Hierarchical Set Decision Diagrams and Regular Models (DRAFT) ⋆

Yann Thierry-Mieg, Denis Poitrenaud, Alexandre Hamez, and Fabrice Kordon

Université P. & M. Curie, LIP6 - CNRS UMR 7606 - 4 Place Jussieu, Paris, France
first.last@lip6.fr

Abstract. This paper presents algorithms and data structures that exploit a compositional and hierarchical specification to enable more efficient symbolic model-checking. We encode the state space and transition relation using hierarchical Set Decision Diagrams (SDD) [9]. In SDD, arcs of the structure are labeled with sets, themselves stored as SDD.
To exploit the hierarchy of SDD, a structured model representation is needed. We thus introduce a formalism integrating a simple notion of *type* and *instance*. Complex composite behaviors are obtained using a synchronization mechanism borrowed from process calculi. Using this relatively general framework, we investigate how to capture similarities in regular and concurrent models. Experimental results are presented, showing that this approach can outperform in time and memory previous work in this area.

1 Introduction

Model checking is a formal verification approach that suffers from the state-space explosion problem. One approach which has been successfully used to tackle this problem is symbolic model-checking using binary decision diagrams [3].

Shared reduced ordered Binary Decision Diagrams (Binary DD or BDD) offer in many cases a very compact representation of a binary function of n Boolean variables, i.e. a function $\mathbb{B}^n \mapsto \mathbb{B}$. BDD rely on a unique table to avoid creating nodes more than once: the decision tree is built from the leaves (the terminals 0 and 1) up to the unique root. This yields a canonical representation for a boolean function given an ordering of its variables. Thus comparison of two BDD is of constant complexity. Using a cache, it is possible to obtain algorithms in complexity polynomial to the number of nodes in the data structure, rather than to the number of paths. For instance, union (or) and intersection (and) of two BDD a and b has a complexity proportional to the product of the number of nodes in the representation a and b.

Since their introduction, many extensions to BDD have been proposed. One family of extensions consists in Multi-Terminal DD (MTBDD [5]) a.k.a. Algebraic DD [1] which allow to represent functions $\mathbb{B}^n \mapsto \mathbb{N}$, and by extension when the set of terminals remains of manageable size $\mathbb{B}^n \mapsto \mathbb{R}$. This type of DD has been successfully used for

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

probabilistic model-checking [7], as well as being competitive with sparse representations for matrix computations [1].

Another family of extensions is Multiway DD [5], or Data DD [8] which allow to store functions $\mathbb{N}^n \mapsto \mathbb{B}$, or even $\mathbb{N}^n \mapsto \mathbb{N}$ when combining with multi-terminals. Although comparable to binary encodings when variables are bounded, they allow to handle *a priori* unbounded variables, and may provide more efficient solutions.

Finally, many dedicated data structures that use the same basic concepts of canonical representation and dynamic programming have emerged to tackle timed (e.g. Clock Difference Diagrams CDD [2], Clock Region Diagrams CRD [12]) or probabilistic systems (e.g. MatriX Decision Diagrams MxD [7]).

In hierarchical Set Decision Diagrams (SDD [9]) arcs are labeled by a set of values rather than a single valuation. They represent assignment sequences of the form $\omega_1 \in s_1; \cdots; \omega_n \in s_n$ where ω_i are variables and s_i are sets of values. Since sets are compactly represented using decision diagrams, the arcs of the structure may be labeled by SDD (or indeed any other variant of DD), introducing hierarchy in the data structure. This produces a fundamental difference with other decision diagram types by allowing similar subsystems of a larger specification to share their representation. In the case of very regular systems, they may even provide an exponential compression factor with respect to usual DD [11].

Contributions: In this paper we investigate how SDD can be used to provide an efficient representation of the state space of composite systems. We define a general framework to express systems as a composition of smaller (possibly similar) subsystems using a notion of *type* and *instance*. Subproblems are composed using an event-based synchronization model borrowed from process calculi.

We investigate *when* gains from increased sharing can be expected from using SDD and how to maximize the gain when applicable. For a standard benchmark set of parametric models borrowed from [5] we show that the SDD solution is more efficient in both time and memory than the current state of the art in symbolic representations.

Outline: Section 2 defines SDD and formalizes operations over SDD as inductive homomorphisms. Section 3 defines a general formalism that allows to closely match the requirements of SDD based solutions. Section 4 then investigates diverse ways of encoding a problem. Finally, section 5 compares the different encodings proposed across a benchmark of models.

