A Symbolic Symbolic State Space Representation

Yann Thierry-Mieg, Jean-Michel Ilié, and Denis Poitrenaud

SRC - LIP6 - UMR 7606, Université Paris 6
4, Place Jussieu, 75252 Paris cedex 05
(firstname.lastname@lip6.fr)

KEYWORDS: Decision Diagrams, Symbolic Model-checking, Symbolic Reachability Graph, Well-Formed Petri Nets, Symmetry Detection

Abstract. Symmetry based approaches are known to attack the state space explosion problem encountered during the analysis of distributed systems. In another way, BDD-like encodings enable the management of huge data sets. In this paper, we show how to benefit from both approaches automatically. Hence, a quotient set is built from a coloured Petri net description modeling the system. The reachability set is managed under some explicit symbolic operations. Also, data representations are managed symbolically based on a recently introduced data structure, called Data Decisions Diagrams, that allow flexible definition of application specific operators. Performances yielded by our prototype are reported in the paper.

1 Introduction

In this paper, we exploit both data symmetries to construct a set of reachable equivalence classes of states, and a symbolic coding of these classes and of the transition relation using a BDD-like representation. The construction of this reachability set is the basis for model-checking, verification of safety constraints, deadlock detection, etc.

Roughly speaking, model checking of symmetrical systems exploits the fact that many synchronization and communication protocols, involving parallel composition of n processes differing only by their identity, often exhibit considerable symmetries. This can be viewed as a redundancy of information in the state graph, as states identical up to a permutation can be aggregated into equivalence classes, yielding a possibly exponentially smaller quotient state space. The efficiency of this type of approach is demonstrated by tools like GreatSPN, SMC, Mur-ϕ, which offer mechanisms to define the symmetries allowed by the model (for instance the Mur-ϕ scalarset).

The core of the problem consists in determining whether two states are equivalent; one approach found in literature is to define a canonization operation that yields a unique representative for each equivalence class, thus only representatives need to be stored. However it has been proved that the BDD coding the orbit relation required to find a unique representative of an equivalence class would need an exponential number of nodes [3]. The approach used here is based on the work of [1] which uses dedicated data-structures to represent equivalence classes, termed Symbolic Markings (*SM*), instead of concrete states designated as representatives. In practice, such a direct coding

of the equivalence classes can speed up the computation of the quotient graph. In the literature, one can find other codings which use close concepts [8].

Furthermore, the symmetries exploited to construct an SM are computed automatically through a structural exploration of a model. This procedure uses the algorithm described in [10] and assumes a modeling in a coloured Petri net to produce a Well-Formed net. It is fully automatic and has a polynomial complexity over the size of the model. The symbolic reachability set (SRS) is then built by means of a symbolic firing rule, that does not require the actual concrete states to be explored. The steps required for this construction are: from a set of SM, a symbolic firing rule is applied yielding a new set of (intermediate) symbolic states. These last SMs are then canonized yielding a set of canonical representatives, which can be compared to already obtained canonical SMs.

The challenge we address here is to define these operations on a BDD-like representation, although the structures used to describe equivalence classes of states are calculated dynamically, use *a priori* unbounded integer domains, have a variable domain size according to the dynamically grouped elements, and require quite complex data-structures to represent them. We show how the specific dag library called *Data Decision Diagrams* (DDD), that allows flexible operation definition possibilities through inductive homomorphisms, meets our needs. Application of DDD to uncoloured "extended Petri nets" [4] has shown their expression power and dynamic capabilities. We make full use of available DDD features: variable repeats, variable length vectors, and we rely on hierarchical computations to maximize cache hit ratio.

This paper is organized as follows: section 2 presents a brief overview of DDD capabilities and introduces the Well-Formed nets (*WN*) formalism; section 3 gives the principles used in the coding of a state; section 4 presents the main symbolic firing operation and section 5 the minimization and canonization procedures; section 6 describes the state space construction procedure; finally, section 7 reports the performances of the tool implementing this technique on a few classical examples.

2 Context

2.1 DDD basic concepts and inductive homomorphism

Data Decision Diagrams (DDD) are a data structure for representing finite sets of assignments sequences of the form $(x_1 := v_1) \cdot (x_2 := v_2) \cdots (x_n := v_n)$ where x_i are variables and v_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, DDDs correspond to the specialized version of the Multi-valued Decision Diagrams representing characteristic function of sets [2]. However DDDs 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.

DDDs have three terminals : 1, 0 and $\top$. 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 DDDs is important as it allows the use of variables of finite domain with *a priori* unknown bounds. The $\top$ terminal is introduced to resolve conflicts introduced by the fact that no variable ordering is required and that a same variable can appear several times in a same sequence.

In the following, X denotes a set of variables, and for any x in X, $\mathrm{Dom}(x)$ represents the domain of x.

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

- $d \in \{0, 1, \top\}$ *or*
- $d = \langle x, \alpha \rangle$ *with:*
  - $x \in X$
  - $\alpha : \mathrm{Dom}(x) \to \mathbb{D}$*, such that* $\{v \in \mathrm{Dom}(x) \mid \alpha(v) \neq 0\}$ *is finite.*

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

As usual, DDDs are encoded as (shared) decision trees. Hence, a DDD of the form $\langle x, \alpha \rangle$ is encoded by a node labeled x and for each $v \in Dom(x)$ such that $\alpha(v) \neq 0$, there is an arc from this node to the root of $\alpha(v)$. By the definition 1, from a node $\langle x, \alpha \rangle$ there can be at most one arc labeled by $v \in \mathrm{Dom}(x)$ and leading to $\alpha(v)$. This may cause conflicts when computing the union (noted $+$) of two DDDs. Consider for instance $d = (a \xrightarrow{1} b \xrightarrow{2} 1) + (a \xrightarrow{1} a \xrightarrow{3} 1) = a \xrightarrow{1} (b \xrightarrow{2} 1 + a \xrightarrow{3} 1)$. We need to compute $(b \xrightarrow{2} 1 + a \xrightarrow{3} 1)$. If $a = b$, this can be resolved by creating a node of variable a having two arcs to the terminal 1 labeled with values 2 and 3. However if $a \neq b$ we cannot decide what variable the resulting node should bear. The result is therefore undefined, and is noted as such by the terminal $\top$. Thus d will evaluate to $d = a \xrightarrow{1} \top$. More formally, $\top$ represents *any* set of finite assignment sequences, therefore $\top$ is the worst approximation of a finite set of assignment sequences. When $\top$ does not appear in a DDD d, d represents a unique finite set of assignment sequences. Such DDDs are said *well-defined*. We require that the DDDs we manipulate be well-defined, and we will detail in section 3.1 how we ensure this property. For a complete definition of DDDs and particulary of their operations, please refer to [4].

