Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Data Decision Diagrams for ProMeLa Systems Analysis

Vincent Beaudenon, Emmanuelle Encrenaz, Sami Taktak

Laboratoire d'Informatique de Paris 6, University Pierre et Marie Curie (Paris 6), CNRS UMR 7606, 12, rue Cuvier, 75252 Paris cedex 05, France.
E-Mail : {Vincent.Beaudenon, Emmanuelle.Encrenaz, Sami.Taktak}@lip6.fr

Received: date / Revised version: date

Abstract. In this paper, we show how to verify CTL properties, using symbolic methods, on systems written in ProMeLa. Symbolic representation is based on Data Decision Diagrams (DDDs) which are n-valued DAGs designed to represent dynamic systems with integer domain variables. We describe principal components used for the verification of ProMeLa systems (DDD, representation of ProMeLa programs with DDD, the transposition of the execution of ProMeLa instructions into DDD). Then we compare and contrast our method with the model checker SPIN or classical BDD techniques, to highlight what system classes whether SPIN or our tool is more relevant for.

Key words: SPIN, CTL, Symbolic Verification Methods, BDDs, DDDs.

1 Introduction

1.1 Context

Model checking is a set of techniques intended to automatically decide if some property holds on a finite-state system. If the system does not satisfy the property checked, a counter-example may be expected to correct the system's (or the property's) specification. To determine whether the system satisfies a property, all *computation sequences* have to be explored. This means that, at least, all reachable states have to be stored to ensure the ending of computation.

Unlike simulation methods that can handle huge systems (10^{15} states or more), without giving any *formal* conclusion, exhaustive methods *formaly* ensures validity of conclusions, but cannot handle more than modest sized systems (up to 10^9 states) in case of explicit enumerative construction. Memory resources needed are linked to the amount of states explored and the information they hold. If we want to use exhaustive methods on huge systems, we first have to reduce these parameters. The abstracted model must be specified regarding the property that will be checked, which means that system's reduction (concerning variables' domains and computation abstraction) must be carefully expressed.

We chose an abstraction level which is the input for high-level hardware synthesis [17] and its industrial applications [11]. The system to analyse is modelled by a finite set of concurrent asynchronous processes. Communications are performed using buffered channels or shared variables. These systems are described in ProMeLa, the input language of the SPIN model-checker [14] which proposes verification of safety or liveness properties, expressed in linear temporal logic (LTL, [23]) or with Büchi automata.

1.2 Explicit state-space construction

The model-checker SPIN has proven its efficiency for communication protocols or C programs [14,13]. It performs LTL properties validation by finding acceptance cycles in the synchronous product of the system's automaton and a Büchi automaton. This latest recognizes any infinite sequence validating the negation of the property to be checked. The synchronous product is also a Büchi automaton, its size grows exponentially with the systems's size and the complexity of the checked property. The states are explicitly stored using an efficient compression algorithm [15] and methods limiting the state-space exploration [16,21]. Nevertheless, the SPIN model-checker cannot handle systems whose size exceeds 10^7 states, and a knowledge about the tool behaviour is essential to deal with real systems [1].

1.3 Symbolic Methods

As state-space representation is a crucial factor for these verification methods, *symbolic methods* emerged for representing systems with a greater size (10^{20} states) [5]. These methods deal with *set* of states contrary to explicit methods handling states one-by-one. Symbolic Methods usualy lend theirselves to Computation Tree Logic [8] and led to the implementation of efficient validation tools such as VIS [2] or SMV [19]. Symbolic representation used by these tools is based on Binary Decision Diagrams [3] described in section 1.4.

An experimental comparison between the enumerative method (performed by SPIN) and the symbolic one (performed by VIS) was proposed on the ATM protocol [22]. Unlike SPIN, VIS is a *symbolic* model-checker over networks of *synchronous* processes using *signal* communications. For a better comparison, the most adapted models were implemented for each tool. It emerges that SPIN is more relevant for small-sized system but VIS can handle huge systems.

Other comparative studies, such as [25] exhibit the better performance of exhaustive methods over symbolics ones in case of livelock analysis of a variant of the sliding-window protocol (the GNU i-protocol).

Our approach consists in using symbolic methods on ProMeLa systems, in order to verify safety and liveness properties of some systems that cannot be handled by SPIN. We chose *Data Decision Diagrams* (DDDs) [9] that present the efficient properties of canonicity and compacity of BDDs and are better suited to represent dynamic systems and non boolean variables (but having finite domains). Systematic comparison of verification of systems described in ProMeLa exhibits some characteristics of systems that are better suited for SPIN analysis, while others give the advantage to symbolic techniques (BDD or DDD). Moreover, for systems presenting a high parallelism where a "natural" ordering of variables can be found, DDD outperforms BDD for all (static or dynamic) ordering strategies of BDD.

Organisation of the paper follows : the rest of this section describes DAG-based state-space representations. section 2 briefly recalls preliminaries about CTL, and in section 3 we describe the Data Decision Diagram structure and the way they are manipulated. In section 4 we exhibit the links between a ProMeLa program and the set of transformations to be applied to DDD. In section 5 we compare, on systems described with the *same* semantics (ProMeLa) the performances of explicit methods using SPIN and symbolic methods based on DDDs and BDDs, and bring out some characteristics of systems giving an advantage to one verification approach or to the other one.

1.4 DAG-based state-space construction

A Binary Decision Diagram (BDD [3]) represents a boolean function on a graph with each nonterminal node labelled by a binary variable. Each node has two outgoing edges, corresponding to the cases where the variable evaluates to 1 or to 0. Terminal nodes are labelled with 0 or 1, corresponding to the possible functions values.

Binary Decision Diagrams (BDDs [3] represent Boolean functions in a both canonical and (often) compact form. Most logical operation can be efficiently implemented using OBDDs and constant assignment can be performed in linear time (regarding the size of modified BDD). These diagrams allow state space construction when considered as characteristic function of a set of states. The success of OBDDs inspired researchs to improve their efficiency. Main members of the BDD family are shown on figure 1. Two parts of this figure are separated with a dashed line. The left side contains decision diagrams used to represent functions over boolean variables. Reader may find more details in [4].

The right side of the figure shows recent improvements of DAGs made for state-space representation purpose. They all deal with variables that are not necessary boolean.

Functions over numeric variables avoid bit-to-bit construction. They propose an easier way to manipulate transitions in case of state generation. "Multi-valued Decision Diagrams" (MDDs) where developped by [20] for set of states generation and storage. "Additive Edge-Valued MDD" (EV+MDD [6]) where proposed to represent numeric valued function over numeric values and set generation of shortest paths.

At least two topics in MDDs caught our attention: First, avoiding bit-to-bit construction is a simple way to deal with high level design; There is no need, for instance, to built a complete (bit-level) adder to perform an integer addition; Second, DAG based representation conserves sharing properties, but it needs knowledge about the range of each variable to be constructed. And third, the use of *event locality* that accelerates treatments during states generation [7] gives excellents results on Petri nets analysis. *Event locality* consists in reaching directly selected nodes in the middle of paths in DAGs rather than traversing the DAGs variable node by variable node from root to leaf. This is applyable only if few and close layers have to be modified by particularizing each transition with each enabling precondition. Unfortunately ProMeLa systems do not have this locality property.

