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.

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 formally 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 system’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 (100 states) [5]. These methods deal with set of states contrary to explicit methods handling states one-by-one. Symbolic Methods usually lend themselves 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 symbolic 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.

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 developed 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 excellent 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 applicable 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 instantiated as often as necessary, which allows representation of dynamic systems. DDDs are endowed with a formalism of inductive methods that are
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 substructures.

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.
- $L : S \rightarrow 2^P$ 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 $\pi^i$ the suffix of $\pi$ 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.
  
  \[ s \models f \iff p \in L(s) \]

- Let $f$ and $g$ be to CTL formulas, then $\neg f$, $f \land g$, $\text{EX}f$, $\text{EG}f$ et $\text{EF}f \cup \text{UG}g$ are CTL formulas.
  
  \[ s_0 \models \neg f \iff s_0 \not\models f \]
  
  \[ s_0 \models f \land g \iff s_0 \models f \text{ and } s_0 \models g \]
  
  \[ s_0 \models \text{EX}f \iff \text{there exists a path } \pi \text{ such as } \pi^0 \models f \]
  
  \[ s_0 \models \text{EG}f \iff \text{there exists a path } \pi \text{ such as } \forall i \geq 0, \pi^i \models f \]
  
  \[ s_0 \models \text{EF}f \cup \text{UG}g \iff \text{there exists a path } \pi \text{ such as } \exists k \geq 0 \text{ and } \forall 0 \leq i < k, \pi^i \models f \]

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 $\text{post}$ and $\text{pre}$ [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 $\text{post}$ transformer computes the set of states that are reachable from states grouped into a set $X$:

\[ \text{post}(X) = \bigsqcup_{s \subset X} s' / R(s, s') \]
The \texttt{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 \( s \models \text{EX}(X) \).

\[
\text{pre}(X) = \bigcup_{s \in X} s' / R(s', s)
\]

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

\[
M, s \models \text{EG} f \iff M, s \models f \land \text{EX}(\text{EG} f)
\]

\[
M, s \models \text{EU} f g \iff \forall M, s \models (f \land \text{EX}(\text{EU} f g))
\]

In section 3 we describe the abstract data type we use to represent sets of states and the associated formalism to construct \textit{post} and \textit{pre} operators.

3 Data Decision Diagrams

3.1 Presentation

Data Decision Diagrams (DDDs,\cite{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 outgoing edges are encoded as DAGs. Each node is labeled with a variable. The set \( \{ a = 1, b = 2 \} \), \( \{ a = 2, b = 1 \} \), \( \{ a = 2, b = 2, b = 3 \} \) for the left-hand side.

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

is an homomorphism.

This last expression \( \sum_{x \in \text{Dom}(e)} \phi(e, x)(\alpha(x)) \) is only applied on DDD nodes. In such cases the homomorphism uses only local information: The variable \( e \) and its possible values \( x \in \text{Dom}(e) \). For each of these values, and

![Fig. 2. Two well-defined DDD](image-url)
its associated outgoing edge $a(x)$, 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, $\text{var}$ be a variable of $E$ appearing in $d$, and $\text{cst} \in \text{Dom}(\text{var})$. The homomorphism $\langle \text{setCst}(\text{var}, \text{cst}) \rangle$ that performs the assignment $\text{var} = \text{cst}$ is defined as below:

$$\langle \text{setCst}(\text{var}, \text{cst}) \rangle(e, x) =
\begin{cases}
  \text{if } \text{var} = e \\
  \text{else } \frac{\text{cst}}{\top} (\text{id})
\end{cases}

\text{up} (\text{var}, \text{val}) = e \overset{\text{val}}{\rightarrow} \text{var}, \text{val} (\text{id})$$

And:

$$\langle \text{setExpr}(\text{var}, \text{expr}) \rangle(e, x) =
\begin{cases}
  \text{if } \text{var} = e \\
  \text{if } e \in \text{expr} \\
  \text{if } \|\text{expr}^{(e=\text{var})}\| = 0 \\
  e \overset{\top}{\rightarrow} (\text{id}) \\
  \text{else } \text{down}(\text{var}, \text{expr}^{(e=\text{var})}) \\
  \text{else } \text{down}(\text{var}, \text{expr}^{(e=\text{var})}) \\
\end{cases}