DDDs 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 DDDs, it corresponds to a cartesian product. In addition, homomorphisms are defined to allow flexibility in the definition of application specific operations.

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

Furthermore, application-specific mappings can be defined by *inductive* homomorphisms. An inductive homomorphism Φ is defined by its evaluation on the 1 terminal

$\Phi(1) \in \mathbb{ID}$, and its evaluation $\Phi' = \Phi(x \xrightarrow{v} d)$ for any $x \xrightarrow{v} d$. Φ' is itself a (possibly inductive) homomorphism, that will be applied on the successor node d. The result of $\Phi(\langle x, \alpha \rangle)$ is then defined as $\sum_{v \in Dom(x)} \Phi(x \xrightarrow{v} \alpha(v))$.

Inductive homomorphism examples : $inc(x_1)$ increments the value of the first occurrence of the variable x_1. It returns $\top$ if x_1 is not part of the sequence. $setCst(x_1, v_1, v_2)$ assigns to each occurrence of x_1 the values in the range $[v_1, v_2]$. The application of *setCst* to a simple DDD is shown below.

$$
\begin{array}{l}
inc(x_1)(x, v) = \\
\begin{cases} x \xrightarrow{v+1} Id & \text{if } x = x_1 \\ x \xrightarrow{v} inc(x_1) & \text{otherwise} \end{cases} \\
inc(x_1)(1) = 1
\end{array}
\quad \Bigg| \quad
\begin{array}{l}
setCst(x_1, v_1, v_2)(x, v) = \\
\begin{cases} \sum_{v' \in [v_1, v_2]} x \xrightarrow{v'} setCst(x_1, v_1, v_2) & \text{if } x = x_1 \\ x \xrightarrow{v} setCst(x_1, v_1, v_2) & \text{otherwise} \end{cases} \\
setCst(x_1, v_1, v_2)(1) = 1
\end{array}
$$

$$
\begin{aligned}
&setCst(a,1,2)(a \xrightarrow{1} b \xrightarrow{2} a \xrightarrow{3} 1) = \\
&\quad a \xrightarrow{1} setCst(a,1,2)(b \xrightarrow{2} a \xrightarrow{3} 1) + a \xrightarrow{2} setCst(a,1,2)(b \xrightarrow{2} a \xrightarrow{3} 1) \\
&= a \xrightarrow{1} b \xrightarrow{2} setCst(a,1,2)(a \xrightarrow{3} 1) + a \xrightarrow{2} b \xrightarrow{2} setCst(a,1,2)(a \xrightarrow{3} 1) \\
&= a \xrightarrow{1} b \xrightarrow{2} \begin{pmatrix} a \xrightarrow{1} setCst(a,1,2)(1) + \\ a \xrightarrow{2} setCst(a,1,2)(1) \end{pmatrix} + a \xrightarrow{2} b \xrightarrow{2} \begin{pmatrix} a \xrightarrow{1} setCst(a,1,2)(1) + \\ a \xrightarrow{2} setCst(a,1,2)(1) \end{pmatrix} \\
&= a \xrightarrow{1} b \xrightarrow{2} a \xrightarrow{1} 1 + a \xrightarrow{1} b \xrightarrow{2} a \xrightarrow{2} 1 + a \xrightarrow{2} b \xrightarrow{2} a \xrightarrow{1} 1 + a \xrightarrow{2} b \xrightarrow{2} a \xrightarrow{2} 1
\end{aligned}
$$

Like in BDD packages, a hash table is used to ensure the unicity of DDD nodes. Moreover, a cache is maintained by the library to store the result of the application of a homomorphism to a DDD. Thus, although the expression $setCst(a,1,2)(b \xrightarrow{2} a \xrightarrow{3} 1)$ needs to be evaluated twice, the second evaluation will constitute a constant time cache hit.

2.2 Well-Formed Net and Symbolic Reachability Set (SRS)

Symbolic markings (SM) are equivalence classes of states constructed using symmetries that are computed before starting to explore the state space. The tokens which have structurally similar behaviour, i.e. that can be exchanged at any point in the evolution of the system with no impact on the sequences of fireable transitions, are grouped into "static subclasses" (slow and fast processors for instance), which are not modified during the construction. In contrast "dynamic subclasses" are introduced to represent sets of tokens that have the same *distribution* throughout the places of the model. Although the number and cardinality of these dynamic subclasses evolve during the SRS construction, dynamic subclasses always constitute a partition of static subclasses (the slow processors that are waiting and those that are at rest for instance). Thus dynamic subclasses concisely represent the permutations that are permitted on an SM without modifying future sequences of fireable transitions.

We now informally explain the SRS construction through a simple example. A coding of SM is also introduced, and will be reused in section 3 in our DDD representation. For a more formal description of SRS algorithms, please refer to [1].

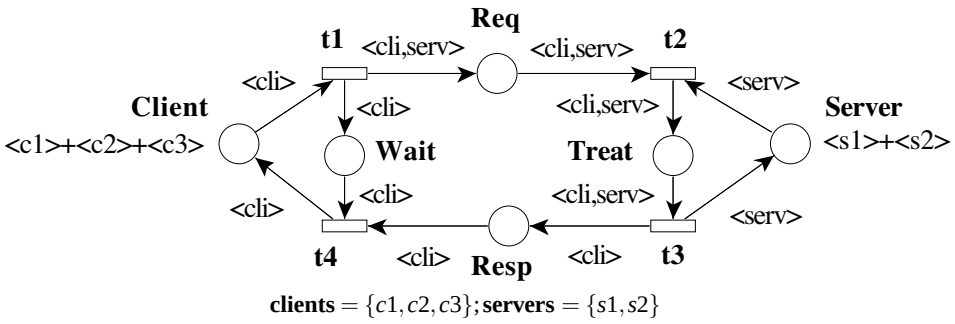


Fig. 1. Client Server Protocol

Figure 1 represents a simple client server protocol. The system is composed of **clients**, initially positioned in place *Client* and of **servers** initially in place *Server*. At some point, a client can emit a request for treatment by firing transition t_1, thus generating a request in *Req* for an arbitrary server *serv* (the parameter *serv* is "free" and may be bound to any server). The client then waits for a response from the server in place *Wait*. When the chosen server is available, it will fire t_2, consuming the request and placing the server in place *Treat*. When treatment is finished, the server generates a response in *Resp* for the client and returns to place *Server* by firing t_3. Finally the client can acknowledge the reply by t_4, and return to its initial state.

