the one used, for instance, in the model checker MEC [Arn89]. It is basically synchronous, even if it allows the description of non-synchronous phenomena.

The idea we use, first proposed by Mac Millan & al [BMDH90], is to encode transition systems in a symbolic way.

Individual Processes In order to illustrate this section, we model in `Toupie` the well-known Milner's scheduler [Mil89], a standard benchmark for process algebra tools [Bou93, EFT93]. The methodology remains the same for other problems such as the verification of mutual exclusion algorithms (see [CGR93]).

The scheduler consists of one starter process and N processes which are scheduled. The communication is organized in a ring. The transition system describing each cycler is depicted figure 1.

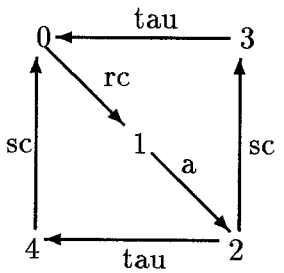


Fig. 1. The transition system encoding a cycler

Each cycler process C_i awaits the permit (rc) to start, performs the action a, and passes the turn (sc) to the next cycler either before or after some internal

computation (*tau*). The starter just initializes the process. A cycler is modeled in **Toupie** as follows:

```
let cycler_state 0..4      % definition of the domain "cycler_state"
let cycler_label {e,tau,a,sc,rc}

cycler(S:cycler_state,L:cycler_label,T:cycler_state) += (
    ((L=e) & (S=T))
  | ((S=0) & (L=rc)  & (T=1))  |  ((S=1) & (L=a)   & (T=2))
  | ((S=2) & (L=sc)  & (T=3))  |  ((S=2) & (L=tau) & (T=4))
  | ((S=3) & (L=tau) & (T=0))  |  ((S=4) & (L=sc)  & (T=0))
  )
```

The variables **S** and **T** stand for the sources and the targets of the transitions, the variable **L** stands for the labels of the transitions. Note that a transition labelled with **e** has been added. In the Arnold-Nivat Model, one considers that some action in some process can be executed only simultaneoulsy with some other action in the other processes. In order to represent asynchronous actions, one adds transitions of the form $s \xrightarrow{e} s$, where s is a state and e is the label of the empty transition.

Synchronization Vector Now, one must synchronize the different processes, that is to constrain, for instance, the cycler i to emit a message (transition *sc*) when the cycler $i+1$ receives the message (transition *rc*) and the other processes remain idle (transition **e**). The synchronization vector written as a **Toupie** rule is as follows:

```
synchronization_vector(
   SL:starter_label,C1L:cycler_label, ... ,CnL:cycler_label) += (
     ((SL=sc) & (C1L=rc)  & (C2L=e)  & ... & (CnL=e))
   | ((SL=e)  & (C1L=a)   & (C2L=e)  & ... & (CnL=e))
   | ((SL=e)  & (C1L=tau) & (C2L=e)  & ... & (CnL=e))
   | ((SL=e)  & (C1L=sc)  & (C2L=rc) & ... & (CnL=e))
   ...
   )
```

Synchronized Product The computation of the synchronized product, that is the automaton modeling the behavior of the system of processes, can be now performed. The set of the reachable states of this product is computed as shown figure 2. It requires a fixpoint computation since the reachable states are found in a breadth-first way. The variable indices are not given in the figure. As remarked for instance in [EFT93] (for the Boolean case), the best order is the interleaved one, that is: $SS < LS < TS < C1S < C1L < C1T < \ldots < CnS < CnL < CnT$. In the remaining we assume such an order.

3.2 Computing Properties

The predicates **reachable** and **edge** allow the verification of properties of the system:

```
edge(                          % allowed edges in the synchronized product
  SS:starter_state, ST:starter_state, % source and target in the starter
  C1S:cycler_state, C1T:cycler_state, % source and target in the cycler 1
  ...
  CnS:cycler_state, CnT:cycler_state) % source and target in the cycler n
  +=
  exist SL:starter_label, C1L:cycler_label, ... , C2L:cycler_label
    (
        starter(SS,SL,ST)
      & cycler(C1S,C1L,C1T) & ... & cycler(CnS,CnL,CnT)
      & synchronization_vector(SL,C1L,...,C2L)
    )

reachable(ST:stater_state,C1T:cycler_state, ..., CnT:cycler_state) += (
    initial_state(ST,C1T,...,CnT)
  | exist SS:starter_state, C1S:cycler_state, ... , C2S:cycler_state
      (reachable(SS,C1S,...,CnS) & edge(SS,ST,C1S,C1T,...,CnS,CnT))
  )
```

