

# Symbolic State-space Generation of Asynchronous Systems Using Extensible Decision Diagrams

Min Wan and Gianfranco Ciardo

Department of Computer Science and Engineering  
University of California, Riverside  
{mwan, ciardo}@cs.ucr.edu

**Abstract.** We propose a new type of canonical decision diagrams, which allows a more efficient symbolic state-space generation for general asynchronous systems by allowing on-the-fly extension of the possible state variable domains. After implementing both breadth-first and saturation-based state-space generation with this new data structure in our tool SMART, we are able to exhibit substantial efficiency improvements with respect to traditional “static” decision diagrams. Since our previous works demonstrated that saturation outperforms breadth-first approaches, saturation with this new structure is now arguably the state-of-the-art algorithm for symbolic state-space generation of asynchronous systems.

## 1 Introduction

Symbolic state-space generation has received much attention in the past two decades. In particular, techniques based on decision diagrams have been successfully applied to various asynchronous models, including asynchronous circuits, distributed software systems, and globally-asynchronous-locally-synchronous systems (GALS). The compactness and efficiency of decision diagrams lie in the node sharing and intensive usage of operation caches.

The simplest symbolic state-space generation uses a breadth-first strategy [12] and performs an image computation at each iteration, until reaching a global fixpoint. This method can be improved through chaining [16], to compound the effect of asynchronous events within a given breadth-first iteration. The saturation algorithm [3] uses instead a completely different iteration strategy to compute a fixpoint for each decision diagram node in ascending order and has been shown to excel when applied to asynchronous systems. The algorithm was first defined for transition relations that admit a Kronecker encoding. While this encoding always exists, it might be unwieldy, thus the approach was not always practical. [11] avoids the Kronecker encoding by conjunctively partitioning the transition relation of each event into groups, encoded by matrix diagrams with an identity reduction. [5] allows an even finer partition, where conjuncts can share state variables, resulting in a more efficient computation for the symbolic encoding of the transition relation. Fully-reduced decision diagrams encode each conjunct and fully-identity-reduced decision diagrams (similar to identity-reduced matrix diagrams) encode the transition relation of each event.

When computing the fixpoint for a node associated to variable  $x_k$ , saturation applies the union of all events whose root nodes are associated to  $x_k$ . If the state variables have known bounds, the symbolic encoding for the transition relation of each event, as well as arbitrary unions among them, can be computed a priori, and the decision diagram structures chosen by [5, 11] work well. If the bounds on the state variables are unknown, however, we need to update the transition relation of any event affected by or affecting  $x_k$  when the domain of  $x_k$  is extended during state-space generation, resulting in an on-the-fly discovery process. Then, the domain of skipped variables can affect the result of a union operation, thus operation caches must be invalidated. This inefficiency comes from the reduction rules chosen for the decision diagrams, which produce a compact encoding but cause semantic errors when the variable domain is growing.

To tackle this problem, we propose a new data structure, *extensible decision diagrams*, which, using appropriate reduction rules and symbolic operation algorithms, still ensures canonicity, never requires to invalidate the operation cache, allows more node reuse, and still produces a compact symbolic encoding. As it allows efficient bookkeeping of the transition relation for each events and their union, the data structure works well for both breadth-first and saturation.

We provide revised breadth-first and saturation algorithms, as well as a framework to update variable domains on-the-fly and symbolic operations for these new decision diagrams. Experimental results support our contribution.

## 2 Generally-reduced decision diagrams

Consider a discrete-state model  $(\mathcal{S}^{init}, \mathcal{E}, \{\mathcal{R}_e | e \in \mathcal{E}\})$ , whose state is a tuple  $\mathbf{x} = (x_L, \dots, x_1) \in \mathbb{N}^L$  of  $L$  variables with a predefined order  $x_L \succ \dots \succ x_1$  imposed on them,  $\mathcal{S}^{init} \subset \mathbb{N}^L$  is a finite set of *initial* states,  $\mathcal{E}$  is a set of *asynchronous* events, and, for each event  $e \in \mathcal{E}$ ,  $\mathcal{R}_e$  is a transition relation, i.e., a finite set of state pairs, such that  $(\mathbf{i}, \mathbf{i}') \in \mathcal{R}_e$  if the model can move from  $\mathbf{i}$  to  $\mathbf{i}'$  in one step when  $e$  occurs. The overall transition relation is defined as  $\mathcal{R} = \bigcup_{e \in \mathcal{E}} \mathcal{R}_e$ .

### 2.1 Symbolic encoding of sets of states

*Multiway decision diagrams (MDDs)* [9] extend BDDs [1] by allowing integer-valued variables. Given  $N$  variables  $\mathbf{v} = (v_N, \dots, v_1)$  with an order  $v_N \succ \dots \succ v_1$ , each  $v_k$  taking value in a finite set  $\mathcal{V}_{v_k} = \{0, 1, \dots, n_{v_k}\} \subset \mathbb{N}$ , an MDD defined on  $\mathbf{v}$  is a directed acyclic edge-labeled multi-graph where:

- A *nonterminal* node  $p$  is associated to a variable  $p_v = v_k$ , with  $N \geq k \geq 1$ .
- $\mathbf{0}$  and  $\mathbf{1}$  are the *terminal* nodes. Let  $\mathbf{0}_v = \mathbf{1}_v = v_0$  and  $v_k \succ v_0$ , for  $N \geq k \geq 1$ .
- A nonterminal node  $p$  associated with  $v_k$  has  $|\mathcal{V}_{v_k}|$  edges, each labeled with a different  $i \in \mathcal{V}_{v_k}$  and pointing to a node  $q$  with  $p_v \succ q_v$ , we write  $p[i] = q$ . At least one  $p[i]$  must not be  $\mathbf{0}$ .
- *Duplicate* nodes are not allowed, i.e., given two distinct nonterminal nodes  $p$  and  $q$  with  $p_v = q_v$ , there must be an index  $i \in \mathcal{V}_{p_v}$  such that  $p[i] \neq q[i]$ .

