Hierarchical Set Decision Diagrams and Automatic Saturation[⋆]

Alexandre Hamez[1,2], Yann Thierry-Mieg[1], and Fabrice Kordon[1]

[1] Université P. & M. Curie
LIP6 - CNRS UMR 7606
4 Place Jussieu, 75252 Paris cedex 05, France

[2] EPITA
Research and Development Laboratory
F-94276 Le Kremlin-Bicetre cedex, France

Alexandre.Hamez@lip6.fr, Yann.Thierry-Mieg@lip6.fr, Fabrice.Kordon@lip6.fr

Abstract. Shared decision diagram representations of a state-space have been shown to provide efficient solutions for model-checking of large systems. However, decision diagram manipulation is tricky, as the construction procedure is liable to produce intractable intermediate structures (a.k.a peak effect). The definition of the so-called saturation method has empirically been shown to mostly avoid this peak effect, and allows verification of much larger systems. However, applying this algorithm currently requires deep knowledge of the decision diagram data-structures, of the model or formalism manipulated, and a level of interaction that is not offered by the API of public DD packages.
Hierarchical Set Decision Diagrams (SDD) are decision diagrams in which arcs of the structure are labeled with sets, themselves stored as SDD. This data structure offers an elegant and very efficient way of encoding structured specifications using decision diagram technology. It also offers, through the concept of inductive homomorphisms, unprecedented freedom to the user when defining the transition relation. Finally, with very limited user input, the SDD library is able to optimize evaluation of a transition relation to produce a saturation effect at runtime. We further show that using recursive folding, SDD are able to offer solutions in logarithmic complexity with respect to other DD. We conclude with some performances on well known examples.

Keywords Hierarchical Decision Diagrams, Model Checking, Saturation

1 Introduction

Parallel systems are notably difficult to verify due to their complexity. Non-determinism of the interleaving of elementary actions in particular is a source of errors difficult to detect through testing. Model-checking of finite systems or exhaustive exploration of the state-space is very simple in its principle, entirely automatic, and provides useful counter-examples when the desired property is not verified.

However model-checking suffers from the combinatorial state-space explosion problem, that severely limits the size of systems that can be checked automatically. One

[⋆] This work has been partially supported by the ModelPlex European integrated project FP6-IP 034081 (Modeling Solutions for Complex Systems)

solution which has shown its strength to tackle very large state spaces is the use of shared decision diagrams like BDD [1,2].

But decision diagram technology also suffers from two main drawbacks. First, the order of variables has a huge impact on performance and defining an appropriate order is non-trivial [3]. Second, the way the transition relation is defined and applied may have a huge impact on performance [4,5].

The objective of this paper is to present novel optimization techniques for hierarchical decision diagrams called Set Decision Diagrams (SDD), suitable to master the complexity of very large systems. Although SDD are a general all-purpose compact data-structure, a design goal has been to provide easy to use off the shelf constructs (such as a fixpoint) to develop a model-checker using SDD. These constructs allow the library to control operation application, and harness the power of state of the art saturation algorithms [5] with low user expertise in DD.

No specific hypothesis is made on the input language, although we focus here on a system described as a composition of labeled transition systems. This simple formalism captures most transition-based representations (such as automata, communicating processes like in Promela [6], Harel state charts, or bounded Petri nets).

Our hierarchical Set Decision Diagrams (section 2) offer the following capabilities:

- Using the structure of a specification to introduce hierarchy in the state space, it enables more possibilities for exploiting pattern similarities in the system (section 3),
- Automatic activation of saturation ; the algorithms described in this paper allow the library to enact saturation with minimal user input (sections 4 and 5),
- A *recursive* folding technique that is suitable for very symmetric systems (section 7).

We also show that our openly distributed implementation: *libDDD* [7], is efficient in terms of memory consumption and enables the verification of bigger state spaces.

2 Definitions

We define in this section Data Decision Diagrams (based on [8]) and Set Decision Diagrams (based on [9]).

2.1 Data Decision Diagrams

Data Decision Diagrams (DDD) [8] are a data structure for representing finite sets of assignments sequences of the form $(e_1 := x_1) \cdot (e_2 := x_2) \cdots (e_n := x_n)$ where e_i are variables and x_i are the assigned integer values. When an ordering on the variables is fixed and the values are booleans, DDD coincides with the well-known Binary Decision Diagrams. When the ordering on the variables is the only assumption, DDD correspond closely to Multi-valued Decision Diagrams (MDD)[5].

However DDD assume no variable ordering and, even more, the same variable may occur many times in a same assignment sequence. Moreover, variables are not assumed to be part of all paths. Therefore, the maximal length of a sequence is not fixed, and sequences of different lengths can coexist in a DDD. This feature is very useful when dealing with dynamic structures like queues.

DDD have two terminals : as usual for decision diagram, 1-leaves stand for accepting terminators and 0-leaves for non-accepting ones. Since there is no assumption on the variable domains, the non-accepted sequences are suppressed from the structure. 0 is considered as the default value and is only used to denote the empty set of sequences. This characteristic of DDD is important as it allows the use of variables of finite domain with *a priori* unknown bounds. In the following, E denotes a set of variables, and for any e in E, $\mathrm{Dom}(e) \subseteq \mathbb{N}$ represents the domain of e.

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

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

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

Although no ordering constraints are given, to ensure existence of a canonic representation, DDD only represent sets of *compatible DDD sequences*. Note that the DDD 0 represents the empty set and is therefore compatible with any DDD sequence. The *symmetric compatibility relation* $\approx$ is defined inductively for two DDD sequences:

Definition 2 (Compatible DDD sequences). *We call DDD sequence a DDD of the form* $e_1 \xrightarrow{x_1} e_2 \xrightarrow{x_2} \dots 1$*. Let* s_1*,* s_2 *be two sequences,* s_1 *is* compatible *with* s_2*, noted* $s_1 \approx s_2$ *iff.:*

- $s_1 = 1 \wedge s_2 = 1$ *or*
- $s_1 = e \xrightarrow{x} d \wedge s_2 = e' \xrightarrow{x'} d'$ *such that* $\begin{cases} e = e' \wedge \\ x = x' \Rightarrow d \approx d' \end{cases}$

As usual, DDD are encoded as (shared) decision trees (see Fig. 1 for an example). Hence, a DDD of the form $\langle e, \alpha \rangle$ is encoded by a node labeled e and for each $x \in Dom(e)$ such that $\alpha(x) \neq 0$, there is an arc from this node to the root of $\alpha(x)$. By the definition 1, from a node $\langle e, \alpha \rangle$ there can be at most one arc labeled by $x \in \mathrm{Dom}(e)$ and leading to $\alpha(x)$. This may cause conflicts when computing the union of two DDD, if the sequences they contain are incompatible, so care must be taken on the operations performed.

DDD are equipped with the classical set-theoretic operations (union, intersection, set difference). They also offer a concatenation operation $d_1 \cdot d_2$ which replaces 1 terminals of d_1 by d_2. It corresponds to a cartesian product. In addition, homomorphisms are defined to allow flexibility in the definition of application specific operations.

