Hierarchical Decision Diagrams to Exploit Model Structure

Jean-Michel Couvreur[1] and Yann Thierry-Mieg[2]

[1] Laboratoire Bordelais de Recherche en Informatique, France
Couvreur@labri.u-bordeaux.fr
[2] Laboratoire d'Informatique de Paris 6, France
Yann.Thierry-Mieg@lip6.fr

Abstract. Symbolic model-checking using binary decision diagrams (BDD) can allow to represent very large state spaces. BDD give good results for synchronous systems, particularly for circuits that are well adapted to a binary encoding of a state. However both the operation definition mechanism (using more BDD) and the state representation (purely linear traversal from root to leaves) show their limits when trying to tackle globally asynchronous and typed specifications. Data Decision Diagrams (DDD) [7] are a directed acyclic graph structure that manipulates(*a priori* unbounded) integer domain variables, and which offers a flexible and compositional definition of operations through inductive homomorphisms.

We first introduce a new transitive closure unary operator for homomorphisms, that heavily reduces the intermediate peak size effect common to symbolic approaches. We then extend the DDD definition to introduce hierarchy in the data structure. We define Set Decision Diagrams, in which a variable's domain is a *set of values*. Concretely, it means the arcs of an SDD may be labeled with an SDD (or a DDD), introducing the possibility of arbitrary depth nesting in the data structure. We show how this data structure and operation framework is particularly adapted to the computation and representation of structured state-spaces, and thus shows good potential for symbolic model-checking of software systems, a problem that is difficult for plain BDD representations.

1 Introduction

Model checking of concurrent systems is a difficult problem that faces the well known state-space explosion problem. Efficient techniques to tolerate extremely large state spaces have been developed however, using a compact representation based on decision diagrams [1,2]. However, these symbolic techniques suffer from the intermediate peak size effect : the size of intermediate results is sometimes out of proportion with the size of the result. This is particularly true of globally asynchronous systems [9]. To fight this effect, and its dual on the size of the BDD representing the transition relation, the works of [11,13,9] for example have shown how to exploit modularity to decompose a transition relation in various ways. This allows large gains with respect to a purely linear encoding in traditional BDD approaches. More recently Ciardo in [4] showed in the context of modular verification how a fixpoint evaluation that is guided by the variable ordering can dramatically reduce the peak size effect. This is due to a saturation

F. Wang (Ed.): FORTE 2005, LNCS 3731, pp. 443–457, 2005.
© IFIP International Federation for Information Processing 2005

algorithm that works from the leaves up, thus a large proportion of the nodes created at each iteration are retained in the result.

The structure of a model is essential from the architectural point of view, but is usually not well captured by symbolic representations. Indeed, in a BDD encoding, a system state is seen as a linear path that traverses all the (binary) variables that represent the system. This is a handicap for symbolic techniques when trying to tackle complex specifications, as structure information is lost in this state encoding. Gupta proposed in [10] an encoding of inductive Boolean functions using hierarchical BDD, in the context of parametric circuits. The complexity of the transition model used in that work has however prevented a more widespread use of this concept.

We define here a new hierarchical data decision structure, SDD, that allows to generalize some of these patterns of good decision diagram usage, in an open and flexible framework, inductive homomorphisms. SDD are naturally adapted to the representation of state spaces composed in parallel behavior, with event based synchronizations. The structure of a model is reflected in the hierarchy of the decision diagram encoding, allowing sharing of both operations and state representation. SDD allow to flexibly compute local fixpoints, and thus our model-checker though still very young offers performance an order above NuSMV [6] and comparable to SMaRT [4]. The DDD/SDD library is available under LGPL from `ddd.lip6.fr`.

The paper is structured as follows : we first present data decision diagrams (2.1) and labeled Petri nets (2.2) as the context in which we work. Section 3 shows how we integrated local saturation in our DDD operation framework. Section 4 introduces our new Set Decision Diagram hierarchical structure and operations. Section 5 explains how they can be used in the context of modular and hierarchical symbolic model checking for labeled transition systems, and in particular our chosen P/T nets. We give performances of our prototype in sections 3 and 5.

2 Context

2.1 Data Decision Diagram Definition

Data Decision Diagrams (DDD) [7] are a data structure for representing finite sets of assignments sequences of the form $(e_1 := x_1) \cdot (e_2 := x_2) \cdots (e_n := x_n)$ where e_i are variables and x_i are the assigned integer values. When an ordering on the variables is fixed and the values are booleans, DDD coincides with the well-known Binary Decision Diagram. When the ordering on the variables is the only assumption, DDD correspond to the specialized version of the Multi-valued Decision Diagrams representing characteristic function of sets [3]. However DDD assume no variable ordering and, even more, the same variable may occur many times in a same assignment sequence. Moreover, variables are not assumed to be part of all paths. Therefore, the maximal length of a sequence is not fixed, and sequences of different lengths can coexist in a DDD. This feature is very useful when dealing with dynamic structures like queues.

DDD have two terminals : as usual for decision diagram, 1-leaves stand for accepting terminators and 0-leaves for non-accepting ones. Since there is no assumption on the variable domains, the non-accepted sequences are suppressed from the structure. 0 is considered as the default value and is only used to denote the empty set of sequence.