It can be noticed that whatever their numbers, all clients and respectively servers, have a symmetric role. The structural symmetry analysis module therefore places all clients in a single static subclass C and all servers in a static subclass S (here, there is no need of further refinements into static subclasses). As they are equal, we will not distinguish C from **clients** and S from **servers**. The initial symbolic marking $\mathcal{S}_0$ is expressed symbolically by the expression $\mathcal{S}_0 = Client(Z_{C0}) + Server(Z_{S0}), |Z_{C0}| = 3, |Z_{S0}| = 2$; this SM corresponds to the concrete initial marking: $Client(c_1 + c_2 + c_3) + Server(s_1 + s_2)$. Z_{C0} and Z_{S0} are the dynamic subclasses which respectively represent the **clients** in place *Client* and the **servers** in place *Server*. As we can see, permuting elements within a Z_i does not modify the marking, as all elements within a Z_i have the same distribution over all places.

From the concrete initial marking, 6 firings of t_1 are possible, since *cli* may be bound to any c_1, c_2, c_3 and independently *serv* can be bound to s_1 or s_2. However, since all elements within a dynamic subclass Z_i are fully equivalent, there is only one way to bind a variable to any Z_i, whatever it's cardinality. Hence, a single symbolic binding is possible from the SM $\mathcal{S}_0$, *cli* is bound to (a value in) Z_{C0} and *serv* to (a value in) Z_{S0}. To compute the firing, we first isolate Z_{C1} as the value bound to *cli* and Z_{S1} as the value bound to *serv*. We can then modify the distribution of these new dynamic subclasses, by applying pre and post arc functions of t_1. We obtain the SM $\mathcal{S}_1 = Client(Z_{C0}) + Server(Z_{S0} + Z_{S1}) + Wait(Z_{C1}) + Req(\langle Z_{C1}, Z_{S1} \rangle), |Z_{C0}| = 2, |Z_{C1}| = 1, |Z_{S0}| = |Z_{S1}| = 1$. This distribution of **clients** into Z_{Ci} can be encoded by **clients**$= \begin{pmatrix} 2 & 1 \end{pmatrix}$ specifying the cardinalities of the Z_{Ci}, and similarly **servers** $= \begin{pmatrix} 1 & 1 \end{pmatrix}$. The place markings can be coded by: $Client = \begin{pmatrix} 1 & 0 \end{pmatrix}; Server = \begin{pmatrix} 1 & 1 \end{pmatrix}; Wait = \begin{pmatrix} 0 & 1 \end{pmatrix}$, indicating that place *Client*

contains $1*Z_{C0}+0*Z_{C1}$, etc... The marking of place *Req* is defined over the colour domain **clients** $\times$ **servers** and can be represented by a matrix $Req=\begin{pmatrix}0&0\\0&1\end{pmatrix}$, indicating it contains $0*\langle Z_{C0},Z_{S0}\rangle+0*\langle Z_{C0},Z_{S1}\rangle+0*\langle Z_{C1},Z_{S0}\rangle+1*\langle Z_{C1},Z_{S1}\rangle$.

We have shown how firing may split dynamic subclasses ; to exhibit the grouping of dynamic subclasses, let us fire t_1 again. *cli* must be bound to a token present in *Client*, therefore must be bound to Z_{C0}, but *serv* may be bound either to Z_{S1} or Z_{S2}. This yields two possible symbolic firings of t_1, instead of the 24 concrete possible firings.

Let us detail the firing $cli \in Z_{C0}, serv \in Z_{S0}$: the second request is addressed to the same server as the first one. Again we split Z_{C0} to distinguish Z_{C2}, the value *cli* is bound to. The marking obtained is $Client(Z_{C0})+Server(Z_{S0}+Z_{S1})+Wait(Z_{C1}+Z_{C2})+Req(\langle Z_{C1},Z_{S0}\rangle+\langle Z_{C2},Z_{S0}\rangle), |Z_{C0}|=|Z_{C1}|=|Z_{C2}|=1$ and $|Z_{S0}|=|Z_{S1}|=1$. This SM can be coded by the tensors: **clients** $=\begin{pmatrix}1&1&1\end{pmatrix}$ and **servers** $=\begin{pmatrix}1&1\end{pmatrix}$ giving the Z_i and

$Server=\begin{pmatrix}1&1\end{pmatrix}$, $Client=\begin{pmatrix}1&0&0\end{pmatrix}$, $Wait=\begin{pmatrix}0&1&1\end{pmatrix}$ and $Req=\begin{pmatrix}0&0\\1&0\\1&0\end{pmatrix}$.

We can observe that Z_{C1} and Z_{C2} have the same distribution in all places in this configuration. Indeed for any place P of domain **clients**, $P[1]=P[2]$ and for place *Req*, the second and third *lines* are equal. We therefore group them in a single subclass of cardinality 2. The resulting SM $\mathcal{S}_3$ is: **clients** $=\begin{pmatrix}2&1\end{pmatrix}$, **servers** $=\begin{pmatrix}1&1\end{pmatrix}$ giving the Z_i and $Server=\begin{pmatrix}1&1\end{pmatrix}$, $Client=\begin{pmatrix}0&1\end{pmatrix}$, $Wait=\begin{pmatrix}1&0\end{pmatrix}$ and $Req=\begin{pmatrix}1&0\\0&0\end{pmatrix}$. It should be noted that the Z_{Ci} dynamic subclasses have been reindexed, in order to have $|Z_{C0}|>|Z_{C1}|$; this reordering is necessary to ensure the unicity of representation of the SM, and will be discussed in detail within Section 5.2.

