Fundamenta Informaticae Petri Nets (2008) 1–25
IOS Press

Building Efficient Model Checkers using Hierarchical Set Decision Diagrams and Automatic Saturation (DRAFT)

Alexandre Hamez
Université P. & M. Curie LIP6 - CNRS UMR 7606
4 Place Jussieu, 75252 Paris cedex 05, France
EPITA Research and Development Laboratory
F-94276 Le Kremlin-Bicêtre cedex, France

Yann Thierry-Mieg
Université P. & M. Curie LIP6 - CNRS UMR 7606
4 Place Jussieu, 75252 Paris cedex 05, France

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

Abstract. Shared decision diagram representations of a state-space 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.

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, flexibility to a user defining a symbolic transition relation. We show in this paper how, 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 build as an example an SDD model-checker for a compositional formalism: Instantiable Petri Nets (IPN). IPN define a *type* as an abstract contract. Labeled P/T nets are used as an elementary type. A composite type is defined to hierarchically contain instances (of elementary or composite type). To compose behaviors, IPN use classic label synchronization semantics from process calculi.

With a particular recursive folding SDD are able to offer solutions for symmetric systems in logarithmic complexity with respect to other DD. Even in less regular cases, the use of hierarchy in the specification is shown to be well supported by SDD. Experimentations and performances are reported on some 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 solution which has shown its strength to tackle very large state spaces is the use of shared decision diagrams like BDD [4, 5].

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 tool performance [19, 10]. Such aspects are difficult to tackle for non specialists of decision diagram technology.

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 [10] with limited user expertise in DD. These high level constructs allow a user to concentrate on specifying the transition relation of the system using *homomorphisms*. The rewriting rules introduced in this paper allow to obtain an equivalent representation (with the same overall effect), but that optimizes its evaluation.

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 [1], communicating processes like in Promela [16] or Harel state charts [15]). To illustrate the use of our approach, we introduce IPN, a hierarchical Petri net representation as a basis for illustration and experimentation.

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

- Exploitation of specification's structure to introduce hierarchy in the state space, it enables more possibilities for exploiting pattern similarities in the system,
- Automatic activation of saturation; the algorithms described in this paper allow the library to enact saturation with minimal user input,
- A *recursive* folding technique that is suitable for very symmetric systems.

The paper is structured as follows. Section 2 presents SDD and section 3 defines a compositional Petri net formalism: Instantiable Petri Nets (IPN), that provide built-in functions for modularity and assembling. Section 4 then explains how to build a model checker for IPN on top of SDD. Sections 5 and 6 show how we can provide transparent optimizations, thus extending the saturation mechanisms introduced in [10]. Section 7 then presents a performance evaluation of our openly distributed implementation: `libddd` [18]. Finally, section 8 presents a way to take advantage of hierarchy in decision diagrams for very regular systems, as well as the associated performance results we get.

2. Hierarchical Set Decision Diagram (SDD)

This section recalls the salient points of Hierarchical Set Decision Diagrams (SDD), a data structure based on the principles of decision diagram technology (node uniqueness thanks to a canonical representation, dynamic programming, ordering issues, etc.). They feature two main original aspects: the support of hierarchy in the representation (section 2.1) and the definition of user operations through a mechanism called *inductive homomorphisms* (section 2.2) which gives freedom and flexibility to the user.

2.1. Structure of SDD

Hierarchical Set Decision Diagrams (SDD) defined in [13], are shared decision diagrams in which arcs are labeled by a *set* of values, instead of a single value. This set may itself be represented by an SDD, thus when labels are SDD, we think of them as hierarchical decision diagrams. Definition 2.1 is taken practically verbatim from [14] where it was adapted for more clarity from [13].

SDD are data structures for representing sets of sequences of assignments of the form $\omega_1 \in s_1; \omega_2 \in s_2; \cdots ; \omega_n \in s_n$ where ω_i are variables and s_i are sets of values.

We assume no variable ordering, and the same variable can occur several times in an assignment sequence. We define the terminal 1 to represent the empty assignment sequence, that terminates any valid sequence. The terminal 0 represents the empty set of assignment sequences. In the following, Var denotes a set of variables, and for any ω in Var, $\mathrm{Dom}(\omega)$ represents the domain of ω which may be infinite.

Definition 2.1. (Set Decision Diagram)
$\delta \in \mathbb{S}$, the set of SDD, is inductively defined by:

- $\delta \in \{0,1\}$ or
- $\delta = \langle \omega, \pi, \alpha \rangle$ with:
  - $\omega \in Var$.
  - $\pi = s_0 \cup \cdots \cup s_n$ is a finite partition of $\mathrm{Dom}(\omega)$, i.e. $\forall i \neq j, s_i \cap s_j = \emptyset, s_i \neq \emptyset, n$ finite.
  - $\alpha : \pi \to \mathbb{S}$, such that $\forall i \neq j, \alpha(s_i) \neq \alpha(s_j)$.

By convention, when it exists, the element of the partition π that maps to the SDD 0 is not represented. We denote by $\omega \xrightarrow{s} \delta'$, the SDD $\delta = \langle \omega, \pi, \alpha \rangle$ with $\pi = s$ and $\alpha(s) = \delta'$ (and $\alpha(\mathrm{Dom}(\omega) \setminus s) = 0$).

Despite its simplicity, this definition supports rich and complex data:

- SDD support domains of infinite size (e.g. $\mathrm{Dom}(\omega) = \mathbb{R}$), provided that the number of elements in the partition remains finite (e.g. $]0..3],]3..+\infty]$). This feature could be used to model clocks for instance (as in [21]). It also places the expressive power of SDD above most variants of DD.

- SDD or other variants of decision diagrams can be used as the domain of variables, introducing hierarchy in the data structure.
- SDD can handle paths of variable lengths, if care is taken when choosing the state encoding to avoid creating so-called incompatible sequences (see [13]). This feature is useful when representing dynamic structures such as queues, lists or variable size arrays.

The definition ensures that any set of assignment sequences has a unique (canonical) SDD representation. The finite size of the partition π ensures we can store α as a finite set of pairs $\langle s_i, \delta_i \rangle$, and let π be implicitly defined by α.

SDD are canonized by construction through the union operator. The canonicity of SDD is due to the two properties (1) π is a partition and (2) no two arcs from a node may lead to the same SDD. Therefore any value $x \in \mathrm{Dom}(\omega)$ is represented on at most one arc, and any time we are about to construct $\omega \xrightarrow{s} \delta + \omega \xrightarrow{s'} \delta$, we will construct an arc $\omega \xrightarrow{s \cup s'} \delta$ instead (fusing arcs).

The opposite effect (splitting arcs) is obtained when building an SDD such that two arcs $\langle s, \delta \rangle$ and $\langle s', \delta' \rangle$ have non empty intersection $s \cap s' \neq \emptyset$. We then produce three arcs, $\langle s \setminus s', \delta \rangle, \langle s' \setminus s, \delta' \rangle$ and $\langle s \cap s', \delta \cup \delta' \rangle$.