2 Context

This section recalls the salient points of Hierarchical Set Decision Diagrams, a data structure based on the principles of decision diagram technology (node uniqueness thanks to a canonical representation, dynamic programming, ordering issues ...). 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. Usually, the next state function of a system is encoded using one or more decision diagrams, with two variables per variable of the state signature.

2.1 Set Decision Diagrams

Hierarchical Set Decision Diagrams (SDD) defined in [9], 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 1 is taken practically verbatim from [11] where it was adapted for more clarity from [9].

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 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.*

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 partition size remains finite (e.g. $]0..3],]3..+\infty]$). This feature could be used to model clocks for instance (as in [12]). 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 [9]). This feature is useful when representing dynamic structures such as queues, lists or variable size arrays.

2.2 Operations and Homomorphisms

SDD support standard set theoretic operations ($\cup, \cap, \setminus$). 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 [8].

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. We widely use the left concatenation of a single assignment $(\omega \in s)$, noted $\omega \xrightarrow{s} Id$. Many basic homomorphisms are hard-coded.

Furthermore, application-specific mappings can be defined by *inductive* homomorphisms. An inductive homomorphism ϕ is defined by its evaluation on the 1 terminal $\phi(1) \in \mathbb{S}$, and its evaluation $\Phi' = \phi(\omega, s)$ 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 3. It is defined by:

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

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 recent result is that we have defined a set of rewriting rules for homomorphisms [11], allowing to automatically make use of the decision diagram saturation algorithms originally due to Ciardo [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. Futhermore a more complex CTL model-checker can then be constructed using nested transitive closures over the transition relation or its reverse [4].

3 Instantiable Transition System

This section introduces a framework to define formalisms in a way that allows to take advantage of the characteristics of SDD. Previous manual encoding of some particular systems [11] has shown that a best case exponential compression factor can be reached by SDD with respect to other DD. To generalize these results, we define Instantiable Transition System (ITS), a minimal Labeled Transition System (LTS) style formalism that makes use of the notions of *type* and *instance* to emphasize locality of actions. This helps identify similar subproblems. The requirements we express through this formalism on the input language are sufficiently wide to encompass many types of description

languages. Any formalism that can fit this generic description is likely to gain from using SDD rather than other "flat" decision diagram types.

3.1 ITS definition

Notations: $\mathrm{Bag}(A)$ denotes a multiset over a set A. Let $\oplus$ be a commutative operation $A \times A \mapsto A$. Let $\tau \in \mathrm{Bag}(A)$, 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. We note $tuple.X, tuple.Y \ldots$ the element X (resp. $Y \ldots$) of a tuple $tuple = \langle X, Y, \cdots \rangle$.

The generic definition of an Instantiable Transition System (ITS) 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 2 sets an abstract contract or interface that must be realized by concrete ITS types. The principle is to build hierarchical models in which elementary bricks are homogeneous to composite models, as they both conform to the notion of ITS type.

Definition 2 (ITS Concepts). *An **ITS 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.*
- $Succ : S \times \mathrm{Bag}(T) \mapsto 2^S$ *is the transition function satisfying* $\forall s \in S, Succ(s, \emptyset) = \{s\}$.

*Let Types denote a set of ITS types. An **ITS instance** i is defined by its ITS type, noted* $type(i) \in Types$*. An ITS instance i may be associated to a state* $s \in type(i).S$*. We use the terminology: "assign a state s to instance i".*

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. *Succ* takes a multiset of transition labels as argument, to resolve ambiguities that may occur when synchronizing several labels of a given instance The definition of the *Succ* as returning a set of successors (and not a single successor state) offers good generality, and allows in particular to capture non-deterministic transition relations as we will show in section 3.4. This feature allows a compact transition relation representation, as shown by the example of section 4.

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. The definition of composition as a synchronization of independent effects on parts of a system (rather than

data or channel sharing) is favorable to using various verification algorithms that exploit compositional verification [5, 10] and locality of actions.

These functions will be used to define the semantics of a composite type below. To encode a system using SDD, one must define an SDD encoding of a state $s \in S$, and a homomorphism encoding of each of the two functions *Locals* and *Succ*.

As an example, consider the graphical type declaration of a process and buffer type depicted in Fig. 1. This example taken from [5] describes a round robin protocol allowing to share a single buffer to communicate among n processes. The buffer will initially be *empty*. Initially, a single process will be *active* (has the token), all others will be *passive*. This round robin model will be used as a running example through the rest of the paper.

