

Data Decision Diagrams for Petri Nets Analysis

Jean-Michel Couvreur¹, Emmanuelle Encrenaz², Emmanuel Paviot-Adet²,
Denis Poitrenaud², and Pierre-André Wacrenier¹

¹ LaBRI, Université Bordeaux 1, Talence, France
<name>@labri.fr

² LIP6, Université Pierre et Marie Curie, Paris, France
<name>@lip6.fr

Abstract. This paper presents a new data structure, the Data Decision Diagrams, equipped with a mechanism allowing the definition of application-specific operators. This mechanism is based on combination of inductive linear functions offering a large expressiveness while alleviating for the user the burden of hard coding traversals in a shared data structure. We demonstrate the pertinence of our system through the implementation of a verification tool for various classes of Petri nets including self modifying and queuing nets.

Topics. Petri Nets, Decision Diagram, System verification.

1 Introduction

The design and verification of distributed systems present a scientific and technical challenge that must be met by a combination of techniques that scale up to the complexity of systems produced in industrial applications. Simulation is already a recognized tool within industries dealing with complex systems such as telecommunications or aeronautics. Even if an acceptable degree of confidence can be reached via this technique, exhaustiveness cannot generally be reached due to the number of possible states of these systems.

In the 90's, the electronic industries, in search of tools to increase their confidence level in their final product, adopted the Binary Decision Diagrams (BDDs) [1] as a way to deal with the high complexity of their components. BDD can be viewed as a tree structure: binary variables involved in states are ordered, a set of nodes is associated to each variable and the variable valuations are associated to the arcs between nodes. When implemented, the uniqueness of BDDs, combined with the tree structure, ensures an efficient technique to deal with numerous states [6, 4]. Then exhaustiveness could be handled, even with billions and billions of states to verify.

The BDDs expressive power is large enough to deal with a large class of finite state system. Even dynamic systems can be verified using such a technique [5]. Therefore, other domains, like parallel system verification, tried to use BDDs to verify complex systems [8]. Since the number of variables of the studied system is a critical parameter, numerous BDD-like structures were created in order to adapt the tree structure to particular needs [7, 2].

Like BDDs, the shared-tree structures are usually stuck to a precise interpretation. Therefore, dealing with states of a new kind of model usually leads to the design of new kinds of structures or to new kinds of operators on existing structures.

In this paper, a new tree structure, the Data Decisions Diagrams (DDD), is introduced. Our goal is to provide a flexible tool that can be easily adapted to any kind of model and that offers the same storage capabilities as the BDD-like structures. Unlike previous works on the subject, operators on the structure are not hard-coded, but a class of operators, called homomorphisms, are introduced to allow transition rules coding. A special kind of homomorphisms, called inductive homomorphisms, allows to define "local" homomorphisms, i.e. homomorphisms that only use information local to a node in its definition. Together with composition, concatenation, union... operations, general homomorphisms are defined.

In our model, a node is associated to a variable and valuations are associated to arcs. The variables domain is integer, but we do not have to know *a priori* bounds.

Another nice feature of the DDDs is that no variable ordering is presupposed in definition. Moreover, variables are not assumed to be part of all paths. They also can be seen many times along the same path. Therefore, the maximal length of a path in the tree is not fixed. This feature is very useful when dealing with dynamic models like queuing Petri nets, but also when a temporary variable is needed (cf the mechanism we used to code self modifying nets firing rules). Even if a global variable ordering is useful to obtain an efficient storage, the fact that this ordering is not part of the definition introduces a great flexibility when one needs to encode a state. Since we use techniques that have been shown efficient to store set of states, since state coding is very flexible and since operator definition is based on a well-founded theoretical foundation, DDD is a tree structure that can be easily adapted to any kind of computational model.

This paper is structured the following way: Section 2 describes DDDs, the homomorphisms, and gives some hints on the implementation we made in our prototype. Section 3 describes a possible use of DDDs for ordinary Petri nets and some of the most popular extensions (inhibitors arcs, capacity places, reset arcs, self modifying nets, queuing nets). Section 4 gives conclusions with perspectives.

2 Data Decision Diagrams

2.1 Definitions of DDDs

Data Decision Diagrams (DDD) are data structures for representing sets of sequences of assignments of the form $e_1 = x_1; e_2 = x_2; \dots e_n = x_n$ where e_i are variables and x_i are values. When an ordering on the variables is fixed and the variables are boolean, DDD coincides with the well-know Binary Decision Diagram. If an ordering on the variables is the only assumption, DDDs are the specialized version of the Multi-valued Decision Diagrams representing characteristic function of sets [7, 2]. For Data Decision Diagram, we assume no variable

ordering and, even more, the same variable may occur many times in an assignment sequence, allowing the representation of dynamic structures.

Traditionally, decision diagrams are often encoded as decision trees. Figure 1, left-hand side, shows the decision tree for the set of sequences of assignments $A = \{(a = 1; a = 1), (a = 1; a = 2; b = 0), (a = 2; b = 3)\}$. As usual, node 1 stands for accepting terminator and node 0 for non-accepting terminator. Since, there is no assumption on the cardinality of the variable domains, we consider node 0 as the default value. Therefore node 0 is not depicted in the figure.

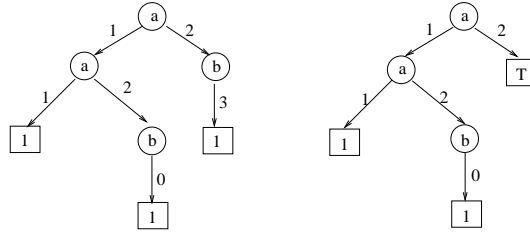


Fig. 1. Two Data Decision Diagrams

Any set of sequences cannot be represented, we thus introduce a new kind of leaf \top for undefined. Figure 1, right-hand side, gives an approximation of the set $A \cup \{(a = 2; a = 3)\}$. Indeed, an ambiguity is introduced since after assigning the first value of variable $a = 2$, one may want to assign both $a = 3$ and $b = 3$. Such assignment sequences cannot be represented in the structure: from a node, only one arc of a given value is allowed.

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

Definition 1 (Data Decision Diagram). *The set \mathbb{ID} of DDDs is defined by $d \in \mathbb{ID}$ if:*

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

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

A straight definition of DDDs equality would lead to a non-unique representation of the empty set (any structure with node 0 as unique terminator). Therefore we introduce the following equivalence relation:

Proposition 1 (Equivalence relation). *Let \equiv the relation inductively defined for $\forall d, d' \in \mathbb{ID}$ as*

- $d = d'$ or
- $d = 0$, $d' = (e', \alpha')$ and $\forall x \in \text{Dom}(e') : \alpha'(x) \equiv 0$ or
- $d = (e, \alpha)$, $d' = 0$ and $\forall x \in \text{Dom}(e) : \alpha(x) \equiv 0$ or
- $d = (e, \alpha)$, $d' = (e', \alpha')$ and $(d \equiv 0) \wedge (d' \equiv 0)$ or
- $d = (e, \alpha)$, $d' = (e', \alpha')$ and $(e = e') \wedge (\forall x \in \text{Dom}(e) : \alpha(x) \equiv \alpha'(x))$

The relation \equiv is an equivalence relation¹.