Fig. 2. Reachable states in the synchronized product

Dead-Locks Let us recall that a dead-lock is a state wherein no transition is possible or only transitions leading to a deadlock state. The `Toupie` program to detect dead-locks is as follows:

$$deadlock(\mathbf{S}) \mathrel{+}= (reachable(\mathbf{S}) \wedge \forall \mathbf{T}\,(reachable(\mathbf{S}) \wedge edge(\mathbf{S},\mathbf{T}) \Rightarrow deadlock(\mathbf{T}))$$

where $\mathbf{S}$ and $\mathbf{T}$ represent the variables ordered as previously.

Live-Locks The detection of live-locks is also a very important feature of a model checker. The problem arises when the modeled processes must share a critical ressource (a printer for instance). In this case, there is a live-lock in the system of processes if there is an infinite execution where:
– two processes attempt to access to their critical section, and never succeed
– none of the processes remains idle for ever.

The methodology consists in recomputing the set of reachable states by forbidding the states in which one process is in its critical section. There is no live-lock if and only if all the states of the obtained synchronized product are dead-locks.

Bisimulation 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 bisimulation generally considered on the Milner's scheduler is the observational equivalence that is to say that two states are equivalent

if and only if there is a path labeled with τ-transition joining them. This bisimulation is computed in **Toupie** in two steps: First, compute the τ-closure of the synchronized product, that is the paths of the form $\tau^* t \tau^*$, where t is any transition (predicate **tau_path**). Second, compute the equivalence relation between states using the extended edges above.

The second step is performed with a greatest fixpoint predicate:

$$
\begin{array}{l}
equivalent(\mathbf{X},\mathbf{Y}) \text{ -= (} \\
\quad reachable(\mathbf{X}) \wedge reachable(\mathbf{Y}) \\
\wedge \forall \mathbf{L} \text{ \% } \mathbf{L} \text{ is a vector of labels, } \mathbf{U} \text{ and } \mathbf{V} \text{ are vectors of states} \\
\qquad \forall \mathbf{U}\, tau\_path(\mathbf{X},\mathbf{L},\mathbf{U}) \Rightarrow \exists \mathbf{V}\, (tau\_path(\mathbf{Y},\mathbf{L},\mathbf{V}) \wedge equivalent(\mathbf{U},\mathbf{V})) \\
\quad \wedge\ \forall \mathbf{V}\, tau\_path(\mathbf{Y},\mathbf{L},\mathbf{V}) \Rightarrow \exists \mathbf{U}\, (tau\_path(\mathbf{X},\mathbf{L},\mathbf{U}) \wedge equivalent(\mathbf{U},\mathbf{V})))
\end{array}
$$

The point is that the predicate **equivalent** mimics exactly the formal definition of the observational equivalence. Note also that if one wants to compute another bisimulation, it suffices to change the definition of the predicate **tau_path**.

Other Properties In [CGR93], we show also how the fairness and the safety of a mutual exclusion algorithm can be studied in **Toupie**. In mutual exclusion algorithm the fairness is achieved whenever the following fact holds:
– If a process P_i wants to access to its critical section, it succeeds in finite time.
The safety is achieved if :
– Whenever a process P_i still remains in its non critical section (it does not attempt to reach critical section), then the mutual exclusion algorithm works.

All of these properties are expressed in **Toupie** in a very natural and declarative way.