\text{eval} = \langle \text{expr} \rangle$$

The second part of the expression (if $e \neq \text{var}$) is immediate, the homomorphism propagates itself with (if necessary) a completed expression. Then if the expression is assessable ($\|\text{expr}^{(e=\text{var})}\| = 0$) the propagated homomorphism becomes $\langle \text{setCst}() \rangle$. The first part of the expression deals with cases where the assigned variable $\text{var}$ is reached before the expression $\text{expr}$ 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 $\langle \text{up]() \rangle$ and $\langle \text{down()} \rangle$ homomorphisms computes these (complete, store, and replace) operations in a unique traversal.

**Example 1 (\langle \text{up()} \rangle and \langle \text{down()} \rangle usage).** We aim at performing the assignment $b = a + b + d$ on the DDD
\[
\begin{align*}
\begin{array}{l}
\text{\( a \triangleright b \triangleleft c \triangleright d \triangleleft e \triangleleft 1. \)} \\
\langle \text{setExpr} (b, a + b + d) \rangle (a \triangleright b \triangleleft c \triangleright d \triangleleft e \triangleleft 1) \\
= a \triangleright \langle \text{setExpr} (b, 1 + b + d) \rangle (b \triangleright c \triangleright d \triangleleft e \triangleleft 1) \\
= a \triangleright \langle \text{down} (b, 1 + 2 + d) \rangle (c \triangleright d \triangleleft e \triangleleft 1) \\
= a \triangleright \langle \text{up} (c, 3) \rangle \circ \langle \text{down} (b, 3 + d) \rangle (d \triangleleft e \triangleleft 1) \\
= a \triangleright \langle \text{up} (c, 3) \rangle (b \triangleright d \triangleleft e \triangleleft 1) \\
= a \triangleright b \triangleright c \triangleright d \triangleleft e \triangleleft 1
\end{array}
\end{align*}
\]

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 assignment (or FIFO’s reading) occurred?

The set of states that precedes 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:

\[
\langle \text{preSetCst} (\text{var}, \text{cst}, C) \rangle (\text{TO}) = \langle \text{sweet} (\text{var}, \text{cst}, \text{TO}) \rangle (C)
\]

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

1. There is no predecessor (nor successor) for the empty set:

\[
\langle \text{sweet} (v, c, \text{TO}) \rangle = \emptyset
\]

if \( \text{TO} = \emptyset \).

2. Before encountering var, a local intersection is made:

\[
\langle \text{sweet} (v, c, \text{TO}) \rangle (e, x) = e \triangleright \langle \text{sweet} (v, c, \text{TO}[x]) \rangle
\]

if \( e \neq v \) where \( \text{TO}[x] \) is the node reached by outgoing edge of \( \text{TO} \), 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 \( \langle \text{sweet} () \rangle \) works on candidate states represented on C). After this an intersection on remaining variables is obtained with a classical product of DDDs.

\[
\langle \text{sweet} (v, c, \text{TO}) \rangle (e, x) = e \triangleright \langle (\text{id}) \cap \text{TO}[x] \rangle
\]