Data Decision Diagrams (DDDs)[9] are very close to MDDs, but they have significant differences: They use the same principle of multivalued variables but only reached values have corresponding outgoing edge. Any variable can be instanciated as often as necessary, which allows representation of dynamic systems. DDDs are endowed with a formalism of inductive methods that are

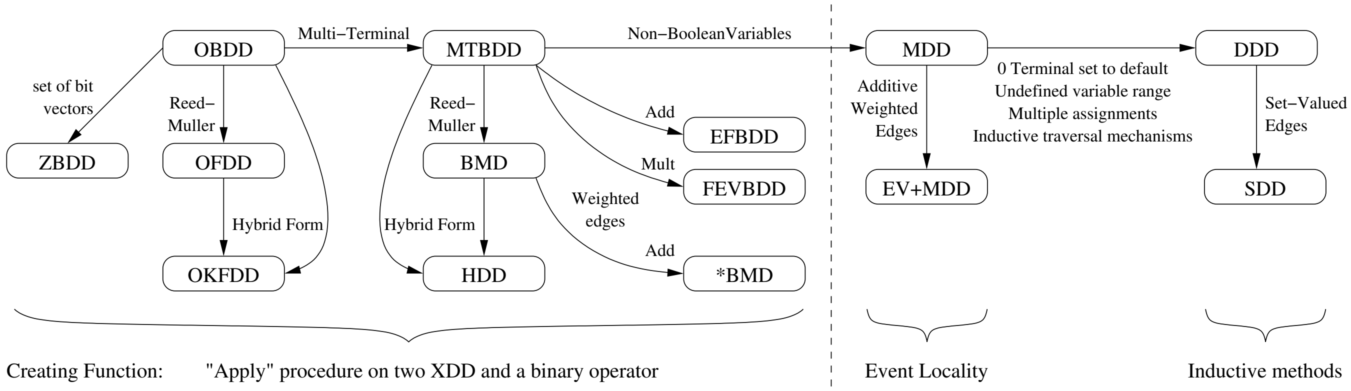

Fig. 1. Graph-based function representation

particularized traversal mechanisms expressing operators and transitions in canonical form. This last difference led us to use DDDs (rather than MDDs) to handle ProMeLa systems as we will see in section 3. Set Decision Diagrams (SDDs [27]) is the last evolution of DDDs. Edges of a SDD are labelled with a set of values (a DDD, for instance), which increases the sharing of sub structures.

2 The CTL Temporal Logic

2.1 Syntax and Semantics

Computation Tree Logic (CTL) is a propositional, branching-time, temporal logic, which was proposed to specify temporal properties on finite concurrent systems [8]. It enlarges classical propositional logics with temporal operators associated with path quantifiers.

The semantic of a CTL formula is defined on a *Kripke structure.*

Definition 1 (Kripke Structure). A Kripke structure $M = (AP, S, L, R, S_0)$ is defined as follows:

- AP is a finite set of atomic propositions.
- S is a finite set of states.
- $L : S \to 2^P$ is a function labelling each state with a set of atomic propositions.
- $R \subseteq S \times S$ is a total transition relation: $\forall s \in S$, $\exists s'/(s, s') \in R$.
- S_0 is the set of initial states.

A *path* is an infinite sequence of states $\pi = s_0, s_1, s_2 \ldots$ such as $\forall i \geq 0$, $(s_i, s_{i+1}) \in R$. We denote π^i the *suffix* of π starting at s_i.

Definition 2 (CTL Properties [8]). CTL properties are built on atomic propositions. The syntax and semantics of CTL properties are defined, given a Kripke structure, as follows:

- Each atomic property $p \in AP$ is a CTL formula.

$$s \models p \iff p \in L(s)$$

- Let f and g be to CTL formulas, then $\neg f$, $f \wedge g$, $\mathbf{EX}f$, $\mathbf{EG}f$ et $\mathbf{E}f\mathbf{U}g$ are CTL formulas.

$$\begin{aligned}
s_0 \models \neg f &\iff s_0 \not\models f \\
s_0 \models f \wedge g &\iff s_0 \models f \text{ and } s_0 \models g \\
s_0 \models \mathbf{EX}f &\iff \text{there exists a path } \pi \\
&\qquad \text{such as } \pi^1 \models f \\
s_0 \models \mathbf{EG}f &\iff \text{there exists a path } \pi \\
&\qquad \text{such as } \forall i \geq 0, \pi^i \models f \\
s_0 \models \mathbf{E}f\mathbf{U}g &\iff \text{there exists a path } \pi \\
&\qquad \text{such as } \exists k \geq 0 \\
&\qquad \text{such as } \pi^k \models g \text{ and} \\
&\qquad \forall 0 \leq i < k, \pi^i \models f
\end{aligned}$$

2.2 Verifying CTL properties

The use on an efficient abstract data type to represent and operate on *sets* is a key issue to perform symbolic model checking. In most cases, for finite systems, characteristic functions of sets of states are considered and encoded into BDD as introduced by [19] and implemented into VIS and SMV. But other representations of sets might be considered, provided they are equipped with efficient set operations computation, *and* two set transformers *post* and *pre* [26]. Some examples of alternative set representations are *predicate structures* applied to algebraic Petri Nets [24] or *convex union of (convex) polyhedra*, used in various verification tools for hybrid systems [10] [12].

The *post* transformer computes the set of states that are reachable from states grouped into a set X:

$$post(X) = \bigsqcup_{s \subset X} s'/R(s, s')$$

The *pre* transformer computes the set of predecessor states of states that are grouped into a set X. This corresponds to the computation of all states that verify the CTL formula $s \models \mathbf{EX}(X)$.

$$pre(X) = \bigsqcup_{s \subset X} s'/R(s', s)$$

EG and **EU** CTL operators are computed as fixpoints based on the *pre* (or **EX**) operator, due to their recurrent definition:

$$M, s \models \mathbf{EG}f \iff M, s \models f \wedge \mathbf{EX}(\mathbf{EG}f)$$

$$M, s \models \mathbf{E}f\mathbf{U}g \iff \begin{cases} M, s \models g \\ \vee \\ M, s \models (f \wedge \mathbf{EX}(\mathbf{E}f\mathbf{U}g)) \end{cases}$$

In section 3 we describe the abstract data type we use to represent set of states and the associated formalism to construct *post* and *pre* operators.

3 Data Decision Diagrams

3.1 Presentation

Data Decision Diagrams (DDDs,[9]) are data structures for representing sets of sequences of assignments. DDDs are encoded as DAGs. Each node is labeled with a variable and each outgoing edge with an integer value. A path leading from the root of the DDD to the terminal node 1 represents an accepted sequence of assignments. All paths leading to the default terminal node 0 are not represented and undefined sequences are represented on paths leading to the node $\top$. Each variable may occur any times along a path. Canonicity is insured by imposing a total order on variables.

In the following, E denotes a set of variables, and for any $e \in E$, $Dom(e)$ represents the domain of e.

Definition 3 (DDD [9]). The set $\mathbb{D}$ of DDD is defined by $d \in \mathbb{D}$ if

- $d \in \{0, 1, \top\}$ or
- $d = (e, \alpha)$ with :
  - $e \in E$
  - $\alpha : Dom(e) \to \mathbb{D}$, such that $\{x \in Dom(e) | \alpha(x) \neq 0\}$ is finite.

We denote $e \xrightarrow{a} d$, the DDD (e, α) with $\alpha(a) = d$ and $\forall x \neq a, \alpha(x) = 0$.

A DDD is *well-defined* if all paths starting from the root of the DAG lead to the unique terminal node 1.