A basic homomorphism is a mapping Φ from $\mathbb{D}$ to $\mathbb{D}$ such that $\Phi(0) = 0$ and $\Phi(d + d') = \Phi(d) + \Phi(d'), \forall d, d' \in \mathbb{D}$. The sum $+$ and the composition $\circ$ of two homomorphisms are homomorphisms. Some basic homomorphisms are hard-coded. For instance, the homomorphism $d * Id$ where $d \in \mathbb{D}$, $*$ stands for the intersection and Id for the identity, allows to select the sequences belonging to d : it is a homomorphism that can be applied to any d' yielding $d * Id(d') = d * d'$. The homomorphisms $d \cdot Id$ and $Id \cdot d$ permit to left or right concatenate sequences. We widely use the left concatenation that adds a single assignment $(e := x)$, noted $e \xrightarrow{x} Id$.

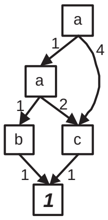


Fig. 1: DDD for $a \xrightarrow{1} a \xrightarrow{1} b \xrightarrow{1} 1$ $+ a \xrightarrow{4} c \xrightarrow{1} 1$ $+ a \xrightarrow{1} a \xrightarrow{2} c \xrightarrow{1} 1$

We also have a **transitive closure** * unary operator that allows to perform a fixpoint computation. For any homomorphism h, $h^\star(d)$, $d \in \mathbb{D}$ is evaluated by repeating $d \leftarrow h(d)$ until a fixpoint is reached. In other words, $h^\star(d) = h^n(d)$ where n is the smallest integer such that $h^n(d) = h^{n-1}(d)$. This computation may not terminate (e.g. h increments a variable). However, if it does, then $h^\star = h^n$ with n finite. Thus, $h^\star$ is itself an inductive homomorphism. This operator is usually applied to $Id + h$ instead of h, allowing to cumulate newly reached paths in the result.

Furthermore, application-specific mappings can be defined by *inductive* homomorphisms. An inductive homomorphism Φ is defined by its evaluation on the 1 terminal $\Phi(1) \in \mathbb{D}$, and its evaluation $\Phi' = \Phi(e, x)$ for any $e \in E$ and any $x \in \mathrm{Dom}(e)$. Φ' is itself a (possibly inductive) homomorphism, that will be applied on the successor node d. The result of $\Phi(\langle e, \alpha \rangle)$ is then defined as $\sum_{(x,d) \in \alpha} \Phi(e, x)(d)$, where $\sum$ represents a union. We give examples of inductive homomorphisms in section 3 which introduces a simple labeled P/T net formalism.

2.2 Set Decision Diagrams

Set Decision Diagrams (SDD) [9], are shared decision diagrams in which arcs of the structure are labeled by a *set* of values, instead of a single valuation. This set may itself be represented by an SDD or DDD, thus when labels are SDD, we think of them as hierarchical decision diagrams. This section presents the definition of SDD, which has been modified from [9] for more clarity (although it is identical in effects).

Set Decision Diagrams (SDD) are data structures for representing sequences of assignments of the form $e_1 \in a_1; e_2 \in a_2; \cdots e_n \in a_n$ where e_i are variables and a_i are sets of values.

SDD can also be seen as a different representation of the DDD defined as: $\bigcup_{x_1 \in a_1} \cdots \bigcup_{x_n \in a_n} e_1 \xrightarrow{x_1} \cdots e_n \xrightarrow{x_n} 1$, however since a_i are not required to be finite, SDD are more expressive than DDD.

We assume no variable ordering, and the same variable can occur several times in an assignment sequence. We define the usual terminal 1 to represent accepting sequences. The terminal 0 is also introduced and represents the empty set of assignment sequences. In the following, E denotes a set of variables, and for any e in E, $\mathrm{Dom}(e)$ represents the domain of *e which may be infinite*.

Definition 3 (Set Decision Diagram). *The set $\mathbb{S}$ of SDD is inductively defined by $s \in \mathbb{S}$ if:*

- $s \in \{0, 1\}$ *or*
- $s = \langle e, \pi, \alpha \rangle$ *with:*

- *$e \in E$.*
- *$\pi = \{a_0, \ldots, a_n\}$ is a finite partition of $\mathrm{Dom}(e)$, i.e. $\mathrm{Dom}(e) = a_0 \uplus \cdots \uplus a_n$ where $\uplus$ is the disjunctive union. We further assume $\forall i, a_i \neq \emptyset$, and n finite.*
- *$\alpha : \pi \to \mathbb{S}$, such that $\forall i \neq j, \alpha(a_i) \neq \alpha(a_j)$.*

We will simply note $s = \langle e, \alpha \rangle$ the node $\langle e, \pi, \alpha \rangle$ as α implicitly defines π. We denote $e \xrightarrow{a} d$, the SDD (e, α) with $\alpha(a) = d, \alpha(\mathrm{Dom}(e) \setminus a) = 0$. By convention, when it exists, the element of the partition π that maps to the SDD 0 is not represented.

SDD are canonized by construction through the union operator. This definition ensures canonicity of SDD, as π is a partition and that no two arcs from a node may lead to the same SDD. Therefore any value x of $\mathrm{Dom}(e)$ is represented on at most one arc, and any time we are about to construct $e \xrightarrow{a} d + e \xrightarrow{a'} d$, we will construct an arc $e \xrightarrow{a \cup a'} d$ instead. This ensures that any set of assignment sequences has a unique SDD representation.

The finite size of the partition π ensures we can store α as a finite set of pairs (a_i, d_i), and let π be implictly defined by α.

Although simple, this definition allows to construct rich and complex data :

- The definition supports domains of infinite size (e.g. $\mathrm{Dom}(e) = \mathbb{R}$), provided that the partition size remains finite (e.g. $]0..3],]3..+\infty]$). This feature could be used to model clocks for instance (as in [10]).
- $\mathbb{S}$ or $\mathbb{D}$ can be used as the domain of variables, introducing hierarchy in the data structure. In the rest of the paper we will focus on this use case, and consider that the SDD variables we manipulate are exclusively of domain $\mathbb{S}$ or $\mathbb{D}$.

Like DDD, to handle paths of variable lengths, SDD are required to represent a set of compatible assignment sequences. An operation over SDD is said partially defined if it may produce incompatible sequences in the result.

Definition 4 (Compatible SDD sequences). *An SDD* sequence *is an SDD of the form $e_0 \xrightarrow{a_0} \cdots e_n \xrightarrow{a_n} 1$. Let s_1, s_2 be two sequences, $s_1 \approx s_2$ iff.:*

- $s_1 = 1 \wedge s_2 = 1$
- $s_1 = e \xrightarrow{a} d \wedge s_2 = e' \xrightarrow{a'} d'$ *such that* $\begin{cases} e = e' \\ \wedge a \approx a' \\ \wedge a \cap a' \neq \emptyset \Rightarrow d \approx d' \end{cases}$

Compatibility is a symmetric property. The $a \approx a'$ is defined as SDD compatibility if $a, a' \in \mathbb{S}$ or DDD compatibility if $a, a' \in \mathbb{D}$. DDD and SDD are incompatible. Other possible referenced types should define their own notion of compatibility.

2.3 SDD Operations

SDD support standard set theoretic operations (union, intersection, set difference) for compatible SDD. Like DDD they also support concatenation, as well as a variant of inductive homomorphisms. Some built-in basic homomorphisms (e.g. $d * Id$) are also provided similarly to DDD.