A computation of \( \langle \text{sweet} () \rangle \) operator is given on figure 3. We are applying the homomorphism \( \langle \text{setExpr} (b, 3, \text{TO}) \rangle \) on the DDD C (see first and second schemes). The DDD TO does not accept \( a = 2 \), thus application of \( \langle \text{sweet} () \rangle \) on \( a \triangleright \ldots \) leads to the empty set (according to formula 1), and the homomorphism propagates in C on the edge \( a \triangleright C_b \) after restricting TO to the suffix of \( a = 1 \) (the outgoing edge of \( \text{TO} \) 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 \( (b \triangleright C_{a1}) + (b \triangleright C_{a2}) \) is intersected with states of \( \text{TO} \) that represents the execution context \( b = 2 \) \( (b \triangleright d2) \) 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 Assignment).** Let \( d \) be a well defined DDD, \( \text{var} \) be a variable of \( d \), \( \text{expr} \) be an arithmetic expression such as any parameter of \( \text{expr} \) is defined on \( d \), and \( C \) the state of reachable states. The homomorphism \( \langle \text{setExpr} (\text{var}, \text{expr}) \rangle \) that performs the pre operator (according to \( C \)) of assignment \( \text{var} = \text{expr} \) is defined as
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-expansion 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 \( i \)th boolean function of an 8-bits adder. The bit-to-bit assignment is performed using bi-implication (boolean function \( \text{xnor} \)). For example, the assignment operator \( a = b \lor c \) for a set of states represented on the boolean function \( f \) is performed in three steps:

1. Abstracting the assigned variable: \( f' = \exists z f \).
2. Constructing the bi-implication operator between \( a \) and \( b \lor c: x = a \oplus b \lor c \).
3. Constructing the new function \( g = f' \land x \).

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

On the other hand, MDDs represent boolean (or even-true 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 layers 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
event enabled by $b_0$: $\langle (1, 1, 1), (2, 1, 1) \rangle$

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

Node $(a0)$, labelled by the variable $a$, with 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 assignment is performed if $b = c = 1$ (other cases have to be handled). The corresponding enabling set of states is $\langle *, 1, 1, * \rangle$ (which means that $b$ and $c$ are set to 1, $a$ and $d$ don’t matter) and the result after transition is $\langle *, 2, 1, * \rangle$. As variable $a$ is not concerned, the prefixes $\langle a \rangle \downarrow$ and $\langle a \rangle \uparrow$ are preserved the computation starts on layer $b$. Corresponding paths are: $\langle b0 \rangle \downarrow \langle c0 \rangle \downarrow \langle [d0] \rangle$ and $\langle b1 \rangle \uparrow \langle c1 \rangle \downarrow \langle [d1] \rangle$. The square brackets denotes suffixes of the paths that are not modified by the treatment. We denote the event $c: \langle *, 1, 1, * \rangle \xrightarrow{\epsilon} \langle *, 2, 1, * \rangle$ they are respectively united with result pattern joint to the same suffixes of enabling patterns $\langle b2 \rangle \xrightarrow{\epsilon} \langle c2 \rangle \downarrow \langle [d0] \rangle$ and $\langle b3 \rangle \xrightarrow{\epsilon} \langle c3 \rangle \downarrow \langle [d1] \rangle$.

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

\[
\text{variable} \xrightarrow{\text{value}} 1
\]

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

\[
\text{program} \xrightarrow{\text{counter}} \text{local} \xrightarrow{\text{var}} 1 \xrightarrow{\text{...}} 1
\]

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

\[
t[0] \rightarrow t[1] \rightarrow \ldots \rightarrow t[n-1] \rightarrow 1
\]

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 held in the FIFO (\#). Thus, FIFO are constructed using the following way:

\[
f \xrightarrow{\text{size}} f \xrightarrow{1^\text{st} \text{elt}} \ldots f \xrightarrow{t^\text{th} \text{elt}} f \xrightarrow{\#} 1
\] (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-emptiness 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:

\[
\langle \text{setCst} (pc, n) \rangle \circ \langle \text{setExpr} (a, b + c) \rangle \circ \langle \text{selectCond} (pc = m) \rangle
\]