Performances The table below indicates the running times for **Toupie** as well as those obtained by Bouali in the one hand [Bou93] and Enders & al in the other hand [EFT93] (the two last have been obtained on a SPARC 2 workstation, which is slightly faster than our own). These authors use BDDs based algorithms. The significant difference of performances in favour of **Toupie** comes, in our opinion, from the use of extended decision diagrams instead of binary onés. The interesting point is that very good performances can be obtained by using a general purpose constraint language instead of a specialized model checker.

processes	6	8	10	12	14	16	18	20
states	577	3073	15361	73729	$3\ 10^6$	$1.2\ 10^7$	$4.8\ 10^7$	$1.8\ 10^8$
transitions	2017	13825	84481	479233	$2\ 10^7$	$8\ 10^7$	$3.2\ 10^8$	$1.28\ 10^9$
reachable	0s70	1s30	2s03	3s36	4s41	5s76	7s36	9s36
Bouali	1s28	2s97	?	?	?	23s42	39s37	53s51
deadlock	0s10	0s13	0s18	0s21	0s26	0s30	0s38	0s38
bisimulation	4s08	6s70	9s95	14s23	18s01	23s20	28s40	34s76
Bouali	19s43	39s07	?	?	?	197s80	255s62	332s54
Enders & al	21s	40s	87s	145s	233s	348s	569s	850s

3.3 Iterative Squaring

A number of properties require to compute the transitive closure of the synchronized product, i.e. the pairs of global states $\langle \mathbf{S}, \mathbf{T} \rangle$ such that there exist a path from $\mathbf{S}$ to $\mathbf{T}$

The transitive closure can be computed in two ways. First as follows:

$$path(\mathbf{S}, \mathbf{T}) += edge(\mathbf{S}, \mathbf{T}) \quad \vee \quad \exists\, \mathbf{U}\ (\mathbf{edge}(\mathbf{S}, \mathbf{U}) \ \wedge \ path(\mathbf{U}, \mathbf{T}))$$

Second, by means of the iterative squaring technique mentioned as a very powerful method by several authors:

$$path(\mathbf{S}, \mathbf{T}) += edge(\mathbf{S}, \mathbf{T}) \quad \vee \quad \exists\, \mathbf{U}\ (\mathbf{path}(\mathbf{S}, \mathbf{U}) \ \wedge \ path(\mathbf{U}, \mathbf{T}))$$

This technique is widely used, for instance for computing powers. Let $\mathcal{K}$ be any ring and $X \in \mathcal{K}$ and $n \in \mathbb{N}$, then $X^{2n} = X \times X^{2n-1} = X^n \times X^n$. The second equality induces an iterative squaring method to compute a power.

Unfortunately, this pretty idea does not work for our purpose.

A critical example is the following: one considers N two states processes. At each step, one and only one of them changes of state. There is a single initial state. This example seems particularly in favour of the iterative squaring because:

- All the 2^N states of the free product are reachable.
- The number of iterations necessary to reach all the states is N while the number of squaring is $log_2(N)$.

Surprisingly, it is not the case, as shown in the following table.

processes	4	8	16	32	64	128
path (iter.)	0s05	0s16	0s76	3s16	14s23	68s96
path (squr.)	0s05	0s20	0s98	6s15	84s40	?

This phenomenon appears on almost all examples we have tried. A possible explanation could be that iterative squaring is efficient when the product of two objects has the same size than the objects themselves. Of course, it is not the case with DDs.

4 Winning Ways

4.1 Artificial Intelligence Classics

`Toupie` can be used to solve classical AI puzzles like N-Queens or Pigeon-Holes problems. Indeed, these problems do not require the expressiveness power of the language. Nevertheless, it is interesting to note that the performances are comparable with those obtained with CLP($\mathcal{FD}$) languages based on the enumeration/propagation paradigm. The tests were performed on a Sparc 1 IPX, with 16MB RAM and 16MB of swap space. The running times given for CLP($\mathcal{FD}$) have been obtained with (Cosytec) CHIP [Hen90] on a Sparc 2.

Queens The following table summarizes the running times for computing the Decision Diagrams that encode all the solutions of the N-queens problem for different values of N.

Queens	5	6	7	8	9	10
Toupie	0*s*05	0*s*06	0*s*25	0*s*58	2*s*20	6*s*73
CHIP	0*s*02	0*s*02	0*s*08	0*s*40	2*s*10	6*s*50

Pigeon-Hole The same for the pigeon-hole problem:

Pigeons/Holes	8/8	9/8	9/9	10/9	10/10	11/10	11/11	12/11	12/12	13/12
Toupie	0*s*05	0*s*45	0*s*25	1*s*25	2*s*73	3*s*11	10*s*31	9*s*75	25*s*63	34*s*40
CHIP	9*s*08	10*s*12	82*s*96	92*s*80	848*s*80	?	?	?	?	?