From now on, a DDD is an equivalence class of the relation \equiv . We will use 0 to represent any empty DDD. This induces a canonical representation of DDDs: in a DDD, nodes equivalent to 0 are replaced by the terminator node 0.

An important feature of DDDs is the notion of approximation of assignment sequence sets based on the terminator node \top : \top represents any set of assignment sequences. When \top does not appear in a DDD, the DDD represents a unique set of assignment sequences; we say that it is well-defined.

Definition 2 (Well-defined DDD). A DDD d is well-defined if

- $d = 0$ or
- $d = 1$ or
- $d = (e, \alpha)$ where $\forall x \in \text{Dom}(e) : \alpha(x)$ is well-defined.

In other words, \top is the worst approximation of a set of assignment sequences. It induces inductively a partial order which formalizes the notion of approximation: the better-defined relation.

Proposition 2 (Better-defined partial order). Let the better-defined relation, denoted \preceq , be inductively defined by $\forall d, d' \in \mathbb{ID}$ as $d \preceq d'$ if

- $d' = \top$ or
- $d \equiv d'$ or
- $d \equiv 0$, $d' = (e', \alpha')$ and $\forall x \in \text{Dom}(e') : 0 \preceq \alpha'(x)$
- $d = (e, \alpha)$, $d' = (e', \alpha')$ and $(e = e') \wedge (\forall x \in \text{Dom}(e) : \alpha(x) \preceq \alpha'(x))$

The relation \preceq is a partial order on equivalent classes of DDDs. Moreover well-defined DDDs are the minimal DDDs.

In order to generalize operators over assignment sequence sets, we observe that better-defined the operands are, better-defined the result is. This leads to the following definition:

Definition 3 (DDD operator). Let f be a mapping $\mathbb{ID}^n \rightarrow \mathbb{ID}$. f is an operator on \mathbb{ID} if f is compatible with the partial order \preceq :

$$\forall (d_i)_i \in \mathbb{ID}^n, \forall (d'_i)_i \in \mathbb{ID}^n : (\forall i : d_i \preceq d'_i) \Rightarrow f((d_i)_i) \preceq f((d'_i)_i)$$

¹ Proofs of propositions are omitted since they are simple but tedious. Indeed, for each proposition, one simply has to enumerate each kind of DDD appearing in it and check the properties.

2.2 Operations on DDDs

First, we generalize the usual set-theoretic operations – sum (union), product (intersection) and difference – to sets of assignment sequences expressed in terms of DDDs. The crucial point of this generalization is that all DDDs are not well-defined and moreover that the result of an operation on two well-defined DDDs is not necessarily well-defined. We propose definitions of these set-theoretic operations which produce the best approximation of the result as possible. In particular, when the two operands are well-defined and when the result may be represented by a well-defined DDD, then the following operators produce exactly this one.

Definition 4 (Set operators). *The sum $+$, the product $*$, the difference \setminus of two DDDs are defined inductively as follows:*

$+$	$0 \vee (e_2, \alpha_2) \equiv 0$	1	\top	$(e_2, \alpha_2) \neq 0$
$0 \vee (e_1, \alpha_1) \equiv 0$	0	1	\top	(e_2, α_2)
1	1	1	\top	\top
\top	\top	\top	\top	\top
$(e_1, \alpha_1) \neq 0$	(e_1, α_1)	\top	\top	$(e_1, \alpha_1 + \alpha_2)$ if $e_1 = e_2$ \top if $e_1 \neq e_2$

$*$	$0 \vee (e_2, \alpha_2) \equiv 0$	1	\top	$(e_2, \alpha_2) \neq 0$
$0 \vee (e_1, \alpha_1) \equiv 0$	0	0	0	0
1	0	1	\top	0
\top	0	\top	\top	\top
$(e_1, \alpha_1) \neq 0$	0	0	\top	$(e_1, \alpha_1 * \alpha_2)$ if $e_1 = e_2$ 0 if $e_1 \neq e_2$

\setminus	$0 \vee (e_2, \alpha_2) \equiv 0$	1	\top	$(e_2, \alpha_2) \neq 0$
$0 \vee (e_1, \alpha_1) \equiv 0$	0	0	0	0
1	1	0	\top	1
\top	\top	\top	\top	\top
$(e_1, \alpha_1) \neq 0$	(e_1, α_1)	(e_1, α_1)	\top	$(e_1, \alpha_1 \setminus \alpha_2)$ if $e_1 = e_2$ (e_1, α_1) if $e_1 \neq e_2$

where, for any $\diamond \in \{+, *, \setminus\}$, $\alpha_1 \diamond \alpha_2$ stands for the mapping in $\text{Dom}(e_1) \rightarrow \mathbb{ID}$ with $\forall x \in \text{Dom}(e_1) : (\alpha_1 \diamond \alpha_2)(x) = \alpha_1(x) \diamond \alpha_2(x)$.

The concatenation operator defined below corresponds to the concatenation of language theory. Nevertheless, the definition takes into account the approximation aspect.

Definition 5 (Concatenation operator). Let d, d' be two DDDs. The concatenation $d \cdot d'$ is inductively defined as follows:

$$d \cdot d' = \begin{cases} 0 & \text{if } d = 0 \vee d' \equiv 0 \\ d' & \text{if } d = 1 \\ \top & \text{if } d = \top \wedge d' \neq 0 \\ (e, \sum_{x \in \text{Dom}(e)} (e \xrightarrow{x} (\alpha(x) \cdot d'))) & \text{if } d = (e, \alpha) \end{cases}$$

The operators respect the definition of operator on DDD (Def. 3) and have the usual properties as commutativity and associativity but $*$ is not associative.

Proposition 3 (Basic operator properties). The operators $*, +, \setminus, \cdot$ are operators on DDDs. Moreover, $*, +$ are commutative and $+, \cdot$ are associative. Operator $*$ is not associative.

Thanks to the basic properties of operator $+$, we may denote a DDD (e, α) as $\sum_{x \in \text{Dom}(e)} (e \xrightarrow{x} \alpha(x))$. Remark that this sum has a finite number of non null DDDs.

