Improving Saturation Efficiency with Implicit Relations

Shruti Biswal and Andrew S. Miner

Iowa State University, Ames IA 50010, USA
{sbiswal,asminer}@iastate.edu

Abstract. Decision diagrams are a well-established data structure for reachability set generation and model checking of high-level models such as Petri nets, due to their versatility and the availability of efficient algorithms for their construction. Using a decision diagram to represent the transition relation of each event of the high-level model, the saturation algorithm can be used to construct a decision diagram representing all states reachable from an initial set of states, via the occurrence of zero or more events. A difficulty arises in practice for models whose state variable bounds are unknown, as the transition relations cannot be constructed before the bounds are known. Previously, on-the-fly approaches have constructed the transition relations along with the reachability set during the saturation procedure. This can affect performance, as the transition relation decision diagrams must be rebuilt, and compute-table entries may need to be discarded, as the size of each state variable increases. In this paper, we introduce a different approach based on an implicit and unchanging representation for the transition relations, thereby avoiding the need to reconstruct the transition relations and discard compute-table entries. We modify the saturation algorithm to use this new representation, and demonstrate its effectiveness with experiments on several benchmark models.

Keywords: Petri nets, decision diagram, saturation, reachability set generation

1 Introduction

High-level formalisms can be used to model complex discrete-state systems. The generation of the reachable state space, or *reachability set*, for such systems is an essential step for different kinds of studies. Formal verification techniques, such as model checking, may require the entire state space of a system to verify that some or all states satisfy certain properties, such as the presence of a safety property at all states. However, the reachability set of a system can be extremely large due to the state explosion problem, making the generation of the reachability set difficult.

Present-day *symbolic* techniques usually outperform the traditional *explicit* techniques used for reachability set generation. The *saturation*[7] algorithm is

one such symbolic strategy for reachability set generation. An efficient implementation [14] of the saturation algorithm uses *multi-valued decision diagram* (MDD) representations [13] for encoding sets of reachable states and *matrix diagram representations* (MxD) for transition relations of the models.

However, a significant complication arises when the variables of the system have unknown bounds. For such systems, *on-the-fly techniques* [7, 19] for transition relation construction are used, that require expansion of transition relations every time new bounds for variables are discovered which amounts to additional changes to compute-table entries. While the information in the transition relation is crucial to the application of symbolic techniques, its repeated construction in the form of MxD affects the overall efficiency of symbolic methods involved in reachability set analysis. When the events of a system are such that the enabling of each event is co-dependent on multiple variables, and the firing of the event affecting each variable is independent of other variables, the re-building of transition relations becomes an overhead because the growth of variable bound is a function of the variable itself. For the aforementioned systems, a more efficient technique would be to use a static representation of transition relations in order to restrict the modification of compute-table entries to the development of the reachability set alone. The focus and contribution of this paper is to devise a data structure, called an *implicit relation forest*, that encodes the events of such systems and uses the saturation algorithm for reachability set generation with reduced time and memory expense. To provide strong evidence of improvement in reachability set generation process, the paper proposes a modified saturation algorithm that uses implicit relations to conduct experiments and compares performance results with that of the on-the-fly saturation algorithm.

The rest of the paper is organized as follows. Section 2 defines the class of models we consider, and briefly recalls decision diagrams and on-the-fly saturation. Section 3 introduces implicit relation forests, details their construction from a model, and presents the saturation algorithm modified to use the alternate representation for model events. Section 4 discusses the related work and compares them with implicit relations. Section 5 describes the experimental evaluation of the devised method on an extensive set of Petri net models collected from the annual Model Checking Competition (MCC) [1]. Finally, Section 6 draws conclusions and discusses future research directions.

2 Background

This section describes the class of models that we consider in the paper, recalls the basics of decision diagrams, MDDs, and MxDs, and the saturation algorithm.

2.1 Model definition

Rather than restricting our discussion to a particular formalism, we consider a class of generic high-level discrete-state models that includes many existing formalisms. A discrete-state model $\mathcal{M}$ is defined by a tuple $(\mathcal{V}, \mathcal{E}, \mathbf{i}_0, \Delta)$ where:

- $\mathcal{V} = \{v_1, v_2, \ldots, v_L\}$ is a finite set of *state variables* of the model. Each state variable v_k can assume a value from the set of natural numbers. A (*global*) state $\mathbf{i}$ of $\mathcal{M}$ is then an L-tuple $(i_1, i_2, \ldots, i_L) \in \mathbb{N}^L$.
- $\mathcal{E} = \{e_1, e_2, \ldots, e_{|\mathcal{E}|}\}$ is a finite set of *events* of the model.
- $\mathbf{i}_0 \in \mathbb{N}^L$ is the initial state of the model.
- $\Delta : \mathbb{N}^L \times \mathcal{E} \nrightarrow \mathbb{N}^L$ is the next state (partial) function. If $\Delta(\mathbf{i}, e)$ is defined, we say event e is *enabled* in state $\mathbf{i}$, and if event e *occurs* then the model changes from state $\mathbf{i}$ into state $\Delta(\mathbf{i}, e)$. Otherwise, if $\Delta(\mathbf{i}, e)$ is undefined, we say event e is *disabled* in state $\mathbf{i}$.