To define a family of inductive homomorphisms Φ, one has just to set the homomorphisms for the symbolic expression $\Phi(e, x)$ for any variable e and set $x \subseteq \mathrm{Dom}(e)$ and the SDD $\Phi(1)$. The application of an inductive homomorphism Φ to a node $s = \langle e, \alpha \rangle$ is then obtained by $\Phi(s) = \sum_{(x,d)\in\alpha} \Phi(e, x)(d)$.

It should be noted that this definition differs from the DDD inductive homomorphism in that $\Phi(e, x)$ is defined over the *sets* $x \subseteq \mathrm{Dom}(e)$. This is a fundamental difference as it requires Φ to be defined in an ensemblist way: we cannot define the evaluation of Φ over a single value of e. However Φ must be defined for the *set* containing any single value. If the user only defined $\Phi(e, x)$ with $x \in \mathrm{Dom}(e)$, since the a_i may be infinite, evaluation could be impossible. Even when $\mathrm{Dom}(e) = \mathbb{D}$, a element-wise definition would force to use an explicit evaluation mechanism, which is not viable when a_i is large (e.g. $|a_i| > 10^7$).

Furthermore, let Φ_1, Φ_2 be two homomorphisms. Then $\Phi_1 + \Phi_2$, $\Phi_1 \circ \Phi_2$ and $\Phi_1^\star$ (transitive closure) are homomorphisms.

We also now define a *local* construction, as an inductive homomorphism. Let $var \in E$ designate a target variable, and h be a SDD or DDD homomorphism (depending on $\mathrm{Dom}(var)$) that can be applied to any $x \subseteq \mathrm{Dom}(var)$,

$$\begin{array}{l} local(h, var)(e, x) = \\ \begin{cases} e \xrightarrow{h(x)} Id & \text{if } e = var \\ e \xrightarrow{x} local(h, var) & \text{otherwise} \end{cases} \\ local(h, var)(1) = 0 \end{array}$$

This construction is built-in, and gives a lot of structural information on the operation. As we will see in section 5, specific rewriting rules will allow to optimize evaluation of *local* constructions.

3 Model Checking with Set Decision Diagrams

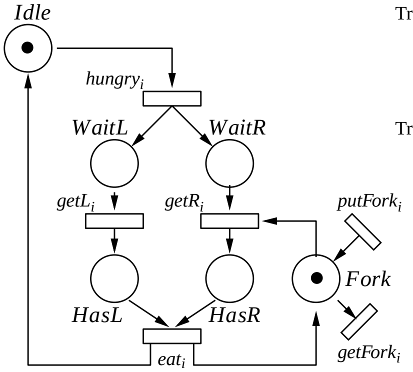
To build a model checker for a given formalism using SDD, one needs to perform the following steps:
1. Define the formalism,
2. Define a representation of states,
3. Define a transition relation using homomorphisms,
4. Define a verification goal.

We exhibit these steps in this section using a simple formalism, labeled P/T nets. Most of what is presented here is valid for other LTS.

1. Defining the Formalism A unitary *Labeled P/T-Net* is a tuple $\langle P, T, Pre, Post, L, label, m_0 \rangle$ where

- P is a finite set of places,

- T is a finite set of transitions (with $P \cap T = \emptyset$),
- Pre and $Post : P \times T \rightarrow \mathbb{N}$ are the pre and post functions labeling the arcs.
- L is a set of labels
- $label : T \rightarrow 2^L$ is a function labeling the transitions
- $m_0 \in \mathbb{N}^P$ is the initial marking of the net.



Transition eat_i is synchronized with $putFork_{i+1modN}$, thus the labels

$$label(putFork_i) = \{\mathcal{R}_{i+1modN}\}$$
$$label(eat_i) = \{\mathcal{R}_i\}$$

Transition $getL_i$ is synchronized with $getFork_{i+1modN}$, thus the labels

$$label(getFork_i) = \{\mathcal{L}_{i+1modN}\}$$
$$label(getL_i) = \{\mathcal{L}_i\}$$

Fig. 2: Labeled P/T net P_i of i^{th} philosopher in the N dining philosophers problem

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 $\mathbb{N}^P$. A transition t is enabled in a marking m if for each place p, the condition $Pre(p,t) \leq m(p)$ holds. The firing of an enabled transition t from a marking m leads to a new marking m' defined by $\forall p \in P, m'(p) = m(p) - Pre(p,t) + Post(p,t)$.

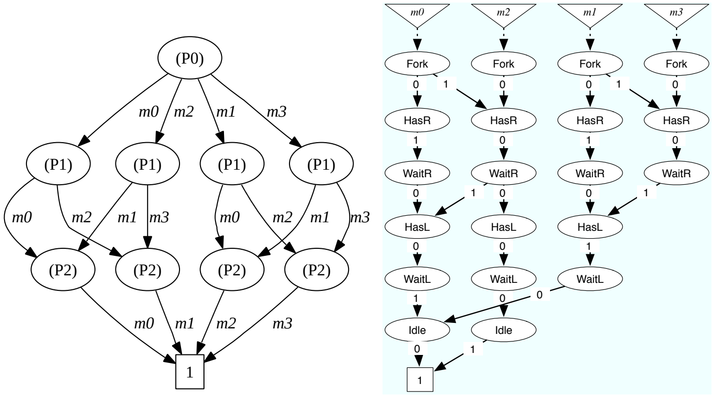
Labeled P/T nets may be composed by synchronization on the transitions that bear the same label. This is a parallel composition noted $/\!\!/$, with event-based synchronizations that can be interpreted as yielding a new (composite) labeled P/T net. Let $M = M_0 /\!\!/ \cdots /\!\!/ M_n$ be such a composite labeled Petri net. Each label of M common to the composed nets M_i gives rise to a *synchronization transition* t_l of M.

Let $\tau_l = \{t_i | t_i \in M_i.T \wedge l \in M_i.label(t_i)\}$ represent *parts of the synchronization*, i.e. the set of transitions that bear this label in the subnets M_i. t_l is enabled iff. $\forall t_i \in \tau_l$, t_i is enabled. The effect firing of t_l is obtained by firing all the parts $t_i \in \tau_l$. In the rest of this paper, we will call *labeled Petri net* a unitary or composite net.

Figure 2 presents an example labeled Petri net, for the classical dining philosophers problem. The composite net $P_0 /\!\!/ P_1$ synchronizes transition $P_0.eat$ with $P_1.putFork$ through label $\mathcal{R}_0$ for instance. This transition corresponds to philosopher P_0 synchronously eating and returning philosopher P_1 his fork.

This is a general compositional framework, adapted to the composition of arbitrary labeled transition systems (LTS).

2. Defining the State Representation Let us consider a representation of a state space of a unitary P/T net in which we use one DDD variable for each place of the system. The



(a) SDD: $(P_0) \mathbin{/\!\!/} (P_1) \mathbin{/\!\!/} (P_2)$

(b) DDD: P_i unitary net

Fig. 3: Hierarchical encoding of the full state-space for 3 philosophers

domain of place variables is the set of natural numbers. The initial marking for a single place is represented by: $d_p = p \xrightarrow{m_0(p)} 1$. For a given total order on the places of the net, the DDD representing the initial marking is the concatenation of DDD $d_{p_1} \cdots d_{p_n}$. For instance, the initial state of a philosopher can be represented by : $Fork \xrightarrow{1} HasR \xrightarrow{0} WaitR \xrightarrow{0} HasL \xrightarrow{0} WaitL \xrightarrow{0} Idle \xrightarrow{1} 1$.