4.2 Games

More exciting is the analysis of two players mathematical games allowed in **Toupie**, thanks to the quantification and fixpoint mechanisms. The Nim game is a good illustration of this technics. Hereafter follows its rules:

The game begin with N lines numbered from 1 to N and containing $2 \times i - 1$ matches at line i. At each step, the player who has the turn takes as many matches as he wants in one of the line (but of course, at least one). Then the turn changes. The winner is the player who takes the last match.

In order to model the Nim game, one takes as many variables as there are lines (each variable taking its value in $0..2 \times i - 1$) plus one variable to model the turn. A move is represented by means of a predicate $move(\mathbf{S}, \mathbf{T})$ where the two vectors $\mathbf{S}$ and $\mathbf{T}$ encode two configurations of the variables (see section 3 for more explanations on the construction of this predicate).

The modeling of the configuration where there is a winning strategy is very simple and pretty in **Toupie**. It is programmed by means of two predicates:

$$winning(\mathbf{S}) += \exists\ \mathbf{T}\ (move(\mathbf{S}, \mathbf{T})\ \wedge\ losing(\mathbf{T}))$$
$$losing(\mathbf{S}) += \forall\ \mathbf{T}\ (move(\mathbf{S}, \mathbf{T}) \Rightarrow winning(\mathbf{T}))$$

Note that the player who has the turn looses when he (or she) cannot play any move. That is when his (or her) position (S) is such that $\forall\ \mathbf{T}\ \neg move(\mathbf{S}, \mathbf{T})$ that is the initial step of the fixpoint computation.

The running times are reported in the following table.

lines	4	5	6	7	8
reachable configurations	384	3840	46080	645120	10321920
time to compute them	0s21	0s43	0s75	1s25	1s93
time to compute winning positions	0s50	1s73	6s23	25s18	141s60

5 Related Works

CLP($\mathcal{FD}$) As mentioned in the introduction, the nature of the problems handled with classical CLP($\mathcal{FD}$) languages is different from the nature of the problems handled with **Toupie**. This comes from the fact that the underlying data-structures are not the same. The use of DDs permits – from a practical point of view – the introduction of a full universal quantification and fixpoint computations but do not permit the use of branch and bound paradigm. There are strong motivations (thanks to applications) to introduce a **Toupie**–like solver in a CLP($\mathcal{FD}$) language complementarily to the currently implemented solvers. The introduction of universal quantification is rather simple. The introduction of a least fixpoint mechanism should not be too difficult by using tabulation mechanism, whilst the introduction of greatest fixpoints is more tedious.

Binary Decison Diagrams have been used in order to implement the Boolean solver of CHIP [BS87]. **Toupie** can be seen as an extension of this work in several ways: extension of BDDs to finite domains, extension of the constraint language to the μ-calculus.

Model Checkers **Toupie** is much more related to model checkers such as MEC [Arn89]. These programs are specialized for the verification of systems of finite state machines and communicating processes. Of course, they present the advantages and disadvantages of a specialized implementation: they are more efficient but far less flexible. It remains that DDs permit to encode in a very compact way even huge automata (DDs capture the regularity of these graphs by means of subtree sharing) — space consumption is the main problem of model checking — and that properties can be written in **Toupie** in a very simple, elegant and declarative way.

Deductive Data Bases The semantics of **Toupie** is close to the semantics of deductive data base languages. In particular, all the **Toupie** formulae can be easily expressed in terms of the relational algebra. The difference comes, here again, from two points: first, in **Toupie** the fixpoints definitions are explicitely declared, and can be either least or greatest fixpoints, with the possibility to mix both (under the condition that formulae remain monotonic). Note also that, since **Toupie** semantics does not make the closed world assumption, the negation is naturally handled. Second, the underlying data structures are not the same. DDs allow a very efficient manipulation of relations, but are limited to small domains. Moreover, in the current implementation of **Toupie**, all the created DDs are stored in memory and cannot be put on an external device.