3 State encoding

3.1 Tensor coding

As our example in section 2.2 has shown, the data that need to be stored are expressed in terms of variable length vectors, matrices or more generally tensors. The mechanism used to store a vector of size n is simply to repeat a same variable V n times. Moreover, a vector is always terminated by an *End* marker, represented by an occurrence of $V\xrightarrow{\sharp}$. Although the number of repeats of V may vary along paths within our DDD, the variables always occur in the same order, thus the $V\xrightarrow{\sharp}$ always leads to the same variable. This ensures that our structures can be safely united, intersected, etc without risk of creating $\top$ terminals.

As we have seen, the marking of a place P of arbitrary domain $D=C_{k_1}\times\cdots C_{k_n}$ can be stored as an n-tensor, n being the number of classes composing D. We therefore need a representation of an n-tensor over what is basically a linear coding, as a DDD stores a sequence of variable/value pairs. Furthermore the operations we need to define manipulate $(n-1)$-tensors extracted from an n-tensor, such as lines of a matrix, or faces of a cubic 3-tensor, so we need to easily determine where to find the elements of an

$(n-1)$-tensor. We have chosen a lexicographic coding, which meets our requirements, and is a generalization of the coding used for simple vectors.

Let t be an n-tensor of dimensions $d_0 \times \cdots \times d_{n-1}$. Let $a_{i_0,i_1,\cdots i_{n-1}}$ be an element of the tensor. The elements of the tensor t are encountered in lexicographic order: $a_{0,\cdots 0,0} \rightarrow a_{0,\cdots 0,1} \rightarrow \cdots \rightarrow a_{0,\cdots 0,(d_{n-1}-1)} \rightarrow a_{0,\cdots 1,0} \rightarrow \ldots \rightarrow a_{(d_1-1),\cdots(d_{n-2}-1),(d_{n-1}-1)}$. We are trying to characterize the elements of a target $(n-1)$-tensor. For instance $\forall i \in [0 \cdots d_1-1], a_{3,i}$ would give the elements of line 3 of a matrix. Let d_k be the target dimension, and v the target index along this dimension. In our example, $k=0$ and $v=3$.

Property 1. Let $\pi = \prod_{j>k} d_j$ and $\mu = \prod_{j \geq k} d_j$; the indexes i of the elements of the v^{th} (n-1)-tensor along dimension k satisfy:

$$v \cdot \pi \leq i < (v+1) \cdot \pi \quad mod(\mu).$$

Where the whole inequation is evaluated modulus μ. The proof of this property is straightforward and is omitted here.

Let us apply this property to an example 2-tensor (a matrix) of dimension 3×4 :

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

The indexes i of the second line are given by $k=0; v=1; \pi = d_1 = 4; \mu = d_0 \cdot d_1 = 12$; thus $4 \leq i < 8 \quad mod(12)$.
In the same way the third column is found at indexes i computed by: $k=1; v=2; \pi = 1; \mu = d_1 = 4$; thus $2 \leq i < 3 \quad mod(4)$.

Generally the indexes of the elements of a (n-1)-tensor are not contiguous within the structure, however operations don't need to read all the values to begin processing the operation. For instance, an operation comparing the second and third columns, needs only to store the value at index 2, to compare it to value at index 3 when it is reached, then if they are not equal the result can directly be given, else values 2 and 3 are "dropped" and the process will be iterated for the next element of index i meeting the $2 \leq i < 3 \quad mod(4)$ criterion.

3.2 States and motifs

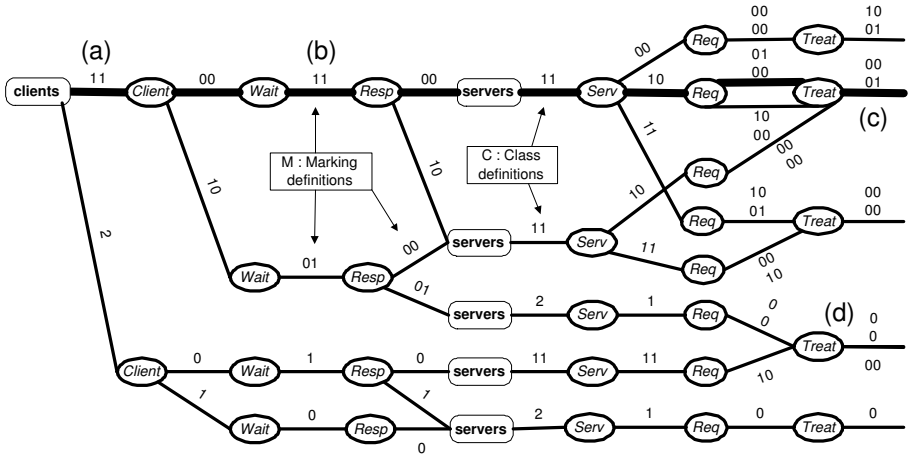
In this section, we present our coding of symbolic states (SM). This coding was developed with two preoccupations: allow the definition of the symbolic firing relation and of the canonization operation with algorithms that only need to perform a unique traversal of the structure to determine their results. This means that the decision of what must be done on a node n only depends on the path traversed from the root to n and not on what may follow. In effect an algorithm that respects this constraint is at most of polynomial complexity over the size in number of nodes of the DDD. The second preoccupation is to obtain a high level of sharing within the representation.

The data stored for each state is organized in motifs. Two primary motifs are distinguished: the class motif corresponding to the definition of the partition of a static subclass C_i into dynamic subclasses Z_i (C motif) and the motif corresponding to the marking of a place (M motif).

C motif, or class definition motif: The current distribution of each class or static subclass C_i into $Z_{C_{ij}}$ subclasses. As we have seen in section 2.2 this can be coded as a variable length vector of integers. As the marking evolves, the number, cardinality and

order of these subclasses is modified, subclasses being split up whenever a particular token is extracted, and merged whenever they have the same distribution.

M motif, or marking motif: Given a partition into dynamic subclasses, a marking of a place is expressed as the number of times m each subclass or combination of subclasses (i.e. $m* < Z_{C_1 1}, Z_{C_2 0} >$) is present in the place. We represent the marking of a place of domain $C_0 \times \cdots C_{n-1}$ by an n-tensor, of dimensions $d_{C_0} \times \cdots d_{C_{n-1}}$ where d_{C_i} is the current number of dynamic subclasses in C_i.