Example 1. Let d_A be the DDD represented in left-hand side of Fig. 1, and d_B the right-hand side one. Notice that any DDD may be defined using the constants 0, 1, \top , the elementary concatenation $e \xrightarrow{x} d$ and the operators $+$.

$$\begin{aligned} d_A &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} b \xrightarrow{3} 1 \\ d_B &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \top \end{aligned}$$

Let us now detail some computations:

$$\begin{aligned} d_A + a \xrightarrow{2} a \xrightarrow{3} 1 &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \left(b \xrightarrow{3} 1 + a \xrightarrow{3} 1 \right) \\ &= a \xrightarrow{1} \left(a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{0} 1 \right) + a \xrightarrow{2} \top \\ &= d_B \\ \left(a \xrightarrow{1} 1 * a \xrightarrow{2} 1 \right) * \top &= 0 * \top = 0 \neq a \xrightarrow{1} 1 * \left(a \xrightarrow{2} 1 * \top \right) = a \xrightarrow{1} 1 * \top = \top \\ d_A \setminus d_B &= a \xrightarrow{2} \left(b \xrightarrow{3} 1 \setminus \top \right) = a \xrightarrow{2} \top \\ d_B \cdot c \xrightarrow{4} 1 &= a \xrightarrow{1} \left(a \xrightarrow{1} c \xrightarrow{4} 1 + a \xrightarrow{2} b \xrightarrow{0} c \xrightarrow{4} 1 \right) + a \xrightarrow{2} \top \end{aligned}$$

The next step of our formalization is to introduce homomorphisms over DDDs to allow the definition of complex operations. In the aim of using the previous operators in the homomorphism context, distributivity is a crucial property. However, it must be adapted to take into account the approximation feature.

Proposition 4 (Weak-distributivity). The product operator $*$ and the concatenation operator \cdot are weakly distributive with the sum operator $+$ and the difference operator \setminus is weakly right distributive with operator $+$:

$$\forall d_1, d_2, d \in \mathbb{ID} : \begin{cases} d * d_1 + d * d_2 \preceq d * (d_1 + d_2) \\ (d_1 \setminus d) + (d_2 \setminus d) \preceq (d_1 + d_2) \setminus d \\ d \cdot d_1 + d \cdot d_2 \preceq d \cdot (d_1 + d_2) \\ (d_1 \cdot d) + (d_2 \cdot d) \preceq (d_1 + d_2) \cdot d \end{cases}$$

The well-foundation of these propositions is that, in the absence of undefined values (\top), we regain the usual properties of these binary operators.

2.3 Homomorphisms on DDDs

Our goal is to generalize the notion of homomorphism to DDD taking into account the approximation introduced by \top . The classical identity $f(d_1) + f(d_2) = f(d_1 + d_2)$ is rewritten using the better-defined relation. One can notice that the weak distributivity that we have just introduced above matches for classical mappings as $d * \text{Id}$, $\text{Id} \setminus d$, $\text{Id} \cdot d$, $d \cdot \text{Id}$ where d is a given DDD and Id is the identity. Another requirement for homomorphisms is to be DDD operators and to map 0 to 0.

Definition 6 (Homomorphism). *A mapping Φ on DDDs is an homomorphism if $\Phi(0) = 0$ and*

$$\forall d_1, d_2 \in \mathbb{D} : \begin{cases} \Phi(d_1) + \Phi(d_2) \preceq \Phi(d_1 + d_2) \\ d_1 \preceq d_2 \Rightarrow \Phi(d_1) \preceq \Phi(d_2) \end{cases}$$

The sum and the composition of two homomorphisms are homomorphisms.

Proposition 5 (Sum and composition). *Let Φ_1, Φ_2 be two homomorphisms. Then $\Phi_1 + \Phi_2, \Phi_1 \circ \Phi_2$ are homomorphisms.*

So far, we have at one's disposal the homomorphism $d * \text{Id}$ which allows to select the sequences belonging to the given DDD d ; on the other hand we may also remove these given sequences, thanks to the homomorphism $\text{Id} \setminus d$. The two other interesting homomorphisms $\text{Id} \cdot d$ and $d \cdot \text{Id}$ permit to concatenate sequences on the left or on the right side. For instance, a widely used left concatenation consists in adding a variable assignment $e_1 = x_1$ that is denoted $e_1 \xrightarrow{x_1} \text{Id}$. Of course, we may combine these homomorphisms using the sum and the composition.

However, the expressive power of this homomorphism family is limited; for instance we cannot express a mapping which modifies the assignment of a given variable. A first step to allow user-defined homomorphism Φ is to give the value of $\Phi(1)$ and of $\Phi(e \xrightarrow{x} d)$ for any $e \xrightarrow{x} d$. The key idea is to define $\Phi(e, \alpha)$ as $\sum_{x \in \text{Dom}(e)} \Phi(e \xrightarrow{x} \alpha(x))$ and $\Phi(\top) = \top$. A sufficient condition for Φ being an homomorphism is that the mappings $\Phi(e, x)$ defined as $\Phi(e, x)(d) = \Phi(e \xrightarrow{x} d)$ are themselves homomorphisms. For instance, $\text{inc}(e, x) = e \xrightarrow{x+1} \text{Id}$ and $\text{inc}(1) = 1$ defines the homomorphism which increments the value of the first variable. A second step is introduce induction in the description of the homomorphism. For instance, one may generalize the increment operation to the homomorphism $\text{inc}(e_1)$ which increments the value of the given variable e_1 . A possible approach is to set $\text{inc}(e_1)(e, x) = e \xrightarrow{x+1} \text{Id}$ whenever $e = e_1$ and otherwise $\text{inc}(e_1)(e, x) = e \xrightarrow{x} \text{inc}(e_1)$. Indeed, if the first variable is e_1 , then the homomorphism increments the values of the variable, otherwise the homomorphism is inductively applied to the next variables. The following proposition formalizes the notion of inductive homomorphisms.

Proposition 6 (Inductive homomorphism). *Let I be an index set. Let $(d_i)_{i \in I}$ be a family of DDDs. Let $(\tau_i)_{i \in I}$ and $(\pi_{i,j})_{i,j \in I}$ be family of homomorphisms. Assume that $\forall i \in I$, the set $\{j \in I \mid \pi_{i,j} \neq 0\}$ is finite. Then the following recursive definition of mappings $(\Phi_i)_{i \in I}$:*

$$\forall d \in \mathbb{D}, \Phi_i(d) = \begin{cases} 0 & \text{if } d = 0 \\ d_i & \text{if } d = 1 \\ \top & \text{if } d = \top \\ \sum_{x \in \text{Dom}(e), j \in I} \pi_{i,j} \circ \Phi_j(\alpha(x)) + \tau_i(\alpha(x)) & \text{if } d = (e, \alpha) \end{cases}$$

defines a family of homomorphisms called inductive homomorphisms. The symbolic expression $\sum_{j \in I} \pi_{i,j} \circ \Phi_j + \tau_i$ is denoted $\Phi_i(e, x)$.