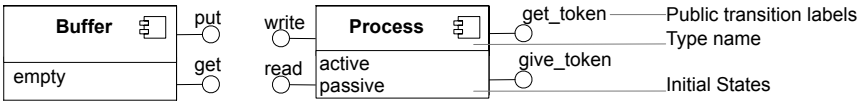

Fig. 1. Two type declarations for a resource and a process. Encapsulation makes implementation details irrelevant: only public transition labels (interface) and initial states are visible. The processes (instances of the Process type) will be connected through "get_token" and "give_token" to form a ring topology. The processes "get" ("read") the message that was "put" ("write") in the buffer by another process.

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

3.2 A Composite Type

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

Notations: Let I designate a set of ITS instances. CS_I designates the set of functions that map instances $i \in I$ to a state of $type(i)$, i.e. $cs \in CS_I \implies \forall i \in I, cs(i) \in type(i).S$. $Syncs_I$ designates the set of functions that map instances $i \in I$ to a multiset of public transition labels of $type(i)$, i.e. $t \in Syncs_I \implies \forall i \in I, t(i) \in \mathrm{Bag}(type(i).T)$. The sum $\oplus : Syncs_I \times Syncs_I \mapsto Syncs_I$ is defined as: $t = t_0 + t_1 \iff \forall i \in I, t(i) = t_0(i) + t_1(i)$ where $+$ designates the standard sum of multisets.

Intuitively, CS_I represents composite states, an element of $Syncs_I$ corresponds to a synchronization of public labels of the set I of subcomponents. The sum $\oplus$ represents an operation cumulating the effect of two synchronizations. For instance, let $I = \{i_0, i_1\}$. Let $s_0, s_1 \in Syncs_I$, $s_0(i_0) = t_0 + 2't_1$, $s_0(i_1) = \emptyset$; $s_1(i_0) = t_0$, $s_1(i_1) = t_3$. Then $s_2 = s_0 \oplus s_1 \implies s_2(i_0) = 2't_0 + 2't_1, s_2(i_1) = t_3$.

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

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

Definition 3 (Composite). *A **composite** is a tuple $C = \langle I, IS, ST, V\rangle$:*

- *I is a finite set of ITS instances, said to* be contained *by C. We further require that the type of each ITS 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 ITS 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*

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

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

Definition 3 is a realization of the generic ITS type contract. It contains either elementary subcomponents (see section 3.3), 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.

As an example consider in Fig. 2 a composite type built to represent the round robin system with two processes.

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

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

The homomorphisms representing *Locals* and *Succ* , $\forall \tau \in \text{Bag}T$, are encoded:
$Locals = \sum_{i\in I} \mathcal{L}(i, type(i).Locals) + \sum_{t\in ST, V(t)=private} Next_{C.I}(\{t\})$
$Succ(\tau) = Next_{C.I}(\tau)$

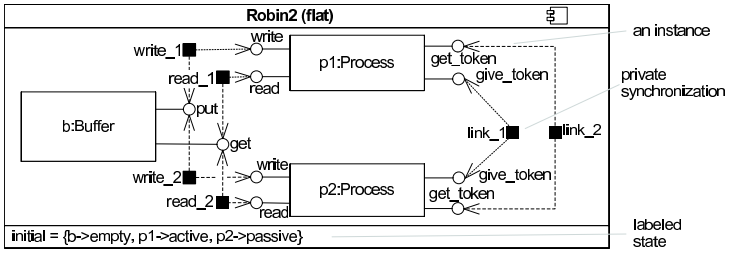

Fig. 2. A composite type declaration, containing three instances and six private synchronizations. For instance, private synchronization $link_2$ is read $link_2(p1) = get\_token, link_2(p2) = give\_token, link_2(b) = \emptyset$.

3.3 An elementary type

To have a fully working model definition, we still need to define an elementary type. We use here a labeled transition system as elementary brick, adapted to the ITS type contract. In practice any finite state model is appropriate.