Figure 2 shows the DDDs representing the following sets of sequences of assignments:

- $\{\ a = 1,\ b = 2\}$, $\{a = 2,\ b = 1\}$, $\{a = 2\ ,b = 2\ ,b = 3\}$ for the left-hand side.
- $\{\ a = 1,\ b = 3\}$, $\{a = 2\ ,b = 2\}$ for the right-hand side.

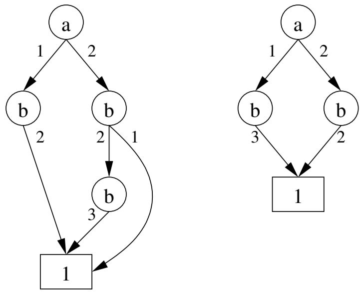

Fig. 2. Two *well-defined* DDD

Two well-defined DDDs are equals if

- $d = d' = 1$ or
- $d = (e, \alpha) \neq 0$ and $d' = (e, \alpha') \neq 0$ and $\forall x \in Dom(e), \alpha(x) = \alpha'(x)$.

Set operators (union $+$, intersection $*$, set-difference $\backslash$) are defined inductively on DDD [9]. They are obtained by a recursive evaluation on DDDs representing the operand sets. Concatenation operator is also defined: $(a \xrightarrow{x} 1).(b \xrightarrow{y} 1) = a \xrightarrow{x} b \xrightarrow{y} 1$.

3.2 Homomorphisms

Our purpose is to represent states of a ProMeLa system on DDDs and use the properties of canonicity and compacity of this structure. Given an initial state, we have to construct a set of operators that satisfies the system's transition rules. DDD allows the use of a class of operators, called homomorphisms, for coding transition rules [9]. Let Φ be an homomorphism and two well-defined DDD d and d', then $\Phi(d + d') = \Phi(d) + \Phi(d')$. We deal with a special kind of homomorphisms called *inductive homomorphisms* which are *locally* defined. Unlike for BDD, only accepted sequences are represented on the DAG thus, it is not necessary to explore the entire DDD.

Definition 4 (Inductive Homomorphism [9]). Let c be a DDD and $\phi(e, x)_{e \in E, x \in Dom(e)}$ be an homomorphism family.

$$\Phi(d) = \begin{cases} 0 & if\ d = 0 \\ \top & if\ d = \top \\ c & if\ d = 1 \\ \sum_{x \in Dom(e)} \phi(e, x)(\alpha(x)) & if\ d = (e, \alpha) \end{cases}$$

is an homomorphism.

This last expression ($\sum_{x \in Dom(e)} \phi(e, x)(\alpha(x))$) is only applied on DDD nodes. In such cases the homomorphism uses only local information: The variable e and its possible values $x \in Dom(e)$. For each of these values, and its associated outgoing edge $\alpha(x)$, a function $\phi(e,x)$ is applied. Then a union of obtained DDD is performed.

Evaluation of set operators in DDD and computation of homomorphisms are stored in an operation cache.

3.2.1 Elementary homomorphisms to compute *post*

In our context, we consider that no homomorphism can be applied on terminal node 1 because it means that the treatment may not be applied correctly (it would refer to used but undeclared variable):

$$\phi(1) = \top, \forall \text{ homomorphism } \phi$$

Thus, we just have to deal with homomorphism on non-terminal node, which means that we only have to define a local treatment for a couple *(variable, edge)*.

Proposition 1 (Constant Assignments). *Let d be a well defined DDD, var be a variable of E appearing in d, and $cst \in Dom(var)$. The homomorphism $\langle\ \mathsf{setCst}\ (var, cst)\ \rangle$ that performs the assignement $var = cst$ is defined as below:*

$$\langle\ \mathsf{setCst}\ (var, cst)\ \rangle(e,x) = \begin{cases} if \quad var = e \\ \qquad e \xrightarrow{cst} \langle\ \mathsf{id}\ \rangle \\ else \\ \qquad e \xrightarrow{x} \langle\ \mathsf{this}\ \rangle \end{cases}$$

Term $cst\ e \xrightarrow{cst}$ means that any encountered outgoing edge of nodes labelled with e receives the unique label cst and $\langle\ \mathsf{id}\ \rangle$ reproduces the original suffixes reached by the edges. Term $\langle\ \mathsf{this}\ \rangle$ means that the homomorphism propagates itself along any outgoing edge if the node is not labelled with var.

Proposition 2 (Expression Assignments). *Let d be a well defined DDD, var be a variable of d, and $expr$ be an arithmetic expression such as any parameter of $expr$ is defined on d. The homomorphism $\langle\ \mathsf{setExpr}\ (var, expr)\ \rangle$ that performs the assignment $var = expr$ is defined as below:*

$$\langle\ \mathsf{setExpr}\ (var, expr)\ \rangle(e,x) =$$
$$\begin{cases} if \quad var = e \\ \quad \begin{cases} if \quad e \in expr \\ \quad \begin{cases} if \quad ||expr_{\{e:=x\}}|| = 0 \\ \qquad e \xrightarrow{eval(expr_{\{e:=x\}})} \langle\ \mathsf{id}\ \rangle \\ else \\ \qquad \langle\ \mathsf{down}\ (var, expr_{\{e:=x\}})\ \rangle \end{cases} \\ else \\ \qquad \langle\ \mathsf{down}\ (var, expr_{\{e:=x\}})\ \rangle \end{cases} \\ else \\ \quad \begin{cases} if \quad e \in expr \\ \quad \begin{cases} if \quad ||expr_{\{e:=x\}}|| = 0 \\ \qquad e \xrightarrow{x} \langle\ \mathsf{setCst}\ (var, eval(expr_{\{e:=x\}})\ \rangle \\ else \\ \qquad e \xrightarrow{x} \langle\ \mathsf{setExpr}\ (var, expr_{\{e:=x\}})\ \rangle \end{cases} \\ else \\ \qquad e \xrightarrow{x} \langle\ \mathsf{this}\ \rangle \end{cases} \end{cases}$$

With:

$$\langle\ \mathsf{up}\ (var, val)\ \rangle(e,x) = e \xrightarrow{x} var \xrightarrow{val} \langle\ \mathsf{id}\ \rangle$$

And:

$$\langle\ \mathsf{down}\ (var, expr)\ \rangle(e,x)$$
$$= \begin{cases} if \quad e \notin expr \\ \qquad \langle\ \mathsf{up}\ (e,x)\ \rangle \circ \langle\ \mathsf{down}\ (var, expr)\ \rangle \\ else \\ \quad \begin{cases} if \quad |expr_{\{e:=x\}}|| > 0 \\ \qquad \langle\ \mathsf{up}\ (e,x)\ \rangle \\ \qquad \circ \langle\ \mathsf{down}\ (var, expr_{\{e:=x\}})\ \rangle \\ else \\ \qquad var \xrightarrow{eval(expr_{\{e:=x\}}} e \xrightarrow{x} \langle\ \mathsf{id}\ \rangle \end{cases} \end{cases}$$

The second part of the expression (if $e \neq var$) in $\langle\ \mathsf{setexpr}\ ()\ \rangle$ is immediate, the homomorphism propagates itself with (if necessary) a completed expression. Then if the expression is assessable ($||expr_{\{e:=x\}} = 0||$) the propagated homomorphism becomes $\langle\ \mathsf{setCst}\ ()\ \rangle$. The first part of the expression deals with cases where the assigned variable var is reached *before* the expression $expr$ is assessable. In such cases the treatment has to *complete* the expression, *store* the result in a temporary node that will be *replaced* at the var level. The composition of $\langle\ \mathsf{up}\ ()\ \rangle$ and $\langle\ \mathsf{down}\ ()\ \rangle$ homomorphisms computes these (*complete*, *store*, and *replace*) operations in a unique traversal.