(a) A sample C motif: $Z \xrightarrow{1} Z \xrightarrow{1} Z \xrightarrow{\sharp}$

(b) A sample M motif: $M \xrightarrow{1} M \xrightarrow{1} M \xrightarrow{\sharp}$

(c) A matrix M motif: $M \xrightarrow{0} M \xrightarrow{0} M \xrightarrow{0} M \xrightarrow{1} M \xrightarrow{\sharp}$

(d) Place *Treat* is marked by: $M \xrightarrow{0} M \xrightarrow{0} M \xrightarrow{\sharp}$ that is interpreted as a 2×1 matrix if coming from the top branch or as a 1×2 matrix if coming from the bottom branch.

Fig. 2. Client Server Protocol, 2 clients, 2 servers: 12 symbolic states.

Figure 2 depicts the state space for our client server protocol with 2 clients and 2 servers. The figure reads from left to right, the root of our DDD being the colour domain definition of **clients**, and the last place described is *Treat*, which leads to the 1 terminal node not represented here. This diagram is an abstraction of the real DDD, as we have directly labeled arcs with tensors (vectors, matrices), instead of repeating the variable and representing the $V \xrightarrow{\sharp} End$ marker. Furthermore, we have named the variables to allow correspondence with the model, but all M motifs use the same variable M, and all C motifs use the same variable Z. Some sharing amongst paths of our representation is obvious here. As the size of the example increases, so does sharing, as we will discuss in section 7.

The state shown in bold corresponds to a state where **clients** are distributed into two dynamic subclasses Z_{C0} and Z_{C1} both of cardinality one, the place *Client* is empty,

the place *wait* contains $Z_{C0} + Z_{C1}$, place *Resp* is empty, the **servers** are distributed into two dynamic subclasses Z_{S0} and Z_{S1} both of cardinality one, place *Server* contains one server Z_{S0}, place *Req* contains one request $\langle Z_{C0}, Z_{S1} \rangle$, and *Treat* contains $\langle Z_{C1}, Z_{S1} \rangle$. Thus in english, both clients are waiting for a reply and their requests target the same server Z_{S1}, one request is being treated and the other has not yet been received. The state that differs only by the marking of *Req* corresponds an analogous state except the client's requests are not for the same server.

3.3 Operation framework

Our prototype is developed in C++, and takes full advantage of both inheritance and template arguments to provide an open framework in which to code operations over these motifs. Thus an abstract operation class provides the functionality required to keep track of the current position within the motif and the concrete operations inherit this behaviour.

When exploring the DDD seen as a tree, this generic operation stores the tensors encountered; upon reaching an $\sharp\, End$ marker, the tensor that has been stored is passed to the concrete operation being run for evaluation. Thus our generic operation reads the tensor values represented on the arcs of Figure 2 into a DDD $\mathcal{M}$, evaluates a concrete operation on the extracted tensors $\mathcal{M}' = op(args)(\mathcal{M})$, and continues evaluation based on the value of $\mathcal{M}'$.

This means that each place marking is isolated before running an operation on it, and since the number of possible place markings (actual different matrices on the arcs of figure 2) is low with respect to the combinations of place markings, sharing is high on the actual operations performed on tensors, although the storage phase has a higher complexity due to lower sharing. Furthermore, except for the transition firing operation, operations are evaluated independently of the actual place being considered, thus the concrete operation run on the extracted tensors have less arguments. In effect it means tensor operations are independent of the position of the tensor within the full state motif, thus operations on different place marking matrices can be shared, increasing the cache hit ratio.

4 Symbolic firing operation

As we have seen in section 2.2, there is only one way to bind a variable to any Z_i, whatever it's cardinality. However, if the dynamic subclass Z_i a variable X is being bound to has a cardinality $c > 1$, a new dynamic subclass Z_X of cardinality 1 must be created to isolate the value X is bound to. With Z_X isolated, one can test if a place contains X by using the number of occurrences of Z_X in P, or add X in P by incrementing the number of occurrences of Z_X in P. An operation "add dim" is defined to create these new dynamic subclasses as needed. This operation simply copies the $(n-1)$-tensor corresponding to Z_i in position $i+1$, prior to evaluating the inhibitor, pre, and post functions for the considered place. On the other hand, if X is bound to a class Z_i of cardinality one, no new subclass need be created, we can use $Z_X = Z_i$.

The transition relation is defined as operating over a set of SM represented by a DDD. But since it is implemented by an inductive homomorphism, we describe the behaviour of the transition as a visit of the word constituting a single SM $\mathcal{S}$. A transition thus has the following behaviour:

1. When encountering a colour domain definition $\mathcal{C}$, all the possible variable bindings are constructed. A variable binding $(vars, adim)$ is composed of the correspondence variable to dynamic subclass index *vars*, and of the associated "add dim" operation $adim$ that should be applied prior to evaluating the arc functions. Illegal variable bindings with respect to the guard of the transition are filtered out in this phase. Then a composite operation defined as a sum of all possible bindings is returned for evaluation on the rest of the SM.
2. When encountering a place marking $\mathcal{M}$, the $adim$ operation corresponding to the current variable bindings is first applied. Then the colour functions associated to the current place are evaluated. If a pre or inhibitor function returns the 0 terminal, the transition stops evaluation and it's homomorphism returns 0 thus pruning the partially constructed resulting marking from the DDD. Otherwise the newly obtained marking is inserted and the operation continues.
3. If a transition reaches the 1 terminal the transition has been successfully fired with the current variable bindings.

Colour functions are compositions of the three basic colour functions: diffusion noted S, identity represented by a formal parameter X, and only for ordered colour domains the successor operation $X++$. These colour functions may have a multiplicative factor associated. For instance $2\cdot < X > + < Y >$, or $< S, X > +2\cdot < X, Y >$ are possible colour functions labeling an arc to or from a place of domain respectively C and $C \times C$.