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 2.2. (Compatible SDD sequences)
An SDD *sequence* is an SDD of the form $\omega_0 \xrightarrow{s_0} \cdots \omega_n \xrightarrow{s_n} 1$. Let σ_1, σ_2 be two sequences, $\sigma_1 \approx \sigma_2$ iff.:

- $\sigma_1 = 1 \wedge \sigma_2 = 1$
- $\sigma_1 = \omega \xrightarrow{s} \delta \wedge \sigma_2 = \omega' \xrightarrow{s'} \delta'$ such that $\begin{cases} \omega = \omega' \\ \wedge s \approx s' \\ \wedge s \cap s' \neq \emptyset \implies \delta \approx \delta' \end{cases}$

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

While this notion of compatible sequences may seem restrictive, it is more permissive than usual for DD, where the norm is to use a fixed set of variables, in a fixed order along all paths. In practice, we use this compatible sequence definition to handle dynamic structures such as queues. To encode a queue, we repeat the same variable ω. The last occurrence of ω along any path is then artificially labeled with a special marker, noted $\sharp$. Hence, $\omega \xrightarrow{\sharp} 1$ represents an empty queue, and $\omega \xrightarrow{s_1} \omega \xrightarrow{\sharp} 1$ represents a queue with one element (chosen from s_1). These two SDD are compatible, and can be stored inside a single SDD. Furthermore, using homomorphisms we can define appropriate operations to manipulate such dynamic structures (see [12]).

2.2. Operations and Homomorphisms

Usually in symbolic methods (e.g. BDD), the next state function of a system is encoded using one or more decision diagrams, with two variables per variable of the state signature. These variables are usually interlaced in the transition relation representation. A dedicated synchronized product operation then allows to compute the successor image for a set of states.

In contrast to BDD, SDD operations are encoded as homomorphisms $\mathbb{S} \mapsto \mathbb{S}$. SDD support standard set theoretic operations ($\cup, \cap, \setminus$ respectively noted $+, *, -$). They also offer a concatenation operation $\delta_1 \cdot \delta_2$ which replaces 1 terminal of δ_1 by δ_2. This corresponds to a Cartesian product. In addition, basic and inductive homomorphisms are introduced as a powerful and flexible mechanism to define application specific operations. A detailed description of homomorphisms including many examples can be found in [12].

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

Furthermore, application-specific mappings can be defined by *inductive* homomorphisms over $\mathbb{S}$. An inductive homomorphism ϕ is defined by its evaluation on the 1 terminal $\phi(1) \in \mathbb{S}$, and its evaluation $\phi(\omega, s)$ for any $\omega \in Var$ and any $s \subseteq \mathrm{Dom}(\omega)$. The expression $\phi(\omega, s)$ is itself a (possibly inductive) homomorphism, that will be applied on the successor node $\alpha(s)$. The result of $\phi(\langle \omega, \pi, \alpha \rangle)$ is then defined as $\sum_{s \in \pi} \phi(\omega, s)(\alpha(s))$, where $\sum$ represents a union.

As an example, the local construction $\mathcal{L}$ allows to "carry" a homomorphism h to a certain variable v, and apply h to the current state of v. Thus, it implements an operation local to the variable v. This homomorphism will be used in section 4. It is defined by:

$$\begin{aligned}
\mathcal{L}(v,h)(\omega,s) &= \begin{cases} \omega \xrightarrow{s} \mathcal{L}(v,h) & if\, \omega \neq v \\ \omega \xrightarrow{h(s)} Id & else \end{cases} \\
\mathcal{L}(v,h)(1) &= 0
\end{aligned}$$

The reader will find several examples of homomorphisms throughout this paper. Section 6 shows simple homomorphisms that select and increment assignment sequences, while Section 4 shows more complex homomorphisms that describe the transition relation of a Petri net.

The **transitive closure** * unary operator allows to perform a least fixpoint computation. For any homomorphism h and any node $\delta \in \mathbb{S}$, $h^\star(\delta)$ is evaluated by repeating $\delta \leftarrow h(\delta)$ until a fixpoint is reached. In other words, $h^\star(\delta) = h^n(\delta)$ where n is the smallest integer such that $h^n(\delta) = h^{n+1}(\delta)$. This operator is often applied to $(Id + h)$ instead of just h, allowing to accumulate newly computed assignment sequences in the result.

An important contribution of [14] is the definition of a set of rewriting rules for homomorphisms, allowing to automatically make use of the decision diagram saturation algorithms originally due to Ciardo [10]. In this extended version of [14], these rules will be detailed in section 6. When computing the least fixpoint of a transition relation over a set of states, this algorithm offers gains of one to three orders of magnitude over classical BFS fixpoint algorithms.

For the user, these rewriting rules are transparent. Given a set of homomorphisms $\{t_1, \ldots, t_n\}$ that represent a partition of the transition relation of the system, the application of $(t_1 + \ldots + t_n + Id)^\star$ to a node automatically triggers the saturation algorithm for the evaluation. Note that this is a central operation in any symbolic model-checking problem since reachability is defined as a transitive closure over the full transition relation. A more complex model-checker, for instance a CTL model-checker, can then be constructed using nested transitive closures over the transition relation or its reverse [5]. The predecessor transition relation can also be encoded using homomorphisms, though care must be taken to remain within a forward reachable region.

3. Instantiable Petri Nets (IPN)

This section defines Instantiable Petri Nets (IPN). This hierarchical Petri Net notation will be used as a running example to demonstrate our SDD based encodings. The definition is split into two parts, first an abstract contract for IPN types. then two concrete realization of this contract.

We show how to adapt labeled Petri nets to match this contract. Then we define a composite type that is a container for instances (of composite or P/T net nature). The abstract contract is introduced to allow a composite type to contain instances of elementary or composite nature homogeneously.

The definitions of this section are new with respect to the conference version of this paper [14]. The definitions we use here are both more generic and better detailed. They reflect a mature formalization of compositionality and hierarchy that closely matches the capabilities of SDD.

3.1. Instantiable Types

The generic definition of an Instantiable Petri NEt (IPN) builds upon the notion of model type and instance. It uses a composition mechanism based solely on transition *synchronization* (no explicit shared memory or channel). Definition 3.1 sets an abstract contract or interface that must be realized by concrete IPN types. The definition is split in two parts: we first define a abstract contract, then two concrete realizations of this contract for Petri nets and a composite type.

Notations: Bag(A) denotes a multiset over a set A. Let $\oplus$ designate a commutative operation $A \times A \mapsto A$. Let $\tau \in \text{Bag}(A)$, we note $S = \bigoplus_{a \in \tau} a$ where if an element $a \in A$ occurs n times in τ it will be $\oplus$-ed n times in S.

Definition 3.1. (IPN Concepts)
An **IPN type** must provide a tuple $type = \langle S, InitStates, T, Locals, Succ \rangle$:

- S is a set of states;
- $InitStates \subseteq S$ is a finite subset of designated initial states;
- T is a finite set of public transition labels;
- $Locals : S \mapsto 2^S$ is the local successors function.
- $Succ : S \times \text{Bag}(T) \mapsto 2^S$ is the transition function satisfying $\forall s \in S, Succ(s, \emptyset) = \{s\}$.

Let Types denote a set of IPN types. An **IPN instance** i is defined by its IPN type, noted $type(i) \in Types$. We will further use $type(i).S$ (resp. $type(i).InitStates, \ldots$) to refer to the states (resp. initial states, ...) of an instance's type.

(**Reachability**) A state s' is reachable by an instance i from the state s_0 iff. $\exists s_1, \dots s_n \in type(i).S$ s.t. $s' = s_n \wedge \forall 1 \leq j \leq n, s_j \in type(i).Locals(s_{j-1})$.