Example 1 ($\langle\ \mathsf{up}\ ()\ \rangle$ and $\langle\ \mathsf{down}\ ()\ \rangle$ usage). We aim at performing the assignment $b = a + b + d$ on the DDD

$a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1$.

$$\begin{aligned}
&\langle\ \mathsf{setExpr}\ (b, a+b+d)\ \rangle(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1)\\
&= a \xrightarrow{1} \langle\ \mathsf{setExpr}\ (b, 1+b+d)\ \rangle(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1)\\
&= a \xrightarrow{1} \langle\ \mathsf{down}\ (b, 1+2+d)\ \rangle(c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1)\\
&= a \xrightarrow{1} \langle\ \mathsf{up}\ (c, 3)\ \rangle \circ \langle\ \mathsf{down}\ (b, 3+d)\ \rangle(d \xrightarrow{4} e \xrightarrow{5} 1)\\
&= a \xrightarrow{1} \langle\ \mathsf{up}\ (c, 3)\ \rangle(b \xrightarrow{7} d \xrightarrow{4} e \xrightarrow{5} 1)\\
&= a \xrightarrow{1} b \xrightarrow{7} c \xrightarrow{3} d \xrightarrow{4} e \xrightarrow{5} 1
\end{aligned}$$

3.2.2 Elementary homomorphisms to compute *pre*

The main difficulty for evaluating the *pre* operator is the non-reversibility of some kinds of instruction such as reading a variable in a FIFO, or some assignments: What can be, for a given context, the value(s) taken by the assigned variable *before* the assignement (or FIFO's reading) occured?

The set of states that preceeds the $var = cst$ instruction's execution cannot be computed without *a priori* knowledge about *var*'s range, or better, about knowledge of the effective values taken in the execution context of the instruction $var = cst$.

For a given constant assignment, two sets of states are used to compute the *pre* operator.

- The set of states that we want to know the set of predecessors states: TO,
- and a set of candidates states C, that approximates the execution context of the instruction $var = cst$. It is a subset of reachable states.

We use the following assumptions:

- Initial value of assigned variable is arbitrary.
- Only *var* is modified.

The solution is a composition of three treatments:

1. Selecting in TO states that satisfies $var = cst$. As *var* is the only modified variable, we obtain the execution context of instruction $var = cst$.
2. Extending, in this last DDD, the domain of *var* to all possible values *given by candidates states* C (we use C because *var*'s range is unknown).
3. Intersect the obtained DDD with the set of candidate states: It gives all candidates respecting the execution context by abstracting *var*.

The homomorphism *sweet* computes these three treatments in a unique traversal of C coupled with the traversal of TO. Its parameters are *var*, *cst* and states whom we want to compute predecessors TO. It is applied on candidates states C:

$$\begin{aligned}
&\langle\ \mathsf{preSetCst}\ (var, cst, C\ \rangle(TO)\\
&= \langle\ \mathsf{sweet}\ (var, cst, TO)\ \rangle(C)
\end{aligned}$$

We detail local treatment of the homomorphism $\langle\ \mathsf{sweet}\ (v, c, To)\ \rangle$ on an arbitrary node (e, x).

1. There is no predecessor (nor successor) for the empty set:
$$\langle\ \mathsf{sweet}\ (v, c, To)\ \rangle = \emptyset \tag{1}$$
if $To = \emptyset$.
2. Before encountering *var*, a local intersection is made:
$$\begin{aligned}
&\langle\ \mathsf{sweet}\ (v, c, To)\ \rangle(e, x)\\
&= e \xrightarrow{x} \langle\ \mathsf{sweet}\ (v, c, To|_x)\ \rangle
\end{aligned} \tag{2}$$
if $e \neq v$ where $To|_x$ is the node reached by outgoing edge of To, labelled with x.
3. After reaching *var*, its domain is extended to the values taken in candidate states (the values of each outgoing edges of current node, as $\langle\ \mathsf{sweet}\ ()\ \rangle$ works on candidate states represented on C). After this an intersection on remaining variables is obtained with a classical product of DDDs.
$$\begin{aligned}
&\langle\ \mathsf{sweet}\ (v, c, To)\ \rangle(e, x)\\
&= e \xrightarrow{x} (\langle\ \mathsf{id}\ \rangle \cap To|_c)i
\end{aligned} \tag{3}$$

A computation of $\langle\ \mathsf{sweet}\ ()\ \rangle$ operator is given on figure 3. We are applying the homomorphism $\langle\ \mathsf{sweet}\ (b, 3, TO)\ \rangle$ on the DDD C (see first and second schemes). The DDD TO does not accept $a = 2$, thus application of $\langle\ \mathsf{sweet}\ ()\ \rangle$ on $a \xrightarrow{2}$ ... leads to the empty set (according to formula 1), and the homomorphism propagates in C on the edge $a \xrightarrow{1} C_b$ after restricting TO to the suffix of $a = 1$ (the outgoing edge of TO labelled by 1 according to the formula 2, see third scheme). By definition, each value labelling outgoing edge of the node C_b is allowable. The set of states they handle $(b \xrightarrow{2} C_{c1}) + (b \xrightarrow{3} C_{c2})$ is intersected with states of TO that represents the execution context $b = 2$ $(b \xrightarrow{2} d2)$ given lower layers (according to formula 3, see fourth scheme).

The result is given in the fifth and last scheme. Predecessors of TO were selected in C, respecting the execution context of the instruction $b = 2$.

We found that for any non-reversible instruction, the *pre* operator can be computed using the evaluation of constant assignment's one. In case of expression assignment, it is defined as below:

Proposition 3 (*pre* operator for Expression Assignement). *Let d be a well defined DDD, var be a variable of d, $expr$ be an arithmetic expression such as any parameter of $expr$ is defined on d, and C the sate of reachable states. The homomorphism* $\langle\ \mathsf{setExpr}\ (var, expr)\ \rangle$ *that performs the pre operator (according to C) of assignment $var = expr$ is defined as*

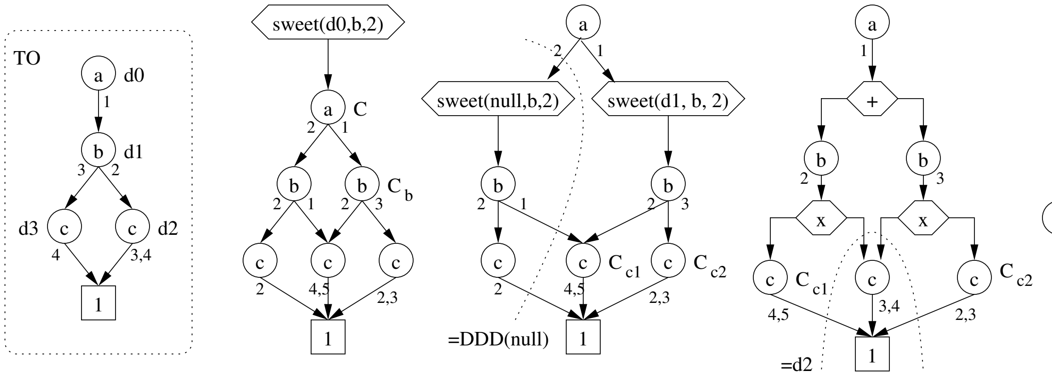

Fig. 3. Computation of the ⟨ sweet () ⟩ operator

below:

$$\langle\ \mathsf{preSetExpr}\ (var, expr, C)\ \rangle(e, x) =$$

$$\begin{cases} \textit{if} \quad var = e \\ \quad \langle\ \mathsf{selectCond}\ (x = expr)\ \rangle \\ \quad \circ\langle\ \mathsf{preSetCst}\ (e, x, c)\ \rangle(e \xrightarrow{x} \langle\ \mathsf{id}\ \rangle) \\ \textit{else} \\ \quad \begin{cases} \textit{if} \quad e \in expr \\ \quad \begin{cases} \textit{if} \quad \|expr_{\{e:=x\}}\| = 0 \\ \quad e \xrightarrow{x} \\ \quad \langle\ \mathsf{preSetCst}\ (var, eval(expr_{\{e:=x\}}, C|_x)\ \rangle \\ \textit{else} \\ \quad e \xrightarrow{x} \\ \quad \langle\ \mathsf{preSetExpr}\ (var, expr_{\{e:=x\}} C|_x)\ \rangle \end{cases} \\ \textit{else} \\ \quad e \xrightarrow{x} \langle\ \mathsf{preSetExp}\ (var, expr, C|_x)\ \rangle \end{cases} \end{cases}$$

3.3 Differences with BDDs and MDDs

Save the range of the variables, there are crucial differences between BDDs and DDDs: On a BDD, each path leading to a terminal (1 or 0) is represented on the DAG but some variable may not occur along a path leading from root to leaf, in case it is not a decision node. On a DDD only paths leading to the terminal 1 are represented, hence along each path from root to terminal 1, each variable appears at least once.

On BDDs, It is possible to represent the transition of a static ProMeLa system using the "Apply" procedure proposed by [3]. The "Apply" procedure recursively traverses the two BDD operands from root to leaves (either 0 or 1). On the opposite, homomorphisms do not have to traverse the whole DDD down to leaf.

On a BDD, assignment of a variable is performed while considering the variable's and the expression's bit-to-bit decomposition. This decomposition supposes an *a priori* knowledge of the domain of each ProMeLa variable (oftenly given by the `type` of the variable), and some type-conversion or bit-expension facilities. For instance, considering three variables x, y and z of `byte` type, and the assignment $x = y + z$, each bit x_i of x will be modified with the ith boolean function of an 8-bits adder. The bit-to-bit assignment is performed using bi-implication (boolean function `xnor`). For example, the assignment operator $a = b \vee c$ for a set of states represented on the boolean function f is performed in three steps:

1. Abstracting the assigned variable: $f' = \exists_a f$.
2. Constructing the bi-implication operator between a and $b \vee c$: $x = \bar{a} \oplus b \vee c$.
3. Constructing the new function $g = f' \wedge x$.

A performances comparison of BDDs and DDDs on ProMeLa systems is given in section 5.

On the other hand, MDDs represent boolean (or eventualy arithmetic) function over *integer* variables. Each internal node represent an integer variable, pointing out the nodes representing another variable. The arity of each node is *a priori* bounded (contrary to DDD). Variables have to be ordered, and MDD nodes representing a given variable are said to belong to a given layer, that can be reached directly (without traversing the MDD from its root to this layer). This implementation is well-suited to perform *local modifications* of the MDD, without having to traverse it (from root to leaves as in BDD or from root to the concerned variable as in DDD). In MDD, representing the firing of transitions is performed using *event locality* [7]. In case of transitions using few variables represented on close layerss on the DAG, [7] proposes to modify these only layers, considering the independence of the others. First, each transition is decomposed into an enumerative set of particularized transitions regarding variables' domains. These transitions are represented on a couple of states: Enabling states and new states. These states are represented on MDDs corresponding to a set of layers containing all concerned variables. A union is performed between paths containing enabling states and new states generated.

An example is given on figure 4. The central MDD represents a set of states made of variables (a, b, c, d). Variables' domain is $\{0, 1, 2\}$, each node is assimilate to an array of three edges implicitly labelled by one of these three values and leading to a node to the following layer. For instance, in current state space, there is a unique

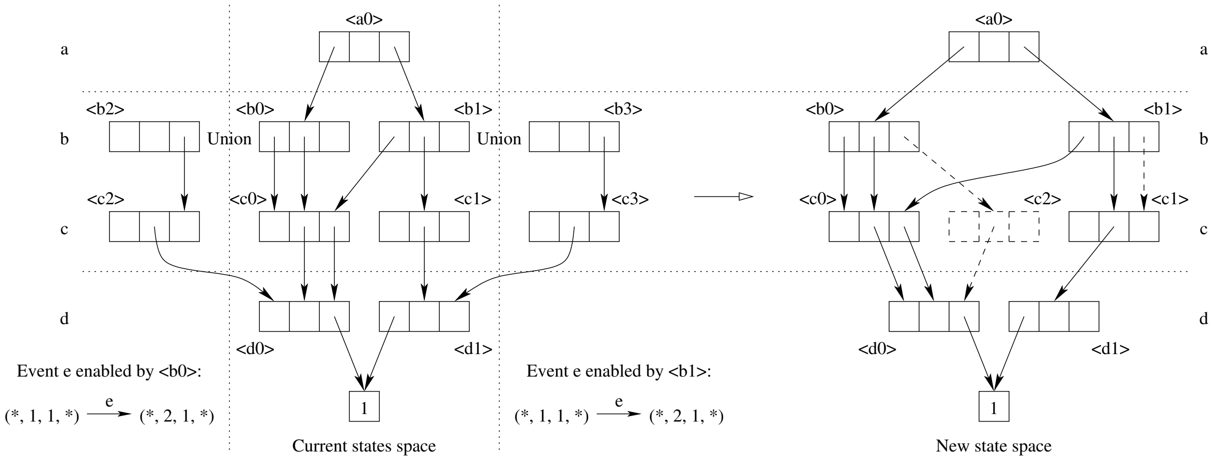

Fig. 4. Example of an MDD-modification in response to *firing an event*

node ($\langle a0 \rangle$), labelled by the variable a, whith two outgoing edges corresponding to the first and the last value of the variables' domain: 0 and 2. There is no state that satisfies $a = 1$. Considering the assignment $b = b + c$, concerned variables are b and c and modifications only occur on these two consecutive layers. In the figure, the assignement is performed if $b = c = 1$ (other cases have to be handled). The corresponding enabling set of states is $(*, 1, 1, *)$ (wich means that b and c are set to 1, a and d don't matter) and the result after transition is $(*, 2, 1, *)$. As variable a is not concerned, the *prefixes* $\langle a \rangle \xrightarrow{0}$ and $\langle a \rangle \xrightarrow{2}$ are preserved the computation starts on layer b. Corresponding pathes are: $\langle b0 \rangle \xrightarrow{1} \langle c0 \rangle \xrightarrow{1} [\langle d0 \rangle]$ and $\langle b1 \rangle \xrightarrow{1} \langle c1 \rangle \xrightarrow{1} [\langle d1 \rangle]$. The square brackets denotes suffixes of the pathes that are not modified by the treatment. We denote the event e: $(*, 1, 1, *) \xrightarrow{e} (*, 2, 1, *)$ they are respectively united with result pattern joint to she same suffixes of enabling patterns $\langle b2 \rangle \xrightarrow{2} \langle c2 \rangle \xrightarrow{1} [\langle d0 \rangle]$ and $(\langle b3 \rangle \xrightarrow{2} \langle c3 \rangle) \xrightarrow{1} [\langle d1 \rangle]$.