  We require that each event e can be expressed using L *local* next state (partial) functions, $\Delta_{e,1}, \ldots, \Delta_{e,L}$, such that

  $$\Delta((i_1, \ldots, i_L), e) \; = \; (\Delta_{e,1}(i_1), \ldots, \Delta_{e,L}(i_L)).$$

  In other words, the value of a single local variable v_k is enough to disable an event, and the change in state variable v_k when an event occurs may depend only on state variable v_k. Furthermore, each event is deterministic: for a given input state, an event can produce at most one output state. However, the model may be nondeterministic, as several events may be enabled in a given state.

For example, an ordinary Petri net [16] can be expressed using our model: the set $\mathcal{V}$ can correspond to the set of Petri net places, the set $\mathcal{E}$ can correspond to the set of Petri net transitions, the initial state $\mathbf{i}_0$ will correspond to the initial marking of the Petri net, and Δ will correspond to the Petri net firing rules. Specifically, for a place p_k and a transition t, $\Delta_{t,k}(i_k)$ is defined if i_k is greater or equal to the number of edges from p_k to t, and $\Delta_{t,k}(i_k) = j_k$ if $j_k - i_k$ equals the number of edges from t to p_k minus the number of edges from p_k to t. Petri nets with inhibitor arcs can also be expressed. An example "fork-join" Petri net model is shown in Figure 1, where the circles correspond to places and the squares correspond to transitions. Transition t_1 performs a fork operation and transition t_6 performs a join operation.

Petri nets with marking-dependent arc cardinalities can sometimes be expressed using our model. Effectively, if the cardinality on an edge from p_i to t or from t to p_i depends on the number of tokens in a place p_j, then places p_i and p_j could be grouped together in a single model variable $v_l \in \mathcal{V}$. Alternatively, transition t can be split into several model events, each representing a portion of t. For example, we might use events $e_1, e_2, e_3, \ldots$ where event e_n simulates transition t but only when there are exactly n tokens present in place p_i. These modifications are necessary to express $\Delta(\mathbf{i}, t)$ in terms of local functions $\Delta_{e,l}(i_l)$ that depend only on the local model state variable. Transition guards can be handled in a similar manner. This is the *Kronecker consistency* requirement discussed in [6], and in practice it limits the applicability of our approach to models that can be obtained by merging only a few places together and whose number of events (including those obtained from splitting transitions) is small.

For a given model $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathbf{i}_0, \Delta)$, we can define the following.

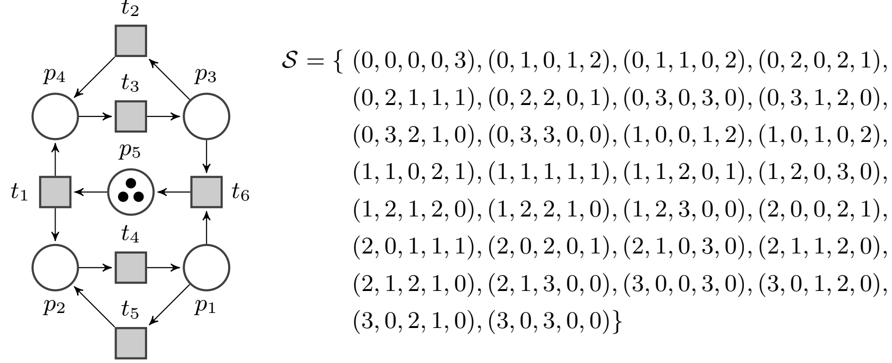

Fig. 1: A fork-join Petri net model (left) and its reachability set (right).

- The next state function for event e, $\mathcal{N}_e : \mathbb{N}^L \to 2^{\mathbb{N}^L}$, is defined as $\mathcal{N}_e(\mathbf{i}) = \{\mathbf{j} : \Delta(\mathbf{i}, e) = \mathbf{j}\}$. We then define the overall next state function as $\mathcal{N}(\mathbf{i}) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(\mathbf{i})$, which gives the set of states reachable via the occurrence of one event from a single starting state, and further extend this to sets of starting states: $\mathcal{N}(\mathcal{I}) = \bigcup_{\mathbf{i} \in \mathcal{I}} \mathcal{N}(\mathbf{i})$.
- The *reachability set* $\mathcal{S} \subseteq \mathbb{N}^L$ is the set of states reachable via the occurrence of zero or more events from the initial state $\mathbf{i}_0$, and is the least fixed point satisfying $\mathcal{S} = \{\mathbf{i}_0\} \cup \mathcal{S} \cup \mathcal{N}(\mathcal{S})$.

As an example, the reachability set $\mathcal{S}$ is shown in Figure 1 for the fork-join Petri net model, where a state is shown as $(p_1, p_2, p_3, p_4, p_5)$.

The focus of this paper is on algorithms to generate the set $\mathcal{S}$, using decision diagrams. This is a necessary first step for many types of analysis, including verification of safety properties or model checking of more complex properties specified in a temporal logic. For this work, we assume that $\mathcal{S}$ is finite (in general there is no guarantee of this). Note that the set $\mathcal{S}$ is finite if and only if every state variable is bounded. We do not require knowledge of these bounds *a priori*; instead, our reachability set generation algorithm will discover these bounds.