To introduce structure in the representation, we introduce the role of parenthesis in the definition of a composite net. We will thus exploit the fact the model is defined as a composition of (relatively independent) parts in our encoding. If we disregard any parenthesizing of the composition we obtain an equivalent "flat" composite net, however using different parenthesizing(s) yields a more hierarchical vision (nested submodules), that can be accurately represented and exploited in our framework.

Definition 5 (Structured state representation). *Let M be a labeled P/T net, we inductively define its initial state representation $r(M)$ by :*

- *If M is a unitary net, we use the encoding $r(M) = d_{p_1} \cdots d_{p_n}$, with $d_p = p \xrightarrow{m_0(p)} 1$.*
- *If $M = M_1 \mathbin{/\!\!/} M_2$, $r(M) = r(M_1) \cdot r(M_2)$. Thus the parallel composition of two nets will give rise to the concatenation of their representations.*
- *If $M = (M_1)$, $r(M) = m_{(M_1)} \xrightarrow{r(M_1)} 1$, where $m_{(M_1)}$ is an SDD variable. Thus parenthesizing an expression gives rise to a new level of hierarchy in the representation.*

A state is thus encoded hierarchically in accordance with the parenthesized composition definition. If we disregard parenthesizing, we obtain a flat representation using only DDD. We use in our benchmark set many models taken from literature that are defined using "modules", that is a net $N = (M_1) \mathbin{/\!\!/} \cdots \mathbin{/\!\!/} (M_n)$ where each M_i is a unitary net called a module (yielding a single level of hierarchy in the SDD). Figure 3 shows an example of this type of encoding, where figure 3(a) is an SDD representing the full composite net, and labels of the SDD arcs refer to DDD nodes of figure 3(b).

3. Defining the Transition encoding The symbolic transition relation is defined arc by arc in a modular way well-adapted to the further combination of arcs of different net sub-classes (e.g. inhibitor arcs, reset arcs, capacity places, queues…). Homomorphisms allowing to represent these extensions were previously defined in [8], and are not presented here for sake of simplicity. The two following homomorphisms are defined to deal respectively with the pre (noted h^-) and post (noted h^+) conditions. Both are parameterized by the connected place (p) as well as the valuation (v) labeling the arc entering or outing p .

$$h^-(p,v)(e,x) = \begin{cases} e \xrightarrow{x-v} Id & \text{if } e = p \land x \geq v \\ 0 & \text{if } e = p \land x < v \\ e \xrightarrow{x} h^-(p,v) & \text{otherwise} \end{cases}$$
$$h^-(p,v)(1) = 0$$

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

These basic homomorphisms are composed to form a transition relation. We use $\bigcirc_{h\in H}$ to denote the composition by $\circ$ of the elements h in the set H.

Definition 6 (Inductive homomorphism transition representation). *Let t be a transition of labeled P/T net M. We inductively define its representation as a homomorphisms $h_{Trans}(t)$ by :*

- *If M is a unitary net, we use the encoding*

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

- *If $M = (M_1) /\!\!/ \cdots /\!\!/ (M_n)$, and t represents a synchronization of transitions on a label $l \in L$. The homomorphism representing t is written :*

$$h_{Trans}(t) = \bigcirc_{t_i\in\tau_l} local(h_{Trans}(t_i), m_{(M_i)})$$

For instance the transition $hungry_i$ in the model of Fig. 2, would have as homomorphism : $h_{Trans}(hungry) = h^+(WaitL,1) \circ h^+(WaitR,1) \circ h^-(Idle,1)$. When on a path a precondition is unsatisfied, the h^- homomorphism will return 0, pruning the path from the structure. Thus the h^+ are only applied on the paths such that all preconditions are satisfied.

To handle synchronization of transitions bearing the same label in different nets of a compositional net definition we use the local application construction of SDD homomorphisms. The fact that this definition as a composition of local actions is possible stems from the simple nature of the synchronization schema considered. A transition relation that is decomposable under this form has been called Kronecker-consistent in various papers on MDD by Ciardo et al like [5].

For instance, let us consider the dining philosophers example for $N = 3$, $M = (P_0) /\!\!/ (P_1) /\!\!/ (P_2)$. The transition $t_{\mathcal{R}_0}$ is written :

$$\begin{aligned} h_{Trans}(t_{\mathcal{R}_0}) &= local(h_{Trans}(eat), m_{(P_0)}) \\ &\quad \circ local(h_{Trans}(putFork), m_{(P_1)}) \\ &= local(h^+(Idle,1) \circ h^+(Fork,1) \circ h^-(HasL,1) \circ h^-(HasR,1), m_{(P_0)}) \\ &\quad \circ local(h^+(Fork,1), m_{(P_1)}) \end{aligned}$$

4. Defining the Verification Goal The last task remaining is to define a set of target (usually undesired) states, and check whether they are reachable, which involves generating the set of reachable states using a *fixpoint* over the transition relation. The user is then free to define a selection inductive homomorphism that only keeps states that verify an atomic property. This is quite simple, using homomorphisms similar to the pre condition (h^-) that do not modify the states they are applied to. Any boolean combination of atomic properties is easily expressed using union, intersection and set difference.

A more complex CTL logic model-checker can then be constructed using nested *fixpoint constructions* over the transition relation or its reverse [2]. Algorithms to produce witness (counter-example) traces also exist [11] and can be implemented using SDD.

4 Transitive Closure : State of the Art

The previous section has allowed us to obtain an encoding of states using SDD and of transitions using homomorphisms. We have concluded with the importance of having an efficient algorithm to obtain the transitive closure or fixpoint of the transition relation over a set of (initial) states, as this procedure is central to the model-checking problem.

Such a transitive closure can be obtained using various algorithms, some of which are presented in algorithm 1. Variant *a* is a naive algorithm, *b* [2] and *c* [4] are algorithms from the literature. Variant *d*, together with automatic optimizations, is our contribution and will be presented in the next section.

Symbolic transitive closure ('91)[2] Variation *a* is adaptated from the natural way of writing a fixpoint with explicit data structures: it uses a set *todo* exclusively containing unexplored states. Notice the slight notation abuse: we note $T(todo)$ when we should note $(\sum_{t \in T} t)(todo)$.

Variant *b* instead applies the transition relation to the full set of currently reached states. Variant *b* is actually much more efficient than variant *a* in practice. This is due to the fact that the size of DD is not directly linked to the number of states encoded, thus the *todo* of variant *a* may actually be much larger in memory. Variant *a* also requires more computations (to get the difference) which are of limited use to produce the final result. Finally, applying the transition relation to states that have been already explored in *b* may actually not be very costly due to the existence of a cache.

Variant *b* is similar to the original way of writing a fixpoint as found in [2]. Note that the standard encoding of a transition relation uses a DD with two DD variables (before and after the transition) for each DD variable of the state. Keeping each transition DD isolated induces a high time overhead, as different transitions then cannot share traversal. Thus the union of transitions T is stored as a DD, in other approaches than our DDD/SDD. However, simply computing this union T has been shown in some cases to be intractable.