To define a family of inductive homomorphisms $(\Phi_i)_{i \in I}$, one has just to set the homomorphisms $\Phi_i(e, x)$ and the DDDs $\Phi_i(1)$. The two following examples illustrate the usefulness of these homomorphisms to design new operators on DDD. The first example formalizes the increment operation. The second example is a swap operation between two variables. It gives a good idea of the techniques used to design homomorphisms for some variants of Petri net analysis.

Example 2. This is the formal description of increment operation:

$$\begin{aligned} inc(e_1)(e, x) &= \begin{cases} e \xrightarrow{x+1} Id & \text{if } e = e_1 \\ e \xrightarrow{x} inc(e_1) & \text{otherwise} \end{cases} \\ inc(e_1)(1) &= 1 \end{aligned}$$

Let us now detail the application of *inc* over a simple DDD:

$$\begin{aligned} inc(b)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) &= a \xrightarrow{1} inc(b)(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) \\ &= a \xrightarrow{1} b \xrightarrow{3} Id(c \xrightarrow{3} d \xrightarrow{4} 1) \\ &= a \xrightarrow{1} b \xrightarrow{3} c \xrightarrow{3} d \xrightarrow{4} 1 \end{aligned}$$

Example 3. The homomorphism *swap*(e_1, e_2) swap the values of variables e_1 and e_2 . It is designed using three other kinds of homomorphisms: *rename*(e_1), *down*(e_1, x_1), *up*(e_1, x_1). The homomorphism *rename*(e_1) renames the first variable into e_1 ; *down*(e_1, x_1) sets the variable e_1 to x_1 and copies the old assignment of e_1 in the first position; *up*(e_1, x_1) puts in the second position the assignment $e_1 = x_1$.

$$\begin{aligned} \text{swap}(e_1, e_2)(e, x) &= \begin{cases} \text{rename}(e_1) \circ \text{down}(e_2, x) & \text{if } e = e_1 \\ \text{rename}(e_2) \circ \text{down}(e_1, x) & \text{if } e = e_2 \\ e \xrightarrow{x} \text{swap}(e_1, e_2) & \text{otherwise} \end{cases} \\ \text{swap}(e_1, e_2)(1) &= \top \end{aligned}$$

$$\begin{aligned} \text{rename}(e_1)(e, x) &= e_1 \xrightarrow{x} \text{Id} \\ \text{rename}(e_1)(1) &= \top \end{aligned}$$

$$\begin{aligned} \text{down}(e_1, x_1)(e, x) &= \begin{cases} e \xrightarrow{x} e \xrightarrow{x_1} \text{Id} & \text{if } e = e_1 \\ \text{up}(e, x) \circ \text{down}(e_1, x_1) & \text{otherwise} \end{cases} \\ \text{down}(e_1, x_1)(1) &= \top \end{aligned}$$

$$\begin{aligned} \text{up}(e_1, x_1)(e, x) &= e \xrightarrow{x} e_1 \xrightarrow{x_1} \text{Id} \\ \text{up}(e_1, x_1)(1) &= \top \end{aligned}$$

Let us now detail the application of *swap* over a simple DDD which enlightens the role of the inductive homomorphisms:

$$\begin{aligned} \text{swap}(b, d)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) &= a \xrightarrow{1} \text{swap}(b, d)(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) \\ &= a \xrightarrow{1} \text{rename}(b) \circ \text{down}(d, 2)(c \xrightarrow{3} d \xrightarrow{4} 1) \\ &= a \xrightarrow{1} \text{rename}(b) \circ \text{up}(c, 3) \circ \text{down}(d, 2)(d \xrightarrow{4} 1) \\ &= a \xrightarrow{1} \text{rename}(b) \circ \text{up}(c, 3)(d \xrightarrow{4} d \xrightarrow{2} 1) \\ &= a \xrightarrow{1} \text{rename}(b)(d \xrightarrow{4} c \xrightarrow{3} d \xrightarrow{2} 1) \\ &= a \xrightarrow{1} b \xrightarrow{4} c \xrightarrow{3} d \xrightarrow{2} 1 \end{aligned}$$

One may remark that $\text{swap}(b, e)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) = a \xrightarrow{1} \top$.

2.4 Implementing Data Decision Diagrams

In order to write object oriented programs handling DDDs, a programmer needs a class hierarchy translating the mathematical concepts of DDDs, of set operators, of concatenation, of homomorphisms and of inductive homomorphisms. These concepts are translated in our interface by the definitions of three classes (`DDD`, `Hom` and `InductiveHom`) where all the means to construct and to handle DDDs and homomorphisms are given. Indeed an important goal of our work is to design an easy to use library interface; so, we have used C++ overloaded operators in order to have the most intuitive interface as possible.

From the theoretic point of view, an inductive homomorphism Φ is an homomorphism defined by a DDD $\Phi(1)$ and an homomorphism family $\Phi(e, x)$. Inductive homomorphisms have in common their evaluation method and this leads to the definition of a class that we named `InductiveHom` that contains the inductive homomorphism evaluation method and gives, in term of abstract methods, the components of an inductive homomorphism: $\Phi(1)$ and $\Phi(e, x)$. In

order to build an inductive homomorphism, it suffices to define a derived class of the class `InductiveHom` implementing the abstract methods $\Phi(1)$ and $\Phi(e, x)$.

The implementation of our interface is based on the three following units:

- A DDD management unit: thanks to hash table techniques, it implements the sharing of the memory and guarantees the uniqueness of the tree structure of the DDDs.
- An HOM management unit: it manages data as well as evaluation methods associated to homomorphisms. Again the syntactic uniqueness of a homomorphism is guaranteed by a hash table. We use the notion of derived class to represent the wide range of homomorphism types.
- A computing unit: it provides the evaluation of operations on the DDDs, as well as the computation of the image of a DDD by an homomorphism. In order to accelerate these computations, this unit uses an operation cache that avoids to evaluate twice a same expression during a computation. The use of cached results reduces the complexity of set operations to polynomial time. Since inductive homomorphisms are user-defined, we cannot express their complexity.

3 Data Decision Diagrams for ordinary Petri nets and some of their extension

In this section, we show how DDDs can be used as a toolkit for the verification of a large class of Petri nets. At first, we introduce Petri nets with marking-dependent valuation enriched by place capacity. From this definition, usual subclasses are defined from ordinary nets to self modifying ones. In the following of the section, we present the key element of a model checker, i.e. the inductive homomorphism encoding the symbolic transition relation associated to each subclass of net. Finally, we show how DDDs can be used in the context of queuing nets.