Let  $\delta(v_k, v_h) = k - h$  be the distance of variables  $v_k$  and  $v_h$ , with  $v_k \succeq v_h$ . We say that edge  $p[i] = q \neq \mathbf{0}$ , with  $p_v = v_k$ ,  $q_v = v_h$ , skips  $v_{k-1}, \dots, v_{h+1}$  if  $\delta(v_k, v_h) > 1$ . A *reduction rule* [1] is then required to ensure canonicity and, implicitly, define the meaning of such edges. We associate a possibly different reduction rule  $\rho(v_k)$  to each variable  $v_k$ . In particular, we consider the following three reduction rules:



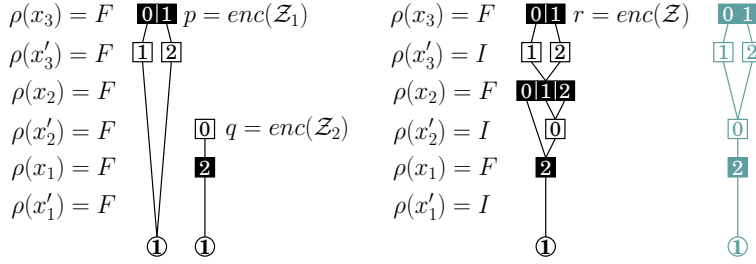


Fig. 2. MDDs encoding  $\mathcal{Z}$  and its sub-relations.

transition relations is a transition relation satisfying  $spt(\mathcal{W}) = spt(\mathcal{Z}) \cup spt(\mathcal{Y})$  and  $top(\mathcal{W}) = \max\{top(\mathcal{Z}), top(\mathcal{Y})\}$ . For example, if  $\mathbf{x} = (x_3, x_2, x_1)$  and  $\mathcal{Z}$  is defined by the boolean expression  $\mathcal{Z} \equiv [x'_3 = x_3 + 1 \wedge x'_2 = x_1 - 2]$ , this transition relation is the natural join of the three sub-relations  $\mathcal{Z}_1 \equiv [x'_3 = x_3 + 1]$ ,  $\mathcal{Z}_2 \equiv [x'_2 = x_1 - 2]$ , and  $\mathcal{Z}^{eq} \equiv [x'_1 = x_1]$ . Then,  $spt(\mathcal{Z}_1) = \{x_3, x'_3\}$ ,  $spt(\mathcal{Z}_2) = \{x_2, x'_2, x_1\}$ , and  $spt(\mathcal{Z}) = \{x_3, x'_3, x_2, x'_2, x_1\}$ , thus  $top(\mathcal{Z}) = x_3$ .

Using a *fully-identity-reduced* (FI) MDD to encode  $\mathcal{Z}$ , where  $\rho(x_k) = F$  and  $\rho(x'_k) = I$ , for  $L \geq k \geq 1$ , every node is associated with a variable in  $spt(\mathcal{Z})$ . Each  $\mathcal{Z}_c$  is instead encoded in a *fully-reduced* (FR) MDD, where  $\rho(x_k) = \rho(x'_k) = F$ , for  $L \geq k \geq 1$ , so that, again, its nodes can only be associated with a variable in  $spt(\mathcal{Z}_c)$ . Assuming all variables take values in  $\{0, 1, 2\}$ , Fig. 2 shows  $p = enc(\mathcal{Z}_1)$  and  $q = enc(\mathcal{Z}_2)$  under FR and  $r = enc(\mathcal{Z})$  under FI. The rightmost MDD, also under FI, is incorrect, demonstrating why the identity-reduced rule does not allow edges from nodes associated with  $x'_3$  to skip over  $x_2$ , which is fully-reduced. Without this, the two rightmost MDDs would both encode  $\mathcal{Z}$ , thus canonicity would be lost. Using FI MDD affords an enormous advantage: neither storing  $\mathcal{Z}^{eq}$  nor performing a natural join with it is needed. The FR MDD for  $\bowtie_{c=1}^C \mathcal{Z}_c$ , interpreted as an FI MDD, encodes  $(\bowtie_{c=1}^C \mathcal{Z}_c) \bowtie \mathcal{Z}^{eq}$  “for free”; minor changes to the FR MDD suffice to transform it into an FI MDD and take into account the new reduction rules for the primed variables in  $spt(\mathcal{Z})$ .

In general discrete-state systems, both asynchronous and synchronous behavior can be present. Then,  $\{\mathcal{R}_e | e \in \mathcal{E}\}$  gives a disjunctive partition of  $\mathcal{R}$  and describes the asynchronous behavior of the system. Each  $\mathcal{R}_e$ , when expressed as a natural join of sub-relations as in Eq. 1, corresponds to the synchronous behavior of the system. Each  $\mathcal{R}_e$  and its sub-relations can be encoded into MDDs as discussed above, and a *union* operation over each  $enc(\mathcal{R}_e)$  results in  $enc(\mathcal{R})$ .

### 2.3 Symbolic generation of the reachable states

Let  $Img(\mathcal{X}, \mathcal{Z}) = \{\mathbf{i}' : \exists \mathbf{i} \in \mathcal{X}, (\mathbf{i}, \mathbf{i}') \in \mathcal{Z}\}$  denote the *image* of the set of states  $\mathcal{X}$  under the transition relation  $\mathcal{Z}$ . The set of reachable states  $\mathcal{S} \subset \mathbb{N}^L$  is the minimal set satisfying  $\mathcal{S} \supseteq \mathcal{S}^{init}$  and  $\mathcal{S} \supseteq Img(\mathcal{S}, \mathcal{R})$ . If  $\mathcal{R}$  can be computed a priori, i.e., the bounds on the state variables are known, we can start from  $\mathcal{S}^{init}$  and repeatedly perform image computations under  $\mathcal{R}$  until reaching a fixpoint. As  $\mathcal{R} = \bigcup_{e \in \mathcal{E}} \mathcal{R}_e$ , we can use the transition relations  $\mathcal{R}_e$  instead of  $\mathcal{R}$  for state-space generation, as long as each  $\mathcal{R}_e$  is applied often enough. Alternatively, we can define transition relations indexed by state variables,  $\mathcal{R}_{x_k} = \bigcup_{top(\mathcal{R}_e) = x_k} \mathcal{R}_e$ , for  $L \geq k \geq 1$ , which leave variable  $x_l$  un-