This characteristic of DDD is important as it allows the use of variables of finite domain with *a priori* unknown bounds. In the following, E denotes a set of variables, and for any e in E, $\mathrm{Dom}(e) \subseteq \mathbb{N}$ represents the domain of e.

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

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

We denote $e \xrightarrow{x} d$, the DDD (e, α) with $\alpha(x) = d$ and for all $y \neq x$, $\alpha(y) = 0$. We call DDD sequence a DDD of the form $e_1 \xrightarrow{x_1} e_2 \xrightarrow{x_2} \ldots 1$.

Although no ordering constraints are given, DDD represent sets of *compatible DDD sequences*. Note that the DDD 0 represents the empty set and is therefore compatible with any DDD sequence. The symmetric compatibility property is defined inductively for two DDD sequences:

Definition 2 (Compatible DDD sequences).

- *Any DDD sequence is compatible with itself.*
- *Sequences* 1 *and $e \xrightarrow{x} d$ are incompatible*
- *Sequences $e \xrightarrow{x} d$ and $e' \xrightarrow{x'} d'$ are compatible iff. $e = e' \wedge (x = x' \Rightarrow d$ and d' are compatible)*

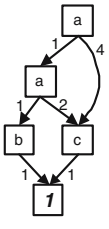

Fig. 1.
$a \xrightarrow{1} a \xrightarrow{1} b \xrightarrow{1} 1$
$+ a \xrightarrow{4} c \xrightarrow{1} 1$
$+ a \xrightarrow{1} a \xrightarrow{2} c \xrightarrow{1} 1$

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

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

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

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

2.2 Labeled P/T Nets Definition

In this section, we introduce a class of modular Petri nets. We chose P/T nets for their simple semantics, but most of what is presented here is valid for other LTS.

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

- P is a finite set of places,
- T is a finite set of transitions (with $P \cap T = \emptyset$),
- Pre and $Post : P \times T \to \mathbb{N}$ are the pre and post functions labelling the arcs.
- L is a set of labels
- $label : L \times T \to \{True, False\}$ is a function labeling the transitions.

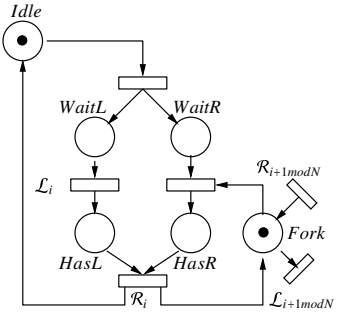

Fig. 2. Labeled PT net model of the philosophers

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

Two labeled P/T nets may be composed by synchronization on the transitions that bear the same label. This is a parallel composition noted $/\!/$, with event-based synchronizations that yields a new labeled P/T net. This is a general compositional framework, adapted to the composition of arbitrary labeled transition systems (LTS).

This paper focuses on the representation of a state-space defined by composition of LTS, more than on a given formalism. We thus limit our discussion to ordinary nets, with only constant arc functions, and only pre and post arcs. However our implementations actually encompass a wider class of nets (with FIFO, queues…) described fully in [7]. Let us consider an encoding of a state space of a P/T net in which we use one variable for each place of the system. The domain of place variables is the set of natural numbers. The initial marking for a single place is encoded by: $d_p = p \xrightarrow{m_0(p)} 1$. For a given total order on the places of the net, the DDD encoding the initial marking is the concatenation of DDD $d_{p_1} \cdots d_{p_n}$. For instance, the initial state of a philosopher can be represented by : $Idle \xrightarrow{1} WaitL \xrightarrow{0} WaitR \xrightarrow{0} HasL \xrightarrow{0} HasR \xrightarrow{0} Fork \xrightarrow{1} 1$.

The symbolic transition relation is defined arc by arc in a modular way well-adapted to the further combination of arcs of different net sub-classes. The two following homomorphisms are defined to deal respectively with the pre (h^-) and post (h^+) conditions.

Both are parameterized by the connected place (p) as well as the valuation (v) labelling the arc entering or outing p .

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

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

These basic homomorphisms are composed to form a transition relation. For a transition t, ${}^\bullet t$ (resp. $t^\bullet$) denotes the set of places $\{p \in P \mid Pre(p,t) \neq 0\}$ (resp. $\{p \in P \mid Post(p,t) \neq 0\}$). The full homomorphism h_{Trans} for a given transition t is obtained by : $h_{Trans}(t) = \bigcirc_{p \in t^\bullet} h^+(p, Post(p,t)) \circ \bigcirc_{p \in {}^\bullet t} h^-(p, Pre(p,t))$

For instance the top-most transition in the model of Fig. 2 would have as homomorphism : $h_{Trans}(t) = h^+(WaitL, 1) \circ h^+(WaitR, 1) \circ h^-(Idle, 1)$.

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.

3 Introducing Saturation

A first extension to the DDD homomorphism model was made to introduce the concept of local saturation. The idea, inspired by [4], is to compute fixpoint computations starting from internal nodes of the decision diagram structure, instead of having all fixpoint computations performed at the highest level. This is shown experimentally to considerably reduce the intermediate peak size effect that is one of the critical problems related to symbolic approaches (see [9] for a good overview of other intermediate size reduction techniques). In effect, by computing local fixpoints starting from the leaves of the structure and going up, and re-saturating lower nodes each time a variable is modified favors creation of "low" nodes that will indeed appear in the final result. The canonization process of decision diagrams is inductively based on the unicity of the terminal node(s), thus creating more saturated low nodes reduces the number of intermediate nodes at all levels in the structure.