Definition 4. *An elementary Labeled Transition System (LTS) is a tuple* $LTS = \langle N, N_0, L, E, V \rangle$*:*

- N *is a finite set of nodes;*
- $N_0 \subseteq N$ *is a subset of designated initial states;*
- L *is a finite set of labels for transitions;*
- $E \subseteq N \times L \times N$ *is a set of labeled edges;*
- $V : L \mapsto \{public, private\}$ *is a function assigning a* visibility *to each label;*

The ITS type which corresponds to an elementary LTS, is defined as:

- $S = N$
- $InitStates = N_0$
- $T = \{l \in L \mid V(l) = public\}$
- $Locals : S \mapsto 2^S$ is defined as $n' \in Locals(n)$ iff $\exists l \in L, V(l) =$private$\wedge \langle n, l, n' \rangle \in E$;
- $Succ : S \times \mathrm{Bag}(T) \mapsto 2^S$: $Succ(n, \tau) = \emptyset$ if τ contains more than one transition label. Else, when $\tau = \{l\}$, $\forall n, n' \in S, n' \in Succ(n, \{l\})$ iff $\langle n, l, n' \rangle \in E$.

As an example, Fig. 3 represents an implementation of the Process type introduced earlier (Fig. 1) using an LTS.

Encoding: we index the states of LTS, and use an SDD with a single variable of integer domain reflecting the current state. The transition relation is easily realized using a precomputed transition function $f : S \times L \mapsto 2^S$ such that $f(n,l) = \{n' \mid \langle n, l, n' \rangle \in E\}$. Furthermore, for any set $s \subseteq S$ we define the set $priv(s) = \bigcup_{x \in s} \bigcup_{l \in L \wedge V(l) = private} f(x,l)$.

Locals and *Succ* are defined using inductive homomorphisms. Note that as we have a single variable in the encoding these homomorphisms do not need to propagate. *Succ* is defined below when the argument $\tau \in \mathrm{Bag}(T)$ contains a single transition label l. Otherwise it returns the terminal 0.

$$\begin{cases} Locals(\omega, s) = e \xrightarrow{priv(s)} Id \\ Locals(1) \quad = 1 \end{cases} \quad \begin{cases} Succ(l)(\omega, s) = e \xrightarrow{\bigcup_{x \in s} f(x,l)} Id \\ Succ(l)(1) \quad = 1 \end{cases}$$

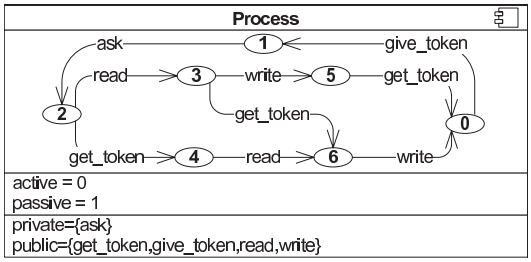

Fig. 3. An LTS representing an implementation of the Process type in the round robin protocol.

3.4 An Extended Composite Type

The concepts introduced up to this point offer basic support for the definition of hierarchical models. Extending this definition is possible provided that the ITS type contract is preserved. We consider here such an extension, which proposes an additional type of synchronization to handle non-determinism.

This additional construct allows more compact modeling, and a more efficient encoding of the transition relation. An example using this extended composite definition and illustrating its benefits is presented in section 4.

Notations: We define an additional operator to combine $Syncs_I$. Let the product $\otimes : 2^{Syncs_I} \times 2^{Syncs_I} \mapsto 2^{Syncs_I}$ be defined as:

$$\forall A, B \subseteq Syncs_I, A \otimes B = \{a \oplus b \mid a \in A \wedge b \in B\}$$

We then define a slightly more complex composite type :

Definition 5. *A **non-deterministic composite** is a tuple $NDC = \langle C, K \rangle$:*

- *C is a composite type*
- *$K : C.T \mapsto \{AND, XOR\}$ partitions transitions into basic AND kind synchronizations and non deterministic choice XOR kind synchronizations;*

We define the determinize function $Det : \mathrm{Bag}(T) \mapsto 2^{Syncs_I}$:

$$Det(\tau) = \begin{cases} \{\bigoplus_{\{t \in \tau | K(t)=AND\}} t\} \otimes \bigotimes_{\{t \in \tau | K(t)=XOR\}} t & \text{if } \exists t \in \tau, K(t) = XOR \\ \{\bigoplus_{\{t \in \tau | K(t)=AND\}} t\} & \text{otherwise} \end{cases}$$

Det allows to map the semantics of XOR synchronizations to a set of deterministic AND synchronizations. To realize the ITS type definition, we define:

- *$S = C.S$, $InitStates = C.InitStates$, $T = C.T$*
- *$Locals : S \mapsto 2^S$. $\forall s, s' \in S, s' \in Locals(s)$ iff*

$$\begin{cases} \exists i \in C.I, s'(i) \in type(i).Locals(s(i)) \wedge \forall j \in C.I, j \neq i, s'(j) = s(j) \\ or\ \exists \theta \in C.ST, C.V(\theta) = private, \exists t \in Det(\theta), s' \in Next_{C.I}(s, \{t\}) \end{cases}$$

- *$Succ : S \times \mathrm{Bag}(T) \mapsto 2^S$. $\forall s, s' \in S, \forall \sigma \in \mathrm{Bag}(T)$, $s' \in Succ(s, \sigma)$ iff* $\exists \tau \in Det(\sigma), s' \in Next_{C.I}(s, \tau)$.