3.1 P/T-Nets with marking-dependent cardinality arcs and capacity places

A *P/T-Net* is a tuple $\langle P, T, Pre, Post, Cap \rangle$ where

- P is a finite set of places,
- T is a finite set of transitions (with $P \cap T = \emptyset$),
- Pre and $Post : P \times T \rightarrow (\mathbb{N}^P \rightarrow \mathbb{N})$ are the marking-dependent pre and post functions labelling the arcs.
- $Cap : P \rightarrow \mathbb{N} \cup \{\infty\}$ defines the capacity of each place.

For a transition t , $\bullet t$ (resp. $t\bullet$) denotes the set of places $\{p \in P \mid Pre(p, t) \neq 0\}$ (resp. $\{p \in P \mid Post(p, t) \neq 0\}$).

A marking m is an element of \mathbb{N}^P satisfying $\forall p \in P, m(p) \leq Cap(p)$. A transition t is enabled in a marking m if for each place p , the two conditions

$Pre(p, t)(m) \leq m(p)$ and $m(p) - Pre(p, t)(m) + Post(p, t)(m) \leq Cap(p)$ hold. The firing of a transition t from a marking m leads to a new marking m' defined by $\forall p \in P, m'(p) = m(p) - Pre(p, t)(m) + Post(p, t)(m)$. As usual, this firing rule is extended to sequence of transitions. Moreover, we denote by $Reach(N, m_0)$ the set of markings of the P/T-Net N reachable from the initial marking m_0 .

From this definition of nets, we define the following sub-classes by restricting the functions labelling the arcs as well as the capacities associated to the places. Assume $\alpha^-, \alpha^+ \in \mathbb{N}^{P \times T}$.

A *net without capacity restriction* is a P/T-Net satisfying $\forall p \in P, Cap(p) = \infty$.

A *constant net* (or *capacity net*) is a P/T-Net for which all the functions on the arcs are constant (i.e. $\forall p \in P, \forall t \in T, Pre(p, t) = \alpha^-(p, t)$ and $Post(p, t) = \alpha^+(p, t)$).

An *ordinary net* is a constant net without capacity restriction.

An *inhibitor net* is a net without capacity restriction and for which the functions labelling the arcs satisfy $\forall p \in P, \forall t \in T, Pre(p, t) = \alpha^-(p, t) \vee Pre(p, t) = 2m(p)$ and $Post(p, t) = \alpha^+(p, t)$. If a transition t has an input place p such that $Pre(p, t) = 2m(p)$ then if $m(p) \neq 0$ then t is not enabled in m . Such an arc is called an inhibitor arc.

A *reset net* is a net without capacity restriction and for which the arc valuation functions satisfy $\forall p \in P, \forall t \in T, Pre(p, t) = \alpha^-(p, t) \vee Pre(p, t) = m(p)$ and $Post(p, t) = \alpha^+(p, t)$. If a transition t has an input place p such that $Pre(p, t) = m(p)$ and t is enabled in m then if m' is the marking reached by firing t , we have $m'(p) = 0$. Such an arc is called a reset arc.

Assume $\beta^-, \beta^+ \in \{0, 1\}^{P \times T \times P}$. A *self modifying net* is a net without capacity restriction and for which the arc valuation functions satisfy : $\forall p \in P, \forall t \in T$,

$$Pre(p, t) = \alpha^-(p, t) + \sum_{r \in P} \beta^-(p, t, r)m(r)$$

$$Post(p, t) = \alpha^+(p, t) + \sum_{r \in P} \beta^+(p, t, r)m(r)$$

This last definition is more restrictive than the one in [9] for which the coefficients β are natural numbers.

3.2 DDD and ordinary Petri nets

First, we show how to encode the states of an ordinary net. We use one variable for each place of the system. The domain of place variables is the set of natural numbers. The initial marking for a single place is encoded by:

$$d_p = p \xrightarrow{m_0(p)} 1$$

For a given total order on the places of the net, the DDD encoding the initial marking is the concatenation of DDDs $d_{p_1} \cdots d_{p_n}$.

The symbolic transition relation is defined arc by arc. There exists alternative definitions and the one we chose is not the most efficient. Indeed, it induces that the DDD representing a set of markings must be traversed for each arc adjacent to a same transition. It is clear that the definition of a more complicated homomorphism, taking into account all the input and output places of a transition, would be more efficient. However, the arc by arc definition is modular and well-adapted for the further combination of arcs of different net sub-classes. We adopt it for sake of presentation clarity. Notice that all homomorphisms defined in this section are independent of the chosen order.

Two homomorphisms are defined to deal respectively with the pre (h^-) and post (h^+) conditions. Both are parameterized by the connected place (p) as well as the valuation (v) labelling the arc entering or outing p .

$$h^-(p, v)(e, x) = \begin{cases} e \xrightarrow{x-v} Id & \text{if } e = p \wedge x \geq v \\ 0 & \text{if } e = p \wedge x < v \\ e \xrightarrow{x} h^-(p, v) & \text{otherwise} \end{cases}$$

$$h^-(p, v)(1) = \top$$

$$h^+(p, v)(e, x) = \begin{cases} e \xrightarrow{x+v} Id & \text{if } e = p \\ e \xrightarrow{x} h^+(p, v) & \text{otherwise} \end{cases}$$

$$h^+(p, v)(1) = \top$$

The symbolic relation of a given transition t is given by the definition:

$$h_{Trans}(t) = \bigcirc_{p \in t \bullet} h^+(p, \alpha^+(p, t)) \circ \bigcirc_{p \in \bullet t} h^-(p, \alpha^-(p, t))$$

where the positive coefficients α^+ and α^- are the ones of the Section 3.1. It is important to note that the state coding together with the homomorphisms h_{Trans} ensure that the produced DDDs are well-defined. This algorithm has been evaluated on a set of examples taken from [2].

The table 1 presents the obtained results (on a PowerBook G4, 667Mhz with 512Mo and running MacOS X). For each model is given the value of the parameter, the cardinality of the reachable marking set, the size of the corresponding DDD, the size of the corresponding tree (i.e a DDD without sharing) and the computation time (in second). We can remark that the size of the DDDs remains reasonable for all these results (linear for the dining philosophers) as well as the computation time when the used homomorphisms can be optimized. However, it is clear that this prototype does not compete with the solution presented in [2]. Indeed, in [2], the computation of the reachable marking set for 50 philosophers takes less than 1s.

3.3 Nets with inhibitor arcs, capacity places and reset arcs

The necessary homomorphisms to deal with inhibitor and reset arcs or places with capacities are just adaptations of the previous ones. Indeed, the firing rule of these particular elements implies only local operations. As an example, the following homomorphism is a simple adaptation of h^- to inhibitor arcs.