<pre> mdd GenerateByBFS() 1 <math>s \leftarrow \text{enc}(\mathcal{S}^{\text{init}})</math>; 2 repeat <math>s \leftarrow \text{AddStates}(s)</math>; 3 until <math>s</math> does not change; 4 return <math>s</math>; </pre>	<pre> mdd GenerateBySaturation() 1 mdd <math>s \leftarrow \text{enc}(\mathcal{S}^{\text{init}})</math>; 2 return <math>\text{Saturate}(s)</math>; </pre>
<pre> mdd AddStates(mdd <math>s</math>) 1 <math>r \leftarrow \text{enc}(\mathcal{R})</math>; • <i>standard breadth-first</i> 2 return <math>\text{Or}(s, \text{RelProd}(s, r))</math>; </pre>	<pre> mdd Saturate(mdd <math>s</math>) 1 if <math>\text{CacheHit}(\text{SAT}, s, t)</math> then return <math>t</math>; 2 <math>t \leftarrow \text{NewNode}(s_v)</math>; 3 foreach <math>i \in \mathcal{V}_{s_v}</math> s.t. <math>s[i] \neq \mathbf{0}</math> do 4 <math>t[i] \leftarrow \text{Saturate}(s[i])</math>; • <i>saturate below</i> 5 <math>t \leftarrow \text{UniqueTablePut}(\text{DoFixPoint}(t))</math>; 6 <math>\text{CacheAdd}(\text{SAT}, s, t)</math>; 7 return <math>t</math>; </pre>
<pre> 1' for <math>k = 1</math> to <math>L</math> do • <i>chain by events</i> 2' foreach <math>e \in \mathcal{E}</math> s.t. <math>\text{top}(\mathcal{R}_e) = x_k</math> do 3' <math>r \leftarrow \text{enc}(\mathcal{R}_e)</math>; 4' <math>s \leftarrow \text{Or}(s, \text{RelProd}(s, r))</math>; 5' return <math>s</math>; </pre>	<pre> mdd DoFixPoint(mdd <math>t</math>) 1 <math>r \leftarrow \text{enc}(\mathcal{R}_{t_v})</math>; • <i>by variables</i> 2 repeat 3 foreach <math>i \in \mathcal{V}_{t_v}</math> do 4 foreach <math>i' \in \mathcal{V}_{r[i]_v}</math> s.t. <math>r[i][i'] \neq \mathbf{0}</math> do 5 <math>u \leftarrow \text{RelProd}(t[i], r[i][i'])</math>; • <i>use 8s</i> 6 <math>t[i'] \leftarrow \text{Or}(t[i'], u)</math>; 7 until <math>t</math> does not change; </pre>
<pre> 1'' for <math>k = 1</math> to <math>L</math> do • <i>chain by variables</i> 2'' <math>r \leftarrow \text{enc}(\mathcal{R}_{x_k})</math>; 3'' <math>s \leftarrow \text{Or}(s, \text{RelProd}(s, r))</math>; 4'' return <math>s</math>; </pre>	<pre> 1' repeat • <i>by events</i> 2' foreach <math>e \in \mathcal{E}</math> s.t. <math>\text{top}(\mathcal{R}_e) = t_v</math> 3' mdd <math>r \leftarrow \text{enc}(\mathcal{R}_e)</math>; 4' foreach <math>i \in \mathcal{V}_{t_v}</math> do 5' foreach <math>i' \in \mathcal{V}_{r[i]_v}</math> s.t. <math>r[i][i'] \neq \mathbf{0}</math> 6' <math>u \leftarrow \text{RelProd}(t[i], r[i][i'])</math>; • <i>use 8s</i> 7' <math>t[i'] \leftarrow \text{Or}(t[i'], u)</math>; 8' until <math>t</math> does not change; </pre>
<pre> mdd RelProd(mdd <math>s</math>, mdd <math>r</math>) 1 if <math>s = \mathbf{1}</math> and <math>r = \mathbf{1}</math> then return <math>\mathbf{1}</math>; 2 if <math>\text{CacheHit}(\text{RPR}, s, r, t)</math> then return <math>t</math>; 3 <math>t \leftarrow \text{NewNode}(s_v)</math>; 4 foreach <math>i \in \mathcal{V}_{s_v}</math> s.t. <math>s[i], r[i] \neq \mathbf{0}</math> do 5 foreach <math>i' \in \mathcal{V}_{r[i]_v}</math> s.t. <math>r[i][i'] \neq \mathbf{0}</math> do 6 <math>u \leftarrow \text{RelProd}(s[i], r[i][i'])</math>; 7 <math>t[i'] \leftarrow \text{Or}(t[i'], u)</math>; </pre>	
<pre> 8b <math>t \leftarrow \text{UniqueTablePut}(t)</math>; </pre>	
<pre> 8s <math>t \leftarrow \text{Saturate}(\text{UniqueTablePut}(t))</math>; </pre>	
<pre> 9 <math>\text{CacheAdd}(\text{RPR}, s, r, t)</math>; 10 return <math>t</math>; </pre>	

**Fig. 3.** State-space generation: breadth-first vs. saturation.

changed if  $x_l \succ x_k$ . As  $\mathcal{X}$  and  $\mathcal{Z}$  are encoded as MDDs, the MDD encoding  $\text{Img}(\mathcal{X}, \mathcal{Z})$  is given by the *relational product* of these MDDs.

Two classes of symbolic state-space generation algorithms exist: *breadth-first* [12, 16] and *saturation* [3, 5]. Fig. 3 shows the breadth-first algorithm and two variations based on *chaining* [14]: breadth-first with chaining by events or with chaining by variables. For simplicity, the pseudocode uses QR MDDs, i.e., no edge skips variables. The advantage of chaining is that more states can be found by each global iteration. When chaining by events, the number of iterations required to reach the fixpoint is affected by the order in which events are applied; experimentally, applying them in increasing order of *top* tends to work well [3].

Fig. 3 shows two saturation algorithms, by variables and by events. These differs from breadth-first generation in that they recursively compute a fixpoint at each node, in a low-to-high order of the associated variables. In the pseudocode, breadth-first uses the “8b” variant of *RelProd*, saturation the “8s” one. The two variants differ in that, with “8b”, *RelProd*( $s, r$ ) computes only a *one-*