2.2 Multi-valued decision diagrams and matrix diagrams

An ordered multi-valued decision diagram (MDD) [13] defined over the sequence of L domain variables $(u_L, \ldots, u_1)$, with a given variable order such that $u_l \succ u_k$ iff $l > k$ and a specified domain for each variable $\mathcal{D}(u_k) = \{0, 1, 2, \ldots, n_k - 1\}$, is a directed acyclic edge-labelled graph where:

- Each node m is associated with some variable, denoted as $m.var$.
- There are two *terminal* nodes, $\mathbf{0}$ and $\mathbf{1}$. These are associated with a special variable u_0, satisfying $u_k \succ u_0$ for any domain variable u_k.
- Each *non-terminal* node m is associated with a domain variable u_k and $\forall i_k \in \mathcal{D}(u_k)$, there is an edge labelled with i_k pointing to a child $m[i_k]$.

– The variable associated with any child $m[i_k]$ of a non-terminal node m is guided by the variable order such that $m.var \succ m[i_k].var$.

A node m in an MDD encodes a function $f_m : \mathcal{D}(u_L) \times \cdots \times \mathcal{D}(u_1) \to \{0, 1\}$, defined recursively by

$$f_m(i_L, \ldots, i_1) = \begin{cases} m, & \text{if } m.var = u_0 \\ f_{m[i_k]}(i_L, \ldots, i_1), & \text{if } m.var = u_k \succ u_0 \end{cases}$$

Non-terminal node n is a *duplicate* of node m if $n.var = m.var$, and if $n[i] = m[i], \forall i \in \mathcal{D}(n.var)$. Note that duplicate nodes n and m encode the same function: $f_m = f_n$. Non-terminal node m is *redundant* if $m[i] = m[0], \forall i \in \mathcal{D}(m.var)$; note that f_m is independent of variable $m.var$ in this case and $f_m = f_{m[0]}$. An MDD is *fully reduced* if it contains no duplicate nodes and no redundant nodes. It can be shown that fully reduced MDDs are a canonical form: any function can be represented uniquely ($f_m = f_n$ if and only if $m = n$). For our work, we instead use *zero reduced* MDDs, which contain no duplicate nodes, and *require* redundant nodes except for terminal node $\mathbf{0}$. More formally, for any non-terminal node m with $m.var = u_k$, we require, for all i, that either $m[i].var = u_{k-1}$ or $m[i] = \mathbf{0}$. This is done because we allow the MDD domain variables to grow, or equivalently, we assume the MDD domain variables are unbounded but each MDD node contains only finitely many non-zero children.

Given a model $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathbf{i}_0, \Delta)$, a finite set of states $\mathcal{X} \subset \mathbb{N}^L$ can be encoded as an MDD as follows.

– The MDD domain variables $(u_L, \ldots, u_1)$ correspond to the model state variables $\mathcal{V}$. For simplicity of presentation, we assume that $\forall i, u_i = v_i$; in practice, the variables can be ordered differently and there can be non-trivial mappings from state variables to domain variables (for example, several state variables could be collected into a single domain variable).
– The set $\mathcal{X}$ can be encoded by an MDD node m such that f_m is the characteristic function for the set $\mathcal{X}$: $f_m(i_L, \ldots, i_i) = 1$ iff $(i_1, \ldots, i_L) \in \mathcal{X}$.

The MDD encoding of $\mathcal{S}$, for the fork-join Petri net, is shown in Figure 2. To increase readability, only paths that lead to terminal node **1** are shown.

An ordered matrix diagram (MxD) [14] is defined similarly to an MDD, except that each non-terminal edge is labelled with a *pair*:

– Each *non-terminal* node m is associated with a domain variable u_k and $\forall (i_k, j_k) \in \mathcal{D}(u_k) \times \mathcal{D}(u_k)$, there is an edge labelled with (i_k, j_k) pointing to a child $m[i_k, j_k]$ such that $m.var \succ m[i_k, j_k].var$.

A node m in an MxD encodes a function $f_m : \mathcal{D}^2(u_L) \times \cdots \times \mathcal{D}^2(u_1) \to \{0, 1\}$, given by $f_m = f_{m,L}$, where $f_{m,L}$ is defined recursively as

$$f_{m,L}(i_L, j_L, \ldots, i_1, j_1) = \begin{cases} m, & \text{if } L = 0 \\ f_{m[i_L,j_L],L-1}(i_L, j_L, \ldots, i_1, j_1), & \text{if } m.var = u_L \\ f_{m,L-1}(i_L, j_L, \ldots, i_1, j_1), & \text{if } u_L \succ m.var \ \wedge \ i_L = j_L \\ 0, & \text{otherwise.} \end{cases}$$

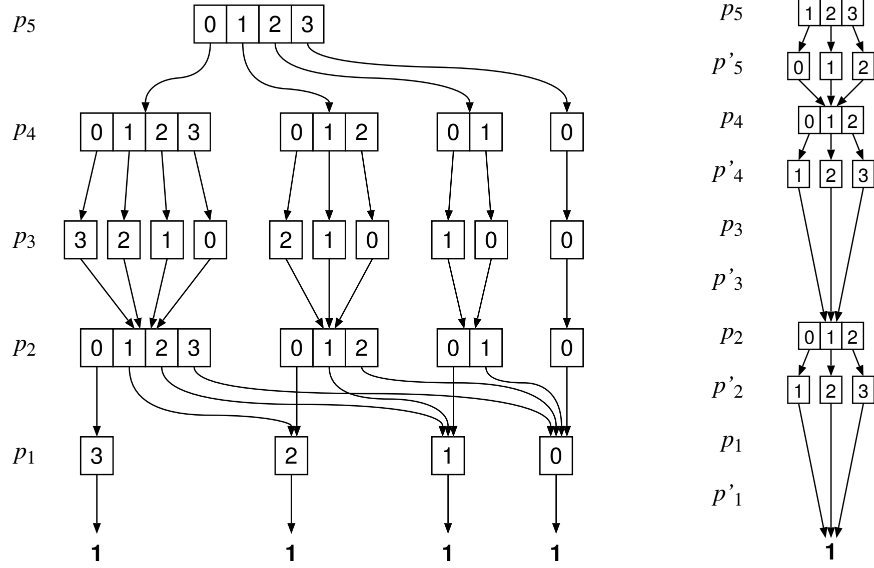

Fig. 2: MDD for $\mathcal{S}$ of fork-join Petri net (left) and MxD/MDD for t_1 (right)