6 Conclusion and Future Works

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

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

Toupie can be considered from two 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 classic ones. However, it remains some problems to integrate it smoothly (see section 5).

– As a new paradigm for constraint logic languages. In this case, the μ-calculus 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 [CC92]. This approach could be of a particular interest to analyse higher order functional languages as well as to introduce disjunction in constraints over continuous domains.

References

[Arn89] A. Arnold. MEC: a System for Constructing and Analysing Transition Systems. In *Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.

[BMDH90] J.R. Burch, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *IEEE transactions on computers*, 1990.

[Bou93] A. Bouali. *Etudes et mises en œuvre d'outils de vérification basée sur la bisimulation.* PhD thesis, Université Paris VII, 03 1993. in french.

[Bry92] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 1992.

[BS87] W. Buettner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, 1987.

[CC92] R. Cousot and P. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. research report LIX/RR/92/09, Ecole Polytechnique, 1992.

[CCMR93] 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*, Estonie, 1993.

[CGR93] M-M. Corsini, A. Griffault, and A. Rauzy. Yet another Application for Toupie: Verification of Mutual Exclusion Algorithms. In *proceedings of Logic Programming and Automated Reasonning, LPAR'93.* LNCS, 1993.

[Col90] A. Colmerauer. An introduction to prologIII. *Communications of the ACM*, 28 (4), july 1990.

[CR93] M-M. Corsini and A. Rauzy. First Experiments with Toupie. Technical Report 577–93, LaBRI - Université Bordeaux I, 1993.

[EFT93] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. *Journal of Distributed Computing*, 6:155–164, June 1993.

[Hen90] P. Van Hentenryck. *Constraint Handling in Logic Programming.* Logic Programming. MIT Press, 1990.

[JL87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of Principle of Programming Languages (POPL'87)*, january 1987.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.

[Ull86] J. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10, 03 1986.

A A Denotational Semantics of Toupie

The semantics of Toupie formulae is determined with respect to a structure $S = \langle Const, \mathcal{V} \rangle$ where *Const* is an interpretation domain, $\mathcal{V}$ is a denumerable set of variables including all the variables of the program. As in DATALOG [Ull86], we assume 1) the unicity of names (two distinct constant symbols denotes two distinct constants) and 2) the closure of the domain, that is to say that *Const* is the set of all the constants occurring in the considered formula or program.

Definition 8. Individual Variable Assignments
An *individual variable assignment* is a mapping α from $\mathcal{V}$ into *Const* such that $\alpha(X) \in dom(X)$ for all the variables X occurring in the program.

For sake of brievety, we assume in the following the condition $\alpha(X) \in dom(X)$.

Definition 9. Relations
A *relation* on S is a mapping from $\mathcal{V} \rightarrow Const$ into $\mathcal{B}$, where $X \rightarrow Y$ stands for the set of mappings from X to Y and $\mathcal{B}$ stands for the Boolean values.

Definition 10. Predicate Variable Interpretations
Let Pr be the set of predicates occurring in the program. A *predicate variable interpretation* is a mapping from Pr into $(\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$, where $\mathbb{N}$ stands for the set of natural numbers.

This definition avoids the complications due to the different arities of the predicates. For a predicate of arity n, it suffices to consider that the corresponding function depends only on the first n numbers.

The semantics of a formula is thus a relation, and the semantics of a predicate (defined with a fixpoint equation) is a mapping from $(\mathbb{N} \rightarrow Const)$ into $\mathcal{B}$.

A Toupie program P assigns a meaning to a set of predicate symbols Pr. The semantics of the program is defined as the least fixpoint of a transformation $\mathcal{T}$. Let us note $\mathcal{PR}$ the set $Pr \rightarrow (\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$ of predicate variable interpretations. and $\mathcal{RE}$ the set $(\mathcal{V} \rightarrow Const) \rightarrow \mathcal{B}$ of relations.

The program defines a continuous transformation:

$$\mathcal{T} : \mathcal{PR} \rightarrow \mathcal{PR}$$

Each formula f defines a function:

$$\mathcal{T}[\![\mathtt{f}]\!] : \mathcal{PR} \rightarrow \mathcal{RE}$$

And each equation defines a function:

$$\mathcal{T}[\![\mathrm{Eq}]\!] : \mathcal{PR} \rightarrow (\mathbb{N} \rightarrow Const) \rightarrow \mathcal{B}$$

The definition of $\mathcal{T}$ will use the following notation.