More complex homomorphisms 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ProMeLa code</th>
<th>Main Homomorphism</th>
<th>Sub-Homomorphisms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td>goto label; <code>{ selectAndSet (pc, pcVal, npcVal)</code> }</td>
<td><code>{ selectCond (Expression)</code> }</td>
<td><code>{ setCst () }</code> <code>{ up () }</code> <code>{ down () }</code></td>
</tr>
<tr>
<td>Guard</td>
<td>(a==b); <code>{ selectCond (Expression)</code> }</td>
<td><code>{ notFull (fifo)</code> } <code>{ checknotFull (n)</code> }</td>
<td>Compares the size of the Fifo and the number (n) of used places.</td>
</tr>
<tr>
<td>Constant Assignment</td>
<td>x=2; <code>{ setCst (var, val)</code> }</td>
<td><code>{ notEmpty (fifo)</code> } <code>{ addTailCst (cst)</code> }</td>
<td>(Tail” means encounter of a ♯)</td>
</tr>
<tr>
<td>Expression Assignment</td>
<td>x=a+b; <code>{ setExpr (var, Expression)</code> }</td>
<td><code>{ writeExpr (fifo, expression)</code> }</td>
<td><code>{ writeCst () }</code> <code>{ up () }</code> <code>{ downFifoExpr () }</code></td>
</tr>
<tr>
<td>Fifo not Full</td>
<td>f![].; <code>{ notFull (fifo)</code> }</td>
<td><code>{ writeCst (fifo, cst)</code> } <code>{ addTailCst (cst)</code> }</td>
<td>(Tail” means encounter of a ♯)</td>
</tr>
<tr>
<td>Fifo not Empty</td>
<td>f?[.]; <code>{ notEmpty (fifo)</code> }</td>
<td><code>{ writeCst (fifo, cst)</code> } <code>{ addTailCst (cst)</code> }</td>
<td>(Tail” means encounter of a ♯)</td>
</tr>
<tr>
<td>Write Constant in Fifo</td>
<td>f!2; <code>{ writeCst (fifo, cst)</code> }</td>
<td><code>{ writeExpr (fifo, expression)</code> }</td>
<td><code>{ writeCst () }</code> <code>{ up () }</code> <code>{ downFifoExpr () }</code></td>
</tr>
<tr>
<td>Write Expression in Fifo</td>
<td>f!(a+b); <code>{ writeExpr (fifo, expression)</code> }</td>
<td><code>{ writeCst (fifo, expression)</code> }</td>
<td><code>{ writeCst () }</code> <code>{ up () }</code> <code>{ downFifoExpr () }</code></td>
</tr>
<tr>
<td>Read variable in Fifo</td>
<td>f?v; <code>{ readVar (fifo, var)</code> }</td>
<td><code>{ readVar (fifo, var)</code> } <code>{ setCst () }</code> <code>{ up () }</code> <code>{ downVarFifo () }</code></td>
<td></td>
</tr>
</tbody>
</table>

state-space construction. Furthermore, 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 start 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 (particular 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

<table>
<thead>
<tr>
<th>N</th>
<th>SPIN</th>
<th>BDD</th>
<th>DDD</th>
<th>DDD-O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>reach</td>
<td>check</td>
<td>reach</td>
<td>check</td>
</tr>
<tr>
<td></td>
<td>(sec.)</td>
<td>mem.</td>
<td>(sec.)</td>
<td>mem.</td>
</tr>
<tr>
<td>5</td>
<td>244</td>
<td>&lt;1</td>
<td>2.3</td>
<td>&lt;1</td>
</tr>
<tr>
<td>10</td>
<td>5.9e4</td>
<td>99</td>
<td>36.7</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>1.7e5</td>
<td>7.6</td>
<td>41.6</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>5.3e5</td>
<td>29.1</td>
<td>90.1</td>
<td>23</td>
</tr>
<tr>
<td>13</td>
<td>1.6e6</td>
<td>193</td>
<td>114.8</td>
<td>55</td>
</tr>
<tr>
<td>14</td>
<td>4.8e6</td>
<td>660</td>
<td>737</td>
<td>137</td>
</tr>
<tr>
<td>15</td>
<td>1.4e7</td>
<td>2251</td>
<td>1067</td>
<td>380</td>
</tr>
<tr>
<td>20</td>
<td>3.5e9</td>
<td>*</td>
<td>&gt;1GB</td>
<td>*</td>
</tr>
<tr>
<td>50</td>
<td>7.2e23</td>
<td>*</td>
<td>&gt;1GB</td>
<td>*</td>
</tr>
</tbody>
</table>