Enumerative particularization of a transition T is necessary and increases the computation complexity. However, this method prove its efficiency for representing huge state-space on Petri nets whith strong locality, controled (and small) variables' domains (places' capacity), and an *ad-hoc* variable ordering.

Representing reachable state space for ProMeLa systems on MDDs presents three major difficulties. First, practitioner may have no idea about variables'range save the C type used for its encoding, which means, for instance, creating nodes of 256 outgoing edges for a variable encoded on a byte, even if its effective set of values is much smaller. Second, some variables are shared between many processes and transitions may concern many of them, thus the locality factor is not as preeminent as in Petri nets. Third, the enumerative particularization of transitions may become the main factor of combinatorial explosion (notably when dealing with arithmetic expressions), all the more since variables' domains are too great.

Data Decision Diagrams solve the first problem: They handle effectively only reached values. Homomorphisms on DDDs solve the two last problems: Even if reaching concerned layers is faster on MDDs (but only appliable when locality is present), representing transition does not depend on variable ordering and has a canonical form whatever the reached values are.

4 ProMeLa systems components

We show how DDDs are used to represent a state of a ProMeLa system. Then we define homomorphisms corresponding to *pre* and *post* operators that match ProMeLa semantics.

4.1 Object Model of a ProMeLa program

Figure 5 shows our object construction of a ProMeLa program. The `program` class possesses global variables, communication channels and ProMeLa *program*'s processes. Each `process` class possesses local variables of a *process* and its program counter.

There are two main instruction's type in ProMeLa: Elementary instructions (labelled, guarded) and blocks separators (selects, loops, etc.). These latests contain specialised structured sets of instructions but they are built on the same homomorphisms as elementary instructions are.

4.2 State representation

A ProMeLa program describes a dynamic collection of processes which communicate with channels or shared variables. We consider a *static* subset of ProMeLa, no process nor variable can be dynamically instanciated.

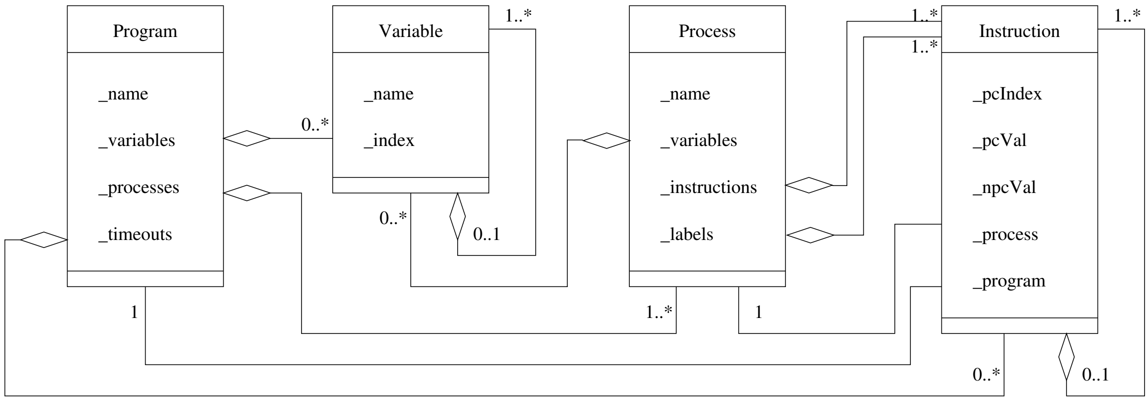

Fig. 5. Object model of a ProMeLa program

This assumption allow us to represent state-space as a collection of variables described as follows. Each integer variable is represented by a DDD :

$$variable \xrightarrow{value} 1$$

A process is obtained by the concatenation of DDDs that represent its local variables and program counter:

$$program\_counter \xrightarrow{pc} local\_var_1 \xrightarrow{val_1} ... \xrightarrow{...} 1$$

Each static structured type is flattened without changing the system's behaviour. For example, an n sized array t is represented with n variables:

$$t[0] \rightarrow t[1] \rightarrow ... \rightarrow t[n-1] \rightarrow 1$$

There are two types of communication using channels: Rendez-vous that are replaced with guarded assignments; and FIFOs that are represented on a DDD by using the encoding proposed by [9]: We use the same variable for each place of the FIFO. All the nodes are represented successively on the DDD framed with two nodes labelled with the same variable: The first one indicates the channel's size and the last one indicates that there is no other occupied place in the FIFO. This last one, has a unique outgoing edge labelled with a value that cannot be hold in the FIFO (#). Thus, FIFO are constructed using the following way:

$$f \xrightarrow{size} f \xrightarrow{1^{st}elt} ...f \xrightarrow{i^{th}elt} f \xrightarrow{\#} 1 \qquad (4)$$

The state of a ProMeLa program is obtained by the concatenation of all global variables, channels and processes.

4.3 Instructions

Each elementary instruction is guarded by a given program counter value. Another condition may be added (like non-emptyness of a FIFO, for instance). Each instruction proceeds to the program counter evolution and modifications described in ProMeLa.

In static ProMeLa systems (no dynamic process creation), each instruction can be constructed using these following elementary treatments:

1. Selection of states satisfying a given boolean formula.
2. Integer expression assignment.
3. Expression's writing in FIFO.
4. Variable's reading in FIFO.
5. Catching information on a FIFO (full, non-full, empty and non-empty).

Elementary instructions and corresponding homomorphisms are given in table 4.3 the last column gives additional homomorphisms that may be called when using the homomorphism in previous column. Given the promela code `a = b+c;`, which assigns the expression $b + c$ to the variable a and sets the program counter pc from an arbitrary value m to another arbitrary value n, the treatment can be performed by applying the following homomorphism:

$$\begin{array}{rl} & \langle\ \mathsf{setCst}\ (pc, n)\ \rangle \\ & \circ \langle\ \mathsf{setExpr}\ (a, b+c)\ \rangle \\ & \circ \langle\ \mathsf{selectCond}\ (pc = m)\ \rangle \\ = & \langle\ \mathsf{selectAndSet}\ (pc, m, n)\ \rangle \\ & \circ \langle\ \mathsf{setExpr}\ (a, b+c)\ \rangle \end{array}$$

More complex homomorphims were built using these elementary ones to perform a complete instruction (including the evolution of the program counter) on a unique traversal.

5 Results

5.1 performances

In this section we compare the performances of SPIN versus our tool using DDDs or BDDs concerning static systems. As checking LTL properties using Büchi automaton is an additional source of complexity, we chose to compare SPIN and our tool only on the reachable

Table 1. Elementary homomorphisms for ProMeLa instructions

Instruction	ProMeLa `code`	Main Homomorphism	Sub-Homomorphisms
Jump	`goto label;`	$\langle$ selectAndSet $(pc, pcVal, npcVal)$ $\rangle$	
Guard	`(a==b);`	$\langle$ selectCond $(Expression)$ $\rangle$	
Constant Assignment	`x=2;`	$\langle$ setCst (var, val) $\rangle$	
Expression Assignment	`x=a+b;`	$\langle$ setExpr $(var, Expression)$ $\rangle$	$\langle$ setCst () $\rangle$ $\langle$ up () $\rangle$ $\langle$ down () $\rangle$
Fifo not Full	`f![_];`	$\langle$ notFull $(fifo)$ $\rangle$	$\langle$ checknotFull (n) $\rangle$ (Compares the size of the Fifo and the number (n) of used places.
Fifo not Empty	`f?[_];`	$\langle$ notEmpty $(fifo)$ $\rangle$	
Write Constant in Fifo	`f!2;`	$\langle$ writeCst $(fifo, cst)$ $\rangle$	$\langle$ addTailCst (cst) $\rangle$ ("Tail" means encounter of a $\sharp$)
Write Expression in Fifo	`f!(a+b);`	$\langle$ writeExpr $(fifo, expression)$ $\rangle$	$\langle$ writeCst () $\rangle$ $\langle$ up () $\rangle$ $\langle$ downFifoExpr () $\rangle$
Read variable in Fifo	`f?v;`	$\langle$ readVar $(fifo, var)$ $\rangle$	$\langle$ setCst () $\rangle$ $\langle$ up () $\rangle$ $\langle$ downVarFifo () $\rangle$

state-space construction. Futhermore, reachable state-space doesn't depend on the property to check and it can be computed and stored *before* checking all properties we want. As the properties checked may be not relevant, it is useful to check new properties without having to restart reachable state-space build-up.