InitStates is introduced to avoid violating encapsulation: to initialize an instance we need to be able to designate its initial configuration(s) without knowing the internal structure of the instance.

Locals will typically return states reachable through occurrence of local events. It represents transitions that may occur within an instance autonomously or independently from the rest of the system.

The function *Succ* allows to obtain successors by explicitly synchronizing over a multiset of public transition labels. Synchronizing on an empty multiset of transitions leaves the state of the instance locally unchanged. Note that *Succ* is the only way to control the behavior of a (sub)system from outside. Thus the transition relation of a full system can only be defined in terms of transition synchronizations using *Succ* and of independent local behaviors.

A full system is defined by an instance of a particular type in a specific initial state. As a full system is self-contained, the definition of reachability only depends on the definition of *Locals*.

As an example, figure 1 presents two IPN type declarations. They could be used to model the classical dining philosophers. For each type declared (Fork and Philo), only elements that are publicly visible are represented: *InitStates* and T. Of course, the transition relation itself (*Succ* and *Locals*) is not represented, as this part is defined in the implementation of a type (e.g a Petri net, see figure 2).

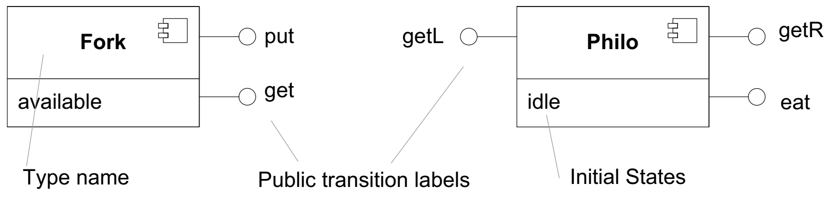

Figure 1. Type declarations for the dining philosophers. A *Fork* which one can *put* or *get*. A *Philo* with transition labels allowing interaction with other types when he gets his left (resp. right) fork with *getL* (resp. *getR*) or returns them (simultaneously in this version) with *eat*. *Philo* are initially *idle* and *Fork* are *available*. An implementation of these types using Petri Nets is provided in figure 2.

3.2. Petri Nets as Elementary Type

We show here how to adapt classic labeled P/T definitions to match this IPN type contract. In practice, any finite state machine based formalism could be used as realization of IPN.

Definition 3.2. A labeled Petri net (LPN) is a tuple $\langle Pl, Tr, Pre, Post, L, label, m_0 \rangle$ where

- Pl is a finite set of places,
- Tr is a finite set of transitions (with $Pl \cap Tr = \emptyset$),
- Pre and $Post : Pl \times Tr \rightarrow \mathbb{N}$ are the pre and post functions labeling the arcs.
- $L \subseteq Tr$ is a set of labeled transitions
- $M_0 \subset \mathbb{N}^{Pl}$ is a set of designated *markings* of the net.

So that LPN fulfill the IPN type contract, we further define:

- $S = \mathbb{N}^{Pl}$
- $InitStates = M_0$
- $T = L$
- $Locals : S \mapsto 2^S$ is defined by $\forall m, m' \in S, m' \in Locals(m)$ iff $\exists t \in Tr \setminus L$, $\forall p \in Pl, m(p) \geq Pre(p,t)$ and then $m'(p) = m(p) - Pre(p,t) + Post(p,t)$;
- $Succ : S \times \mathrm{Bag}(L) \mapsto 2^S$: is defined by $\forall m, m' \in S, \forall \tau \in \mathrm{Bag}(L), m' \in Succ(m, \lambda)$ iff

  $$\forall p \in Pl, m(p) \geq \sum_{t \in \tau} Pre(p,t) \qquad \textit{(enabling)}$$

  and then

  $$m'(p) = m(p) + \sum_{t \in \tau} (Post(p,t) - Pre(p,t)) \qquad \textit{(firing)}$$

As an example, figure 2 represents an implementation of the types introduced in figure 1 for the Philosophers dinner. Note the use of private ($Tr \setminus L$) and public (L) transitions. Public transitions cannot be fired in isolation (by *Locals*), but they are offered to the environment as transition labels.

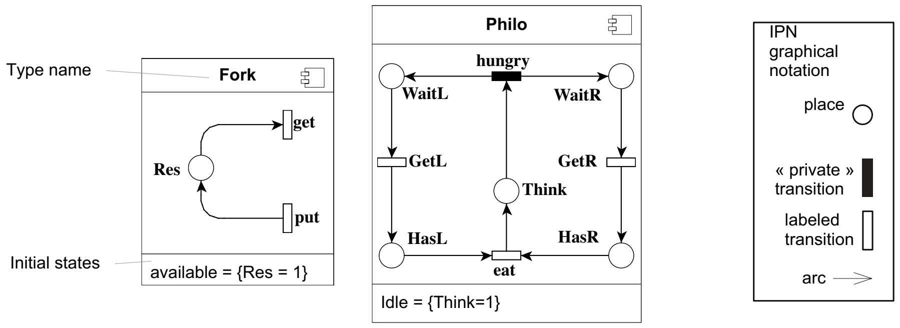

Figure 2. Implementation of the types defined in figure 1.

3.3. A Composite Type

We now define a composite IPN type to offer support for the hierarchical composition of IPN instances.

Notations: Let $I = \{i_1, \ldots, i_n\}$ designate a set of IPN instances. CS_I is the set $type(i_1).S \times \ldots \times type(i_n).S$ and $Syncs_I$ designates the set $\mathrm{Bag}(type(i_1).T) \times \ldots \times \mathrm{Bag}(type(i_n).T)$. We will note $\forall i \in I, \pi_i$ the projection operator $Syncs_I \mapsto \mathrm{Bag}(type(i).T)$. The sum $\oplus : Syncs_I \times Syncs_I \mapsto Syncs_I$ is defined as: $t = t_0 \oplus t_1$ iff $\forall i \in I, \pi_i(t) = \pi_i(t_0) + \pi_i(t_1)$ where + designates the standard sum of multisets.

Intuitively, CS_I represents composite states, and $Syncs_I$ represents synchronizations of public labels of the set I of subcomponents. The sum $\oplus$ represents an operation cumulating the effects of two synchronizations. For instance, let $I = \{i_0, i_1\}$. Let $t, t' \in Syncs_I$, $t = (t_0 + 2't_1) \times (\emptyset)$; $t' = (t_0) \times (t_3)$. Then $t'' = t \oplus t' \implies t'' = (2't_0 + 2't_1) \times (t_3)$.

We define the next state function $Next_I$, which is used when defining *Locals* and *Succ* below, $Next_I : CS_I \times \text{Bag}(Syncs_I) \mapsto 2^{CS_I}. \forall s, s' \in CS_I, \forall \tau \in \text{Bag}(Syncs_I)$,

$$s' \in Next_I(s,\tau) \quad \textit{iff} \quad \forall i \in I, s'(i) \in type(i).Succ(s(i), \pi_i(\bigoplus_{t\in\tau} t))$$

In words, the successors by a multiset of synchronizations is computed as the states obtained by applying the projection of their cumulated effects (obtained with $(\bigoplus_{t\in\tau} t)$) to the current state of each instance i in the set I.