<p><i>AddValue(variable <math>x_k</math>, index <math>i</math>)</i></p> <ol style="list-style-type: none"> <li>1 <math>\mathcal{V}_{x_k} \leftarrow \mathcal{V}_{x_k} \cup \{i\};</math></li> <li>2 <b>foreach</b> <math>\mathcal{R}_{e,c}</math> s.t. <math>x_k \in \text{spt}(\mathcal{R}_{e,c})</math> <b>do</b></li> <li>3   <b>build</b> <math>\mathcal{R}_{e,c}^{x_k=i}</math> <b>by querying the model;</b></li> <li>4   <math>p \leftarrow \text{enc}(\mathcal{R}_{e,c}^{x_k=i});</math></li> <li>5   <math>\text{enc}(\mathcal{R}_{e,c}) \leftarrow \text{Or}(\text{enc}(\mathcal{R}_{e,c}), p);</math></li> </ol>	<p><i>UpdateVariableTR(variable <math>x_l</math>)</i></p> <ol style="list-style-type: none"> <li>1 <b>if</b> <math>\exists x_k \in \text{spt}(\mathcal{R}_{x_l}), \mathcal{V}_{x_k}</math> <b>changed then</b></li> <li>2   <b>mdd</b> <math>r \leftarrow \mathbf{0};</math></li> <li>3   <b>foreach</b> <math>e \in \mathcal{E}</math> s.t. <math>\text{top}(\mathcal{R}_e) = x_l</math> <b>do</b></li> <li>4     <math>r \leftarrow \text{Or}(r, \text{enc}(\mathcal{R}_e)).</math></li> <li>5   <math>\text{enc}(\mathcal{R}_{x_l}) \leftarrow r;</math></li> </ol>
<p><i>UpdateEventTR(event <math>e</math>)</i></p> <ol style="list-style-type: none"> <li>1 <b>if</b> <math>\exists x_k \in \text{spt}(\mathcal{R}_e), \mathcal{V}_{x_k}</math> <b>changed then</b></li> <li>2   <b>mdd</b> <math>r \leftarrow \mathbf{1};</math></li> <li>3   <b>foreach</b> <math>\mathcal{R}_{e,c}</math> <b>do</b></li> <li>4     <math>r \leftarrow \text{And}(r, \text{enc}(\mathcal{R}_{e,c}));</math></li> <li>5   <math>\text{enc}(\mathcal{R}_e) \leftarrow r;</math></li> </ol>	<p><i>UpdateOverallTR()</i></p> <ol style="list-style-type: none"> <li>1 <b>mdd</b> <math>r \leftarrow \mathbf{0};</math></li> <li>2 <b>for</b> <math>k = 1</math> <b>to</b> <math>L</math> <b>do</b></li> <li>3   <math>r \leftarrow \text{Or}(r, \text{enc}(\mathcal{R}_{x_k}));</math></li> <li>4   <math>\text{enc}(\mathcal{R}) \leftarrow r;</math></li> </ol>

**Fig. 4.** Updating the transition relations.

*step* image of the set encoded by  $s$  and the relation encoded by  $r$ , while, with “8s”, it returns the *fixpoint* of the computed image w.r.t.  $\mathcal{R}_{x_h}$  for  $s_v \succeq x_h$ .

Experimentally, saturation performs far better than breadth-first methods and saturation by variables is preferable to saturation by events. (i.e., in the best cases it is much better and in the worst case it is only slightly worse).

### 3 On-the-fly discovery of the MDD domain

We now review symbolic on-the-fly state-space generation (where the sets  $\mathcal{V}_{x_k}$  are not a priori known) and present an optimized way to update  $\text{enc}(\mathcal{R}_e)$ .

#### 3.1 Extensible state variable domains

When  $\mathcal{R}$  or each  $\mathcal{R}_e$  is known a priori, we can compute  $\mathcal{S}$  with a simple breadth-first or saturation algorithm. However, the bounds on the state variables are often unknown. Then, a state variable can be considered to have an *extensible* domain, which grows during the generation of  $\mathcal{S}$ . In other words, if we do not know the actual bounds on the state variables, we can generate  $\mathcal{R}_e$  on-the-fly alongside  $\mathcal{S}$ . When using decision diagrams, this means adding boolean variables (for BDDs) or increasing the set  $\mathcal{V}_{x_k}$  (for MDDs), as new values for a state variable  $x_k$  are discovered. The key difference from the algorithms of Fig. 3 is that the transition relations must be updated before calling *RelProd*.

[5, 11] proposed the first on-the-fly saturation algorithms with transition relations encoded by decision diagrams. Fig. 4 summarizes their approach to update transition relations: when a *local state index* is *confirmed*, i.e., a new value  $i$  for an unprimed state variable  $x_k$  is found, each sub-relation  $\mathcal{R}_{e,c}$  (*group* in [11], *conjunct* in [5]) with  $x_k \in \text{spt}(\mathcal{R}_{e,c})$  is updated to include the required new state-to-state transitions. This is done by first generating a sub-relation  $\mathcal{R}_{e,c}^{x_k=i}$  encoding these new pairs, then integrating it into  $\mathcal{R}_{e,c}$ . Finally,  $\text{enc}(\mathcal{R}_e)$  is obtained by *intersection* of the sub-relations  $\text{enc}(\mathcal{R}_{e,c})$  of  $e$ , while each  $\text{enc}(\mathcal{R}_{x_l})$  and  $\text{enc}(\mathcal{R})$  are obtained through *unions*. This approach is applicable to both breadth-first and saturation symbolic on-the-fly state-space generation algorithms.

[11] encodes transition relations with identity-reduced *matrix diagrams*, similar to FI MDDs, and requires the support sets of sub-relations from the same  $\mathcal{R}_e$

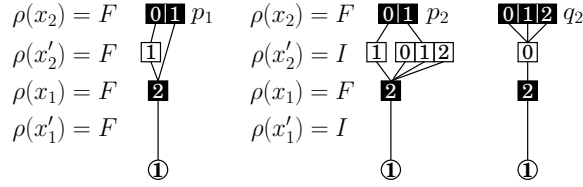


Fig. 5. QFR and QFI MDDs.

to be disjoint. [5], on which our work is based, allows overlapping support sets for the sub-relations from the same  $\mathcal{R}_e$ , thus a finer partitioning of the transition relation and a more efficient state-space generation, using FR MDDs to encode sub-relations and FI MDDs to encode each  $\mathcal{R}_e$ .

### 3.2 An improved implementation

We have seen in Sec. 2.2 that FR MDDs are well suited for sub-relations and FI MDDs for each  $\mathcal{R}_e$ , as these rules produce compact MDDs, respectively [5]. However, two problems arise with these choices. First, the semantic of a skipped variable in a FR MDD changes when its domain grows, so we must be careful when updating the sub-relations  $enc(\mathcal{R}_{e,c})$ . Second, the change from FR to FI when building transition relations from sub-relations requires extra handling.