The relevance of DDD is also compared to a BDD implementation. The experiment compares the time and memory needed to compute the set of reachable states (based on the *post* operator) and the verification of a CTL property (based on the *pre* operator). The BDD and DDD-based tools starts from the same internal representation of the ProMeLa program, and are based on the same verification algorithms save the library used : Our own DDD library or the Buddy package [18]. Buddy offers a set of functions to manage finite sets and finite integers, represented as BDD vectors, and also supports dynamic reordering (that DDD does not handle).

Comparison results are given on table 5.1, we used a 3,2GHz Intel Pentium IV, calculus were automatically aborted after a day or 1GB of used memory. The columns SPIN, BDD, DDD and DDD-O contains performances of a given tool. DDD-O is obtained by forcing a "natural" order on the tree inspired by the system's topology. For each tool, column *reach* means the user time needed to build-up the reachable state-space; column *check* gives the tuser time needed to compute the set of states satisfying the CTL property; column *mem* indicates the memory used to perform the computation of the reachable state-space (once the reachable state-space is built, no additional memory is needed to check the CTL property).

The systems we checked are:

– A massively parallel system using very simple components on a ring: The dining philosophers problem.
– Systems with less (but more complex) components: The leader election on a ring and sliding window protocol.
– Systems dealing with complex arithmetic expression without possibility to force a "natural" order (Bakery's and Peterson's Algorithms).

Results on DDDs (with or without static ordering) were obtained without any optimization (dynamic reorder, state-space compression, partial order reduction, saturation etc.) save the use of a computation cache. Results on BDDs were obtained using the more relevant optimization (particulary Sift or Win2ite reorder algorithms). SPIN results were obtained using *a posteriori* suggested parameters by the SPIN model checker, which led us to recompute some calculus to obtain the best performances.

The column S in table 5.1 gives the number of states generated.

5.2 Discussion

5.2.1 Symbolic vs Enumerative

We only consider the computation of the reachable state-space.

For small sized systems, and systems whose behaviour is highly sequential, SPIN presents better performances in both time and memory than BDD and DDD approaches. As soon as the systems'size grows, BDD and DDD manage better the combinatorial explosion than SPIN.

We can bring out this property by comparing dining philosophers, leader election and sliding window systems. The topology of these examples is the same (a

Table 2. Stats on SPIN, DDDs or BDDs

		SPIN		BDD			DDD			DDD-O		
	$\|S\|$	reach	mem	reach	check	mem.	reach	check	mem	reach	check	mem
N		(sec.)	MB	(sec.)	(sec.)	MB	(sec.)	(sec.)	MB	(sec.)	(sec.)	MB
			Dining Philosophers	$\mathbf{AG}(\mathbf{AF}(eating\ philosopher))$								
5	244	< 1	2.3	< 1	< 1	93	< 1	< 1	4.3	< 1	< 1	5.5
10	5.9e4	2.3	36.7	4	6	94	31	170	17.6	9	2	11
11	1.7e5	7.6	41.6	9	15	94	82	935	30	14	2	12.8
12	5.3e5	29.1	90.1	23	42	94	277	6652	57	21	3	16.6
13	1.6e6	193	114.8	55	117	94	1021	3.7e4	105	30	3	19.6
14	4.8e6	660	737	137	337	95	4534	$> 24h$	203	41	4	24.4
15	1.4e7	2251	1067	380	1231	96	18375	$> 24h$	397	54	5	29.6
20	3.5e9	*	$> 1GB$	*	*	*	*	*	$> 1GB$	182	9	64.1
50	7.2e23	*	$> 1GB$	*	*	*	*	*	$> 1GB$	8006	67	994
			Leader Election	$\mathbf{AF}(best\ candidate\ elected)$								
5	5.4e3	10	23	510	191	94	8	1	6.3	11	1	6.2
6	3.2e4	262	144	5056	5394	144	34	5	11	37	3	11
7	1.8e5	*	$> 1GB$	*	*	*	127	22	28	95	13	19
10	3.3e7	*	$> 1GB$	*	*	*	4533	*	663	716	*	77
			Sliding Window	$\mathbf{AG}(\mathbf{AF}(new\ message\ sent))$								
2	4.0e5	< 1	8.1	552	221	102	486	142	17.8	662	206	17
3	3.5e7	29	450	65429	63216	332	4.8e5		565	4.8e5		475
4	*	*	$> 1GB$	*	*	*	*	*	$> 1GB$	*	*	$> 1GB$
			Peterson	$\mathbf{AF}(query_i\ satisfied)$ and $\mathbf{AG}(query_i \Rightarrow \mathbf{AF}(critical_i))$								
2	208	< 1	5.5	< 1	1	94	< 1	< 1	3.6			
3	2.5e4	< 1	5.8	17	39	93	7	4	5.9			
4	6.4e6	20	402	1674	1093	400	1278	216	113			
5	*	*	$> 1GB$	4.7e5	$> 24h$	900	*	*	$> 1GB$			
			Bakery	$\mathbf{AF}(client_i\ satisfied)$								
4	1.8e5	< 1	5.2	50	< 1	94	17	< 1	8.9			
5	1.6e7	1385	6.4	1231	< 1	93	379	5	46			
6	2.2e9	$> 24h$	*	*	*	*	61901	47	455			

ring) but the complexity of the nodes differs. Figure 6 shows (on a logarithmic scale) the number of *new* states generated at each iteration, by applying the *post* operator. The sliding-window protocol does not produce more than 5000 states at each iteration, and in this case the use of symbolic methods is counter-productive. On the other hand, for systems producing a huge number of new states at each step, (as leaders or philosophers, growing up to 10^8 states), symbolic technics are relevant, but variables' ordering strategies have to be used to outperform significantly enumerative state-space construction.

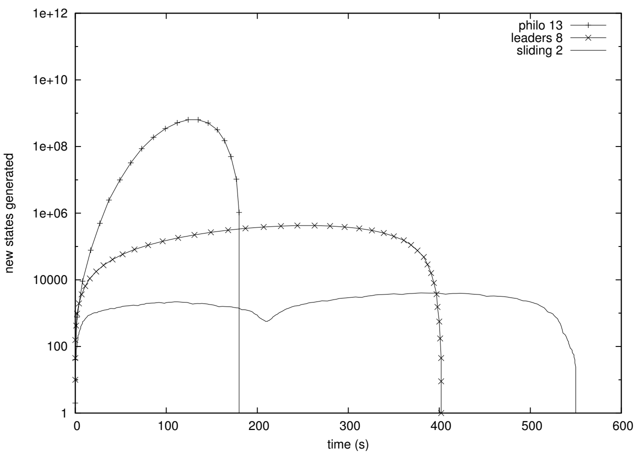

Fig. 6. New states generated by applying *post*

The sequencial aspect of Sliding Window is not the only reason why symbolic methods are outperformed by SPIN: It uses buffered channels containing structured data that have been flattened to be represented on a DDD. First, each assignement or reading/writing operation needs to compose as many homomorphisms as there is fields in the concerned structure. Second, this unexpected growth of the tree's depth increases the number of nodes and homomorphisms to store (and the memory needed), and slow down the computation cache.