Definition 3.3. (Composite)
A **composite** is a tuple $C = \langle I, IS, ST, V\rangle$:

- I is a finite set of IPN instances, said to *be contained* by C. We further require that the type of each IPN instance preexists when defining these instances, in order to prevent circular or recursive type definitions.
- $IS \subseteq \{s \in CS_I \mid \forall i \in I, s(i) \in type(i).InitStates\}$ is a finite set of designated initial states
- $ST \subset Syncs_I$ is the finite set of synchronizations;
- $V : ST \mapsto$ {public,private} assigns a visibility to each synchronization

The IPN type corresponding to a composite, is defined as:

- $S = CS_I$
- $InitStates = IS$
- $T = \{st \in ST \mid V(st) = public\}$
- $Locals : S \mapsto 2^S.\ \forall s, s' \in S, s' \in Locals(s)$ iff

$$\begin{array}{rl} & \exists i \in I, s'(i) \in type(i).Locals(s(i)) \wedge \forall j \in I, j \neq i, s'(j) = s(j) \\ \textit{or} & \exists t \in ST, V(t) = \text{private}, s' \in Next_{C.I}(s, \{t\}) \end{array}$$

- $Succ : S \times \text{Bag}(T) \mapsto 2^S.\ \forall s, s' \in S, \forall \tau \in \text{Bag}(T), Succ(s,\tau) = Next_{C.I}(s,\tau)$

Definition 3.3 is a realization of the generic IPN type contract. It contains either elementary subcomponents (see section 3.2), or recursively other instances of composite nature.

Locals is defined as states reachable through the occurrence of local transitions of any nested component (without affecting the other subcomponents) or states reachable through occurrence of any given private synchronization.

Succ is realized by “summing” the impact of the multiset of transitions given as its argument using the $\oplus$ operator defined over $Syncs_I$, and synchronously updating the state of each subcomponent.

Concerning our running example, consider Figure 3 defines a module that groups one *Philo* and one *Fork* to build a composite type PhiloFork.

We can then consider in Fig. 4 a composite type built to represent the Philosophers system with three instances of PhiloFork.

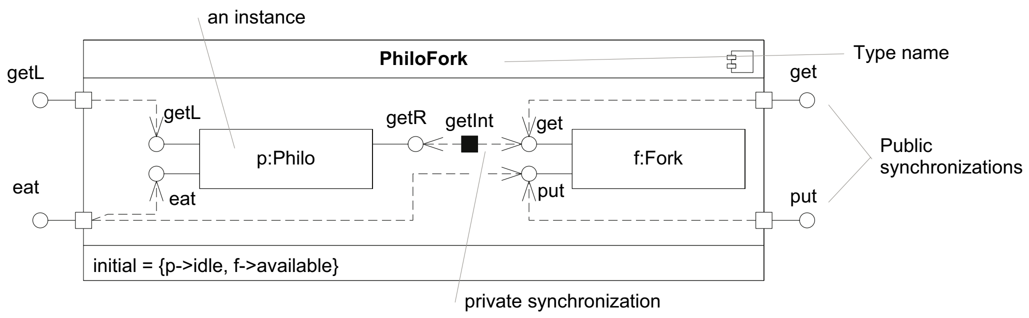

Figure 3. A PhiloFork composite type representing a Philo and the Fork to his right as a block. Note that synchronization *eat* synchronizes the philosopher *p.eat* eating and releasing his right fork *f.put* but is declared public to allow synchronization with the release of the left fork. *getL*, *get* and *put* are simply exported (made visible). An internal event exists to represent the philosopher getting his right fork (synchronize *p.getR* and *f.get*). This event is not visible to the environment (it is fired via *Locals*).

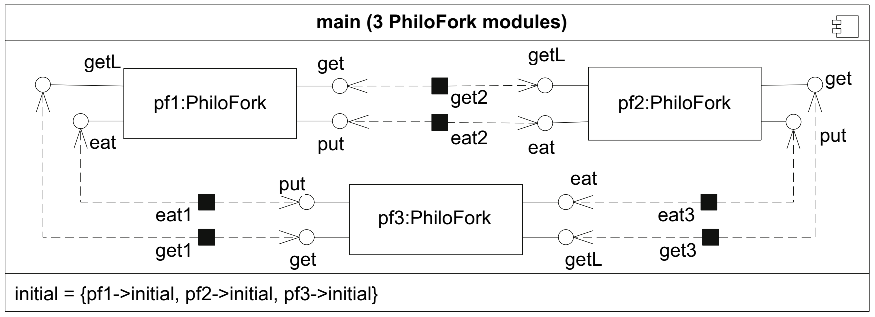

Figure 4. A model for the dinner of 3 philosophers. The composite type declaration contains three instances and six private synchronizations. For instance, private synchronization eat_2 synchronizes *pf2.eat* and *pf1.put*.

Other encodings can be defined. For example, instances of *philo* and *Fork* could be directly assembled as a ring, without defining the PhiloFork type. Or the PhiloFork could be directly implemented by a P/T net. Or as we will show in section 8 PhiloFork can be implemented by a different composite type that represents several philosophers. This example shows that hierarchical specification of a system is possible. Such a feature is of particular interest to describe distributed systems that can be seen as a hierarchical composition of elementary modules. It also allows to exhibit a certain type of symmetry of in system, which can be exploited by SDD.

4. Building a Model-Checker for IPN with Set Decision Diagrams

To build a model-checker for a given formalism using SDD, one needs to perform the following steps:

1. Adapt the formalism to a hierarchical encoding,
2. Define a representation of states,
3. Define a transition relation using homomorphisms,

4. Define a verification goal.

These steps are presented here using the IPN formalism defined in the previous section.

4.1. Step 1: Define a hierarchical formalism

This step is not strictly necessary to enable saturation, however, it allows to profit from a hierarchical state encoding.

The easiest way to do this given our definitions of section 3 is to adapt a formalism to match the IPN type contract. In this way, new elementary types can be defined (e.g. a simple labeled transition system, or any variant of automata). They can then reuse the composite type definition to allow hierarchical modelling.

One could also extend the composite definition, for instance by adding other types of synchronizations (e.g. reset transitions, UML-style history to return to a previous state, non deterministic synchronizations, etc.).

We use as running example the IPN definition of section 3, which is already hierarchical. So this step consisted in adapting the definitions of a P/T net to match the IPN type contract, as presented in section 3.2.

4.2. Step 2: State Encoding

For any IPN type, we need to define an SDD representation of any set of states as an SDD.

IPN To encode the state of an IPN with $n = |P|$ places, we use an SDD with n integer domain SDD variables. Given a total ordering of the places, for any state $m \in S$, we can define a state $\sigma(m) \in \mathbb{S}$ such that $\sigma(m) = p_0 \xrightarrow{\{m(p_0)\}} \cdots p_{n-1} \xrightarrow{\{m(p_{n-1})\}} 1$.

Composite A state $s \in CS_I$ of a composite C will be represented by an SDD of $|I|$ variables, each representing the state of an instance $i \in I$. The domain of each variable is determined by the type of the instance.

Figure 5 shows this type of encoding for the philosopher example (we consider three philosophers). This encoding reproduces the structure of the IPN specification. We thus find three levels:

- The *main* level describes the states of the three instances $pf1$, $pf2$ and $pf3$ of figure 4. Each instance is represented by one variable. The arcs are labeled using SDD of the *PhiloFork* level.