The definition of *duplicates* is similar to MDDs. A non-terminal node m is an *identity* node if (1) $m[i, i] = m[0, 0], \forall i \in \mathcal{D}(m.var)$, and (2) $m[i, j] = \mathbf{0}, \forall i \neq j$. An MxD is *reduced* if it contains no duplicate nodes and no identity nodes. In practice, MxDs can be implemented as MDDs on twice as many domain variables. If the MxD domain variables are $(u_L, \ldots, u_1)$, then the MDD domain variables are $(u_L, u'_L, \ldots, u_1, u'_1)$, with the variable ordering defined such that u'_i immediately follows u_i. However, the MDDs are not fully reduced, but instead use a special *identity reduction* for primed variables [10].

Given a model $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathbf{i}_0, \Delta)$ and bounds for each state variable, function $\mathcal{N}_e$ can be encoded as an MxD as follows. Again, the domain variables correspond to the state variables $\mathcal{V}$. Then we use an MxD node m such that $f_m(i_L, j_L, \ldots, i_1, j_1) = 1$ iff $(j_L, \ldots, j_1) \in \mathcal{N}_e((i_L, \ldots, i_1))$. The MxD encoding for $\mathcal{N}_{t_1}$, for the fork-join Petri net example, is shown in Figure 2. Note that levels p_3, p'_3, p_1, p'_1 are skipped because places p_3 and p_1 are completely unaffected by transition t_1, and thus its occurrence does not change state variables p_3 and p_1. Also note that, because our model requires that an event occurrence changes a state variable in isolation (without considering the values of the other state variables), all MxD encodings of events will have a similar linear shape, where there can be a fanout only from u_k to u'_k and all non-zero pointers from u'_k must point to a single node.

2.3 On-the-fly saturation using extensible decision diagrams

Given two MDD nodes m and n, encoding functions f_m and f_n, the MDD node p encoding function $f_p = f_m \oplus f_n$ for a binary operation $\oplus$ is constructed in a recursive "Apply" operation [4]. An example of this is algorithm Union, shown in Figure 5, which constructs a new MDD encoding the union of two sets, passed as arguments and encoded as MDDs. Like most "apply" operations, Union simultaneously traverses the graphs rooted at nodes m and n, constructing a new graph rooted at p containing the result. Procedure UniqueInsert, called in line 13, eliminates duplicate nodes during construction: if the newly created node p is a duplicate of some other node q, then node p is discarded and node q is returned; otherwise, node p is added to the unique table and is returned. Duplicate computation arising from repeated recursive calls with the same arguments is avoided using a compute-table $\mathcal{C}$ (c.f. lines 4 and 14). This bounds the computational cost and the size of the resulting graph to be at worst the product of the sizes of the input graphs.

For efficiency, specialized relational product operations can be implemented (e.g., [20]) to construct the MDD for $\mathcal{N}(\mathcal{X})$ or $\mathcal{N}_e(\mathcal{X})$, when set $\mathcal{X}$ is encoded as an MDD and $\mathcal{N}$ is encoded as an MxD or MDD. A straightforward breadth-first iteration based on the fixed point equation $\mathcal{S} = \{\mathbf{i}_0\} \cup \mathcal{S} \cup \mathcal{N}(\mathcal{S})$ can then be used to generate $\mathcal{S}$. However, the iteration strategy of *node saturation* [7] can be orders of magnitude more efficient in practice and is known to terminate whenever $\mathcal{S}$ is finite.

Difficulty arises in practice for models whose state variable bounds are difficult or impossible to obtain, or are conservative. This can be alleviated using an "on-the-fly" variant of saturation [8, 10, 15, 19], which allows variable bounds to be discovered while generating $\mathcal{S}$. This is done by distinguishing between *confirmed* local states that are known to appear in at least one reachable global state, and *unconfirmed* local states. The encoding of $\mathcal{N}$ contains all transitions out of confirmed local states, leading to confirmed or unconfirmed local states. During the saturation procedure, when an unconfirmed local state is discovered to be part of a reachable global state, it is confirmed and the encoding of $\mathcal{N}$ must be expanded to include any transitions out of the newly confirmed local state. For a Kronecker-based encoding of $\mathcal{N}$ [8], this expansion is straightforward; for more general encodings of $\mathcal{N}$ [10, 15], this expansion requires rebuilding the MxD/MDD encoding of $\mathcal{N}$, and even worse, often discards compute-table information that could eliminate duplicate computations during the saturation procedure. This happens because, as the sizes of the state variable domains grow, the nodes in the encoding of $\mathcal{N}$ must also grow, and it is possible for the MxD/MDD for a next state function to change shape when the state variable domains increase. *Extensible* MDD nodes were introduced [19] to address exactly this issue, but unfortunately only certain repeating patterns can be exploited by extensible nodes.