Chaining ('95)[4] An intermediate approach is to use clusters. Transition clusters are defined and a DD representing each cluster is computed using union. This produces

Algorithm 1: Four variants of a transitive closure loop.

```
Data: {Hom} $T$ : the set of transitions encoded as $h_{Trans}$ homomorphisms
\$ $m_0$ : initial state encoded as $r(M)$ SDD
\$ $todo$ : new states to explore
\$ $reach$ : reachable states
```

a) Explicit reachability style

```
begin
    $todo := m_0$
    $reach := m_0$
    while $todo \neq 0$ do
        \$ $tmp := T(todo)$
        $todo := tmp \setminus reach$
        $reach := reach + tmp$
end
```

b) Standard symbolic BFS loop

```
begin
    $todo := m_0$
    $reach := 0$
    while $todo \neq reach$ do
        $reach := todo$
        $todo := todo + T(todo) \equiv (T + Id)(todo)$
end
```

c) Chaining loop

```
begin
    $todo := m_0$
    $reach := 0$
    while $todo \neq reach$ do
        $reach := todo$
        for $t \in T$ do
            $todo := (t + Id)(todo)$
end
```

d) Saturation enabled

```
begin
    $reach := (T + Id)^{\star}(m_0)$
end
```

smaller DD, that represent the transition relation in parts. The transitive closure is then obtained by algorithm *c*, where each *t* represents a cluster. Note that this algorithm no longer explores states in a strict BFS order, as when t_2 is applied after t_1, it may discover successors of states obtained by the application of t_1. The clusters are defined in [4] using structural heuristics that rely on the Petri net definition of the model, and try to maximize independence of clusters. This may allow to converge faster than in *a* or *b* which will need as many iterations as the state-space is deep. While this variant relies on a heuristic, it has empirically been shown to be much better than *b*.

Saturation ('01)[5] Finally the saturation method is empirically an order of magnitude better than *c*. Saturation consists in constructing clusters based on the highest DD variable that is used by a transition. Any time a DD node of the state space representation is modified by a transition it is (re)saturated, that is the cluster that corresponds to this variable is applied to the node until a fixpoint is reached. When saturating a node, if lower nodes in the data structure are modified they will themselves be (re)saturated. This recursive algorithm can be seen as particular application order of the transition clusters that is adapted to the DD representation of state space, instead of exploring in BFS order the states.

The saturation algorithm is not represented in the algorithm variants figure because it is described (in [5])on a full page that defines complex mutually recursive procedures, and would not fit here. Furthermore, DD packages such as *cudd* or *Buddy* [12,13] do not provide in their public API the possibility of such fine manipulation of the evaluation procedure, so the algorithm of [5] cannot be easily implemented using those packages.

Our Contribution All these algorithm variants, including saturation (see [9]), can be implemented using SDD. However we introduce in this paper a more natural way of expressing a fixpoint through the $h^\star$ unary operator, presented in variant d. The application order of transitions is not specified by the user in this version, leaving it up to the library to decide how to best compute the result. By default, the library will thus apply the most efficient algorithm curently available: saturation. We thus overcome the limits of other DD packages, by implementing saturation *inside* the library.

5 Automating Saturation

This section presents how using simple rewriting rules we automatically create a saturation effect. This allows to embed the complex logic of this algorithm in the library, offering the power of this technique at no additional cost to users. At the heart of this optimization is the property of *local invariance*.

5.1 Local Invariance

A minimal structural information is needed for saturation to be possible: the highest variable operations need to be applied to must be known. To this end we define :

Definition 7 (Locally invariant homomorphism). *An homomorphism h is locally invariant on variable e iff*

$$\forall s = \langle e, \alpha \rangle \in \mathbb{D} \cup \mathbb{S},\ h(s) = \textstyle\sum_{(x,d)\in\alpha} e \xrightarrow{x} h(d)$$

Concretely, this means that the application of h doesn't modify the structure of nodes of variable e, and h is not modified by traversing these nodes. The variable e is a "don't care" w.r.t. operation h, it is neither written nor read by h. A standard DD encoding [5] of h applied to this variable would produce the identity. The identity homomorphism Id is locally invariant on all variables.

For an inductive homomorphism h locally invariant on e, it means that $h(e,x) = e \xrightarrow{x} h$. A user defining an inductive homomorphism h should provide a predicate $Skip(e)$ that returns *true* if h is locally invariant on variable e. This minimal information will be used to reorder the application of homomorphisms to produce a saturation effect. It is not difficult when writing a homomorphism to define this *Skip* predicate since the useful variables are known, it actually reduces the number of tests that need to be written.

For example, the h^+ and h^- homomorphisms of section 3 can exhibit the locality of their effect on the state signature by defining *Skip*, which removes the test $e = p$ w.r.t. the previous definition since p is the only variable that is not *skipped*:

$$\begin{array}{l}
h^-(p,v)(e,x) = \\
\begin{cases} e \xrightarrow{x-v} Id & \text{if } x \geq v \\ 0 & \text{if } x < v \end{cases} \\
h^-.Skip(e) = (e \neq p) \\
h^-(p,v)(1) = 0
\end{array}$$

$$\begin{array}{l}
h^+(p,v)(e,x) = e \xrightarrow{x+v} Id \\
h^+.Skip(e) = (e \neq p) \\
h^+(p,v)(1) = 0
\end{array}$$

An inductive homomorphism Φ's application to $s = \langle e, \alpha \rangle$ is defined by $\Phi(s) = \sum_{(x,d)\in\alpha} \Phi(e,x)(d)$. But when Φ is invariant on e, computation of this union produces the expression $\sum_{(x,d)\in\alpha} e \xrightarrow{x} \Phi(d)$. This result is known beforehand thanks to the predicate *Skip*.

From an implementation point of view this allows us to create a new node directly by copying the structure of the original node and modifying it in place. Indeed the application of Φ will at worst remove some arcs. If a $\Phi(d)$ produces the 0 terminal, we prune the arc. Else, if two $\Phi(d)$ applications return the same value in SDD setting, we need to fuse the arcs into an arc labeled by the union of the arc values. We thus avoid computing the expression $\sum_{(x,d)\in\alpha} \Phi(e,x)(d)$, which involves creation of intermediate single arc nodes $e \xrightarrow{x} \cdots$ and their subsequent union. The impact on performances of this "in place" evaluation is already measurable, but more importantly it enables the next step of rewriting rules.

5.2 Union and Composition

For built-in homomorphisms the value of the *Skip* predicate can be computed by querying their operands: homomorphisms constructed using union, composition and fixpoint of other homomorphisms, are locally invariant on variable e if their operands are themselves invariant on e.

This property derives from the definition (given in [8,9]) of the basic set theory operations on DDD and SDD. Indeed for two homomorphisms h and h' locally invariant on variable e we have: $\forall s = \langle e, \alpha \rangle \in \mathbb{D} \cup \mathbb{S}$,

$$\begin{aligned}(h+h')(s) &= h(s) + h'(s)\\ &= \textstyle\sum_{(x,d)\in\alpha} e \xrightarrow{x} h(d) + \sum_{(x,d)\in\alpha} e \xrightarrow{x} h'(d)\\ &= \textstyle\sum_{(x,d)\in\alpha} e \xrightarrow{x} h(d) + h'(d)\\ &= \textstyle\sum_{(x,d)\in\alpha} e \xrightarrow{x} (h+h')(d)\end{aligned}$$

A similar reasoning can be used to prove the property for composition.