	N	reached	DDD size	no sharing	time
philosophers	5	1364	127	11108	0.12
	10	1.86×10^6	267	1.52×10^7	0.68
	50	2.23×10^{31}	1387	1.82×10^{32}	24.48
slotted ring	5	53856	350	507762	5.21
	10	8.29×10^9	1281	8.07×10^{10}	136.37
fms	5	2.89×10^6	225	9.97×10^6	0.76
	10	2.5×10^9	580	8.02×10^9	4.05
	20	6.03×10^{12}	1740	1.87×10^{13}	26.09
kanban	5	2.55×10^6	112	8.98×10^6	2.6
	10	1.01×10^9	257	3.29×10^9	34.83

Table 1. Experimentation results for ordinary nets

$$\begin{aligned}
 h^i(p)(e, x) &= \begin{cases} e \xrightarrow{x} Id & \text{if } e = p \wedge x = 0 \\ 0 & \text{if } e = p \wedge x > 0 \\ e \xrightarrow{x} h^i(p) & \text{otherwise} \end{cases} \\
 h^i(p)(1) &= \top
 \end{aligned}$$

Another adaptation is defined to take into account the capacity of the places:

$$\begin{aligned}
 h^c(p, v)(e, x) &= \begin{cases} e \xrightarrow{x} Id & \text{if } e = p \wedge x \leq v \\ 0 & \text{if } e = p \wedge x > v \\ e \xrightarrow{x} h^c(p, v) & \text{otherwise} \end{cases} \\
 h^c(p, v)(1) &= \top
 \end{aligned}$$

In a similar way, h^+ can be adapted to deal with reset arcs.

$$\begin{aligned}
 h^r(p)(e, x) &= \begin{cases} e \xrightarrow{0} Id & \text{if } e = p \\ e \xrightarrow{x} h^r(p) & \text{otherwise} \end{cases} \\
 h^r(p)(1) &= \top
 \end{aligned}$$

The composition of these different homomorphisms allows us to define the symbolic transition relation for nets including inhibitor and reset arcs as well as capacity places. Let t be a transition, its transition relation is:

$$\begin{aligned}
 h_{Trans}(t) &= \bigcirc_{p \in t \bullet \wedge Cap(p) \neq \infty} h^c(p, Cap(p)) \\
 &\quad \circ \bigcirc_{p \in t \bullet} h^+(p, \alpha^+(p, t)) \\
 &\quad \circ \bigcirc_{p \in \bullet t \wedge Pre(p, t) = \alpha^-(p, t)} h^-(p, \alpha^-(p, t))
 \end{aligned}$$

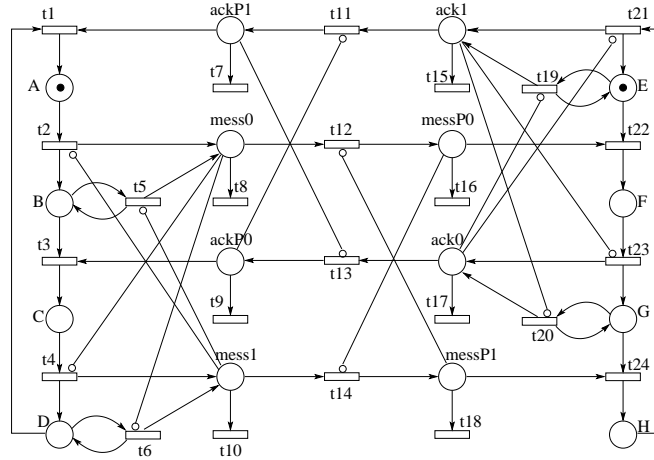


Fig. 2. Inhibitor net (with capacity) of the alternate bit protocol

$$\begin{aligned} \circ & \text{ } \circ_{p \in \bullet t \wedge Pre(p,t)=2m(p)} h^i(p) \\ \circ & \text{ } \circ_{p \in \bullet t \wedge Pre(p,t)=m(p)} h^r(p) \end{aligned}$$

The symbolic transition relation given below has been evaluated on the model of Fig.2 originally presented (and explained) in [3]. The table 2 gives the obtained results. The parameter is here the capacities of the places containing the messages and the acknowledgments. We can remark that the number of DDD nodes is constant. However, the number of arcs (not given in the table) is not.

	N	reached	DDD size	no sharing	time
alternate bit	5	14688	84	43804	1.89
	10	170368	84	480394	11.21
	20	2.23×10^6	84	6.29×10^6	94.66

Table 2. Experimentation results for inhibitor, reset and capacity net

3.4 Self modifying nets

The design of homomorphisms for self modifying nets is more tricky. Indeed, the expression labelling an arc between two nodes can refer any places of the net and then its evaluation is a global operation.

Of course, the adjacent places of the considered transition can be referenced in the function labelling the arcs. Like in previous subsections, the pre condition evaluation as well as the computation of the reached marking are done in several steps (arc by arc). Let m be the marking before the firing of the considered transition t . In order to properly compute the reached marking, for any place p adjacent to t and referenced in an expression used to fire t , the marking $m(p)$ must be stored. Indeed, this value is necessary for the arc by arc computation. To deal with this specific situation, we duplicate such places in the DDD before doing any other operation. The first occurrence of a place is used to store the current value of the marking during the computation whereas the second one represents the value of the marking before the firing. After all the operations are performed, the duplicates are removed. Duplication and removal are expressed in terms of homomorphisms. Due to their weak interest, these homomorphisms are not presented here.

The evaluation of the pre condition of a transition in a self modifying net is based on the following homomorphisms.

$$\begin{aligned} search(p, i)(e, x) &= \begin{cases} e \xrightarrow{x} e \xrightarrow{x} Id & \text{if } e = p \wedge i = 1 \\ up(e, x) \circ search(p, i - 1) & \text{if } e = p \wedge i > 1 \\ up(e, x) \circ search(p, i) & \text{otherwise} \end{cases} \\ search(p, i)(1) &= \top \end{aligned}$$

$search$ is a specialization of the homomorphism $down$ presented in the Section 2.3. It searches for the value of the i^{th} occurrence of a given variable. Indeed, in many cases the value to be subtracted is the one associated to the second occurrence of the variable (the original value of the place marking).

$$\begin{aligned} assign^-(p, v)(e, x) &= \begin{cases} p \xrightarrow{v-x} Id & \text{if } x \leq v \\ 0 & \text{otherwise} \end{cases} \\ assign^-(v)(1) &= \top \end{aligned}$$

$assign^-$ is applied on the variable corresponding to the current value of the marking. Knowing the effective value x of the arc valuation (retrieved by $search$) and the current value v of the marking, it evaluates the pre condition and constructs the new marking if the latter is satisfied.