3 Implicit Relations

Motivated by the requirement of current on-the-fly saturation methods to update (with varying degrees of computational overhead) their encodings of $\mathcal{N}$ as state variable domains increase in size, in this section we introduce *implicit relations* to encode $\mathcal{N}$ independently of the variable domain size. This new representation retains useful properties of MxDs, namely the ability to exploit identity structures and allowing nodes to be shared (i.e., have several incoming pointers). We also modify the on-the-fly saturation algorithm to work with implicit relations.

3.1 Definition

An ordered *implicit relation forest* defined over the sequence of L domain variables $(u_L, \ldots, u_1)$ is a directed acyclic graph where:

- Each node r is associated with some variable, denoted as $r.var$.
- There is a single terminal node, **1**, associated with a special variable u_0 satisfying $u_k \succ u_0$ for any domain variable u_k.
- Each non-terminal *relation node* r is associated with a domain variable u_k, and contains a partial function $r.\delta : \mathcal{D}(u_k) \nrightarrow \mathcal{D}(u_k)$.
- Relation node r contains a single outgoing edge, $r.ptr$, consistent with the variable order such that $r.var \succ r.ptr.var$.

An implicit relation forest contains $|\mathcal{E}|$ implicit relations, where each relation corresponds to an event $e \in \mathcal{E}$ and is uniquely identified by the top-most relation node. A node r in an implicit relation forest encodes a function $f_r : \mathcal{D}^2(u_L) \times \cdots \times \mathcal{D}^2(u_1) \to \{0, 1\}$, given by $f_r = f_{r,L}$, where $f_{r,L}$ is defined recursively as

$$f_{r,L}(i_L, j_L, \ldots, i_1, j_1) = \begin{cases} 1, & \text{if } L = 0 \\ f_{r.ptr,L-1}(i_L, j_L, \ldots, i_1, j_1), & \text{if } r.var = u_L \ \wedge \ j_L = r.\delta(i_L) \\ f_{r,L-1}(i_L, j_L, \ldots, i_1, j_1), & \text{if } u_L \succ r.var \ \wedge \ i_L = j_L \\ 0, & \text{otherwise.} \end{cases}$$

Let $\mathcal{U}_k$ represent the sequence of variables $u_k, \ldots, u_1$. Then $f_{r,k}$ encodes the effect of an event on $\mathcal{U}_k$. Either u_k, a variable associated with node r, participates in the event and the post-event value of u_k is $j_k = r.\delta(i_k)$ with values of $\mathcal{U}_{k-1}$ given by $f_{r.ptr,k-1}$, or u_k does not participate in the event and the value of u_k remains unchanged, $j_k = i_k$. For the latter case, the variable u_k is not associated with r and hence, $f_{r,k}$ defines the effect of the event on the variables $\mathcal{U}_{k-1}$, recursively by $f_{r,k-1}$

A relation node p is a *duplicate* of relation node r if $p.var = r.var$, $p.ptr = r.ptr$, and $p.\delta = r.\delta$. From now on, we assume that the implicit relation forest contains no duplicate nodes.

Algorithm BuildImplicit, shown in Figure 5, constructs a set of relation nodes, $\mathcal{R}$, for a given model $\mathcal{M}$. It builds a relation for each event $e \in \mathcal{E}$, from the bottom up. In the algorithm, we loop over variables u_i where $\Delta_{i,e}$ is not the identity function (c.f. line 4); this corresponds to variables that either affect the enabling

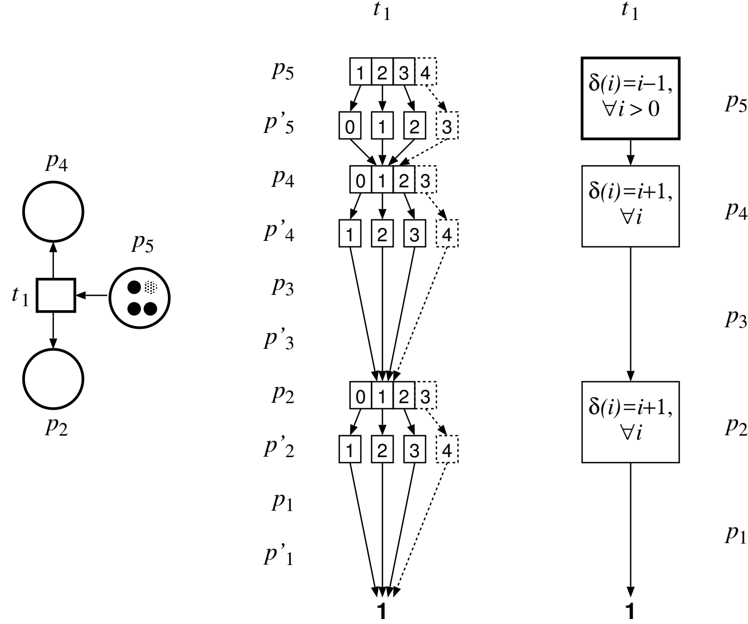

Fig. 3: Representation of transition t_1, from the fork-join Petri net model, in MxD and implicit relation. Dashed entities in the MxD represent the modification in the decision diagram due to additional token in place p_5, represented by dotted-circle.

of e or are changed when e occurs. For each such variable u_i, we create a new relation node with function $\Delta_{i,e}$, pointing to the node below it. After eliminating duplicates (c.f. line 11), the top-level node encoding $\mathcal{N}_e$ is added to the set $\mathcal{R}$.