Note that an extended composite in which no disjunctive synchronizations are defined is identical to definition 3. However, the extended definition introduces "exclusive or" type synchronizations, in which only one of the transition labels that belong to the set $Syncs_I$ is required to occur when the synchronization occurs. The transition label is chosen arbitrarily. The function *Det* selects one transition label from each *XOR* synchronization, and all transition labels from the *AND* transitions. Its output is a set of $Syncs_I$ that can be used to define a Successors rule using the same *Next* function as definition 3.

Encoding: The state encoding is the same as the encoding for a (basic) composite. The homomorphisms representing *Locals* and *Succ* , $\forall \sigma \in \mathrm{Bag}T$, are encoded:
$Locals = \sum_{i \in I} \mathcal{L}(i, type(i).Locals) + \sum_{\tau \in ST, V(\tau) = private} (\sum_{t \in Det(\tau)} Next_{C.I}(\{t\}))$
$Succ(\sigma) = \sum_{\tau \in Det(\sigma)} Next_{C.I}(\tau)$

4 Hierarchical Modeling Strategies

ITS allow to model a given system in a number of equivalent ways, depending on the hierarchy of types that is defined. One way of seeing this is that ITS offer to *parenthesize* a parallel composition of *n* processes. Flattening the representation is always possible, yielding an equivalent composite ITS containing only instances of an elementary type. This can be seen as removing parenthesis from the expression of the synchronization, which does not affect the resulting semantics. However, a model's hierarchy allows to factorize description of similar structures and behaviors. This is exploited to provide a more efficient SDD solution for model-checking.

For instance, to obtain an homogeneous representation of a chain of processes and a single process, we can define a type *ProcessGroup* (Fig. 4) to contain a set of process instances. This homogeneous representation is only possible thanks to the use of XOR synchronizations; a simpler encoding using only AND synchronizations would require that a process group containing *n* process instances make visible *n* versions of the *write* and *read* transitions.

The process group is then identical to a process, from the ITS point of view that sees only public transition labels and designated initial states. We will note such a composite $M2 = (P /\!\!/ P)$, where *P* represents an elementary process type, and $/\!\!/$ denotes a parallel composition. This homogeneous representation of a set of processes allows to build a larger process group by combining two process groups using the same schema. For instance, we can define $M4 = (M2 /\!\!/ M2)$ using two instances of the type defined in Fig. 4 to represent 4 process.

We can then define (Fig. 5) a *ProcessRing* type that models the boundary synchronizations necessary to close a process ring.

We compare here two approaches to encode a system of *n* process. The Recursive(*grain*) approach consists in building process groups such that no process group definition ever contains more than *grain* process group instances.

The Group(*grain*) approach consists in building a process group type containing $n/grain$ subgroups of sizes ranging from *grain* to $grain + 1$. The subgroups are defined in this approach as a simple composition of *grain* (or $grain + 1$) elementary Process

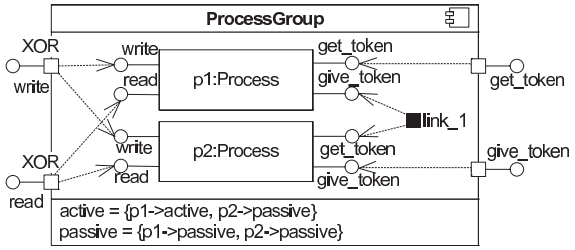

Fig. 4. An extended composite ITS representing a group of Process in a manner homogeneous to a single Process. This pattern can be generalized to contain *k* instances rather than just two.