Definition 11. Substitutions
Let $f : A \rightarrow B$ be a function. Let $a_1, \ldots, a_n$ be distinct elements of A and $b_1, \ldots, b_n$ be arbitrary elements of B. We note

$$f[a_1/b_1, \ldots, a_n/b_n]$$

the function $g : A \rightarrow B$ such that $ga_i = fb_i$ $(1 \leq i \leq n)$ and $ga = fa$ $(\forall a \notin \{a_1, \ldots, a_n\})$.
The notation $[a_1/b_1, \ldots, a_n/b_n]$ stands for $f[a_1/b_1, \ldots, a_n/b_n]$ where f is an arbitrary function.

We are now in position to define the semantic function $\mathcal{T}$. Let π be a predicate variable interpretation, α be an individual variable assignment, and σ be an element of $(\mathbb{N} \rightarrow \mathit{Const})$. $\mathcal{T}$ is defined inductively on the structure of formulae in the following way :

- $\mathcal{T}[\![1]\!]\ \pi\ \alpha = 1$ and $\mathcal{T}[\![0]\!]\ \pi\ \alpha = 0$.
- $\mathcal{T}[\![X_i = X_j]\!]\ \pi\ \alpha = \alpha(X_i) = \alpha(X_j)$.
- $\mathcal{T}[\![X_i = k]\!]\ \pi\ \alpha = \alpha(X_i) = k$.
- $\mathcal{T}[\![f \mid g]\!]\ \pi\ \alpha = \mathcal{T}[\![f]\!]\ \pi\ \alpha \vee \mathcal{T}[\![g]\!]\ \pi\ \alpha$.
- $\mathcal{T}[\![f\ \&\ g]\!]\pi\alpha = \mathcal{T}[\![f]\!]\ \pi\ \alpha \wedge \mathcal{T}[\![g]\!]\ \pi\ \alpha$.
- $\mathcal{T}[\![\forall\ X\ f]\!]\ \pi\ \alpha = \bigwedge_{k \in dom(X)} (\mathcal{T}[\![f]\!]\ \pi\ \alpha[X/k])$.
- $\mathcal{T}[\![\exists\ X\ f]\!]\ \pi\ \alpha = \bigvee_{k \in dom(X)} (\mathcal{T}[\![f]\!]\ \pi\ \alpha[X/k])$.
- $\mathcal{T}[\![P(X_{i_1}, \ldots, X_{i_r})]\!]\ \pi\ \alpha = \pi(P)(\alpha(X_{i_1}), \ldots, \alpha(X_{i_r}))$.
- $\mathcal{T}[\![P(X_1, \ldots, X_n) += f]\!]\ \pi\ \sigma = \mathcal{T}[\![f]\!]\ \pi\ [X_1/\sigma(1), \ldots, X_n/\sigma(n)]$.
- Finally, the transformation associated with the program is:
  $\mathcal{T}[\![Eq_1 \ldots Eq_n]\!]\ \pi = \pi[p_1/\mathcal{T}[\![Eq_1]\!]\ \pi, \ldots, p_n/\mathcal{T}[\![Eq_n]\!]\ \pi]$
  where the p_i are the predicates defined by the equations Eq_i.

Definition 12. Denotation of a Toupie Formula wrt a Program
Let P be a Toupie program. Let f be a Toupie formula. Let D be the set of variables occurring free in f. By definition, the *denotation* of f wrt P is the function $\mathcal{D}[\![f]\!] : (D \rightarrow \mathit{Const}) \rightarrow \mathcal{B}$ such that, for all $\alpha \in (D \rightarrow \mathit{Const})$,

$$\mathcal{D}[\![f]\!]\alpha = \mathcal{T}[\![f]\!](\mu(\mathcal{T}[\![P]\!]))\alpha',$$

where α' is any variable assignment such that

$$\alpha' X = \alpha X\ (\forall X \in D).$$

(The underlying program is kept implicit.)

Note that the introduction of least fixpoint definitions complicates the notations but not the semantics itself.