To this end we introduce a new **transitive closure** * unary operator that allows to perform local fixpoint computations. For any homomorphism h, $h^*(d), d \in$ is evaluated by repeating $d \leftarrow h(d)$ until a fixpoint is reached. While this computation may not terminate, if it does its evaluation can be described as a finite composition using $\circ$ of h, thus is itself an inductive homomorphism. This operator is usually applied to $Id + h$ instead of h, allowing to cumulate newly reached paths in the result.

To use this operator efficiently, we need to set a starting level for each transition of the system. We define a transition t's top level, noted $top(t)$, as the variable of highest index (the last variable encountered in the DDD bears index 0) that is affected by a firing of t. We then define a table $TopTrans$ of size the number of variables of the system. This table contains in $TopTrans[i]$ a homomorphism that is a sum of the homomorphisms (constructed using the usual equation presented in section 2.2) of all transitions t such that $top(t) = i$, and of Id.

$$\mathcal{L}(e,x) = \begin{cases} e \xrightarrow{x} (TopTrans[e-1] \circ \mathcal{L}^*)^* & \text{if } e > 0 \\ e \xrightarrow{x} Id & \text{otherwise} \end{cases}$$
$$\mathcal{L}(1) = 1$$

We then define this new transition homomorphism $\mathcal{L}$ that exploits these aspects. $TopTrans$ is never modified, and is thus simply referenced in $\mathcal{L}$. The main application case consists in applying over the successor node of index $e-1$ a fixpoint of $\mathcal{L}$ (thus saturating all lower nodes of indexes strictly smaller than $e-1$) followed by all the transitions that start from node $e-1$ (plus Id) in a fixpoint. The last place in the structure is indexed by 0, and we stop the operation when we reach this level. For a system composed of N places, we compute the full state space by applying $(TopTrans[N-1] \circ \mathcal{L}^*)^*$ to the DDD representing the initial state.

Table 1 measures the impact of the use of fixpoint operators on state space construction. We use a benchmark of four models taken from [13]. Performance measures were run on a P4/2.4GHz/2GbRAM. The table shows the huge gain in complexity offered by local saturation. The number of nodes explored is however sometimes higher in the saturation approach ; this effect is explained by the fact that every step of the fixpoint computations are cached, thus some transitions may be fired more times in the version with saturation. Ciardo et al. suggest in [4] that only the result of the full fixpoint

Table 1. Comparing our tool PNDDD with saturation activated or not. Benchmark models taken from [13]. We give for each model the final number of nodes and the total number of nodes explored (i.e. constructed at some point). Garbage collecting was deactivated to allow to measure this value; this is the default behavior anyway, we lazily collect garbage only once at the end of the construction.

				PNDDD no sat		PNDDD sat	
Model	N	Nb. States	final nodes	total nodes	time (s)	total nodes	time (s)
Dining Philosophers	50	2.23e+31	1387	13123	11.6	10739	0.09
	100	4.97e+62	2787	26823	54.19	21689	0.18
	200	2.47e+125	5587	54223	234	43589	0.39
	1000	9.18e+626	27987	-	-	218789	2.1
Slotted Ring Protocol	10	8.29e+09	1281	35898	83.07	45970	0.8
	15	1.46e+15	2780	118054	595	132126	2.26
	50	1.72e+52	29401	-	-	3.58e+06	61.58
Flexible Manufacturing System	10	2.50+09	580	8604	2.06	11202	0.17
	25	8.54e+13	2545	50489	28.75	85962	1.58
	50	4.24e+17	8820	231464	240.4	490062	9.78
	80	1.58e+20	21300	-	-	1.72e+06	37.06
Kanban	10	1.01e+09	257	26862	20.47	5837	0.06
	50	1.04e+16	3217	-	-	209117	3.96
	100	1.73e+19	11417	-	-	1.32e+06	28.09
	200	3.17e+22	42817	-	-	9.23e+06	238.95

evaluation be cached, not its steps ; however we did not implement this tuning of our caching policy, which might reduce the *measure* (if we only count nodes actually created), but not truly the time/space complexity (as the extra nodes that we do count could be garbage colected at any time). Note that our fixpoint operator allows any library user to profit from leaves to root saturation instead of the traditional external fixpoint, thus generalizing the concept introduced in [4] to other applications.

The time complexity explosion in the version without local fixpoint is due to the cost of traversals of the structure and to the number of iterations required in this breadth-first evaluation scheme. The saturation process naturally applies transitions from their starting level as it is reached, whereas in a fixpoint "from outside", the transitions that target the bottom of the structure need to traverse a very large number of nodes.

4 Set Decision Diagrams

4.1 SDD Definition

DDD are a flexible data structure, and inductive homomorphisms give the user unprecedented freedom to define new symbolic operations. Since the work in [7], DDD have been used for instance to implement model checkers for a subset of VHDL in a project for the "Direction Générale des Armées" (DGA) called Clovis (the reports are however not public), for formal verification of LfP in MORSE, an RNTL project [12], and to construct a quotient state-space by exploiting the symmetries of a colored Well-Formed net model [15]. However, as we manipulated more and more complex data structures, such as the dynamically dimensioned tensors of [15], we encountered problems linked to the lack of structure of DDD. We therefore decided to extend the DDD definition to allow hierarchical nesting of DDD. This new data structure is called Set Decision Diagrams, as an arc of the structure is labeled by a *set* of values, instead of a single valuation. The set is itself represented by an SDD or DDD, thus in effect we label the arcs of our structure with references to SDD or DDD, introducing hierarchy in the data structure.