Dining Philosophers

AG(AF(eating philosopher))

<table>
<thead>
<tr>
<th>N</th>
<th>SPIN</th>
<th>BDD</th>
<th>DDD</th>
<th>DDD-O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>reach</td>
<td>check</td>
<td>reach</td>
<td>check</td>
</tr>
<tr>
<td></td>
<td>(sec.)</td>
<td>mem.</td>
<td>(sec.)</td>
<td>mem.</td>
</tr>
<tr>
<td>5</td>
<td>5.4e3</td>
<td>10</td>
<td>23</td>
<td>151</td>
</tr>
<tr>
<td>6</td>
<td>3.2e4</td>
<td>262</td>
<td>144</td>
<td>5056</td>
</tr>
<tr>
<td>7</td>
<td>1.8e5</td>
<td>*</td>
<td>&gt;1GB</td>
<td>*</td>
</tr>
<tr>
<td>10</td>
<td>3.3e7</td>
<td>*</td>
<td>&gt;1GB</td>
<td>*</td>
</tr>
</tbody>
</table>

Leader Election

AF(best candidate elected)

<table>
<thead>
<tr>
<th>N</th>
<th>SPIN</th>
<th>BDD</th>
<th>DDD</th>
<th>DDD-O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>reach</td>
<td>check</td>
<td>reach</td>
<td>check</td>
</tr>
<tr>
<td></td>
<td>(sec.)</td>
<td>mem.</td>
<td>(sec.)</td>
<td>mem.</td>
</tr>
<tr>
<td>2</td>
<td>4.0e5</td>
<td>&lt;1</td>
<td>8.1</td>
<td>952</td>
</tr>
<tr>
<td>3</td>
<td>3.5e7</td>
<td>29</td>
<td>450</td>
<td>65429</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>*</td>
<td>&gt;1GB</td>
<td>*</td>
</tr>
</tbody>
</table>

Sliding Window

AG(AF(new message sent))

<table>
<thead>
<tr>
<th>N</th>
<th>SPIN</th>
<th>BDD</th>
<th>DDD</th>
<th>DDD-O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>reach</td>
<td>check</td>
<td>reach</td>
<td>check</td>
</tr>
<tr>
<td></td>
<td>(sec.)</td>
<td>mem.</td>
<td>(sec.)</td>
<td>mem.</td>
</tr>
<tr>
<td>2</td>
<td>208</td>
<td>&lt;1</td>
<td>5.5</td>
<td>&lt;1</td>
</tr>
<tr>
<td>3</td>
<td>2.5e4</td>
<td>&lt;1</td>
<td>5.8</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>6.4e6</td>
<td>20</td>
<td>402</td>
<td>1674</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>*</td>
<td>&gt;1GB</td>
<td>4.7e5</td>
</tr>
</tbody>
</table>

Peterson

AF(query, satisfied) and AG(query⇒AF(critical))

<table>
<thead>
<tr>
<th>N</th>
<th>SPIN</th>
<th>BDD</th>
<th>DDD</th>
<th>DDD-O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>reach</td>
<td>check</td>
<td>reach</td>
<td>check</td>
</tr>
<tr>
<td></td>
<td>(sec.)</td>
<td>mem.</td>
<td>(sec.)</td>
<td>mem.</td>
</tr>
<tr>
<td>2</td>
<td>1.8e5</td>
<td>&lt;1</td>
<td>5.2</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>1.6e7</td>
<td>1385</td>
<td>6.4</td>
<td>1231</td>
</tr>
<tr>
<td>6</td>
<td>2.2e9</td>
<td>&gt;24h</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

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’ domains. Thus, without changing global behavior of processes, we had to modify some instruction in the ProMeLa code to simulate 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 prove 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 manually 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
– sequential 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 hierarchically 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 launches 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 inefficient. 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 symbolic methods more efficient than explicit 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