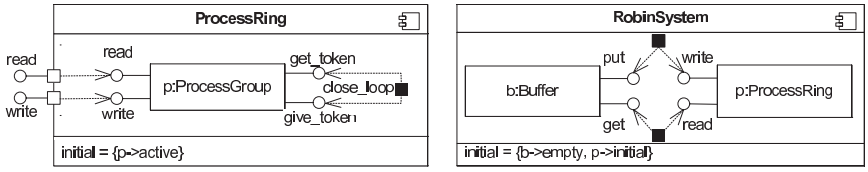

Fig. 5. A composite ITS representing the actions closing a Process ring and a full system as a composite of a Buffer and a ProcessRing.

instances. The overall depth is thus two: the Process group composite contains subgroup composite instances that contain elementary type instances.

For instance, let us compare for $n = 18$ the encodings of Robin(18). Recursive(2) would yield $M2 = (P /\!\!/ P), M4 = (M2 /\!\!/ M2), M8 = (M4 /\!\!/ M4), M16 = (M8 /\!\!/ M8)$, and finally $M18 = (M16 /\!\!/ M2)$. Recursive(3) would yield $M3 = (P /\!\!/ P /\!\!/ P), M9 = (M3 /\!\!/ M3 /\!\!/ M3), M18 = (M9 /\!\!/ M9)$ with more shallow hierarchy. Group(4) would build a model $M4 = (P /\!\!/ P /\!\!/ P /\!\!/ P), M5 = (P /\!\!/ P /\!\!/ P /\!\!/ P /\!\!/ P), M18 = (M5 /\!\!/ M5 /\!\!/ M4 /\!\!/ M4)$.

Comparing Strategies The Recursive approach fixes the maximum number of variables in an SDD assignment sequence, and lets "depth" in the hierarchy grow with $log_{grain}(n)$. The Group approach fixes the depth at 2 and fixes the length of assignment sequences in the "deeper" level to *grain*. The number of variables in the outer level thus grows with $n/grain$.

We have run experiments with these strategies, for three examples taken from Smart benchmarks [5]. **Robin** is the protocol we have used as running example in this paper. **Philosophers** models Dijkstra's classical dining philosophers. **Slotted Ring** describes a ring communication protocol with one slot per participant. These three models are regular, i.e. parametric in the number of participants in the protocol. They are thus all appropriate to apply our encoding strategies.

We report performance for Robin using the XOR construct. Without it, locality of events which is critical to saturation performance is broken. As a result, even if the final state space representation size is the same, the poorer encoding of the transition relation yields high degree polynomial complexity in *n* due to peaks in representation size.

Table 1 presents the results obtained with two of these regular models. In this experiment we let the grain vary in both Recursive and Group approaches.

		Recursive				Group					
grain ⇒		3		5		1		5		10	
Model Size	States #	T. (s)	Mem. (MB)	T. (s)	Mem. (MB)	T. (s)	Mem. (MB)	T. (s)	Mem. (MB)	T. (s)	Mem. (MB)
Slotted Ring											
50	1.7×10^{52}	4.9	59.7	2.2	48.1	2.5	112.7	1.9	55.9	2.5	50.2
100	2.6×10^{105}	94.7	463.9	27.9	300.1	19.7	814.4	14.5	410.5	17.6	342.6
200	8.4×10^{211}	-	-	489.9	2285.2	-	-	-	-	128.7	2735.7
Robin											
100	2.8×10^{32}	0.24	12.5	0.26	11.8	26.9	102.8	1.0	23.8	0.7	14.8
400	2.3×10^{123}	1.0	45.4	1.2	43.8	998.2	1118.7	62.9	318.5	15.5	169.2
1000	2.4×10^{304}	3.0	117.8	3.12	106.6	*> 1000*	*> 2473.2*	*> 1000*	*> 1718.1*	307.7	1006.5

Table 1. Compared performances of Recursive and Group approaches for a sampling of *grain* parameter. "–" entries indicate failure due to exhaustion of memory.

Our experiments shows on Robin and Philosophers that the Recursive strategy can be extremely efficient w.r.t. to the Group strategy.

The results on Philosophers have been omitted due to space constraints, but performance is sublinear $O(n)$ in the Group approach (with larger grain yielding better performance) and logarithmic $O(ln(n))$ in recursive approaches.

For Robin, the final state space representation size is $O(ln(n))$, like Philosophers. However, the assymetry of the initial state enforces n iterations to cover all positions of the token in the ring, yielding overall linear complexity in n.

However, the recursive encoding of the model description, even when it is possible (i.e. the model is regular) does not ensure that the state-space computation will be easy, as the Ring example shows. In this model the recursive approach does not yield a final representation size in $O(ln(n))$. Although the system is regular, system-wide dependencies between component states force a larger representation, in which most arcs bear a single value. The increased depth in the data structure even introduces overhead in this case, thus the Group approach is more effective.