Set Decision Diagrams (SDD) are data structures for representing sequences of assignments of the form $e_1 \in a_1; e_2 \in a_2; \cdots e_n \in a_n$ where e_i are variables and a_i are sets of values. SDD can therefore simply be seen as a different encoding for set of assignment sequences of the same form as those of DDD, obtained by flattening the structure, i.e. as a DDD defined as $\bigcup_{x_1 \in a_1} \bigcup_{x_2 \in a_2} \cdots \bigcup_{x_n \in a_n} e_1 \xrightarrow{x_1} e_2 \xrightarrow{x_2} \cdots e_n \xrightarrow{x_n} 1$.

In this section we base our reasoning on the actual data structure that is used to store them, and as *sets* will label the arcs of our data structure, we use the $e_1 \in a_1 \cdots$ presentation. However the basic linearity operation properties over the sequences of the equivalent DDD must be ensured to allow correct computations. We assume no variable ordering, and the same variable can occur several times in an assignment sequence. We also make no assumptions on the domain of the variables. We encode SDD as shared decision trees. We define the usual terminals 0 and 1 to represent non-accepting and accepting sequences respectively. In the following, E denotes a set of variables, and for any e in E, Dom(e) represents the domain of e,

Definition 3 (Set Decision Diagram). *The set of SDD is defined by $d \in$ if:*

- $d \in \{0,1\}$ *or*
- $d = (e,\alpha)$ *with:*
  - $e \in E$
  - α*: is a finite set of pairs* (a_i,d_i) *where* $a_i \subseteq \mathrm{Dom}(e)$ *and* $d_i \in$

We denote $e \xrightarrow{a_i} d_i$*, the SDD* (e,α) *with* $\alpha(a_i) = d$ *and for all* $a_j \neq a_i$*,* $\alpha(a_j) = 0$*. We call SDD sequence an SDD of the form* $e_1 \xrightarrow{a_1} e_2 \xrightarrow{a_2} \ldots 1$ *where* $\forall i, |a_i| = 1$.

We further introduce a canonical representation for SDD, essential to allow use of a unicity table and cache. The SDD we manipulate are canonized by construction, through the union operation given in proposition 1 below.

Definition 4 (Canonical Set Decision Diagram). *An SDD d is said to be canonical if and only if :*

- $d = 0$ *or* $d = 1$
- $d = (e,\alpha)$ *and* $\forall (a_i,d_i),(a_j,d_j) \in \alpha, i \neq j,$ $\begin{cases} 1.\ a_i \cap a_j = \emptyset \\ 2.\ d_i \neq d_j \\ 3.\ a_i \neq \emptyset \text{ and } d_i \neq 0 \end{cases}$

Intuitively this definition sets the constraints that:

1. The number of sets of values that are mapped to a non-zero SDD be finite. This is required so that the number of arcs leading from a node be finite, since *only the arcs labeled with sets that map to a non-zero SDD are stored in the data structure*;
2. For a value x of $\mathrm{Dom}(e)$, at most one non zero SDD is associated. In other words the sets referenced on the arcs outgoing from a node are disjoint. This is required to allow existence of a unique canonical representation of sets, hence unicity and comparison of SDD nodes.
3. No two arcs from a node may lead to the same SDD. This is the crucial point, any time we are about to construct $e \xrightarrow{a} d + e \xrightarrow{a'} d$, we will construct $e \xrightarrow{a \cup a'} d$ instead. This corresponds to fusing arcs that would have led to the same node.
4. By definition, the empty set maps to 0 and is not represented.

Some immediate effects of this definition should be highlighted :

- This definition assumes that sets of values can be (efficiently) represented, as an arc of the shared decision tree representing the SDD is labeled with a set of values. As an SDD itself represents a set, we can use variables of domain itself, introducing hierarchy in the data structure.
- In practice, the requirements on the data sets that label the arcs of an SDD are that they offer the usual set theoretic operations (union, intersection and set difference) and the ability to compute a hash key for the set stored. These requirements are captured by an abstract interface class, thus labeling an SDD with any type of decision diagram (i.e. from existing libraries) should be very easy if it is written in C or C++.

– Another effect is that we no longer have the constraint of DDD that the number of values taken by a variable x be finite. This constraint expressed in DDD that the number of outgoing arcs from a node be finite, but is reduced for SDD to the constraint that the number of sets of values that lead to different nodes be finite. This subtle difference means that we could represent infinite sets provided an efficient set representation is used (intervals in $\mathbb{R}$ for instance). This possibility has not yet been fully explored, and stresses the limits of our definition however, as we can no longer consider our model equivalent to a linear DDD like finite representation.

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 5 (Compatible SDD sequences).