It allows homomorphisms nested in a union to share traversal of the nodes at the top of the structure as long as they are locally invariant. When they no longer *Skip* variables, the usual evaluation definition $h(s) + h'(s)$ is used to affect the current node. Until then, the shared traversal implies better time complexity and better memory complexity as they also share cache entries.

We further support natively the n-ary union of homomorphisms. This allows to dynamically create clusters by top application level as the union evaluation travels downwards on nodes. When evaluating an nary union $H(s) = \sum_i h_i(s)$ on a node $s = \langle e, \alpha \rangle$ we partition its operands into $F = \{h_i | h_i.Skip(e)\}$ and $G = \{h_i | \neg h_i.Skip(e)\}$. We then rewrite the union $H(s) = (\sum_{h\in F} h)(s) + (\sum_{h\in G} h)(s)$, or more simply $H(s) = F(s) + G(s)$. The F union is thus locally invariant on e and will continue evaluation as a block. The G part is evaluated using the standard definition $G(s) = \sum_{h\in G} h(s)$

Thus the minimal *Skip* predicate allows to automatically create clusters of operations by adapting to the structure of the SDD it is applied to. We still have no requirements on the order of variables, as the clusters can be created dynamically. To obtain efficiency, the partitions $F+G$ are cached, as the structure of the SDD typically has limited variation during construction. Thus the partitions for an nary union are computed at most once per variable instead of once per node.

The computation using the definition of $H(s) = \sum_i h_i(s)$ requires each h_i to separately traverse s, and forces to fully rebuild all the $h_i(s)$. In contrast, applying a union H allows sharing of traversals of the SDD for its elements, as operations are carried to their application level in clusters before being applied. Thus, when a strict BFS progression (like algorithm 1.b) is required this new evaluation mechanism has a significant effect on performance.

5.3 Fixpoint

With the rewriting rule of a union $H = F+G$ we have defined, we can now examine the rewriting of an expression $(H+Id)^\star(d)$ as found in algorithm 1.d :

$$\begin{aligned}(H+Id)^\star(s) &= (F+G+Id)^\star(s)\\ &= (G+Id+(F+Id)^\star)^\star(s)\end{aligned}$$

The $(F+Id)^\star$ block by definition is locally invariant on the current variable. Thus it is directly propagated to the successor nodes, where it will recursively be evaluated using the same definition as $(H+Id)^\star$.

The remaining fixpoint over G homomorphisms can be evaluated using the chaining operation order (see algorithm 1.c), which is reported empirically more effective than other approaches [14], a result also confirmed in our experiments.

The chaining application order algorithm 1.c can be written compactly in SDD as :

$$reach = (\bigcirc_{t\in T}(t+Id))^\star(s_0)$$

We thus finally rewrite:

$$(H+Id)^\star(s) = (\bigcirc_{g\in G}(g+Id)\circ(F+Id)^\star)^\star(s)$$

5.4 Local Applications

We have additional rewriting rules specific to SDD homomorphisms and the *local* construction (see section 2.3):

$$\begin{aligned}&local(h,var)(e,x) = e \xrightarrow{h(x)} Id\\ &local(h,var).Skip(e) = (r \neq var)\\ &local(h,var)(1) = 0\end{aligned}$$

Note that h is a homomorphism, and its application is thus linear to the values in x. Further a local operation can only affect a single level of the structure (defined by var). We can thus define the following rewriting rules, exploiting the locality of the operation :

$$\begin{array}{lr}
(1) & local(h,v)\circ local(h',v) = local(h\circ h',v)\\
(2) & local(h,v)+local(h',v) = local(h+h',v)\\
(3)\ v\neq v' \implies & local(h,v)\circ local(h',v') = local(h',v')\circ local(h,v)\\
(4) & (local(h,v)+Id)^\star = local((h+Id)^\star,v)
\end{array}$$

Expressions (1) and (2) come from the fact that a local operation is locally invariant on all variables except v. Expression (3) asserts commutativity of composition of local operations, when they do not concern the same variable. Indeed, the effect of applying $local(h,v)$ is only to modify the state of variable v, so modifying v then v' or modifying v' then v has the same overall effect. Thus two local applications that do not concern the same variable are independent. We exploit this rewriting rule when considering a composition of *local* to maximize applications of the rule (1), by sorting the composition by application variable. A final rewriting rule (4) is used to allow nested propagation of the fixpoint. It derives directly from rules (1) and (2).

With these additional rewriting rules defined, we slightly change the rewriting of $(H+Id)^\star(s)$ for node $s=\langle e,\alpha\rangle$: we consider $H(s)=F(s)+L(s)+G(s)$ where F contains the locally invariant part, $L = local(l,e)$ represents the operations purely local to the current variable e (if any), and G contains operations which affect the value of e (and possibly also other variables below). Thanks to rule (4) above, we can write :

$$\begin{aligned}
(H+Id)^\star(s) &= (F+L+G+Id)^\star(s)\\
&= (G+Id+(L+Id)^\star+(F+Id)^\star)^\star(s)\\
&= (\bigcirc_{g\in G}(g+Id)\circ local((l+Id)^\star,e)\circ(F+Id)^\star)^\star(s)
\end{aligned}$$

As the next section presenting performance evaluations will show, this saturation style application order heuristically allows to gain an order of magnitude in the size of models that can be treated.

6 Performances of Automatic Saturation

Impact of Propagation We have first measured how the propagation alone impacts on memory size, that is without automatic saturation. We have thus measured the memory footprint when using a chaining loop with propagation enabled or not. We have observed a gain from 15% to 50%, with an average of about 40%. This is due to the shared traversal of homomorphisms when they are propagated, thus inducing much less creation of intermediary nodes.

Impact of Hierarchy and Automatic Saturation Table 1 shows the results obtained (on a Xeon @ 1.83GHz with 4GB of memory) when generating the state spaces of several models with automatic saturation (Algo. 1.d) compared to those obtained using a standard chaining loop (Algo. 1.c). Moreover, we measured how hierarchical encoding of state spaces perform compared to flat encoding (DDD).