$$\begin{aligned} setCst^-(p, v)(e, x) &= \begin{cases} e \xrightarrow{x-v} Id & \text{if } e = p \wedge v \leq x \\ 0 & \text{if } e = p \wedge v > x \\ e \xrightarrow{x} setCst^-(p, v) & \text{otherwise} \end{cases} \\ setCst^-(p, v)(1) &= \top \end{aligned}$$

$setCst^-$ is used in the opposite case. The effective value v of the arc valuation is known *a priori*. When the current value of the marking of the place p is found (the first occurrence of the variable), the pre condition is evaluated and the marking modified in case of success.

$$\begin{aligned}
set^-(p, p', i)(e, x) &= \begin{cases} assign^-(p, x) \circ search(p', 2) & \text{if } e = p \wedge p \neq p' \\ assign^-(p, x) \circ search(p', 1) & \text{if } e = p \wedge p = p' \\ p' \xrightarrow{x} setCst^-(p, x) & \text{if } e = p' \wedge p \neq p' \wedge i = 1 \\ p' \xrightarrow{x} set^-(p, p', i - 1) & \text{if } e = p' \wedge p \neq p' \wedge i > 1 \\ e \xrightarrow{x} set^-(p, p', i) & \text{otherwise} \end{cases} \\
set^-(p, p', i)(1) &= \top
\end{aligned}$$

set^- subtracts to the current value of the place p (its first occurrence), the original value of the place p' (its second occurrence). This assignment is performed according to the order between p and p' . The case when p precedes p' is treated by the first statement of the previous definition by combining a search followed by an assignment. The special situation of a reset arc is taken into account by the second statement of the definition. The search is then limited to the next occurrence of the variable. When p' precedes p , the effective value of the arc valuation is then known before visiting p . The value to be subtracted to the current value of p is the value associated to the second occurrence of p' . Then, an index i is used to count how many occurrences have been already visited. As a consequence, we define $h^{sm^-}(p, p')$ as $set^-(p, p', 2)$.

On the other hand, a set of homomorphisms ($assign^+$, $setCst^+$ and set^+) is defined to deal with the post condition. These homomorphisms are symmetric to those presented above. The main difference is there is no condition $x \leq v$ (resp. $v \leq x$) in $assign^+$ (resp. $setCst^-$) and a sum $v + x$ is produced in these homomorphisms. We define $h^{sm^+}(p, p')$ as $set^+(p, p', 2)$.

The symbolic relation for nets containing self modifying arcs is given below. This transition relation can be easily combined with the one of Section 3.3 to deal with a more general class of net. This combined transition relation has been implemented in our prototype.

$$\begin{aligned}
h_{Trans}(t) &= \bigcirc_{p \in \bullet t} (h^+(p, \alpha^+(p, t)) \circ \bigcirc_{r \in P \wedge \beta^+(p, t, r)=1} h^{sm^+}(p, r)) \\
&\quad \circ \bigcirc_{p \in \bullet t} (h^-(p, \alpha^-(p, t)) \circ \bigcirc_{r \in P \wedge \beta^-(p, t, r)=1} h^{sm^-}(p, r))
\end{aligned}$$

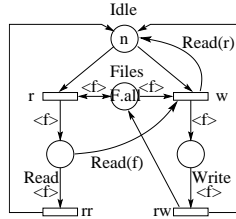


Fig. 3. Self modifying net of readers and preemptive writers

The previous transition relation has been evaluated on the example of Fig. 3 originally presented in [9]. The parameters are the numbers of processes and of

files. The results are presented in Tab. 3. We remark that the size of the DDD is proportionally linear to the value of the parameters. In the case of finite systems (i.e. nets with a finite number of reachable markings), it is well-known that self modifying nets have the same expressive power as ordinary nets. The net of Fig. 3 can then be imitated by an equivalent ordinary net where the transition w is duplicated to take into account the different situations (the different possible number of readers of a given file). Notice that some complementary places must be introduced to test the strict equality and then that the place bounds must be known *a priori*. The last column of Tab. 3 indicates the time needed to treat the equivalent ordinary net. We remark that the concision of the self modifying net allows to take advantage on this alternative.

	N	reached	DDD size	no sharing	time	ord. time
preemptive writers	5×5	873	120	6440	0.62	0.72
	10×10	1.94×10^6	485	1.46×10^7	18.35	24.35
	15×15	4.95×10^9	1100	3.77×10^{10}	137.5	211.9

Table 3. Experimentation results for self modifying net

3.5 Queuing nets

In this section, we present how DDD can be used to verify ordinary net enriched by lossy queues. In our model, a loss can occur at any position in the queue. Notice that this model cannot be simulated by an ordinary net without introducing intermediary markings.

A *queuing net* is a tuple $\langle \Sigma, P, Q, Q_{loss}, T, Pre_P, Post_P, Pre_Q, Post_Q \rangle$ where:

- Σ is a finite alphabet,
- $\langle P, T, Pre_P, Post_P \rangle$ forms an ordinary net,
- Q is a finite set of queues,
- $Q_{loss} \subseteq Q$ is the set of lossy queues,
- Pre_Q and $Post_Q : Q \times T \rightarrow \Sigma \cup \{\epsilon\}$ are the pre and post conditions of the queues.

For a transition t , we denote by ${}^{\triangleright}t$ (resp. t^{\triangleright}), the set of queues $\{q \in Q \mid Pre_Q(q, t) \neq \epsilon\}$ (resp. $\{q \in Q \mid Post_Q(q, t) \neq \epsilon\}$).

A marking m is an element of $\mathbb{N}^P \times (\Sigma^*)^Q$. A transition t is enabled in a marking m if for each place p , the condition $Pre_P(p, t) \leq m(p)$ holds and if for each queue q , there exists a word $\omega \in \Sigma^*$ such that $m(q) = Pre_Q(q, t) \cdot \omega$. The firing of t from m leads to a set of new markings constructed from the marking m' defined by $\forall p \in P, m'(p) = m(p) - Pre_P(p, t) + Post_P(p, t)$ and $\forall q \in Q$, if $m(q) = Pre_Q(q, t) \cdot \omega$ then $m'(q) = \omega \cdot Post_Q(q, t)$. A marking reached from m by

firing t is any marking obtained by erasing any number of letters of the words $m'(q)$ for the lossy queue $q \in Q_{loss}$.