– *Any SDD sequence is compatible with itself.*
– 1 *and* $e \xrightarrow{a} d$ *are incompatible*
– $e \xrightarrow{a} d$ *and* $e' \xrightarrow{a'} d'$ *are compatible if* $\begin{cases} e = e' \\ \wedge a \text{ and } a' \text{ are compatible} \\ \wedge (a = a' \Rightarrow d \text{ and } d' \text{are compatible}) \end{cases}$

The compatibility of a and a′ is defined as SDD compatibility if $a, a' \in$ *or DDD compatibility if* $a, a' \in$ *. DDD and SDD are incompatible. Other possible referenced types should define their own notion of compatibility.*

4.2 Operations on SDD

Set Theoretic Operations. First, we generalize the usual set-theoretic operations – sum (union), product (intersection) and difference – to sets of set assignment sequences expressed in terms of SDD.

Definition 6 (Finite Mapping and Union). *A mapping* $\alpha : 2^{\mathrm{Dom}(e)} \rightarrow$ *is said to be finite if it respects the property that* $\{a \subseteq \mathrm{Dom}(e) | \alpha(a) \neq 0\}$ *is finite. Such a mapping has a finite number k of sets* $a_i \subseteq \mathrm{Dom}(e)$ *such that* $\alpha(a_i) \neq 0$*, and can be explicitly represented by the enumeration of the non-zero mappings it defines :* $\alpha = \bigcup_{i=1}^{k} \{a_i \rightarrow d_i\}$ *where* $\forall i, a_i \subseteq \mathrm{Dom}(e), d_i \in$ *. Let* $\alpha = \bigcup_{i=1}^{k} \{a_i \rightarrow d_i\}$ *and* $\alpha' = \bigcup_{i=1}^{k'} \{a'_i \rightarrow d'_i\}$ *be two finite mappings.*

We define the square union $\sqcup$ *as :*

$$\begin{aligned} \alpha \sqcup \alpha' &= \bigcup_{i=1}^{k} \{a_i \rightarrow d_i\} \sqcup \bigcup_{i=1}^{k'} \{a'_i \rightarrow d'_i\} \\ &= \bigcup_{i=1}^{k} \bigcup_{j=1}^{k'} \{a_i \cup a'_j \rightarrow d_i \text{ if } d_i = d'_j\} \\ &\quad \cup \bigcup_{i=1}^{k} \{a_i \rightarrow d_i \text{ if } \forall j \in [1 \cdots k'], d_i \neq d'_j\} \\ &\quad \cup \bigcup_{i=1}^{k'} \{a'_i \rightarrow d'_i \text{ if } \forall j \in [1 \cdots k], d'_i \neq d_j\} \end{aligned}$$

Intuitively this operation performs part of the son-based canonization scheme necessary for SDD : it ensures that no two arcs from an SDD node lead to the same SDD (requirement 3 of SDD definition 3). It is easily implemented by a hash map of keys d_is and values a_is. However it should be noted that this operation does not preserve requirement 2 of definition 3, as nothing ensures that the sets mentionned on the arcs

are disjoint. Indeed, a given value x in $\mathrm{Dom}(e)$ may be included in more than one a_i set of the result. But this $\sqcup$ operation over mappings will serve as a basis to define the sum $+$, the difference $\setminus$, and the product $*$ of two SDD Mappings.

Definition 7 (Compatible SDD set theoretic operations). *By definition, set theoretic operations are only offered over compatible SDD.*

- $0 + d = d + 0 = d, 0 * d = d * 0 = 0,\ 0 \setminus d = 0$ *and* $d \setminus 0 = d, \forall d \in$;
- $1 + 1 = 1 * 1 = 1,\ 1 \setminus 1 = 0$
- $\langle e, \alpha \rangle \diamond \langle e, \alpha' \rangle = \langle e, \alpha \diamond \alpha' \rangle, \forall \diamond \in \{+, *, \setminus\}$

Proposition 1 (Mapping operations). *The sum $+$ (respectively product $*$ and difference $\setminus$) of two SDD mappings $\alpha = \bigcup_{i=1}^{k} \{a_i \to d_i\}$ and $\alpha' = \bigcup_{i=1}^{k'} \{a'_i \to d'_i\}$ can be defined inductively by :*

$$\alpha + \alpha' = \bigsqcup_{i=1}^{k} \{(a_i \setminus \bigcup_{j=1}^{k'} (a'_j)) \to d_i\} \sqcup \bigsqcup_{i=1}^{k'} \{(a'_i \setminus \bigcup_{j=1}^{k} (a_j)) \to d'_i\} \sqcup \bigsqcup_{i=1}^{k} \bigsqcup_{j=1}^{k'} \{a_i \cap a'_j \to d_i + d'_j\}$$

$$\alpha * \alpha' = \bigsqcup_{i=1}^{k} \bigsqcup_{j=1}^{k'} \{(a_i \cap a'_j) \to d_i * d'_i\}$$

$$\alpha \setminus \alpha' = \bigsqcup_{i=1}^{k} \{(a_i \setminus \bigcup_{j=1}^{k'} (a'_j)) \to d_i\} \sqcup \bigsqcup_{i=1}^{k} \bigsqcup_{j=1}^{k'} \{a_i \cap a'_j \to d_i \setminus d'_j\}$$

Proof. We need to show the equivalence of the above propositions with a straight definition reasoning on the actual individual assignments in a sequence. The proof is relatively straightforward and is based on considering the different intersection possibilities between the operands' α mappings. It is omitted here due to lack of space as it requires introduction of additional definitions and notations for reasoning with the sequences of the equivalent DDD.