5.2.2 DDDs vs BDDs

In the examples, all variables'domains can be *a priori* defined, hence are representable either with DDD or with BDD (for BDD, all the values of the type domain are encoded). DDDs shows more flexibility when using arithmetic expression using arrays (Bakery and Peterson). In fact, expressions concerning arrays cannot be computed with an orthogonal product ($t[n] = \sum_{i=0}^{n-1} t[i] \times (i = n)$) on BDDs because of the necessity two operands of the same domain to build operators (this is a Buddy limitation). DDDs doesn't suffer this particularity as they don't care about variables's domains. Thus, without changing global behavior of processes, we had to modify some instruction in the ProMeLa code to simulates the orthogonal product to fit with Buddy's interface.

Handling expression with many operands made BDDs a little more efficient than DDDs when checking the Peterson's algorithm (one step beyond).

When any static ordering is imposed, DDDs show comparable performances with BDDs using the best ordering configuration. When using a "natural" order, DDDs overcomes BDDs, but applying this good DDD-order on BDD does not improve BDD performance (it is just the opposite !). Considering speed, DDDs proove the efficiency of inductive methods as computation time gets lower for DDDs than BDDs as the system's complexity grows.

Concerning CTL formulas and computing *pre* operator, the structural differences inclined in favour of DDDs. Even if abstracting a variable on a BDD is more efficient than using reachable states, the principle of a unique coupled traversal in DDD avoids to compute other treatments (selection and intersection) on the whole BDD. The locality evaluation due to homomorphism is the main factor explaining the gain of DDD over BDD.

5.2.3 Future Works

As stats shown on table 5.1 were given without any optimization on DDDs (save ordering variables manualy in the last column), DDDs'formalism (Shared tree and inductive methods) prove its efficiency for state-space construction, and its ability to contain state-space explosion. Without variables' ordering, DDDs present comparable performances with explicit and compressed representation in SPIN or with symbolic representation on ordered BDDs. Save variable's order that improves sharing quality, two DDDs disfavourable parameters were not minimized:

- variables' number that increases the tree's depth and
- sequencial procedures that makes too thin the new states frontier.

The last evolution of DDDs, the Set Decision Diagrams (SDDs [27]) that labels the edges of the tree with data sets (DDDs or SDDs) allows best sharing properties on hierarchical modelled systems and decreases the depth of the tree on hierarchicaly modeled systems. Concerning sequencial aspect of some kind of systems, a saturation algorithm may help to produce more states by applying the *post* operator for a unique process until a fixpoint is reached. After saturating a process, some pattern are outlined and the saturation of other processes will take into account more execution contexts. This method can be implemented on homomorphism that launchs the saturation after having reached a variable that concerns the saturated process. For instance, on DDDs with "natural" order, saturating processes in increasing then decreasing order allow us to construct the state-space of 50 dining philosophers in 11 seconds with "only" 7MB of memory. These first results encourage us to pursue our investigation of DDDs and the derived structures to build verification tools.

6 Conclusion and future works

We developped a CTL symbolic model checker for static ProMeLa systems that can verify safety and liveness properties when SPIN is ineficient. This tool is based on Data Decision Diagrams that reproduces shared and tree based canonical representation of OBDDs. This structure, and the associated formalism of homomorphisms, helps to handle numeric values and allows complex transitions representation. As with BDDs, variables' ordering is critical. We indentify the main characteristics that makes symbolics methods more efficient than explicits methods (SPIN gives best results only on strongly sequential systems). Comparison with OBDDs prove the relevance to work with non boolean variables for state-space representation and inductive methods for state-space construction (rather than all-BDD construction). Mains lacks are linked to the depth of the tree, specially when many structured types are flattened. As DDDs and homomorphisms' formalism prove their efficiency, our future works resides in build a Model-Checker that exploits hierarchical properties of the checked systems by the way of Set Decision Diagrams that will overcome variable's multiplicity problem and will limit needed material resources by reducing the size of computation cache.

References

1. Vincent Beaudenon, Emmanuelle Encrenaz, and Jean-Lou Desbarbieux. Design validation of zcsp with spin. In *Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)*, page 102. IEEE Computer Society, 2003.
2. R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA, / 1996. Springer Verlag.
3. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

4. Randal E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD, San Jose/CA*, pages 236–243. IEEE CS Press, Los Alamitos, 1995.
5. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.
6. G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths, 2002.
7. Gianfranco Ciardo, Gerald Luettgen, and Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. Technical report, 1999.
8. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
9. Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier. Data decision diagrams for petri net analysis. In *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, pages 101–120. Springer-Verlag, 2002.
10. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 208–219. Springer-Verlag New-York , Inc., 1996.
11. E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi: application modeling for signal processing systems. In *Proceedings of the 37th conference on Design automation*, pages 402–405. ACM Press, 2000.
12. T. Henzinger and H. Wong-Toi P.-H. Ho. Hytech: a model-checker for hybrid systems. 1:110–122, 1997.
13. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
14. G. J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
15. G.J. Holzmann. State Compression in Spin. Twente University, The Netherlands, April 1997.
16. G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.
17. Denis Hommais, Frédéric Pétrot, and Ivan Augé. A practical tool box for system level communication synthesis. In *Proceedings of the ninth international symposium on Hardware/software codesign*, pages 48–53. ACM Press, 2001.
18. Jørn Lind-Nielsen and Henrik Reif Andersen. Stepwise CTL model checking of state/event systems. In *CAV'99: Computer Aided Verification* . Lecture Notes in Computer Science, 1999.
19. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
20. Andrew S. Miner and Gianfranco Ciardo. Efficient reachability set generation and storage using decision diagrams. In Donatelli, Susanna and Kleijn, Jetty, editors, *Lecture Notes in Computer Science: Application and Theory of Petri Nets 1999, 20th International Conference, ICATPN'99, Williamsburg, Virginia, USA*, volume 1630, pages 6–25. Springer-Verlag, june 1999.
21. D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. Sixth Int Conf. Computer Aided Verification (CAV94)*, pages 377–390, 1994.
22. Hong Peng, Sofiène Tahar, and Ferhat Khendek. Comparison of SPIN and VIS for protocol verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):234–245, February 2003.
23. A. Pnueli. The Temporal Logic of Programs. In *18th IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.
24. P. Racloz. *Symbolic proof of temporal properties for various kinds of Petri nets*. PhD thesis, University of Geneva, Switzerland, April 1994.
25. Y.S. Ramakrishna, O. Sokolsky, Xiaoqun Du, S. Smolka, and E. Stark. Fighting livelock in the i-protocol: A comparative study of verification tools. In *Proc. of the 5th international conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'99*, pages 74–88. Springer-Verlag, 1999.
26. J. Sifakis. A unified approach for studying the properties of transition systems. *Theoretical Computer Science*, 18(3):227–258, June 1982.
27. Yann Thierry-Mieg. *Techniques pour le model-checking de spcification de Haut Niveau*. PhD thesis, Universit Pierre et Marie Curie (UPMC, Paris 6) - LIP6, 2004.