To illustrate the first problem, consider the example of Fig. 2, modified so that  $\mathcal{Z}_2 \equiv [x'_2 = x_1 - 2]$  applies only if  $x_2 \leq 2$ . When  $\mathcal{V}_{x_2} = \{0, 1, 2\}$ , the FR MDD  $q$  correctly encodes  $\mathcal{Z}_2$ . However, if we later add 3 to  $\mathcal{V}_{x_2}$ ,  $q$  now (incorrectly) also encodes transitions of the form  $(i_3, 3, 2) \rightarrow (i_3, 0, i_1)$ , for  $i_3 \in \mathcal{V}_{x_3}$  and  $i_1 \in \mathcal{V}_{x_1}$ . For the second problem, Fig. 5 assumes a transition relation  $\mathcal{Z}$  with one sub-relation  $\mathcal{Z}_1$ , encoded by node  $p_1$  under FR, in addition to  $\mathcal{Z}^{eq} = [x'_1 = x_1]$ . After intersecting all sub-relations (there is only one), we should obtain  $p_2 = enc(\mathcal{Z})$ . No intersection is performed, but a transformation from  $p_1$  to  $p_2$  is needed since  $x'_2$  changes reduction rule, from  $\rho(x'_2) = F$  to  $\rho(x'_2) = I$ .

We solve these problem through two alternate reduction approaches. With *QFR*,  $\rho(x) = Q$  if  $x \in spt(\mathcal{Z}_c)$ , and  $\rho(x) = F$  otherwise. With *QFI*,  $\rho(x'_k) = I$ , while  $\rho(x_k) = Q$  if  $x_k \in spt(\mathcal{Z})$ , and  $\rho(x_k) = F$  otherwise. We now give an optimized implementation using a QFR MDD to encode each sub-relation  $\mathcal{R}_{e,c}$ . The intersection algorithm of Fig. 6, applied to all sub-relations  $enc(\mathcal{R}_{e,c})$ , produces an MDD  $r$ , which, interpreted as a QFI, is  $enc(\mathcal{R}_e)$ . Although QFR and QFI MDDs might not be as compact as FR and FI, respectively, we gain efficiency. With the new reductions, in Fig. 5,  $p_2 = enc(\mathcal{Z}_1)$  under QFR, while  $p_2 = enc(\mathcal{Z})$  under QFI, no transformation is needed. Also, node  $q$  in Fig. 2, when changed to a QFR MDD, is  $q_2$  in Fig. 5, still encoding the correct set after  $\mathcal{V}_{x_2}$  is enlarged.

## 4 Extensible MDDs to encode the transition relation

Consider a call *UpdateVariableTR*( $x_l$ ) (Fig. 4) after a variable domain  $\mathcal{V}_{x_k}$  has grown, with  $x_l \succeq x_k$ . It seems redundant to perform the union of all  $enc(\mathcal{R}_e)$  with  $top(\mathcal{R}_e) = x_l$  to recompute  $enc(\mathcal{R}_{x_l})$ , when only those  $enc(\mathcal{R}_e)$  containing  $x_k$  in their domain might change. Ideally, we could simply add the new  $enc(\mathcal{R}_e)$ , which have changed due growing  $\mathcal{V}_{x_k}$ , to the original  $enc(\mathcal{R}_{x_l})$ . However, the otherwise very effective (Q)FI reduction used for  $\mathcal{R}_e$  and  $\mathcal{R}_{x_l}$  makes this problematic,

<pre> mdd And(mdd p, mdd q) 1 if p = 0 or q = 0 then return 0; 2 if p = 1 or p = q then return q 3 if q = 1 then return p; 4 if q_v &gt; p_v then Swap(p, q); 5 if CacheHit(AND, p, q, r) then return r; 6 r ← NewNode(p_v); </pre>	<pre> 7 if p_v &gt; q_v then 8   foreach i ∈ V_{p_v} do r[i] ← And(p[i], q); 9 else 10  foreach i ∈ V_{p_v} do r[i] ← And(p[i], q[i]); 11  r ← UniqueTablePut(r); 12  CacheAdd(AND, p, q, r); 13  return r; </pre>
---	--

Fig. 6. Intersection of two QFR MDDs.

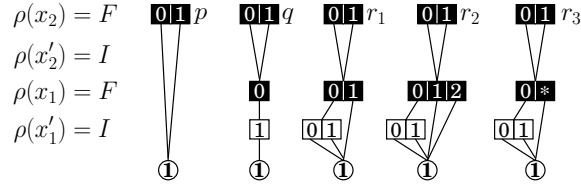


Fig. 7. Problem with updating  $\mathcal{R}_{x_2}$ .

as Fig. 7 illustrates. Assume that  $\mathcal{V}_{x_2} = \{0, 1, 2\}$ ,  $\mathcal{V}_{x_1} = \{0, 1\}$ , and that  $r_1 = \text{enc}(\mathcal{R}_{x_2})$  is obtained as the union of  $p$  and  $q$ . If a new value 2 is added to  $\mathcal{V}_{x_1}$  but  $p$  and  $q$  do not change due to this, one would imagine that  $r_1$  still correctly encodes  $\mathcal{R}_{x_2}$ , thus nothing needs to be done. In fact,  $r_2$ , not  $r_1$ , now correctly encodes  $\mathcal{R}_{x_2}$ , while  $r_1$  lacks some of the required state-to-state transitions. Using  $r_1$  instead of  $r_2$  could then miss reachable states during image computation. Worse yet, the cache for union operations needs to be cleared, further reducing efficiency, since the union of two MDDs becomes incorrect when a domain  $\mathcal{V}_{x_k}$  changes and  $x_k$  is in the support of one but not the other.

We both solve this correctness issue and improve efficiency by introducing a new MDD structure, *extensible MDDs*, with the following modifications from the original MDD definition:

- A nonterminal node  $p$  has an *infinite* set of edges, each labeled with a different index  $i \in \mathbb{N}$  and pointing to a node  $q$ , with  $p_v > q_v$ . However, there is a value  $p_f \in \mathcal{V}_{p_v}$  such that, for  $i > p_f$ , all edges  $p[i]$  point to the same node  $p.\text{ext}$ , while  $p[p_f] \neq p.\text{ext}$ . We write this as  $p[*] = p.\text{ext}$ ,