It should be noted that using $\sqcup$ to compose the terms constituting the result may produce some simplifications, as sets that map to the same value will be unioned, and the $d_i \diamond d'_j$ terms may produce already existing SDD. Furthermore, by definition the empty set $\emptyset$ maps to 0, this produces further simplification as both the $(a_i \setminus \bigcup_{j=1}^{k'} (a'_j))$ and the $a_i \cap a'_j$ terms are liable to be empty sets. We should remind here that the union $+$ operation defined above is the core of the canonisation procedure, as it is in charge of ensuring the canonicity of SDD by construction.

SDD Homomorphisms. By analogy with DDD, SDD allow the definition of user defined operations through a recursive and compositional definition : inductive homomorphisms. The essential constraint over homomorphisms is linearity over the set of sequences contained in an SDD. Homomorphisms can then be combined by sum and composition.

Definition 8 (Homomorphism). *A mapping Φ on SDD is a fully defined* homomorphism *if $\Phi(0) = 0$ and $\forall d_1, d_2 \in \quad : \Phi(d_1) + \Phi(d_2) = \Phi(d_1 + d_2)$*

Proposition 2 (Sum and composition). *Let Φ_1, Φ_2 be two homomorphisms. Then $\Phi_1 + \Phi_2$, $\Phi_1 \circ \Phi_2$ are homomorphisms.*

The **transitive closure** * is also introduced, and allows to perform a local fixpoint computation. It follows the same definition as for DDD transitive closure : for a homomorphism h, $h^*(d)$ is computed by repeating $d \leftarrow h(d)$ until a fixpoint is reached. Again we usually use $(h + Id)^*$ in our fixpoint computations. From here we can allow the definition of user-defined inductive homomorphisms:

Proposition 3 (Inductive homomorphism). *The following recursive definition of mappings $(\Phi_k)_k$ defines a family of homomorphisms called inductive homomorphisms.:*

$$\forall d \in \quad , \Phi_k(d) = \begin{cases} 0 & \text{if } d = 0 \\ d' \in & \text{if } d = 1 \\ \alpha' = \sum_{i=1}^{k} \Phi_k(e, a_i)(d_i) & \text{if } d = (e, \alpha = \bigcup_{i=1}^{k} \{a_i \to d_i\}) \end{cases}$$

$\Phi_k(e,a)$ is inductively defined as a sum $\Phi_k(e,a) = \sum_l \pi_l(e,a) \circ \Phi_l + \pi_0(e,a)$ where all $\pi_l(e,a)$ are SDD homomorphisms, linear over the elements of a ($\forall a, a' \subseteq \mathrm{Dom}(e)$: $\pi_l(e, a \cup a') = \pi_l(e,a) + \pi_l(e,a')$).

To define a family of inductive homomorphisms Φ, one has just to set the homomorphisms for the symbolic expression $\Phi(e, a_i)$ for any variable e and set a_i and the SDD $\Phi(1)$. It should be noted that this definition differs from the DDD inductive homomorphism in that $\Phi(e, a_i)$ is defined over the *sets* ($a_i \subseteq \mathrm{Dom}(e)$) of values of the variable e's domain $\mathrm{Dom}(e)$. This is a fundamental difference as it requires Φ to be defined in an ensemblist way: we cannot by this definition define the evaluation of Φ over a single value of e. However Φ must be defined for the *set* containing any single value.

In addition we must respect the linearity constraint over the sequences of the equivalent DDD. Thus $\pi(e,a)$ must be an SDD homorphism linear over the element of a.

We use most commonly homomorphisms of the form $e \xrightarrow{\phi(a)} Id$ which allows a linear operation on the values labeling the arc, and by composition with another inductive homomorphism, to realize an operation on the rest of the paths of the SDD.

As in [13], we require that the partition of the system into modules be *consistent*. This constraint allows the definition of a transition relation $\mathcal{N}$ in a partitioned disjunctive or conjunctive (i.e. $\mathcal{N} = \bigwedge_i \mathcal{N}_i$) form [11]. This allows one not to explicitly construct the full BDD (or Kronecker representation [13,5]) that corresponds to $\mathcal{N}$, allowing to tackle larger systems. In effect, consistency means each term composing $\mathcal{N}$ can be evaluated independently and in any order, and $\mathcal{N}(S) = \bigcap_i \mathcal{N}_i(S)$. For our ordinary Petri net model, this is not a problem as any partition is consistent [13], however more complex operations require some care in the definition of modules.

5 SDD and Modular Petri Nets

In this section we present how the SDD hierarchy can be exploited to efficiently generate and store the state-space of a labeled Petri net, itself a composition of labeled Petri nets.

In previous works, we had shown how to use DDD for non-modular Petri nets [7]. The key idea is to use a variable to represent the state of a *set* of places, instead of having one variable per place of the net. In Miner and Ciardo's work on Smart [13], a variable represented a set of places or module, but the states of the module were represented in an explicit fashion using splay trees. Here we propose a purely symbolic approach as we use an SDD variable to represent the state of a module entering the composition of the full model, the value domain of which is a DDD with one variable per place of the module.