Figure 3 compares the structure of implicit relation and MxD for transition t_1 of the fork-join Petri net model from Figure 1. Note that for every additional token in place p_5, the MxD for this transition undergoes modification in terms of expansion of variable bound for each place. On the contrary, the implicit relation remains unchanged. The figure showcases the benefit of representing transitions of a model using implicit relation over MxD.

Figure 4 shows the implicit relation forest for all events of the fork-join Petri net model. The set $\mathcal{R}$ contains the top-most node for each event. Each node r is annotated with $r.\delta$ along with the values for which $r.\delta$ is defined. Note that the terminal node **1** is repeated for clarity purpose only. The figure also demonstrates merging of implicit relations for transitions t_6 and t_5 which is possible due to equal effects of each transition, t_6 and t_5, on the low lying variables, namely p_1. This merging of implicit relations allow f_r to encode the effect of more than one event, where r is a relation node common to multiple transitions.

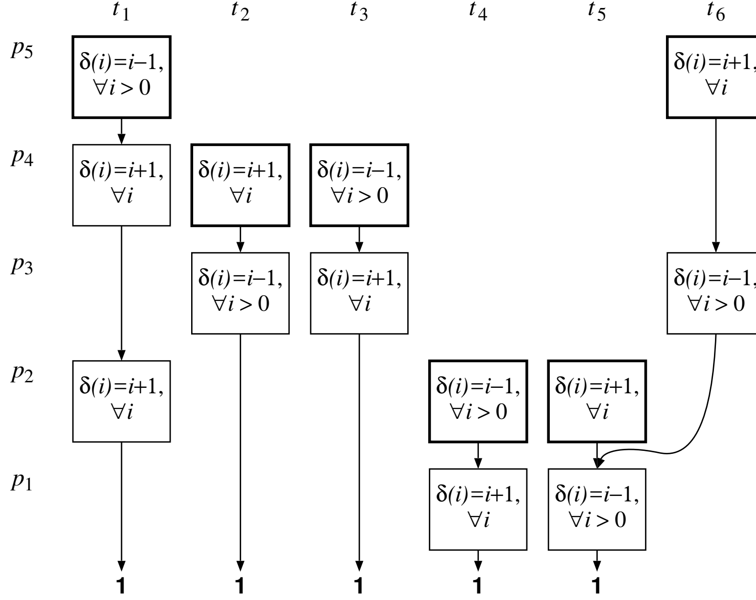

Fig. 4: The implicit relation forest for the fork-join Petri net model.

3.2 Saturation using relation nodes

To use saturation, we must partition the set of relation nodes $\mathcal{R}$ into $\mathcal{R}_L, \ldots, \mathcal{R}_1$ with $\mathcal{R}_k = \{r \in \mathcal{R} : r.var = u_k\}$. Note that any event whose relation belongs to $\mathcal{R}_k$ will not be disabled by, and will not modify if it occurs, any variable $u_l \succ u_k$. The idea of saturation [7] is that, every time a node for variable u_k is created, it is *saturated* by applying the relations in set $\mathcal{R}_k$ repeatedly until a fixed point is reached. The saturation algorithm, modified to use relation nodes, is shown in Figure 5 as procedure Saturate. It operates "in place" on a node n that has been created, but not yet added to the unique table, by repeatedly firing (c.f. line 10) the events in $\mathcal{A}[j]$ (c.f. the loop in line 8), a subset of events from $\mathcal{R}_k$ that produce same local index j on firing (c.f. the loop in line 5), over all possible values of j and adding those states to the current node (c.f. line 11). Procedure MultiRecFire, in line 10 is used to invoke RecFire for such subset of events and union the result (c.f. the lines 2, 3 of MultiRecFire) before saturating the node. It has been observed [5] that the order in which local states are explored (c.f. line 3 of Saturate) can significantly affect the efficiency of the iteration. The differences with respect to saturation using MxDs for relations are in lines 6 and 7, which obtain the local index j produced when an event fires on i using the relation node r, and the use of the common downward pointer $r.ptr$ in line 10.

Procedure RecFire, also shown in Figure 5, is used to "fire" a relation r on node n (this determines the relational product of the set encoded by MDD node n and the relation encoded by relation node r), except that any created nodes

are saturated immediately (c.f. line 15). Note that lines 8–9 handle the case where the relation graph skips a level (corresponding to an identity function), while lines 11–14 handle the case where the relation node and MDD node are at the same level. We do not give the case where the MDD graph skips a level, as this can only happen with edges that point directly to terminal node **0**. Again, the differences with respect to saturation using MxDs for relations are in lines 11, 12, and 13 which use the relation node r.

For fixed variable bounds, our modified saturation algorithm has the same complexity as the on-the fly saturation using extended decision diagrams. However, as bounds expand during saturation, relations stored using Kronecker representations, MxDs, or extended decision diagrams must all be updated to some extent for the increased bounds, with various costs for this reconstruction based on the type of storage. Even worse, some methods require discarding entries of the compute table, which can require significant duplication of computation. In contrast, no adjustments are needed to the implicit relation forest when bounds expand, and no compute table entries ever need to be discarded.

3.3 Implementation notes

We briefly discuss some ideas for efficient implementation of implicit relation forest nodes. For a node r, its partial function $r.\delta$ can be implemented using a function pointer, an abstract class with a virtual function, or with a parse tree or similar representation for expressions.