The setting Group(1) closely mimics a flattened composition of processes of the form $(P /\!\!/ P /\!\!/ \cdots /\!\!/ P)$. Since this is the encoding other DD based tools would use (no hierarchy), it is a good baseline comparison.

We can observe that using larger variable domains (i.e. increasing the *grain*) tends to reduce complexity in both approaches. This trend is reversed when the grain is so large that the depth is very shallow in *Recursive*, or the outer assignment sequence is too short in *Group*.

5 Comparative Performance Analysis

This section presents performance comparisons of our tool that relies on ITS and SDD to the tool Smart based on MDD [5]. To allow us to easily use Smart's benchmark models, our tool uses place transition nets as an elementary type rather than LTS.

Comparison to Smart is indicated as it is, to our knowledge, the only other symbolic model-checker that uses a saturation algorithm. Comparisons to NuSMV were also performed, but are not really comparable, as without saturation it cannot compete. In fact,

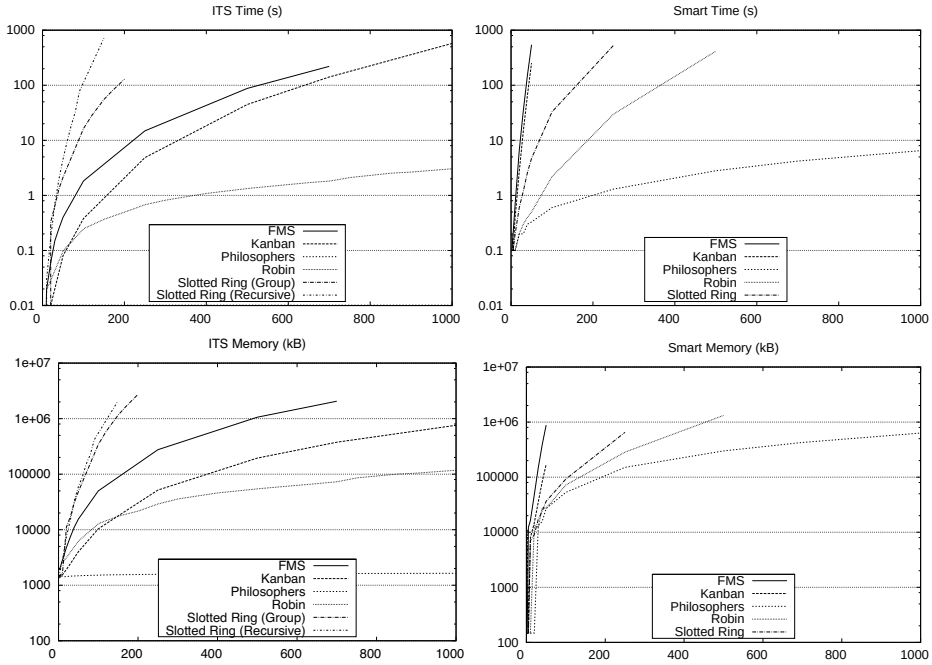

Fig. 6. Compared performances of ITS/SDD (left) vs. Smart (right), using the best settings for each tool. The *x* axis is the parameter *n* setting model's complexity. The logarithmic *y* axis represents time in seconds (top) and memory in kilo-bytes (bottom).

no answer in a reasonable time was given for the parameters we use in our benchmark. This confirms the experimentations presented in [5].

Figure 6 presents the comparisons run for five parametric models taken from the Smart's own benchmark. These models were chosen as being representative of both tools behavior, with extreme cases represented by Philo (SDD more efficient) and Ring (MDD more efficient). Three of them are the regular models introduced in section 4. In those models, the number of variables increases with *n*, while variable domains are fixed and typically small (less than 20 values). We also included two parametric models which are not regular, and could not be encoded using the approaches of the previous section. **FMS** and **Kanban** model flexible manufacturing systems. Parameter *n* defines the number of available resources rather than the size of the manufacturing plant. In these models the number of variables is fixed, while variable domains evolve with *n*.

Both tools use the best available settings, and compute the full reachable state-space. The state space size is in all cases exponential in the parameter *n*.

Philosophers: the *Recursive* approach (see section 4) is so successful that time and memory are still negligible for the value $n = 1000$. The complexity is in $O(n)$ in Smart, thanks to saturation, but it is $O(ln(n))$ with SDD. For instance, we compute reachable states of the Philosophers for $n = 10^{3000}$ in 36 seconds using 386 MB of RAM.