This new definition changes the set of edges to be infinite for each node  $p$ , but still allows a finite encoding, since we only need to store the  $p_f + 2$  edges  $p[0], p[1], \dots, p[p_f], p[*]$ . We stress that we can still talk about the (finite) variable domains  $\mathcal{V}_{x_k}$ , since sets of states are still encoded using ordinary MDDs, which can also be thought of as the special case of extensible MDDs where  $p[*] = \mathbf{0}$  for all nodes. Of course,  $p[*]$  does not need to be stored explicitly in this case and, strictly speaking, an ordinary MDD node  $p$  stores exactly  $|\mathcal{V}_{p_v}|$  edges while its extensible MDD equivalent does not store edges  $p[i] = \mathbf{0}$  for  $p_f < i \leq n_{p_v}$ , but this can be seen as an implementation detail and, in fact, exactly corresponds to the *truncated full* node format of [3]. Finally, the old reduction rules still apply, if treat an extensible MDD node  $p$  with  $p_v = v_k$  and edges  $p[0], p[1], \dots, p[p_f], p[*]$  as an ordinary MDD node  $q$  with  $q_v = v_k$  and edges

<pre> mdd Or(mdd p, mdd q) 1 if p = 0 or p = q then return q; 2 if q = 0 then return p; 3 if q_v &gt; p_v then Swap(p, q); 4 mdd r; 5 if CacheHit(OR, p, q, r) then return r; 6 r ← NewNode(p_v); 7 if p_v &gt; q_v then 8   r[*] ← Or(r[*], q); • extensible part 9   foreach i ∈ {0, 1, ..., p_f} 10    r[i] ← CopyOf(p[i]); 11   if p[i] ≠ 0 and δ(p, p[i]) = 1 then 12     r[i][i] ← Or(p[i][i], q); 13   r[i] ← UniqueTablePut(r[i]); 14 else • p_v = q_v 15   r_f ← max{p_f, q_f}; 16   foreach i ∈ {0, 1, ..., r_f} do </pre>	<pre> 17   if δ(p, p[i]) = δ(q, q[i]) = 1 then 18     r[i] ← NewNode(p[i]_v) 19     foreach j ∈ V_{p[i]_v} ∪ V_{q[i]_v} do 20       r[i][j] ← Or(p[i][j], q[i][j]); 21   else if δ(p, p[i]) = 1 then 22     r[i] ← CopyOf(p[i]) 23     r[i][i] ← Or(r[i][i], q); 24   else if δ(q, q[i]) = 1 25     r[i] ← CopyOf(q[i]); 26     r[i][i] ← Or(r[i][i], p); 27   else 28     r[i] ← Or(p[i], q[i]); 29   r[i] ← UniqueTablePut(r[i]); 30   r[*] ← Or(p[*], q[*]); • extensible part 31   r_f ← max{i : r[i] ≠ r[*]}; 32   r ← UniqueTablePut(r); 33   CacheAdd(OR, p, q, r); 34   return r; </pre>
--	--

Fig. 8. Union of two extensible QFI MDDs.

<pre> UpdateVariableTR(variable x_l) 1 if ∃x_k ∈ spt(R_{x_l}), V_{x_k} changed then 2   foreach e ∈ E s.t. top(R_e) = x_l do 3     if enc(R_e) changed then 4       enc(R_{x_l}) ← Or(enc(R_{x_l}), enc(R_e)); </pre>	<pre> UpdateOverallTR() 1 for k = 1 to L do 2   if enc(R_{x_k}) changed then 3     enc(R) ← Or(enc(R), enc(R_{x_k})); </pre>
---	--

Fig. 9. Updating the transition relations using extensible MDDs.

$q[0] = p[0], \dots, q[p_f] = p[p_f], q[p_f+1] = p[*], \dots, q[n_{x_k}+1] = p[*]$ , where  $n_{x_k} = \max\{s_f : s_v = x_k\}$ . The set encoded by an extensible MDD node is analogously defined.

This solves the problem of Fig. 7: the union of  $p$  and  $q$  is now  $r_3$  and correctly encodes  $\mathcal{R}_{x_2}$  even if  $\mathcal{V}_{x_1}$  grows. Fig. 8 shows the version of union for two QFI extensible MDDs. This is the *only* operation that can create a node  $r$  with  $r[*] \neq \mathbf{0}$  in our algorithm, and the result of this union is not affected by a later increase in the variable domains. In the pseudocode, the recursive union is guaranteed to be invoked on nodes  $p$  and  $q$  only when  $p_v$  and  $q_v$  are unprimed variables:

$$\begin{aligned} &\text{if } p_v > q_v \text{ then } r_v = p_v, r_f = p_f, r[*] = Or(p[*], q); \\ &\text{if } p_v = q_v \text{ then } r_v = p_v, r_f = \max\{p_f, q_f\}, r[*] = Or(p[*], q[*]). \end{aligned}$$

The intersection of Fig. 6 is not affected by this change either, since  $p[*] = \mathbf{0}$  in any node  $p$  of our QFR MDDs. Using our new extensible MDDs, the update of  $enc(\mathcal{R}_{x_l})$  and  $enc(\mathcal{R})$  can be done more efficiently, as shown in Fig. 9.

## 5 Experimental results

We implemented the proposed approach in SMART [2] and report on experiments run on an Intel Xeon 3.0Ghz workstation with 16GB RAM under SuSE Linux 9.1. Each experiment set (run on a variety of models, referenced in Table 1) is denoted

by a combination X-Y, corresponding to state-space generation algorithm X and reduction Y. For the reduction choices, we use:

- FR: sub-relations and transition relations encoded by ordinary FR MDDs
- QFI: sub-relations encoded by ordinary QFR MDDs, transition relations encoded by ordinary QFI MDDs (Sec. 3.2); on-the-fly updates of Fig. 4.
- EQFI: sub-relations encoded by extensible QFR MDDs, transition relations encoded by extensible QFI MDDs; on-the-fly updates of Fig. 9.

For the state-space algorithm choices, we use:

- B: Standard breadth-first. B-FR is close to NuSMV’s [6] approach, which uses CUDD’s [17] FR BDDs.
- CV: breadth-first with chaining by variables.
- CE: breadth-first with chaining by events (no union of transition relations).
- V: saturation by variables. V-QFI is the improvement of [5] in Sec. 3.2.
- E: saturation by events (as for CE, no union of transition relations is needed).

Table 1 shows runtime, peak memory consumption and number of unions invoked in procedure *UpdateVariableTR* and *UpdateOverallTR* for state-space generation, on a set of models. From it, we can make the following observations:

- For the same algorithm, QFI and EQFI is much better than FR, in both time and memory.
- For the same algorithm, EQFI is 20–80% faster than QFI for most models, while it is at worst 2% slower in the remaining models. QFI and EQFI have similar memory consumption.
- The improvement due to extensible MDDs is more substantial for saturation than for breadth-first.
- CV is preferable to CE, and V is preferable to E. Clearly, it is best to merge events according to their top state variable, even at the cost of additional unions. This further stresses the benefits of extensible MDDs.
- Saturation (V, E) with QFI or EQFI performs much better than breadth-first (B, CV, CE) with any reduction.