The encoding of the states of a queuing net is obtained by generalizing the one used for P/T-nets. We use one variable for each place and each queue of the system. The domain of place variables is the set of natural numbers, while the domain of queue variables is $\Sigma \cup \{\#\}$, the set of messages that may contain a queue and a terminal character “#” (we assume that $\# \notin \Sigma$). The initial marking for a single place and a single queue is encoded by:

$$d_r = \begin{cases} r \xrightarrow{x} 1 & \text{if } r \in P, m_0(r) = x \\ r \xrightarrow{a_0} r \xrightarrow{a_1} r \dots \xrightarrow{a_n} r \xrightarrow{\#} 1 & \text{if } r \in Q, m_0(r) = a_0 \cdot a_1 \dots a_n \end{cases}$$

For a given total order $r_1, r_2 \dots r_n$ on $P \cup Q$, the DDD encoding the initial marking is the concatenation of DDDs $d_{r_1} \dots d_{r_n}$.

We are now in position to define the homomorphisms used for the arcs related to the queues.

$$h^{q^-}(q, v)(e, x) = \begin{cases} Id & \text{if } e = q \wedge x = v \\ 0 & \text{if } e = q \wedge x \neq v \\ e \xrightarrow{x} h^{q^-}(q, v) & \text{otherwise} \end{cases}$$

$$h^{q^-}(q, v)(1) = \top$$

h^{q^-} tests that the first occurrence of the variable q is associated with the value v and in this case, it removes this occurrence.

$$h^{q^+}(q, v)(e, x) = \begin{cases} e \xrightarrow{v} e \xrightarrow{\#} Id & \text{if } e = q \wedge x = \# \\ e \xrightarrow{x} h^{q^+}(q, v) & \text{otherwise} \end{cases}$$

$$h^{q^+}(q, v)(1) = \top$$

h^{q^+} searches the last occurrence of the variable q (the one associated with the value $\#$) and then introduces the value v before the terminal character at the end of the queue.

$$h^l(q)(e, x) = \begin{cases} h^l(q) + e \xrightarrow{x} h^l(q) & \text{if } e = q \wedge x \neq \# \\ e \xrightarrow{x} Id & \text{if } e = q \wedge x = \# \\ e \xrightarrow{x} h^l(q) & \text{otherwise} \end{cases}$$

$$h^l(q)(1) = \top$$

h^l is defined to deal with lossy queues. Its role is to produce all the markings obtained by erasing any combination of letters in the word associated to the queue q .

The symbolic relation of a given transition t is given below. Notice that after the firing of a transition, all the markings reached by losing some messages are produced by applying h^l . In particular, the homomorphism $\bigcirc_{q \in Q_{lost}} h^l(q)$ must be applied to the initial marking.

$$\begin{aligned}
h_{Trans}(t) = & \bigcirc_{q \in Q_{lost}} h^l(q) \\
& \circ \bigcirc_{q \in t^>} h^{q^+}(q, Post_Q(q, t)) \circ \bigcirc_{p \in t \bullet} h^+(p, \alpha^+(p, t)) \\
& \circ \bigcirc_{q \in > t} h^{q^-}(q, Pre_Q(q, t)) \circ \bigcirc_{p \in \bullet t} h^-(p, \alpha^-(p, t))
\end{aligned}$$

The previous transition relation has been evaluated on a model of the alternate bit protocol with bounded lossy queues. In our prototype, we have adapted the homomorphism h^c to deal directly with the queue capacities. The experimental results are presented in Tab. 4 and the parameter is the capacity of the queues. Here again, the size of the DDDs is linearly proportional to the value of N .

	N	reached	DDD size	no sharing	time
alternate bit	5	630	104	2008	0.21
	10	3355	164	10303	0.53
	20	21105	284	63943	1.57
	50	280755	644	845263	8.46
	100	2.12×10^6	1244	6.37×10^6	33.4

Table 4. Experimentation results for queuing net

4 Concluding Remarks

DDD provides a sensible alternative to other decision diagrams and the absence of hypotheses on the variables and their domains gives an important flexibility for the coding of the states (allowing the representation of dynamic data structures or the taking into account of data without *a priori* knowing their domains). The inductive homomorphisms permits the definition of a large class of operators that are specific to a given application domain. While this expressiveness comes necessarily at a cost, our experience shows that intelligent use of advanced techniques such as maximal sharing makes it possible to provide reasonable performance. Moreover, inductive homomorphisms could be adapted to other decision diagrams.

The application to the analysis of different classes of Petri nets presented in this paper has demonstrated the expressiveness of inductive homomorphisms. Indeed, most of the development cost has been spent for the DDD library while the Petri net analyser has been coded in less than one day. We have also shown how this analyser can be completed and extended to provide a CTL model checker.

However, the performance offered by this prototype are not completely satisfactory even if many ways of optimization are possible. At first, DDDs are very sensitive to the variable ordering (like other decision diagrams) and no reordering techniques have been implemented yet. On the other hand, we can remark that many of the operations are local to a variable. Ciardo et al. in [7, 2] have proposed a hierarchical structure to reach directly the block of data where the affected variable is located. DDDs can take advantage of such accelerators. Finally, the symbolic analysis of homomorphism compositions used for a given application and their reordering can also be a way to optimize their evaluation.

In the context of a semi-industrial project, the expressiveness of inductive homomorphisms has been put to the test. We have developed a symbolic model checker for circuits modelled by VHDL programs. A large subset of the language, including discrete timeouts, is covered. These experiments encourage us to integrate temporal aspects in our Petri net model.

References

1. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
2. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *Proc. of ICATPN'2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 103–122. Springer Verlag, 2000.
3. J.M. Couvreur and E. Paviot-Adet. New structural invariants for Petri nets analysis. In *Proc. of ICATPN'94*, volume 815 of *Lecture Notes in Computer Science*, pages 199–218. Springer Verlag, 1994.
4. H. Hulgaard, P. F. Williams, and H. R. Andersen. Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions of Computer-Aided Design*, 18(7), July 1999.
5. T. Kolks, B. Lin, and H. De Man. Sizing and verification of communication buffers for communicating processes. In *Proc. of IEEE International Conference on Computer-Aided Design*, volume 1825, pages 660–664, Santa Clara, USA, November 1993.
6. S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagrams with attributed edges for efficient boolean function manipulation. In L.J.M Claesen, editor, *Proceedings of the 27th ACM/IEEE Design Automation Conference, DAC'90*, pages 52–57, June 1990.
7. A.S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. of ICATPN'99*, volume 1639 of *Lecture Notes in Computer Science*, pages 6–25. Springer Verlag, 1999.
8. E. Pastor, O. Roig, J. Cortadella, and R.M. Badia. Petri net analysis using boolean manipulation. In *Proc. of ICATPN'94*, volume 815 of *Lecture Notes in Computer Science*, pages 416–435. Springer Verlag, 1994.
9. R. Valk. Bridging the gap between place- and floyd-invariants with applications to preemptive scheduling. In *Proc. of ICATPN'93*, volume 691 of *Lecture Notes in Computer Science*, pages 432–452. Springer Verlag, 1993.