Given the definition of a place's colour domain, the number of dynamic subclasses in each of the C_i that form the domain, and the bindings of all variables or formal parameters to their Z_i, we compute the indexes in the M motif that are targeted by a colour function, and the multiplicative factor m associated to these indexes. This computation is common to all types of arcs; given this list of target indexes and multiplicities, a homomorphism specific to the type of arc is applied for each target index i. These homomorphisms are defined by (inhibitor not represented) :

$$\mathcal{W}^-(i,m)(x,v) = \begin{cases} \text{if} \quad i \neq 0 & \Rightarrow x \xrightarrow{v} \mathcal{W}^-(i-1,m) \\ \text{else} \ \text{if } v \geq m & \Rightarrow x \xrightarrow{v-m} Id \\ \quad \text{else} & \Rightarrow 0 \end{cases}$$
$$\mathcal{W}^-(1) = \top$$

$$\mathcal{W}^+(i,m)(x,v) = \begin{cases} \text{if} \quad i \neq 0 \Rightarrow x \xrightarrow{v} \mathcal{W}^+(i-1,m) \\ \text{else} \quad \Rightarrow x \xrightarrow{v+m} Id \end{cases}$$
$$\mathcal{W}^+(1) = \top$$

Where $\mathcal{W}^-, \mathcal{W}^+$ are respectively the pre and post arc operations. None of these operations should ever encounter the 1 terminal as the index i that is targeted will be reached before, and reaching i terminates the operation.

5 Canonization algorithms

This section presents the operations that we have defined to implement the canonization algorithms. The construction of the SRS is based on the notion of canonic representation for an equivalence class of states. From a symbolic state $\mathcal{S}$, a transition is fired yielding a new symbolic state $\mathcal{S}_1$. $\mathcal{S}_1$ is then minimized by groupings of dynamic Z_i subclasses and canonized to ensure the unicity of its representation.

As we have seen, the firing of a transition is liable to create new dynamic subclasses bound to the different formal parameters of the transition, and changes the distribution of dynamic subclasses within an SM. The goal of the minimization operation is to group dynamic subclasses that have the same distribution in all places, yielding a reduced expression of place markings. This can be accomplished by testing for any two dynamic subclasses Z_i and Z_j whether the $(n-1)$-tensor corresponding to them are equal in all M motifs of the SM. Once all possible groupings have been accomplished, we must ensure that a state has a unique representation by finding an indexation of dynamic subclasses that yields a "minimal" or canonic representant for a state. To this end, we define a total order on SMs that are equivalent up to a permutation of Z_i, and use the smallest according to this order as canonic representative. This stage is called the canonization phase.

5.1 Minimization

The group operation consists in testing for all target colour classes C of index tC, whether any two dynamic subclasses Z_i and Z_j can be grouped into a single dynamic subclass such that $|Z_k| = |Z_i| + |Z_j|$. This is possible iff all places have the same marking with respect to Z_i and Z_j.

The group operation follows the generic schema described in section 3. Its specific operation arguments are i, j the indexes of the subclasses to be grouped if possible within the target class tC. The operation is initially created with the values $tC = i = j = -1$ meaning that none of these values are bound yet. A group operation $group(tC, i, j)(\mathcal{S})$ thus has the following behaviour:

1. Upon encountering a new colour domain C_k, if tC is yet unbound the operation binds to C_k and lets run the operation with tC unbound by returning $group(-1,-1,-1) + group(k,-1,-1)$. If tC is already bound the operation follows its course normally.
2. When traversing its definition in terms of Z and their cardinality, all possible bindings of i and j are constructed and summed.
3. Upon reaching the end of an M motif $\mathcal{M}$, a first operation *groupable* is run on $\mathcal{M}$ to test whether the group operation is possible with the currently values of i and j. This operation tests the equality of lines i and j of $\mathcal{M}$ and breaks by returning the 0 terminal at the first difference. If $\mathcal{M}$ allows grouping, a second operation is run that deletes the values of line j from $\mathcal{M}$. If groupable returns 0, the global operation also returns 0, pruning the state being constructed from the full DDD.

It should be noted that the group operation thus defined prunes any state that does not allow any grouping. Moreover, whenever two (or more) groupings are possible on

an SM, a single call of the group operation will create two partially grouped SMs. Given the DDD of the newly reached markings, the group operation is therefore called iteratively to stability as in:

group_iter $(\mathcal{S})$:
while groupable $(\mathcal{S}) \neq 0$ **do**
$\mathcal{S} \leftarrow \mathcal{S} -$ groupable $(\mathcal{S}) +$ group$(\mathcal{S})$
end while

5.2 Canonization

In order to obtain a canonic representative of a state, we need to select one of the permutations of dynamic subclass indexes as the canonic one. This is done by *sorting* our DDD. Any ordering criterion is appropriate, as long as it defines a total order over permutations of the Z_is. But to keep the complexity of our operation reasonable, it is essential to define a criterion that can be evaluated as we travel from the root of our DDD to the terminals.

Our sort is thus based on two levels of sort:

- The first level of sort is cardinality based: we require that dynamic subclasses Z_i be encountered in decreasing order of size. This can be evaluated as soon as a C motif definition is encountered.
- Then for two Z_i, Z_j of equal cardinality, we use a lexical sort that defines a total order over tensors of same size.

We define a swap operation that swaps two adjacent Z_i and Z_{i+1} through a whole SM. The behaviour of our sort homomorphism is defined by:

1. Within a C motif, when comparing $|Z_i|$ to $|Z_{i+1}|$, three cases are possible:
   (a) $|Z_i| > |Z_{i+1}|$: The order is already correct, iterate over the next Z_{i+1}, Z_{i+2}
   (b) $|Z_i| < |Z_{i+1}|$: The order is wrong, the procedure swaps Z_i and Z_{i+1} over the rest of the state
   (c) $|Z_i| = |Z_{i+1}|$: Apply a lexical sort on lines i and $i+1$ then continue over the next Z_{i+1}, Z_{i+2}
2. Upon reading a place marking tensor $\mathcal{M}$ for place P, which will only happen if the cardinality sort failed ($|Z_i| = |Z_{i+1}|$), we compute $\mathcal{M}' \leftarrow$swap$(i, i+1)(\mathcal{M})$ and compare $\mathcal{M}$ to $\mathcal{M}'$ lexicographically.
   (a) If $\mathcal{M} = \mathcal{M}'$, $\mathcal{M}$ is put back as marking of P and the lexical sort operation continues downwards,
   (b) if $\mathcal{M} < \mathcal{M}'$ then $\mathcal{M}'$ replaces the previous $\mathcal{M}$ as marking of P and a swap operation is applied downwards,
   (c) if $\mathcal{M} > \mathcal{M}'$ then the DDD is already sorted, $\mathcal{M}$ is put back and the identity homomorphism is returned.