The elimination of duplicate nodes can be done similarly to MDDs, which utilize a *unique table*. A hash signature for $r.\delta$, with the property that equal functions should produce equal signatures, can be used to reduce the number of (potentially expensive) comparisons between functions. We stress that a stray duplicate node will not affect correctness, but only the efficiency, by potentially requiring duplication in computation.

To reduce the number of calls to $r.\delta$ (another potentially expensive operation), each node can maintain an array that memorizes $r.\delta$, so that $r.\delta(i)$ must be computed at most once for each i. Clearly this is a time/memory tradeoff, and note that the memory cost for this array is similar to and does not exceed other relation representations such as MxDs. However, simple functions, like update by a constant, can be handled directly without using a function pointer or a memorization array.

Finally, we note that the flexibility of defining $r.\delta$ as a (partial) *function* allows us to easily handle the case where MDD variable u_i is not necessarily equal to the number of tokens in place p_i. This is required if u_i corresponds to more than one Petri net place. In our implementation, the value of u_i is the index of a submarking, stored in a collection of submarkings, where the submarking is over the places corresponding to MDD variable u_i.

```
mdd Union(mdd m, mdd n)
 1 if n = 1 ∨ m = 1 then return 1;
 2 if n = 0 then return m;
 3 if m = 0 then return n;
 4 if ∃p s.t. (∪, m, n, p) ∈ C or
 5            (∪, n, m, p) ∈ C then
 6    return p;
 7 k ← max(m.var, n.var);
 8 p ← new MDD node for variable u_k;
 9 for each i ∈ D(u_k) do
10    md ← (u_k ≻ m.var) ? m : m[i];
11    nd ← (u_k ≻ n.var) ? n : n[i];
12    p[i] ← Union(md, nd);
13 p ← UniqueInsert(p);
14 C ← C ∪ {(∪, m, n, p)};
15 return p;
```

```
nodeset BuildImplicit(model M)
• Build set of relation nodes for model M
 1 R ← ∅;
 2 for each e ∈ E do
 3    Split Δ into Δ_{e,1}, ..., Δ_{e,L};
 4    V_e ← {i : Δ_{e,i} ≠ identity func.};
 5    r ← 1;
 6    for i ∈ V_e do
 7       p ← new relation node;
 8       p.var ← u_i;
 9       p.ptr ← r;
10       p.δ ← Δ_{i,e};
11       r ← UniqueInsert(p);
12    R ← R ∪ {r};
13 return R
```

```
MultiRecFire(mdd n, nodeset rSet)
• RecFire all events in rSet on node n
 1 for each r ∈ rSet do
 2       f_r ← RecFire(n, r);
 3       f ← Union(f, f_r);
 4 return f;
```

```
Saturate(level k, mdd n)
• Saturate node n in place using R_k.
 1 Q ← {i : n[i] ≠ 0};
 2 while Q ≠ ∅ do
 3    i ← SelectElement(Q);
 4    Q ← Q \ {i};
 5    for each r ∈ R_k do
 6       if r.δ(i) is defined then
 7          A[r.δ(i)] ← A[r.δ(i)] ∪ r.ptr
 8    for each a ∈ A do
 9       j ← a.index;
10       f ← MultiRecFire(n[i], a.rSet);
11       u ← Union(f, n[j])
12       if u ≠ n[j] then
13          n[j] ← u;
14          Q ← Q ∪ {j};
```

```
mdd RecFire(mdd n, relation r)
• Fire r on node n and then saturate it.
 1 if n = 0 then return 0;
 2 if r = 1 then return n;
 3 if ∃m s.t. (RecFire, n, r, m) ∈ C then
 4    return m;
 5 k ← max(n.var, r.var);
 6 m ← new MDD node for variable u_k;
 7 if n.var ≻ r.var then
 8    for each i ∈ D(u_k) do
 9       m[i] ← RecFire(n[i], r);
10 else
11    for each i s.t. r.δ(i) is defined do
12       j ← r.δ(i);
13       f ← RecFire(n[i], r.ptr);
14       m[j] ← Union(m[j], f);
15 Saturate(k, m);
16 m ← UniqueInsert(m);
17 C ← C ∪ {(RecFire, n, r, m)};
18 return m;
```

Fig. 5: MDD and relation node algorithms.

4 Related Work

This section discusses our approach of saturation with implicit relations in light of related work. We examine the alternative approaches of encoding transitions and compare them against the idea of this paper.

4.1 Kronecker representations

The saturation algorithm originally used a Kronecker representation to encode transitions [7]. Conceptually, such a scheme requires, for each model event e and each state variable v_k, a boolean matrix $\mathbf{N}_{e,k}$ used to encode function $\Delta_{e,k}$, where $\mathbf{N}_{e,k}[i_k, j_k]$ is one if and only if $\Delta_{e,k}(i_k) = j_k$. Note that $\mathbf{N}_{e,k}$ will be the identity matrix if event e does not change or depend on state variable v_k. In practice, identity matrices need not be stored explicitly, and other sophisticated schemes [6] could be used to store each $\mathbf{N}_{e,k}$. The dimension of $\mathbf{N}_{e,k}$ is the bound for state variable v_k, and it was originally assumed that this bound was known. On-the-fly saturation [8] eliminated this requirement, allowing the bounds of state variables to expand during saturation. As bounds expand, the matrices $\mathbf{N}_{e,k}$ also expand in size.