We compare the number of unions for transition relations with or without extensible MDDs, specifically, for algorithms B-QFI vs. B-EQFI, CV-QFI vs. CV-EQFI and V-QFI vs. V-EQFI, which accounts for part of the improvements we achieve. For all experiments, thanks to the updating approach of Fig. 9, the number of unions decreases when using extensible MDDs, and this decrease is positively correlated to the time improvement, especially for saturation-based algorithms. When this decrease is large ( $\geq 50\%$ ), e.g., B with *bitshift* and *bit-toggle* or V with *knight*, *leader*, *queen*, *robin*, or *slot*, we achieve substantial time improvement (20–80%) from fewer unions; when the decrease is relatively small, we achieve a minor or no improvement (V with *bitshift*, *kanban*, *phils*, *polling*). Sometimes, even if the decrease is small or there is no decrease in the number of unions, we still achieve a sharp improvement (e.g., V with *bubsort*, *bqueue*, *dme*, *fms*, *ftolerant*, *swapper*), because extensible MDDs do not require to invalidate the caches, making unions more efficient. *intshift* and *rips* show a sharp decrease

Model	$N$	$ S $	BREADTH-FIRST GENERATION																SATURATION													
			B-FR		B-QFI			B-EQFI			CV-FR		CV-QFI			CV-EQFI			CE-QFI		V-FR		V-QFI			V-EQFI			E-QFI			
			sec	MB	sec	MB	Or	sec	MB	Or	sec	MB	sec	MB	Or	sec	MB	Or	sec	MB	sec	MB	sec	MB	Or	sec	MB	Or	sec	MB		
<i>bitshift</i>	6	$2.62 \cdot 10^5$	0.02	0.44	0.01	0.26	173	0.01	0.26	40	0.01	0.18	0.01	0.16	21	0.00	0.16	20	0.01	0.16	0.01	0.16	0.01	0.10	0.01	0.08	22	0.01	0.08	20	0.01	0.08
[5]	256	$4.63 \cdot 10^{77}$	4.95	140	3.23	94.6	2573	2.84	94.6	520	0.75	33.1	0.72	30.6	261	0.59	30.6	260	0.58	30.6	0.47	17.8	0.28	13.6	262	0.28	13.6	260	0.28	13.6		
<i>bittog- gle</i> [8]	100	$1.26 \cdot 10^{30}$	2.40	10.0	0.13	4.49	20199	0.09	4.50	400	2.23	67.9	1.44	67.8	299	1.40	67.8	200	2.51	117	0.05	0.15	0.02	0.10	299	0.02	0.10	200	0.03	0.09		
	200	$1.60 \cdot 10^{60}$	21.4	39.9	0.60	17.8	80399	0.43	17.8	800	30.0	508	22.5	508	599	22.4	508	400	41.5	717	0.15	0.30	0.06	0.19	599	0.04	0.19	400	0.04	0.18		
<i>bqueue</i>	50	$4.57 \cdot 10^9$	13.3	68.9	2.39	36.5	2167	2.38	36.5	813	10.0	46.9	1.41	35.1	602	1.42	35.1	551	16.7	316	-	-	3.20	11.2	854	1.43	11.2	702	8.29	50.1		
[13]	100	$2.70 \cdot 10^{11}$	118	178	20.3	91.5	4333	20.3	91.0	1620	95.9	187	15.1	145	1202	15.1	145	1101	-	-	-	-	40.1	65.9	1704	18.2	66.0	1402	171	160		
<i>bubsort</i>	11	$3.99 \cdot 10^7$	16.5	38.2	15.1	35.9	650	15.1	35.9	180	1.48	33.1	1.08	25.6	65	1.08	25.6	65	1.09	25.6	-	-	1.27	7.21	210	0.62	7.21	210	0.63	7.21		
[5]	15	$1.30 \cdot 10^{12}$	-	-	-	-	-	-	-	-	108	188	113	259	119	113	259	119	113	259	-	-	61.4	201	210	35.0	201	210	35.1	201		
<i>dme</i>	50	$6.30 \cdot 10^{48}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	362	49.4	1.36	24.5	3314	0.94	25.2	3103	1.46	40.6	
[6]	80	$8.59 \cdot 10^{76}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2.38	41.2	5294	1.70	42.3	4963	2.60	67.2		
<i>fms</i>	10	$2.50 \cdot 10^9$	15.6	165	9.37	127	2087	9.33	127	797	4.24	129	3.44	115	343	3.42	115	292	3.17	112	47.8	12.6	0.36	6.29	442	0.26	6.30	374	0.26	5.63		
[10]	20	$6.02 \cdot 10^{12}$	-	-	-	-	-	-	-	-	121	395	94.6	359	678	94.7	359	582	-	-	-	-	5.31	62.1	872	4.19	62.1	744	4.05	55.1		
<i>ftoler- ant</i> [5]	10	$2.95 \cdot 10^{26}$	3.35	63.9	1.25	46.2	6680	1.24	46.5	3780	0.65	28.6	0.58	27.6	950	0.56	27.6	920	3.75	140	3.22	0.18	3.44	2370	0.14	3.63	2270	0.25	5.50			
	30	$6.47 \cdot 10^{78}$	-	-	-	-	-	-	-	-	20.5	726	19.8	727	2850	19.8	726	2760	-	-	246	83.1	3.89	65.8	17540	2.84	67.4	17220	6.65	113		
<i>kanban</i>	10	$1.00 \cdot 10^9$	1.61	31.3	0.49	16.5	1242	0.50	16.5	404	0.48	9.41	0.15	6.12	296	0.15	6.12	291	0.23	10.1	1.77	3.36	0.07	1.82	370	0.06	1.83	332	0.08	2.65		
[10]	50	$1.04 \cdot 10^{16}$	-	-	-	-	-	-	-	-	89.3	1087	33.9	704	1496	33.8	704	1471	48.6	1040	-	-	13.8	252	1890	13.3	252	1692	32.5	266		
<i>knight</i>	5	$6.76 \cdot 10^7$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	12.8	28.3	1.14	25.9	450	0.72	25.8	192	0.62	20.3	
[7]	6	$1.63 \cdot 10^{11}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	313	64.5	727	259	64.6	320	2.72	75.8		
<i>intshift</i>	32	$9.35 \cdot 10^{49}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	14.2	22.0	1.20	19.0	100	1.19	18.9	66	1.26	20.3	
[5]	45	$2.23 \cdot 10^{76}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	76.9	40.3	4.76	25.7	139	4.76	25.7	92	4.98	28.3	
<i>leader</i>	8	$3.04 \cdot 10^8$	97.9	73.3	82.1	58.1	3360	81.9	58.1	1548	80.1	315	68.5	262	2939	68.4	262	1425	-	-	-	-	3.02	38.7	3439	2.40	38.8	1354	13.1	50.6		
[5]	10	$5.02 \cdot 10^{10}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	45.2	58.8	5516	37.7	58.4	2173	46.1	86.5		
<i>phils</i>	50	$2.22 \cdot 10^{31}$	3.79	16.8	0.19	5.40	5600	0.19	5.41	648	0.19	2.63	0.11	2.26	402	0.10	2.27	304	0.17	6.69	0.96	0.89	0.09	0.42	552	0.08	0.43	449	0.08	0.32		
[10]	100	$4.96 \cdot 10^{62}$	29.6	65.9	0.78	20.9	21200	0.73	28.2	1298	0.66	9.09	0.28	8.34	802	0.27	8.36	604	0.68	26.7	3.80	1.80	0.17	0.83	1102	0.16	0.85	899	0.16	0.65		
<i>polling</i>	15	$3.35 \cdot 10^{19}$	38.4	28.6	0.44	7.04	4244	0.45	7.05	975	1.48	7.17	0.26	4.28	525	0.26	4.29	509	0.27	3.80	5.74	1.25	0.21	0.86	384	0.19	0.86	341	0.24	0.83		
[10]	30	$3.25 \cdot 10^{46}$	-	-	4.99	71.3	30314	4.91	126	3750	38.6	65.1	1.66	42.9	1950	1.65	43.0	1919	1.83	37.9	196	6.26	0.89	4.65	1224	0.86	4.65	1136	0.88	4.62		
<i>queen</i>	12	$8.56 \cdot 10^5$	2.39	83.2	2.30	82.5	356	2.31	82.5	145	3.74	128	3.71	127	266	3.72	127	133	15.9	553	85.1	65.6	2.98	65.2	266	1.79	65.2	133	2.16	65.2		
[7]	13	$4.67 \cdot 10^6$	12.2	381	11.9	378	418	11.9	378	170	19.3	595	19.1	595	314	19.0	595	157	-	536	330	16.7	330	314	10.9	330	157	12.9	330			
<i>rips</i>	5	$2.97 \cdot 10^{13}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	296	810	2103	290	807	949	-	-		
[15]	10	$8.87 \cdot 10^{14}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	414	904	2103	404	902	949	-	-		
<i>robin</i>	100	$2.85 \cdot 10^{32}$	233	517	55.7	333	97540	55.4	331	1699	78.0	439	15.2	520	21494	15.1	520	700	39.3	1004	581	217	3.50	51.3	22088	2.00	51.2	799	2.99	77.6		
[5]	200	$7.23 \cdot 10^{62}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	25.2	331	84188	13.5	331	1599	25.2	593		
<i>slot</i>	50	$1.72 \cdot 10^{52}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1.09	6.00	1256	0.43	6.02	600	1.02	12.5		
[5]	70	$3.12 \cdot 10^{73}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2.64	14.1	1756	1.00	14.1	840	2.54	31.3		
<i>swap- per</i> [5]	20	$1.31 \cdot 10^{11}$	1.89	23.4	0.54	15.1	15716	0.50	15.1	232	0.26	14.4	0.21	13.7	96	0.21	13.7	96	0.21	13.7	0.21	0.22	0.02	0.17	96	0.01	0.17	96	0.01	0.17		
	100	$8.96 \cdot 10^{98}$	-	-	-	-	-	-	-	-	312	443	300	448	496	297	448	496	300	448	25.9	3.50	0.27	2.94	496	0.11	2.94	496	0.11	2.94		