Definition 9 (Structured state representation). *Let M be a labeled P/T net, we inductively define its representation $r(M)$ by :*

- *If M is a unitary net, we use the encoding of section 2.2 $r(M) = d_{p_1} \cdot d_{p_2} \cdots d_{p_n}$, with $d_p = p \xrightarrow{m_0(p)} 1$.*
- *If $M = M_1 /\!\!/ M_2$, $r(M) = m_{M_1} \xrightarrow{r(M_1)} m_{M_2} \xrightarrow{r(M_2)} 1$. Thus the parallel composition of two subnets will give rise to the concatenation of their representations.*
- *If $M = (M_1)$, $r(M) = m_{(M_1)} \xrightarrow{r(M_1)} 1$. Thus parenthesizing an expression gives rise to a new level of hierarchy in the representation.*

A state is thus encoded hierarchically in accordance with the module definitions. We define a total order over the N subnets or modules composing a model, used to index these submodels. Indeed the parallel composition operation is commutative and symmetric, therefore a net can always be seen as a "flat" parallel composition of its subnets. Such a composition which does not use any parenthesizing, would produce a DDD representation. However, using different parenthesizing(s) yields a more hierarchical vision (nested submodules), that can be accurately represented and exploited in our framework.

Thus for a parenthesizing of the composition in the manner $M_0 /\!\!/ (M_1 /\!\!/ (M_2 \cdots /\!\!/ (M_{n-1} /\!\!/ (M_n) \cdots))$ we have n levels of depth in the SDD, with at each level k two variables : a variable m_k with the states of a unitary module of the form M_k, and a variable $m_{(M_k^+)}$ that in effect represents the states of all the modules of index greater than k.

We partition the transitions of the system into *local* and *synchronization* transitions. A transition t is local to unitary module M_n iff $\forall p \in {}^\bullet t \cup t^\bullet, p \in M_n$. For each unitary module of index n, we construct a DDD homomorphism $\mathcal{L}_n$ built using DDD saturation as presented in section 3.

For synchronization transitions that are not local to a single module, we define the projection of a transition on a module of index n as:

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

We further define for a synchronization transition t, $Top(t)$ as the most internal parenthesised group (M_k) such that M_k wholly contains t. So in effect t is local to this group, and this is the most internal level we can apply t from. When nesting occurs at more than one level of depth t has a top and bottom at each level of depth in the structure. $Bot(t)$ is defined as the lowest variable index that is used by t. Our full transition relation is then inductively defined by:

Let $\tau_k = \bigcup_{\{t|Top(t)=(M_k)\}} \tau(t)$ represent the transitions local to a parenthesized group (M_k); we define $\tau_{-1} = Id$.

$$\tau(t)(e,x) = \begin{cases} e \xrightarrow{\mathcal{L}_k^* \circ \Pi_k(t) \circ \mathcal{L}_k^*(x)} \tau_{e-1}^* & \text{if } e = m_{M_k} \wedge bot(t) \geq e \\ e \xrightarrow{\mathcal{L}_k^* \circ \Pi_k(t) \circ \mathcal{L}_k^*(x)} \tau_{e-1}^* \circ \tau(t) & \text{if } e = m_{M_k} \wedge bot(t) < e \\ e \xrightarrow{\tau_k^* \circ \tau(t) \circ \tau_k^*(x)} \tau_{e-1}^* \circ \tau(t) & \text{if } e = m_{(M_k)} \end{cases}$$
$$\tau(t)(1) = 1$$

This algorithm thus performs *local saturation* of nested subnets. Like the algorithms of [4], it performs local saturation on the lower nodes as soon synchronization transitions are fired. This avoids the creation of intermediate nodes which will not appear in the

Table 2. Performances of our prototype over some bench models. We compare our tool's run time with the run times of Smart. Indicatively we also give the runtimes of NuSMV [6], the emblematic tool for symbolic representations (these values were not measured by us, but are directly taken from [5]). Some are missing as indicated by ?. The Lotos model is 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. AGV (automated guided vehicle) is a flexible manufacturing problem, with synthesis of controllers in mind : we give the statistics with and without the controller enabled. Example witness trace construction is possible, yielding the shortest path to (un)desirable states. Run time is 1h20 for finding a *shortest* witness trace enabling each of the 776 transitions of the Lotos model, the longest is 28 transitions in length.

Model	N	States	final		total		PNDDD SDD time	NuSMV time	SMaRT time
	(#)	(#)	SDD (#)	DDD (#)	SDD (#)	DDD (#)	(sec)	(sec)	(sec)
Philosophers	100	4.97+62	398	21	2185	70	0.21	990.8	0.43
	200	2.47+125	798	21	4385	70	0.43	18129	0.7
	1000	9.18e+626	3998	21	21985	70	2.28	-	5.9
	5000	6.52+3134	19998	21	109985	70	11.7	-	83.7
Ring	10	8.29e+09	61	44	2640	150	0.4	6.1	0.11
	15	1.65e+16	288	44	8011	150	1.21	2853	0.29
	50	1.72e+52	2600	44	238400	150	34.01	-	5.6
FMS	25	8.54e+13	55	412	346	11550	0.26	41.6	0.36
	50	4.24e+17	105	812	671	38100	1.02	17321	1.33
	80	1.58e+20	165	1292	1064	89760	2.59	-	4
	150	4.8e+23	305	2412	1971	294300	10.52	-	20.7
Kanban	10	1.01e+09	15	46	129	592	0.02	?	0.48
	50	1.04+16	55	206	1589	9972	1.08	?	43
	100	1.73e+19	105	406	5664	37447	8.79	?	474
	200	3.17e+22	205	806	21314	144897	93.63	?	13920
Lotos [8]	N/A	9.79474e+21	326	759	125773	34298	265.28	?	?
AGV [14]	N/A	3.09658e+07	12	34	135	234	0.01	?	?
AGV Controlled	N/A	1.66011e+07	95	124	2678	349	0.38	?	?

final state-space representation, hence it limits the peak number of nodes that needs to be stored.