As a single application of the sort operation may not be enough to fully sort a set of states, a sort_iter procedure is defined, that simply iteratively calls the sort homomorphism to stability. When stability is reached, we are ensured that all the SMs of the DDD are fully canonic.

6 Building the state space

We have defined the following operations in the previous sections, all applicable to a set of states $\mathcal{S}$:

- *fire*$(\mathcal{S},t)$: fires the transition t for all possible variable bindings over a set of states $\mathcal{S}$(section 4). The states returned are not canonical however.
- *group_iter*$(\mathcal{S})$: Groups the Z_i that are groupable of a set of states $\mathcal{S}$(section 5.1).
- *sort_iter*$(\mathcal{S})$: Sorts the states of a set $\mathcal{S}$ (section 5.2).

The canonized successors of a set of states $\mathcal{S}$ by symbolic firing of a transition t, for all possible bindings of t's variables, is obtained by:

succ $(\mathcal{S},t)$:
$\mathcal{S} \leftarrow$ *fire*$(\mathcal{S},t)$ {Obtain successors (non-canonic)}
$\mathcal{S} \leftarrow$ *sort_iter*$(\mathcal{S})$ {Apply a canonization (sort)}
$\mathcal{S} \leftarrow$ *group_iter*$(\mathcal{S})$ {Apply minimization (group)}
$\mathcal{S} \leftarrow$ *sort_iter*$(\mathcal{S})$ {Canonize the result (sort)}

Let us note that *sort* is applied twice in the *succ* operation, which may seem counter-productive. Indeed, the group operation will operate correctly whether the Z_i are sorted or not. But in fact this accelerates the procedure because the first *sort* may reduce the number of states in $\mathcal{S}$, as it only keeps one "version" of states identical up to a permutation of Z_i. Furthermore the cache for the *group* operation is used more efficiently as the input $\mathcal{S}$ for the *group* operation is always sorted, and the number of sorted SMs (canonic representatives) is very small w.r.t. the number of unsorted SMs. Finally the second *sort* comes at a very low cost, as the result of the first *sort* is still in cache. Thus the full *sort* operation is only fully evaluated on the newly grouped SMs (this last assertion is only mostly true, since it assumes that most of the nodes in group_iter($\mathcal{S}$) already existed in $\mathcal{S}$).

Given this *succ* operation, the state space reached from a set of initial states $\mathcal{S}$ by firing a set of transitions $\mathcal{T}$ is computed by the following algorithm :

srs $(\mathcal{S},\mathcal{T})$:
$\mathcal{S}_1 \leftarrow \mathcal{S}$
repeat
  $\mathcal{S} \leftarrow \mathcal{S}_1$
  for all $t \in \mathcal{T}$ **do**
    $\mathcal{S}_1 \leftarrow$ *saturate*$(\mathcal{S}_1,t)$
    do garbage collection
  end for
until $\mathcal{S}_1 = \mathcal{S}$

With:

saturate $(\mathcal{S},t)$:
$\mathcal{S}_1 \leftarrow \mathcal{S}$
$\mathcal{S}_2 \leftarrow \mathcal{S}$
repeat
  $\mathcal{S} \leftarrow \mathcal{S}_2$
  $\mathcal{S}_1 \leftarrow$ *succ*$(\mathcal{S}_1,t)$
  $\mathcal{S}_2 \leftarrow \mathcal{S}_2 + \mathcal{S}_1$
until $\mathcal{S}_2 = \mathcal{S}$

We initiate a construction of the full SRS by invoking *srs*$(\mathcal{S}_0,\mathcal{T}_{all})$ where $\mathcal{S}_0$ is the initial state and $\mathcal{T}_{all}$ is the set of all transitions of the model. Thus the algorithm is based on two fixpoint computations: the first fires a transition until it is no longer fireable (*saturate*), the second saturates the firings of each transition successively until no new states are reached (our main *srs*). This double fixpoint method heuristically gives good results, because the cache doesn't need to be cleared within *saturate* (though in truth, we do garbage collection whenever memory consumption exceeds reasonable limits). The

cache doesn't overfill as the operations are exactly the same in each loop of saturate, whereas we cannot hope to store all the intermediate results constructed during a loop of *srs*.

It also tends to saturate place markings, thus increasing sharing: when all the possible markings of a place have been reached, sharing between states and operations increases. This can be understood by considering a system composed of two unrelated places P_1 and P_2; if P_1 has n possible markings and P_2 has m possible markings; when adding newly reached states sharing in the structure will be poor at first, until enough states have been added to allow sharing, and ultimately we will only require two nodes ${P_1}^{all \xrightarrow{n\ values}}{P_2}^{all \xrightarrow{m\ values}}1$. It also favours cache hit, as a transition that only touches P_1 will return *Id* upon reaching P_2, thus all reached markings of P_2 will be concatenated to the new value of P_1.

The evaluation order of transitions also plays an important role on the number of iterations required to reach the fixpoint. The heuristic used to order transitions is to follow approximate flows: we start by evaluating transitions which have all their input places initially marked (whatever their marking thus the approximation); all output places of these transitions are then noted as marked, and the next transitions to be evaluated are again those that have all their input places marked, etc ... Although very simple, this heuristic has given good results over the models tested (2 or 3 iterations).

7 Implementation and results

The table below gives an overview of the performances of our prototype over a few examples taken from literature. The models presented are:

- Peterson's mutual exclusion algorithm for n processes [5]. This protocol is not strongly symmetric: although the process identities (*pid*) can be abstracted away, one must keep track of the level of each process.
- A critical section(CS) protocol with waves ensuring fairness [6]. Processes constitute a wave, then the wave is locked: idle processes can no longer enter the wave until the whole wave has passed into CS. Furthermore, processes within a wave are let through into CS in a static order specified by their *pid*. Although this protocol may seem asymmetric, it functions by taking the process with highest *pid* from a set (the wave) and allowing it into CS. If no other transition of the model distinguishes processes by their identities, we can consider all *pids* equivalent: indeed in any case one of the processes *will* be let through. The performances reported isolate the process of lowest pid 1 in a static subclass. Thus the SRS generated could allow verification of properties such as: the process of $pid = 1$ is always last of his wave to be let into CS.
- A distributed database protocol [7]. This model exhibits considerable symmetry, and allows a high level of sharing within the DDD representation.
- The client server protocol presented in section 1. The performances over this model are parameterized by the number of clients (first column) and the number of servers (second column).