Table 1. Results (“sec”: time in seconds; “MB”: memory in Megabytes; “Or”: number of unions; “-”: runtime > 600sec).

in the number of unions under  $V$ , but do not improve runtime much, because unions are a small part of state-space generation for these models.

We can then experimentally surmise that EQFI usually improves the runtime, for some benchmarks greatly so, while never degrades it in appreciable ways, and does not consume more memory, so it is preferable to QFI.

## 6 Conclusion

We presented a novel canonical decision diagram structure, *extensible MDDs*, suitable for symbolic state-space generation on-the-fly, i.e., when the possible values for the variables describing the global system state are not known a priori. Through a set of experiments, we showed how extensible MDDs improve both breadth-first and saturation-based algorithms. In the future, we intend to explore the application of extensible decision diagrams to other classes of symbolic algorithms, beyond state-space generation.

## References

1. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.
2. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.
3. G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *STTT*, 8(1):4–25, Feb. 2006.
4. G. Ciardo and M. Wan. Generally-reduced decision diagrams: definition and applications. Submitted for publication.
5. G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Proc. CHARME*, LNCS 3725, pages 146–161, 2005.
6. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *Software Tools for Technology Transfer*, 2(4):410–425, 2000.
7. H. E. Dudeney. *Amusements in Mathematics*. Dover, 1970.
8. G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
9. T. Kam et al. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
10. A. S. Miner. Implicit GSPN reachability set generation using decision diagrams. *Perf. Eval.*, 56(1-4):145–165, Mar. 2004.
11. A. S. Miner. Saturation for a general class of models. In *Proc. QEST*, pages 282–291, 2004.
12. E. Pastor et al. Petri net analysis using boolean manipulation. In *Proc. ICATPN*, LNCS 815, pages 416–435, 1994.
13. B. Plateau et al. The PEPS software tool. In *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 2794, pages 98–115, 2003.
14. O. Roig et al. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *Proc. ICATPN*, LNCS 935, pages 374–391, 1995.
15. R. Siminiceanu and G. Ciardo. Formal verification of the NASA Runway Safety Monitor. *Software Tools for Technology Transfer*, 9(1):63–76, Feb. 2007.
16. M. Solé and E. Pastor. Traversal techniques for concurrent systems. In *Proc. FMCAD*, LNCS 2517, pages 220–237, 2002.
17. F. Somenzi. CUDD: CU Decision Diagram Package, Release 2.3.1. <http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html>.