The following table shows the performance of our prototype tool over models taken from [13]. We use here a simple $(M_1) /\!\!/ (M_2) /\!\!/ \cdots /\!\!/ (M_n)$ parenthesizing scheme in these performance runs, thus only one level of depth is used in the data structure. This is a parenthesizing scheme that most closely relates to the experiments of [4]; indeed the number of SDD nodes is identical to the number of MDD nodes reported by Smart.

We can observe that the encoding is much more compact than with plain DDD, and the run times are a factor below those obtained with the flat DDD representation. The exception is the philosophers model, which actually gives better run times in the flat DDD representation (though at a cost in terms of representation size). The slotted ring example shows the superiority of the MDD access procedure, which allows direct access to all the nodes of any level k. Thus MDD saturation more efficiently fights the intermediate size problem than our own transitive closure operator. We believe this might be improved by tuning the caching policy, but this remains to be proved.

The Kanban model shows the advantage of SDD over MDD when the number of states per submodel grows : in this model each submodule has a number of states factorial with respect to N, while the number of modules stays constant. Smart's MDD representation represents one arc for each state value of a submodule, as an arc bearing the index of the state in a splay tree explicit (but quite compact) representation. Thus although we may have the same number of nodes, the number of arcs in the MDD representation explodes exponentially with N, while our referenced DDD scheme allows to factorize all the arcs that lead from a node d_1 to a node d_2 in the referenced DDD. This is an important point as larger software examples present modules with a sometimes very large reachability set.

6 Conclusion

We have presented Set Decision Diagrams, a hierarchical directed acyclic graph structure, with a canonical representation that allows use of a BDD-like cache and unicity table. SDD operations are defined through a general and flexible model, inductive homomorphisms, that give exceptional freedom to the user. This structure is particularly well adapted to the construction and storage of the state space of hierarchical and modular models. Thanks to local fixpoint computations and improved sharing in the representation with respect to the purely linear encoding of usual decision diagram libraries, exceptional performances can be attained.

The principles of our parenthesized parallel composition can be generalized to a wide range of models that can be seen as LTS. The choice of a correct parenthesizing is an open problem : the separation into parts should highlight parts that are similar at least structurally. One easy choice for typed specifications is to assign an encoding to each type (record, vectors, lists, basic types...). This will increase representation sharing. Our library, available under LGPL from `ddd.lip6.fr`, can be extended through inheritance to use other decision diagram packages than DDD to represent the states of a module.

We are currently working at extending [15] to exploit symmetries in a hierarchical SDD representation, as coloration can be seen as an important structural information, that can guide the process of choosing an appropriate parenthesizing. We are also using

the SDD in a project to model-check Promela specifications. Finally we aim at defining a framework for operations that do not respect the module consistency constraint, as our current solutions lacks generality in this respect.

References

1. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
2. 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.
3. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *Proc. of ICATPN'2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 103–122. Springer Verlag, June 2000.
4. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In H. Garavel and J. Hatcliff, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, pages 379–393, Warsaw, Poland, April 2003. Springer-Verlag LNCS 2619.
5. Gianfranco Ciardo. Reachability set generation for petri nets: Can brute force be smart. In J. Cortadella and W. Reisig, editors, *Application and Theory of Petri Nets 2004. 25th International Conference, ICATPN 2004.*, volume 3099 of *LNCS*, pages 17–34, 2004.
6. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.
7. J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier. Data decision diagrams for Petri net analysis. In *Proc. of ICATPN'2002*, volume 2360 of *Lecture Notes in Computer Science*, pages 101–120. Springer Verlag, June 2002.
8. H. Garavel. A net generated from lotos by cadp (`http://www.inrialpes.fr/vasy/cadp`). In *`PetriNets@daimi.au.dk` mailing list.*, Posted 28/07/03 and follow-up with performance of 4 tools on 26/09/03.
9. J. Geldenhuys and A. Valmari. Techniques for smaller intermediary bdds. In *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 233–247, 2001.
10. A. Gupta and A. L. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *ICCAD'93*, pages 111–116, 1993.
11. J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, 1991. North-Holland.
12. F. Kordon and M. Lemoine, editors. *Formal Methods for Embedded Distributed Systems How to master the complexity*. Kluwer Academic, 2004.
13. A.S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. of ICATPN'99*, volume 1639 of *Lecture Notes in Computer Science*, pages 6–25. Springer Verlag, 1999.
14. L. Petrucci. Design and validation of a controller. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2000)*, pages 684–688, Orlando, Florida, USA, July 2000.
15. Y. Thierry-Mieg, J-M. Ilie, and D. Poitrenaud. A symbolic symbolic state space. In *Proc. of the 24th IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE'04)*, volume 3235 of *LNCS*, pages 276–291, Madrid, Spain, September 2004. Springer.