For each model, the number of concrete states is given (this in an over-approximation for the CS Wave protocol), the number of SMs, the number of nodes in the full SRS, the average share (80% share means the reduced DDD representation is 20% of the size of the decision tree with no sharing), the average length of the paths of the resulting DDD in number of nodes (DDDs of SMs have variable length path as we have seen), and the time to compute the SRS are given.

Model	N (#)	States (#)	SM (#)	final nodes (#)	Avg share (%)	Avg SM len (# nodes)	SRS time (sec)
peterson	7	692777	320	6159	65.3 %	55.5	15.7
	10	$3.46 \cdot 10^{9}$	3328	42442	85.1%	85.9	247.0
	11	$7.19 \cdot 10^{10}$	7168	74039	89.4 %	97.3	545.4
	12	$1.62 \cdot 10^{12}$	15360	126807	92.4%	109.6	1946.4
CS Wave	40	$1.74 \cdot 10^{20}$	5620	831	99.3%	23.8	14.83
	100	$1.76 \cdot 10^{49}$	35050	1011	99.8%	24.0	127.88
	200	$1.79 \cdot 10^{97}$	140100	1313	99.9%	24.07	1041.75
	300	$1.02 \cdot 10^{371}$	315150	1600	99.98 %	24.09	3h20
dist. DB	40	$1.77 \cdot 10^{97}$	20101	725	99.8%	27.6	56.95
	300	$5.39 \cdot 10^{370}$	45151	875	99.93%	27.6	205.45
	500	$4.01 \cdot 10^{622}$	125251	1175	99.96%	27.6	1841.68
	700	$4.89 \cdot 10^{623}$	245351	1482	99.97%	27.6	8 hours
Cli. Serv.	5 2	5484	82	884	67.1 %	32.7	3
	10 2	$1.35 \cdot 10^{7}$	476	2419	86.3 %	37.11	28.56
	20 2	$4.17 \cdot 10^{13}$	3201	2914	97.7 %	40.4	116.92
	6 6	$2.44 \cdot 10^{7}$	281	4091	72.5 %	53.1	21.2
	8 8	$1.12 \cdot 10^{11}$	964	13123	80.5 %	69.8	170.4
	9 9	$1.05 \cdot 10^{13}$	1698	22016	83.5 %	78.8	450.5

We can observe that the representation is extremely dense, with as many as $4.8 \cdot 10^{623}$ concrete states represented by only 1500 nodes. This is mainly due to the low number of SMs representing such a state space, only 245,000 in this instance. The average length of an SM, which corresponds to the average number of dynamic subclasses in a static subclass, asymptotically tends toward a structural limit imposed by the P-invariants of the models studied, thus does not follow the evolution of N. For the client/server however, we have worst case SMs with M tensors of size N^2.

Unfortunately computation time does not directly follow the number of nodes. Indeed, the number of SMs is a clear component of the time complexity. This is partly due to the fact that the complexity of evaluating an operation on a node is necessarily linear to the number of sons, as the inductive homomorphism is applied to each son. Our compact encoding generates nodes with a very high number of sons, particularly when the P-invariant bounds have been reached, and only the cardinalities of the Z_i change from one SM to another. We also attribute this in part to the fact that the DDD library is still at a prototype stage, and that caching policies in particular are inefficient. This could certainly be improved by integrating into the DDD library caching and accelerated access algorithms developed for other variants of BDDs [11].

8 Conclusion

We have shown in this paper how the algorithms for SRG construction [1] can be implemented over a DDD [4] representation. The key idea is to exploit both explicitly expressed symmetries by building equivalence classes of states (SMs), and implicit symmetry through the similarities in the representation of these SM. The flexibility offered by DDDs and inductive homomorphisms allows to both represent and operate over complex and dynamic structures, such as the tensors representing place markings in an SM. The prototype developed shows that extremely compact encodings of a state space can be obtained, allowing storage of $4.8 \cdot 10^{623}$ states on 1,500 nodes. Memory consumption is thus very low, however time complexity remains high.

Further directions include improving the DDD library core, with respect to our specific needs, and developing a full LTL model checker using the symbolic observation graph method [9]. We are also interested in developing extensions of the SRG construction, such as the ESRG construction [6] that captures partial symmetries, using our symbolic symbolic framework.

References

1. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, 1993.
2. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *Proc. of ICATPN'2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 103–122. Springer Verlag, June 2000.
3. E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2):77–104, 1996.
4. 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.
5. J.-M. Couvreur and E. Paviot-Adet. New structural invariant for Petri net analysis. In *Proc. of ICATPN'1994*, volume 815 of *Lecture Notes in Computer Science*, pages 199–218. Springer Verlag, June 1994.
6. S. Haddad, J.-M. Ilié, and K. Ajami. A model checking method for partially symmetric systems. In *Proc. of FORTE/PSTV'2000*, pages 121–136. Kluwer, 2000.
7. K. Jensen. Coloured Petri nets. In *Petri Nets: Central Model and their Properties, Advances in Petri Nets 86*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Verlag, September 1986.
8. T. Junttila. New canonical representative marking algorithms for place/transition-nets. In *Proc. of ICATPN'2004*, Lecture Notes in Computer Science. Springer Verlag, June 2004.
9. K. Klai. *Réseaux de Petri : Vérification Symbolique et Modulaire (chapter 3)*. PhD thesis, Laboratoire d'Informatique de Paris 6, December 2003.
10. Y. Thierry-Mieg, C. Dutheillet, and I. Mounier. Automatic symmetry detection in well-formed nets. In *Proc. of ICATPN'2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 82–101. Springer Verlag, June 2003.
11. B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In *Proc. of FMCAD'98*, pages 255–289, 1998.