Robin: the complexity in Smart is high degree polynomial, where our *Recursive* solution is $O(n)$.

Ring: We report here both the results of the Recursive strategy (with grain=3) and of the Group strategy (with grain=10). Both strategies remain in high degree polynomial complexity comparable to Smart.

In this model, many arcs carry a single value even when using SDD. Since the resulting tree structure is similar, the theoretical complexity of both solutions is comparable. However, the lower memory footprint per node of MDD vs SDD factors up to give Smart the advantage. It was able to compute *Ring* up to $n = 250$ when ITS failed above $n = 200$.

Kanban and **FMS**: the superiority of SDD over MDD is clearly affirmed when variables have a large domain. In these models, the number of variables is fixed, but the variable domains grow in $O(n)$. In both these models, the complexity in Smart is near exponential where we obtain a low order polynomial.

For **Kanban**, in ITS/SDD we use the natural encoding that synchronizes four instances of a single type, yielding 4 variables of domain $O(n^3)$. However, an alternate flatter partitioning is used in Smart benchmarks, with 16 variables of domain size $O(n)$. This encoding limits the potential number of arcs per node, as the "rough" partition fails past $n \approx 50$.

Papers by Ciardo et al. on Smart (e.g. [5]) that compare the performances of a "rough" partition versus a "fine" one conclude that a fine partition is better for MDD. As our experiments section 4 show, this is not the case for SDD. Smart is based on MDD, which allow to represent integer domain variables rather than on SDD. Thus, when the size of the set of local states of a component grows (i.e. the domain of variables is large), performance is degraded.

This is due to the creation of MDD nodes having a large set of arcs. In contrast SDD are resistant to large local state spaces, as arcs are fused when they lead to the same successor node. Thus the number of arcs per node is not directly related to the number of local states, but rather to the complexity of state dependencies between components.

6 Conclusion

This paper investigates how hierarchical Set Decision Diagrams (SDD) may provide an efficient representation of the state space of composite systems. We have introduced *Instantiable Transition Systems* (ITS) as a framework to define formalisms in a way that allows to take advantage of the characteristics of SDD.

Our contributions are : *1)* we have defined encoding strategies which allow take advantage of the regularity of systems, *2)* ITS allow to capture this regularity using the notions of *type* and *instance*, *3)* the definition of ITS is generic allowing to adapt it to many formalisms.

Experimentation shows that our solution is competitive with existing symbolic solutions such as SMART or NuSMV. In the case of very regular models, we can even obtain a exponential compression factor in both time and memory.

Hierarchical Set Decision Diagrams are available as an open source C++ library `http://ddd.lip6.fr`, which has already been used to build several efficient model-checkers.

Definition of heuristics allowing to detect the regularity of a model and automatically propose an appropriate encoding strategy is left to future work.

References

1. R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.
2. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer Aided Verification, CAV'99*, LNCS, pages 341–353, 1999.
3. R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
4. J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation (Special issue for best papers from LICS90)*, 98(2):153–181, 1992.
5. G. Ciardo, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31(1):63–100, 2007.
6. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Tools and algorithms for the construction and analysis of systems*, pages 379–393. Springer Varlag, LNCS 2619, 2003.
7. G. Ciardo and A. S. Miner. Implicit data structures for logic and stochastic systems analysis. *SIGMETRICS Perform. Eval. Rev.*, 32(4):4–9, 2005.
8. J-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P-A. Wacrenier. Data Decision Diagrams for Petri Net Analysis. In *ICATPN'02*, volume 2360 of *LNCS*, pages 1–101. Springer-Verlag, 2002.
9. J-M. Couvreur and Y. Thierry-Mieg. Hierarchical Decision Diagrams to Exploit Model Structure. *Formal Techniques for Networked and Distributed Systems - FORTE 2005*, pages 443–457, 2005.
10. S. Donatelli and G. Franceschinis. The PSR Methodology: Integrating Hardware and Software Models. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, pages 133–152, London, UK, 1996. Springer-Verlag.
11. A. Hamez, Y. Thierry-Mieg, and F. Kordon. Hierarchical Set Decision Diagrams and Automatic Saturation. In *Applications and Theory of Petri Nets 2008, ICATPN 2008, Xian, China*, volume 5062 of *LNCS*, 2008.
12. Farn Wang. Formal verification of timed systems: A survey and perspective. *IEEE*, 92(8), August 2004.