- The *PhiloFork* level describes the possible states of a *PhiloFork* module (see figure 3). The state of a *PhiloFork* is decomposed into the state of the fork f and the *Philo* instance p.

- The *elementary level* contains IPN states. For more clarity we have represented left the SDD corresponding to Philo states and right the SDD representing Fork states (see figure 1). These SDD use integer domain variables, and one variable per place of the net.

At each level, the possibility of sharing representation is introduced. The labels of the arcs of the upper levels refer to nodes of lower levels. Let us outline in Figure 5 how we can read a state from the structure. In the *main* SDD, bold gray arcs (labeled with $m2$) are linked to the $m2$ entry point in the *PhiloFork* SDD. Similarly, bold black arcs in the *PhiloFork* SDD (labeled with $f0$) are linked to the $f0$

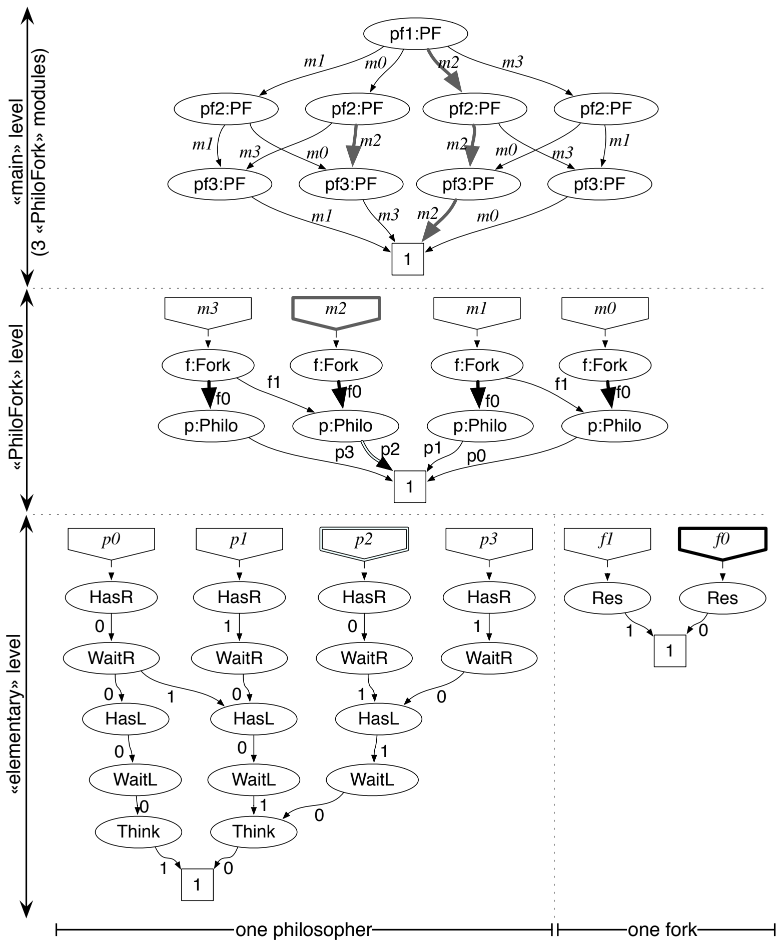

Figure 5. Hierarchical encoding of the full state-space for 3 philosophers

entry in the fork SDD. Finally, double gray arc (labeled with $p2$) are connected to the $p2$ entry in the *one philosopher* SDD. The state where all *PhiloFork* instances are in state $m2$ corresponds to a deadlock state: $m2$ is a state where the fork is not available ($f0$) and the philosopher is in state $p2$ where *WaitR* and *HasL* are marked (he has the left fork and waits for the right one).

This example clearly shows that parts of the representation are shared at each level thanks to these relationships between hierarchical levels. For example, the $m2$ entry of the *PhiloFork* level is referenced four times in the *main* level. With "classical" decision diagrams, this type of sharing between parts of the state space could not be achieved.

4.3. Step 3: Transition Encoding:

The IPN formalism defines two types: IPN and composite. For each of these concrete realizations of the IPN type contract, we need to define *Succ* and *Locals* as homomorphisms.

IPN 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 .

$$h^-(p,v)(\omega,s) = \begin{cases} \omega \xrightarrow{\{n-v \mid n\in s \wedge n \geq v\}} Id & if\, \omega = p \\ \omega \xrightarrow{s} h^-(p,v) & otherwise \end{cases}$$
$$h^-(p,v)(1) = 0$$

$$h^+(p,v)(\omega,s) = \begin{cases} \omega \xrightarrow{\{n+v \mid n\in s\}} Id & if\, \omega = p \\ \omega \xrightarrow{s} h^+(p,v) & otherwise \end{cases}$$
$$h^+(p,v)(1) = 0$$

Note that this definition arc by arc of the semantics is 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 [12], and are not presented here for sake of simplicity.

Locals and *Succ* are then defined as compositions of these inductive homomorphisms. We use $\bigcirc_{h\in H}$ to denote the composition by $\circ$ of the homomorphisms h in the set H.

$$Locals = \sum_{t\in T\setminus L} (\bigcirc_{p\in P} h^+(p, Post(p,t)) \circ h^-(p, Pre(p,t)))$$

$$Succ(\tau) = \bigcirc_{p\in P}(\bigcirc_{t\in\tau}(h^+(p, Pre(p,t)) \circ \bigcirc_{t\in\tau}(h^-(p, Pre(p,t)))))$$

For instance the transition *hungry* in the model of Fig. 1, 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.

Composite The $Next_I$ function is defined using the $\mathcal{L}$ homomorphism introduced in section 2.2. For any $\tau \in \mathrm{Bag}(T)$:

$$Next_I(\tau) = \bigcirc_{i\in I} \mathcal{L}(i, type(i).Succ((\bigoplus_{t\in\tau} t)(i)))$$

The homomorphisms representing *Locals* and *Succ* , $\forall\tau \in \mathrm{Bag}T$, are encoded:

$$\begin{array}{lcl} Locals & = & \sum_{i\in I} \mathcal{L}(i, type(i).Locals) + \sum_{t\in S.T, V(t)=private} Next_{C.I}(\{t\}) \\ Succ(\tau) & = & Next_{C.I}(\tau) \end{array}$$

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 [10].

Figure 2 present a PhiloFork module. The private synchronization *getInt* of this composite net synchronizes transition *p.getL* with *f.get*. This transition corresponds to the philosopher *p* picking up the fork to his right.

The homomorphism encoding this transition *getInt* is written :

$$\begin{aligned} Next_I(\{getInt\}) &= \mathcal{L}(Succ(get), f) \circ \mathcal{L}(Succ(getL), p) \\ &= \mathcal{L}(h^-(Res, 1), f) \circ \mathcal{L}(h^+(HasR, 1) \circ h^-(WaitL, 1), p) \end{aligned}$$

4.4. 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 [5]. Efficient algorithms to produce witness (counter-example) traces also exist [11] and can be implemented using SDD.

5. 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* [5] and *c* [19] are algorithms from the literature. Variant *d*, together with automatic optimizations, is our contribution and will be presented in the next section.

5.1. Symbolic transitive closure (1991) [5]

Variation *a* is adapted 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 [5]. 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 in SDD. However, simply computing this union T has been shown in some cases to be intractable (leading to more elaborate partitioning algorithms [6]).