We have run the benchmarks on 4 parametrized models, with different sizes: the well-known Dining Philosophers and Kanban models; a model of the slotted ring protocol; a model of a flexible manufacturing system. We have also benchmarked a LOTOS

		Final *		Hierarchical *Chaining Loop*			Flat *Automatic Sat.*			Hierarchical *Automatic Sat.*		
Model Size	States #	DDD	SDD	T. (s)	Mem. (MB)	Peak #	T. (s)	Mem. (MB)	Peak #	T. (s)	Mem. (MB)	Peak #
LOTOS Specification												
	9.8e+21	–	1085	–	–	–	–	–	–	1.47	74.0	110e+3
Dining Philosophers												
100	4.9e+62	2792	419	1.9	112	276e+3	0.2	20	18040	0.07	5.2	4614
200	2.5e+125	5589	819	7.9	446	1.1e+6	0.7	58.1	36241	0.2	10.6	9216
1000	9.2e+626	27989	4019	–	–	–	14	1108	182e+3	4.3	115	46015
4000	7e+2507	–	16019	–	–	–	–	–	–	77	1488	184e+3
Slotted Ring Protocol												
10	8.3e+09	1283	105	1.1	48	90043	0.2	16	31501	0.03	3.5	3743
50	1.7e+52	29403	1345	–	–	–	22	1054	2.4e+6	5.1	209	461e+3
100	2.6e+105	–	5145	–	–	–	–	–	–	22	816	1.7e+6
150	4.5e+158	–	11445	–	–	–	–	–	–	60	2466	5.6e+6
Kanban												
100	1.7e+19	11419	511	12	145	264e+3	2.9	132	309e+3	0.4	11	14817
200	3.2e+22	42819	1011	96	563	1e+6	19	809	1.9e+6	2.2	37	46617
300	2.6e+24	94219	1511	–	–	–	60	2482	5.7e+6	7	78	104e+3
700	2.8+28	–	3511	–	–	–	–	–	–	95	397	523e+3
Flexible Manufacturing System												
50	4.2e+17	8822	917	13	430	530e+3	2.7	105	222e+3	0.4	16	23287
100	2.7e+21	32622	1817	–	–	–	19	627	1.3e+6	1.9	50	76587
150	4.8e+23	71422	2717	–	–	–	62	1875	3.8e+6	5.3	105	160e+3
300	3.6e+27	–	5417	–	–	–	–	–	–	33	386	590e+3

Table 1: Impact of hierarchical decision diagrams and automatic saturation

specification obtained from a true industrial case-study (it was generated automatically from a LOTOS specification – 8,500 lines of LOTOS code + 3,000 lines of C code – by Hubert Garavel from INRIA).

All[1] "–" entries indicate that the state space's generation did not finish because of the exhaustion of the computer's main memory.

The *"Final"* grey columns show the final number of decision diagram nodes needed to encode the state spaces for hierarchical (SDD) and flat (DDD) encoding. Clearly, flat DD need an order of magnitude of more nodes to store a state space. This shows how well hierarchy factorizes state spaces. The good performances of hierarchy also show that using a structured specification can help detect similarity of behavior in parts of a model, enabling sharing of their state space representation (see figure 3).

But the gains from enabling saturation are even more important than the gains from using hierarchy on this example set. Indeed, saturation allows to mostly overcome the

[1] We haven't reported results for flat DDs with a chaining loop generation algorithm as they were nearly always unable to handle models of big size.

“peak effect” problem. Thus *“Flat Automatic Saturation”* performs better (in both time and memory) than *“Hierarchical Chaining Loop”*.

As expected, mixing hierarchical encoding and saturation brings the best results: this combination enables the generation of much larger models than other methods on a smaller memory footprint and in less time.

7 Recursive Folding

In this section we show how SDD allow in some cases to gain an order of complexity: we define a solution to the state-space generation of the philosophers problem which has complexity in time and memory *logarithmic* to the number of philosophers. The philosophers system is highly symmetric, and is thus well-adapted to techniques that exploit this symmetry. We show how SDD allow to capture this symmetry by an adapted hierarchical encoding of the state-space. The crucial idea is to use a recursive folding of the model with n levels of depth for 2^n philosophers [2].

7.1 Initial State

Instead of $(P_0) /\!/ (P_1) /\!/ (P_2) /\!/ (P_3)$ which is the parenthesizing that is assumed by default, we parenthesize our composition $(((P_0) /\!/ (P_1)) /\!/ ((P_2) /\!/ (P_3)))$. We will thus introduce $n+2$ levels of hierarchy to represent 2^n philosophers, each level corresponding to a parenthesis group. Since each parenthesis group $((X) /\!/ (Y))$ only contains one composition $/\!/$, its SDD will contain two variables that correspond to the states of (X) and (Y).

The innermost level (level 0, corresponding to the most nested parenthesis of the composition) contains a variable of domain the states of a single philosopher. The most external parenthesis group will be used to close the loop, i.e. connect the first and last philosophers. Hence level 0 represents a single philosopher, level 1 represents the states of two philosophers, and level i represents the states of 2^i philosophers.

The magic in this representation is that each half of the philosophers at any level behaves in the same way as the other half : it’s really $(((P_0) /\!/ (P_0)) /\!/ ((P_0) /\!/ (P_0)))$. Thus sharing is extremely high : the initial state of the system for 2^n philosophers only requires $2n+k$ ($k \in \mathbb{N}$) nodes to be represented.

Let $P_0 = Fork \xrightarrow{1} HasR \xrightarrow{0} WaitR \xrightarrow{0} HasL \xrightarrow{0} WaitL \xrightarrow{0} Idle \xrightarrow{1} 1$ represent the states of a single philosopher as a DDD (as in section 3). Let M_k represent the states of 2^k philosophers using the recursive parenthesizing scheme. Following our definitions of the previous section, M_k is defined inductively by :

$$M_k = h_0 \xrightarrow{M_{k-1}} h_1 \xrightarrow{M_{k-1}} 1 \qquad\qquad M_0 = p \xrightarrow{P_0} 1$$

The most external parenthesis group yields a last variable noted $h_{(M_n)}$ such that $r((M_n)) = h_{(M_n)} \xrightarrow{M_n} 1$. We have thus defined 4 variables: $h_{(M_n)}$ for the external parenthesis, h_0 and h_1 for intermediate levels, and p for the last level ($\mathrm{Dom}(p) \subseteq \mathbb{D}$).

[2] We thank Jean-Michel Couvreur for fruitful input on this idea

7.2 Transition Relation

We define the SDD homomorphisms f and l to work respectively on the first and last philosopher modules of a submodule, as they communicate by a synchronization transition.

$$f(h)(e,x) = \\ \begin{cases} e \xrightarrow{h(x)} Id & \text{if } e = p \\ e \xrightarrow{f(h)(x)} Id & \text{if } e = h_0 \end{cases} \\ f.Skip(e) = (e \neq p) \wedge (e \neq h_0) \\ f(h)(1) = 0$$

$$l(h)(e,x) = \\ \begin{cases} e \xrightarrow{h(x)} Id & \text{if } e = p \\ e \xrightarrow{l(h)(x)} Id & \text{if } e = h_1 \end{cases} \\ l.Skip(e) = (e \neq p) \wedge (e \neq h_1) \\ l(h)(1) = 0$$

We then need to take into account that all modules have the same transitions. Transitions that are purely local to a philosopher module are unioned and stored in a homomorphism which will be noted $\mathcal{L}$ (in fact only *hungry* is purely local). We note $\Pi_i(s)$ the part of a synchronization transition s_L created for label L that concerns the *current* philosopher module P_i and $\Pi_{i+1}(s)$ the part of s_L that concerns $P_{i+1 mod N}$ the right hand neighbor of P_i. We note S the set of synchronization transitions, induced by the labels $\mathcal{L}_i$ and $\mathcal{R}_i$.