Algorithm 1: Four variants of a transitive closure loop.

```
Data: {Hom} $T$ : the set of transitions encoded as $h_{Trans}$ homomorphisms
$ $s_0$ : initial state encoded as an SDD
$ $todo$ : new states to explore
$ $reach$ : reachable states

a) Explicit reachability style
begin
    $todo := s_0$
    $reach := s_0$
    while $todo \neq 0$ do
        $ $tmp := T(todo)$
        $todo := tmp \setminus reach$
        $reach := reach + tmp$
end

b) Standard symbolic BFS loop
begin
    $todo := s_0$
    $reach := 0$
    while $todo \neq reach$ do
        $reach := todo$
        $todo := todo + T(todo) \equiv (T + Id)(todo)$
end

c) Chaining loop
begin
    $todo := s_0$
    $reach := 0$
    while $todo \neq reach$ do
        $reach := todo$
        for $t \in T$ do
            $todo := (t + Id)(todo)$
end

d) Saturation enabled
begin
    $reach := (T + Id)^{\star}(s_0)$
end
```

5.2. Chaining (1995) [19]

An intermediate approach is to use clusters. Transition clusters are defined and a DD representing each cluster is computed using union. This produces 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 [19] 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*.

5.3. Saturation (2001) [10]

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 [10]) on a full page that defines complex mutually recursive procedures, and would not fit here. Furthermore, DD packages such as *CUDD* or *Buddy* [20, 17] do not provide in their public API the possibility of such fine manipulation of the evaluation procedure, so the algorithm of [10] cannot be easily implemented using those packages.

5.4. Our Contribution

All these algorithm variants, including saturation (see [13]), 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 currently available: saturation. We thus overcome the limits of other DD packages, by implementing saturation *inside* the library.

6. 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*.

6.1. Intuition

The key idea behind exploiting local invariance is the *propagation* of operations. Indeed, often operations representing transitions do not affect all variables of the state signature. Thus the homomorphism representing the transition can be propagated, skipping the variables which are not relevant for the transition. This allows to limit the number of (useless) intermediate nodes created during an application of a transition relation.

Let us consider the two following homomorphisms *leq* and *inc*. *leq* returns all assignments sequences in which all values of the variable d are less or equal than k, while inc_d increments all values of d.

$$\begin{aligned}
leq(x,k)(\omega,s) &= \begin{cases} \omega \xrightarrow{\{n \mid n\in s \wedge n\leq k\}} Id & if\, \omega = x \\ \omega \xrightarrow{s} leq(x,k) & else \end{cases} \\
leq(x,k)(1) &= 1
\end{aligned}$$

$$\begin{aligned}
inc(x)(\omega,s) &= \begin{cases} \omega \xrightarrow{\{n+1 \mid n\in s\}} Id & if\, \omega = x \\ \omega \xrightarrow{s} inc(x) & else \end{cases} \\
inc(x)(1) &= 1
\end{aligned}$$

Suppose we have the transition $f_d = leq(d,2)\circ inc(d)$ to apply on S_1 of the figure 6. We want the full state space, i.e. $(f_d + Id)^\star(S_1)$. A basic BFS application would produce intermediate SDD S_2 to S_4. However, if the evaluation mechanism could know that variables a, b and c are not relevant for the

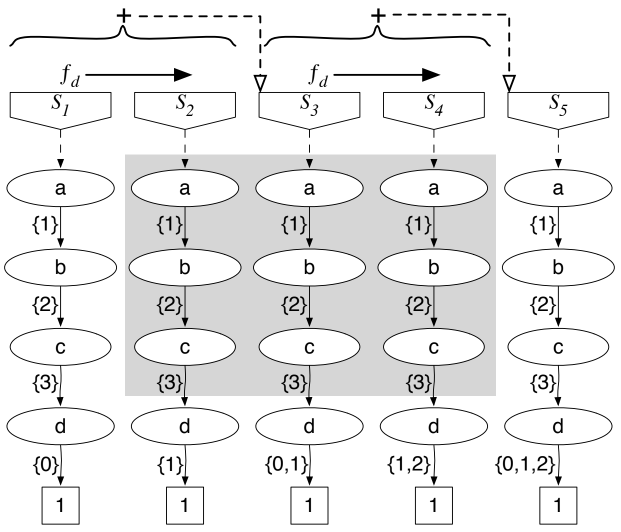

Figure 6. Effects of propagation

operation, we could propagate $(f_d + Id)^\star$ down to the d node of S_1, work on that node until the * fixpoint is reached, then reconstruct the top of the SDD of S_5. This avoids creation of all the intermediate nodes outlined in grey.

The next subsections formalizes this intuition, allowing to embed this logic in the SDD library.

6.2. 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 6.1. (Locally invariant homomorphism)
An homomorphism h is locally invariant on variable ω iff

$$\forall \delta = \langle \omega, \pi, \alpha \rangle \in \mathbb{S},\ h(\delta) = \sum_{\langle s,\delta' \rangle \in \alpha} \omega \xrightarrow{s} h(\delta')$$

Concretely, this means that the application of h doesn't modify the structure of nodes of variable ω, and h is not modified by traversing these nodes. The variable ω is a "don't care" w.r.t. operation h, it is neither written nor read by h. A standard DD encoding [10] 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 ω, it means that $h(\omega, s) = \omega \xrightarrow{s} h$. A user defining an inductive homomorphism h should provide a predicate $Skip(\omega)$ that returns *true* if h is locally invariant on variable ω. This minimal information will be used to reorder the application of homomorphisms to produce a saturation effect. It is not difficult when writing an 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 4 can exhibit the locality of their effect on the state signature by defining *Skip*, which removes the test $\omega = p$ w.r.t. the previous definition since p is the only variable that is not *skipped*:

$$h^-(p,v)(\omega,s) = \omega \xrightarrow{\{n-v \mid n\in s \wedge n\geq v\}} Id$$
$$h^-(p,v).Skip(\omega) = (\omega \neq p)$$
$$h^-(p,v)(1) = 0$$

$$h^+(p,v)(\omega,s) = \omega \xrightarrow{\{n+v \mid n\in s\}} Id$$
$$h^+(p,v).Skip(\omega) = (\omega \neq p)$$
$$h^+(p,v)(1) = 0$$

An inductive homomorphism ϕ's application to $\delta = \langle\omega,\pi,\alpha\rangle$ is defined by $\phi(\delta) = \sum_{\langle s,\delta'\rangle\in\alpha} \Phi(\omega,s)(\delta')$. But when Φ is invariant on ω, computation of this union produces the expression $\sum_{\langle s,\delta'\rangle\in\alpha} \omega \xrightarrow{s} \Phi(\delta')$. 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 $\phi(\delta')$ produces the 0 terminal, we prune the arc. Else, if two $\phi(\delta')$ applications return the same value, we need to fuse the arcs into an arc labeled by the union of the arc values. We thus avoid computing the expression $\sum_{\langle s,\delta'\rangle\in\alpha} \phi(\omega,s)(\delta)$, which involves creation of intermediate single arc nodes $\omega \xrightarrow{s} \cdots$ and their subsequent union. The impact on the efficiency of this "in place" evaluation is already measurable, but more importantly it enables the next step of rewriting rules.

6.3. 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 ω if their operands are themselves invariant on ω.

This property derives from the definition (given in [12, 13]) of the basic set theory operations on DDD and SDD. Indeed for two homomorphisms h and h' locally invariant on variable ω we have: $\forall\delta = \langle\omega,\pi,\alpha\rangle \in \mathbb{S}$,

$$\begin{array}{rcl}(h+h')(\delta) & = & h(\delta)+h'(\delta) \\ & = & \sum_{(s,\delta')\in\alpha} \omega \xrightarrow{s} h(\delta') + \sum_{(s,\delta')\in\alpha} \omega \xrightarrow{s} h'(\delta') \\ & = & \sum_{(s,\delta')\in\alpha} \omega \xrightarrow{s} h(\delta') + h'(\delta') \\ & = & \sum_{(s,\delta')\in\alpha} \omega \xrightarrow{s} (h+h')(\delta')\end{array}$$

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 $h(\delta)+h'(\delta)$ 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 n-ary union $H(\delta) = \sum_i h_i(\delta)$ on a node $\delta = \langle\omega,\pi,\alpha\rangle$ we partition its operands into $F = \{h_i | h_i.Skip(\omega)\}$ and $G = \{h_i | \neg h_i.Skip(\omega)\}$. We then rewrite the union $H(\delta) = (\sum_{h\in F} h)(\delta) + (\sum_{h\in G} h)(\delta)$, or more simply $H(\delta) = F(\delta) + G(\delta)$. The F union is thus locally invariant on ω and will continue evaluation as a block. The G part is evaluated using the standard definition $G(\delta) = \sum_{h\in G} h(\delta)$

Thus the minimal *Skip* predicate allows us 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 n-ary union are computed at most once per variable instead of once per node.

The computation using the definition of $H(\delta) = \sum_i h_i(\delta)$ requires each h_i to separately traverse δ, and forces to fully rebuild all the $h_i(\delta)$. 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.

6.4. Fixpoint

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

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

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 [9], 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:

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

6.5. Local Applications

We have additional rewriting rules specific to SDD homomorphisms and the $\mathcal{L}$ local construction (see section 2.2):

$$\begin{aligned}&\mathcal{L}(h,var)(\omega,s) = \omega \xrightarrow{h(s)} Id\\ &\mathcal{L}(h,var).Skip(\omega) = (\omega \neq var)\\ &\mathcal{L}(h,var)(1) = 0\end{aligned}$$

Note that h is an homomorphism, and its application is thus linear to the values in s. Further a $\mathcal{L}$ 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 :

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

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 $\mathcal{L}(h,v)$ 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 $(H+Id)^\star(\delta)$ for node $\delta=\langle\omega,\pi,\alpha\rangle$: we consider $H(\delta)=F(\delta)+L(\delta)+G(\delta)$ where F contains the locally invariant part, $L=\mathcal{L}(l,\omega)$ represents the operations purely local to the current variable ω (if any), and G contains operations which affect the value of ω (and possibly also other variables below). Thanks to rule (4) above, we can write :

$$\begin{array}{rcl}
(H+Id)^\star(\delta) & = & (F+L+G+Id)^\star(\delta)\\
& = & (G+Id+(L+Id)^\star+(F+Id)^\star)^\star(\delta)\\
& = & (\bigcirc_{g\in G}(g+Id)\circ\mathcal{L}((l+Id)^\star,\omega)\circ(F+Id)^\star)^\star(\delta)
\end{array}$$

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.

7. Efficiency of Automatic Saturation

SDD and automatic saturation have been implemented in the C++ `libddd` library [18], available under the terms of GNU LGPL. Hereafter reported results were obtained with this library on a Xeon @ 1.83GHz with 4GB of memory.

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 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).

7.1. Impact of Propagation

We have first measured on these models 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. Although by itself this is a good optimization, automatic saturation allows to gain orders of magnitude in both memory and time.

7.2. Impact of Hierarchy and Automatic Saturation

Table 1 shows the results obtained 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 a flat encoding. A such encoding means that we do not use the intrinsic hierarchy of models.

		Final #		Hierarchical *Chaining Loop*			Flat *Automatic Sat.*			Hierarchical *Automatic Sat.*		
Model Size	States #	Flat	Hier.	T. (s)	Mem. (MB)	Peak #	T. (s)	Mem. (MB)	Peak #	T. (s)	Mem. (MB)	Peak #
LOTOS Specification												
	9.8×10^{21}	–	1085	–	–	–	–	–	–	1.47	74.0	1.1×10^{5}
Dining Philosophers												
100	4.9×10^{62}	2792	419	1.9	112	2.8×10^{5}	0.2	20	18040	0.07	5.2	4614
200	2.5×10^{125}	5589	819	7.9	446	1.1×10^{6}	0.7	58.1	36241	0.2	10.6	9216
1000	9.2×10^{626}	27989	4019	–	–	–	14	1108	1.8×10^{5}	4.3	115	46015
4000	7×10^{2507}	–	16019	–	–	–	–	–	–	77	1488	1.8×10^{5}
Slotted Ring Protocol												
10	8.3×10^{09}	1283	105	1.1	48	90043	0.2	16	31501	0.03	3.5	3743
50	1.7×10^{52}	29403	1345	–	–	–	22	1054	2.4×10^{6}	5.1	209	4.6×10^{5}
100	2.6×10^{105}	–	5145	–	–	–	–	–	–	22	816	1.7×10^{6}
150	4.5×10^{158}	–	11445	–	–	–	–	–	–	60	2466	5.6×10^{6}
Kanban												
100	1.7×10^{19}	11419	511	12	145	2.6×10^{5}	2.9	132	3.1×10^{5}	0.4	11	14817
200	3.2×10^{22}	42819	1011	96	563	1×10^{6}	19	809	1.9×10^{6}	2.2	37	46617
300	2.6×10^{24}	94219	1511	–	–	–	60	2482	5.7×10^{6}	7	78	1.0×10^{5}
700	2.8×10^{28}	–	3511	–	–	–	–	–	–	95	397	5.2×10^{5}
Flexible Manufacturing System												
50	4.2×10^{17}	8822	917	13	430	5.3×10^{5}	2.7	105	2.2×10^{5}	0.4	16	23287
100	2.7×10^{21}	32622	1817	–	–	–	19	627	1.3×10^{6}	1.9	50	76587
150	4.8×10^{23}	71422	2717	–	–	–	62	1875	3.8×10^{6}	5.3	105	1.6×10^{5}
300	3.6×10^{27}	–	5417	–	–	–	–	–	–	33	386	5.9×10^{5}

Table 1. Impact of hierarchical decision diagrams and automatic saturation

All "–" entries indicate that the state space's generation did not finish because of the exhaustion of the computer's main memory. We have not reported results for flat representation with a chaining loop generation algorithm as they were nearly always unable to handle models of big size.

The *"Final"* grey columns show the final number of decision diagram nodes needed to encode the state spaces for hierarchical and flat 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 efficiency 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 5).

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 "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.

8. Recursive Folding

In this section we show how SDD allow us 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.

8.1. A group of Philosophers

We now introduce a composite type definition *PhiloForkGroup* to represent a *group* of philosophers (figure 7 left), so that it is identical to a single *PhiloFork* as defined in figure 3 (from the point of view of the IPN type contract).