Let $\tau_{loop} = Id + \sum_{s \in S} l(\Pi_i(s)) \circ f(\Pi_{i+1}(s))$
τ_{loop} is an SDD homomorphism operation defined to “close the loop”, that materializes that the last philosophers right hand neighbor is the first philosopher. Our main firing operation that controls the saturation is τ defined as follows :

$$\tau(e,x) = \\ \begin{cases} e \xrightarrow{(\tau \circ \tau_{loop})^{\star}(x)} Id & \text{if } e = h_{(M_n)} \\ e \xrightarrow{\tau^{\star}(x)} \tau + \sum_{s \in S} e \xrightarrow{\tau^{\star} \circ l(\Pi_i(s))} \tau \circ f(\Pi_{i+1}(s)) & \text{if } e = h_0 \\ e \xrightarrow{\tau^{\star}(x)} Id & \text{if } e = h_1 \\ e \xrightarrow{\mathcal{L}^{\star}(x)} Id & \text{if } e = p \end{cases} \\ \tau(t)(1) = 0$$

We can easily adapt this encoding to treat an arbitrary number n of philosophers instead of powers of 2, by decomposing n into it’s binary encoding. For instance, for $5 = 2^0 + 2^2$ philosophers $((P0) \mathbin{/\!/} ((P1 \mathbin{/\!/} P2) \mathbin{/\!/} (P3 \mathbin{/\!/} P4)))$ Such unbalanced depth in the data structure is gracefully handled by the homogeneity of our operation definitions, and does not increase computational complexity.

7.3 Experimentation

We show in table 2 how SDD provide an elegant solution to the state-space generation of the philosophers problem, for up to 2^{20000} philosophers. The complexity both in time and space is roughly linear to n, with empirically $8n$ nodes and $12n$ arcs required to represent the final state-space of 2^n philos

The solution presented here is specific to the philosphers problem, though it can be adapted to other symmetric problems. Its efficiency here is essentially due to the

			Final		Peak	
Nb. Philosophers	States	Time (s)	SDD	DDD	SDD	DDD
2^{10}	1.02337e+642	0.0	83	31	717	97
2^{31}	1.63233e+1346392620	0.02	251	31	2250	97
2^{1000}	*N/A*	0.81	8003	31	72987	97
2^{10000}	*N/A*	9.85	80003	31	729987	97
2^{20000}	*N/A*	20.61	160003	31	1459987	97

Table 2: Performances of recursive folding with 2^n philosophers . The states count is noted *N/A* when the large number library GNU Multiple Precision (GMP) we use reports an overflow.

inherent properties of the model under study. In particular the strong locality, symmetry and the fact that even in a BDD/DDD representation, adding philosophers does not increase the "width" of the DDD representation – only it's height –, are the key factors.

The difficulty in generalizing the results of this example, is that we exploit in the definition of the transition relation the fact that all philosophers have the same behavior, and the circular way they are synchronized. In other words, our formalism is not well adapted to scaling to 2^n, because it lacks an inductive definition of the problem that we could capture automatically. While a simple use of the parenthesizing scheme described in section 3 would produce overall the same effects, the recursive homogeneity captured by τ would be lost. We would then have linear complexity w.r.t. to the number of philosophers, when computing our rewriting rules, which is not viable to scale up to 2^{20000} as we no longer can have overall logarithmic complexity.

Thus our current research direction consists in defining a formalism (e.g. a particular family of Petri nets) such that we could recognize this pattern and obtain the recursive encoding naturally.

However, this example reveals that SDD are potentially exponentially more powerful than other decision diagram variants.

8 Conclusion

In this paper, we have presented the latest evolutions of hierarchical Set Decision Diagrams (SDD), that are suitable to master the complexity of very large systems. We think that such diagrams are well-adapted to process hierarchical high-level specifications such as Net-within-Nets [15] or CO-OPN [16].

We have presented how we optimize evaluation of user homomorphisms to automatically producing a saturation effect. Moreover, this automation is done at a low cost for users since it uses a *Skip* predicate that is easy to define. We thus generalize extremely efficient saturation approach of Ciardo et al. [5] by giving a definition that is entirely based on the structure of the decision diagram and the operations encoded, instead of involving a given formalism. Furthermore, the automatic activation of saturation allows users to concentrate on defining the state and transition encoding.

Also, we have shown how recursive folding allows in very efficient and elegant manner to generate state spaces of regular and symmetric models, with up to 2^{20000} philosophers in our example. Although generalization of this application example is left to further research, it exhibits the potentially exponentially better encoding SDD provide over other DD variants for regular examples.

SDD and the optimizations described are implemented in `libddd`, a C++ library freely available under the terms of GNU LGPL. With growing maturity since the initial prototype developed in 2001 and described in [8], `libddd` is today a viable alternative to Buddy [13] or CUDD [12] for developers wishing to take advantage of symbolic encodings to build a model-checker.

References

1. Bryant, R.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691
2. Burch, J., Clarke, E., McMillan, K.: Symbolic model checking: 10^{20} states and beyond. Information and Computation (Special issue for best papers from LICS90) **98**(2) (1992) 153–181
3. Bollig, B., Wegener, I.: Improving the Variable Ordering of OBDDs Is NP-Complete. IEEE Trans. Comput. **45**(9) (1996) 993–1002
4. Roig, O., Cortadella, J., Pastor, E.: Verification of asynchronous circuits by BDD-based model checking of Petri nets. In: 16th International Conference on the Application and Theory of Petri Nets. Volume 815. (1995) 374–391
5. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: Tools and algorithms for the construction and analysis of systems, Springer Varlag, LNCS 2619 (2003) 379–393
6. Holzmann, G., Smith, M.: A practical method for verifying event-driven software. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 597–607
7. LIP6/Move: the libDDD environment. `www.lip6.fr/libddd` (2007)
8. Couvreur, J.M., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.A.: Data Decision Diagrams for Petri Net Analysis. In: ICATPN'02, Adelaide, Australia. Volume 2360 of LNCS., Springer-Verlag (2002) 1–101
9. Couvreur, J.M., Thierry-Mieg, Y.: Hierarchical Decision Diagrams to Exploit Model Structure. Formal Techniques for Networked and Distributed Systems - FORTE 2005 (2005) 443–457
10. Wang, F.: Formal verification of timed systems: A survey and perspective. IEEE **92**(8) (2004)
11. Ciardo, G., Siminiceanu, R.: Using edge-valued decision diagrams for symbolic generation of shortest paths. In: FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design, London, UK, Springer-Verlag (2002) 256–273
12. Somenzi, F.: CUDD: CU Decision Diagram Package (release 2.4.1), `http://vlsi.colorado.edu/fabio/CUDD/cuddIntro.html` (2005)
13. Lind-Nielsen, J., Mishchenko, A., Behrmann, G., Hulgaard, H., Andersen, H.R., Lichtenberg, J., Larsen, K., Soranzo, N., Bjorner, N., Duret-Lutz, A., Cohen, H.a.: buddy - library for binary decision diagrams (release 2.4), `http://buddy.wiki.sourceforge.net/` (2004)
14. Ciardo, G.: Reachability Set Generation for Petri Nets: Can Brute Force Be Smart? Applications and Theory of Petri Nets 2004 (2004) 17–34
15. Cabac, L., Duvigneau, M., Moldt, D., Rölke, H.: Modeling Dynamic Architectures Using Nets-within-Nets. In: Applications and Theory of Petri Nets 2005, ICATPN 2005, Miami, USA. Volume 3536 of LNCS. (2005) 148–167
16. Biberstein, O., Buchs, D., Guelfi, N.: Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. (2001)