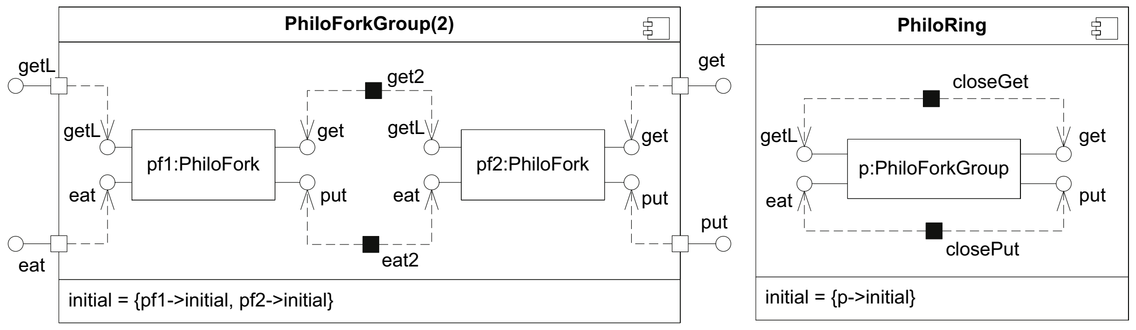

Figure 7. Module containing one *Fork* and one *Philo*

We further introduce (figure 7 right) a composite type definition *PhiloRing* to "close the loop", and connect the first and last philosophers of a group.

These two type definitions can be adapted by setting the type of the instances they contain. In particular, since *PhiloFork* and *PhiloForkGroup* have the same transition labels and initial states, one can build a *PhiloForkGroup* (like in figure 7) that contains instances of (smaller) *PhiloForkGroup* rather than instances of *PhiloFork*.

Such an encoding can be extremely compact. Suppose the initial state of one *PhiloFork* is noted (P_0). The initial state of a *PhiloForkGroup(2)* is noted $M_2 = (P_0) /\!/ (P_0)$. Then the state of *PhiloForkGroup(4)* is noted $M_4 = (M_2) /\!/ (M_2)$. The state of *PhiloForkGroup(8)* would be $M_8 = (M_4) /\!/ (M_4)$, etc... 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.

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 with a group definition containing a single PhiloFork and a PhiloFrokGroup(4). Such unbalanced depth in the data does not increase computational complexity.

8.2. Experimentations

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 philosophers.

Nb. Philosophers	States	Time (s)	Final	Peak
2^{10}	1.02337×10^{642}	0.0	124	814
2^{31}	$1.63233 \times 10^{1346392620}$	0.02	282	2347
2^{1000}	*N/A*	0.81	8034	73084
2^{10000}	*N/A*	9.85	80034	730084
2^{20000}	*N/A*	20.61	160034	1460084

Table 2. Performance evaluation 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.

What is surprising in this problem instance, is that each half of the philosophers at any level actually behaves in the same way as the other half. This is beyond a structural symmetry to global behavioral symmetry. Also when the system is evolving, few system-wide dependencies emerge, the evolution of a Philo depends on its immediate neighbors, not much on people across the table.

The solution presented here is specific to the philosophers problem, though it can be adapted to other symmetric problems. Its efficiency here is essentially due to the inherent properties of the model under study. In particular the strong locality, symmetry and the fact that even in a BDD representation, adding philosophers does not increase the “width” of the DDD representation – only it's height –, are the key factors.

The encoding presented here can be used for other very regular systems. However even when such a recursive encoding is possible, logarithmic complexity is not guaranteed. Even when the memory complexity is low, an asymmetry of the initial state may bound optimal complexity to linear (n iterations to pass a token around a ring for instance).

Our current research direction consists in defining a translation pattern from higher level notations that express symmetries (e.g. Well-Formed Nets [8]) to IPN. Such a translation could help recognize this pattern and obtain the recursive encoding automatically.

In any case, this example reveals that SDD are potentially exponentially more powerful than other decision diagram variants without hierarchy.

9. 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 [7] or CO-OPN [2].

We have presented how we optimize evaluation of user operations to automatically produce 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 the extremely efficient saturation approach of Ciardo et al. [10] 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.

We have shown how to build a symbolic model-checker that exploits a hierarchical model definition. To this end we introduced Instantiable Petri Nets, based on a general compositional notion of type and instance. Petri nets were used as elementary component type.

Finally, we have shown how recursive folding allows in very efficient and elegant manner to generate state spaces of some 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 [12], `libddd` is today a viable alternative to Buddy [17] or CUDD [20] for developers wishing to take advantage of symbolic encodings to build a model-checker.

References

[1] Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., McKenzie, P.: *Systems and Software Verification: Model Checking Techniques and Tools*, chapter Model Checking, Springer Verlag, 2001, 39–46.

[2] Biberstein, O., Buchs, D., Guelfi, N.: Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism, 2001.

[3] Bollig, B., Wegener, I.: Improving the Variable Ordering of OBDDs Is NP-Complete, *IEEE Trans. Comput.*, **45**(9), 1996, 993–1002, ISSN 0018-9340.

[4] Bryant, R.: Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers*, **35**(8), August 1986, 677–691.

[5] 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.

[6] Burch, J. R., Clarke, E. M., Long, D. E.: Symbolic Model Checking with Partitioned Transistion Relations, *VLSI*, 1991.

[7] Cabac, L., Duvigneau, M., Moldt, D., Rölke, H.: Modeling Dynamic Architectures Using Nets-within-Nets, *Applications and Theory of Petri Nets 2005, ICATPN 2005, Miami, USA*, 3536, 2005.

[8] Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications, *IEEE Transactions on Computers*, **42(11)**, 1993, 1343–1360.

[9] Ciardo, G.: Reachability Set Generation for Petri Nets: Can Brute Force Be Smart?, *Applications and Theory of Petri Nets 2004*, 2004, 17–34.

[10] Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound, *Tools and algorithms for the construction and analysis of systems*, Springer Varlag, LNCS 2619, 2003.

[11] Ciardo, G., Siminiceanu, R.: Using Edge-Valued Decision Diagrams for Symbolic Generation of Shortest Paths, *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, Springer-Verlag, London, UK, 2002, ISBN 3-540-00116-6.

[12] Couvreur, J.-M., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.-A.: Data Decision Diagrams for Petri Net Analysis, *ICATPN'02*, 2360, Springer-Verlag, 2002.

[13] 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.

[14] Hamez, A., Thierry-Mieg, Y., Kordon, F.: Hierarchical Set Decision Diagrams and Automatic Saturation, *Applications and Theory of Petri Nets 2008, ICATPN 2008, Xian, China*, 5062, 2008.

[15] Harel, D., Naamad, A.: The STATEMATE Semantics of Statecharts, *ACM Trans. Softw. Eng. Methodol.*, **5**(4), 1996, 293–333.

[16] Holzmann, G., Smith, M.: A practical method for verifying event-driven software, *ICSE '99: Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1999, ISBN 1-58113-074-0.

[17] 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.

[18] LIP6/Move: the libDDD environment, `www.lip6.fr/libddd`, 2007.

[19] Roig, O., Cortadella, J., Pastor, E.: Verification of asynchronous circuits by BDD-based model checking of Petri nets, *16th International Conference on the Application and Theory of Petri Nets*, 815, 1995.

[20] Somenzi, F.: CUDD: CU Decision Diagram Package (release 2.4.1), `http://vlsi.colorado.edu/fabio/CUDD/cuddIntro.html`, 2005.

[21] Wang, F.: Formal Verification of Timed Systems: A Survey and Perspective, *IEEE*, **92**(8), August 2004.