Implicit relation forests have two main advantages as compared to on-the-fly saturation using Kronecker representations. First, implicit relation forests allow for "sharing", i.e., a node can have more than one parent node, which occurs whenever two transitions have the same effect on the bottom-most k state variables. In some models, especially if transitions are split to maintain Kronecker consistency [6], this sharing can be significant, and the primary benefit is reduction of computation time, as this duplication in computation is avoided via the compute table. Second, the saturation algorithm is simpler with implicit nodes, as there is no longer a need to distinguish between "confirmed" and "unconfirmed" local states, nor is it necessary to expand matrices as local states are confirmed.

4.2 MDDs and extensible MDDs

A 2L-variable MDD, also called MxD, suffers from deletion of relevant yet incomplete compute-table entries when the *operand* MxDs remain unchanged but the resultant MxD undergoes modification in case of discovery of new bounds for a variable. Such deletions lead to reduced efficiency, to address which *extensible* MxDs were introduced in [19]. However, implicit relations provide a more efficient format, as demonstrated in Section 5, for encoding transitions to tackle the overhead cost of rebuilding *extensible* MxDs for the subclass of Petri nets.

4.3 Interval Mapping Diagrams

Strehl's work [17] on *interval mapping diagrams* (IMD) provides a generalized encoding of transitions wherein the *state distance* between pre- and post-transition state variable values are stored. The *state distance* is defined by an *action operator* and *action interval*, which together formulate the net-effect of the transition, on a *predicate interval* of the state variable, which refers to the enabling condition of the transition.

Implicit relation forests have the advantage of encoding any (partial) function as an effect of a transition on a variable, in contrast to IMDs, where the *action operator* is restricted to use increment, decrement, and equality operators only.

4.4 Homomorphisms

Couvreur et. al's work [11] offers an efficient way of encoding transitions using the concept of inductive homomorphisms. The encoding is defined to work with Data Decision Diagrams (DDD) [11] and Hierarchical Set Decision Diagrams (SDD)[12]. The approach offers freedom to the user in defining transitions and is more efficient compared to prior works [8, 10, 15, 19].

Implicit relations are an adaptation of inductive homomorphisms that work with MDD and are restrictive in terms of the nature of transitions that can be encoded. Only transitions with "firing" conditions defined as partial functions of the participant variables are compatible with implicit relations. A comparative study between tools implementing homomorphism on DDD and implicit relations on MDD is discussed in Section 5 to get a general overview of their performance on a set of benchmark models.

5 Experimental evaluation

Intra-Tool Comparative Performance Analysis

We implemented the modified saturation algorithm based on implicit relations (SatImp) in SMART [9] using Meddly[3, 2] as the underlying decision diagram library. We conducted experiments to compare the performance of SatImp with the existing "on-the-fly saturation with matrix diagrams" approach (OtfSat) for reachability set generation on a suite of 70 Petri net models that is available as *known-models* in MCC 2018 [1]. An experimental run involves execution of SatImp and OtfSat on a model instance with a timeout of one hour for each approach. All experiments are run on a server of Intel Xeon CPU 2.13GHz with 48G RAM under Linux Kernel 4.9.9.

Every Petri net model in the suite has multiple instances characterized by scaling parameters that affect the size of model ($|\mathcal{V}| + |\mathcal{E}|$) or the initial state of the model ($\mathbf{i}_0$). In order to demonstrate the effect of size and complexity of the models on the performance of SatImp and OtfSat, the set of benchmark models are classified into two categories namely, Type-1 models with scaling parameters affecting model size, and Type-2 models with scaling parameters affecting the initial state. Table 1 summarizes the experiments run on a subset (due to space constraint) of Type-1 and Type-2 models with key metrics of comparison as runtime, measured in seconds, and total number of pings and hits to the compute-table for saturation operation. Since, OtfSat uses MxD, the additional computation is summarized in the column for the total number of pings and hits to the compute-table for MxD operations. The pings and hits to the compute-table provide an overall picture of the number of decision diagram computations for each approach.

It is also important to note that the computation time spent in calculating the next-state of a variable is additional to the time spent for executing the saturation algorithm. SatImp generates the next-state of a variable using the

information stored in the implicit relations and OtfSat spends time modifying MxD when new bounds of the variable are discovered.

Observations from Table 1, for Type-1 models, confirm that the computation time for reachability set remains fairly equal in both implementations. For these models, since the scaling parameter does not affect the bound of the variables in the model, there is comparatively less time spent on modification of MxDs. Hence, the computation time spent by SatImp to calculate the next-state of every variable is close to the time spent by OtfSat in construction of matrix diagrams.

On the contrary, for Type-2 model, a significant improvement in performance of SatImp is observed. For these models, the maximum value of any variable discovered during the reachability set generation is a number greater than 1, determined by the scaling parameter(s), which entails frequent expansion of MxD nodes and possibly, for all variables at the most. Supported by this fact and experimental results in Table 1, a few observations can be noted. First, while SatImp is able to complete models with high scaling parameters quite early, OtfSat either takes long time, generally increased many-fold as compared to implicit relation, or does not finish the task before timeout. In such models, a significant amount of computational time is spent in modification of the matrix diagrams as shown by the number of pings and hits to compute-table for MxDs, which is otherwise absent in implicit relations.

Second, since MxDs expand and contract during manipulation, it may require compute-table entries to be discarded. Hence, the number of pings to the saturation compute-table would be relatively higher in OtfSat as compared to SatImp. The dashes in the table correspond to cases in which the runtime to construct $\mathcal{S}$ exceeded one hour. However, no claims can be made about the models that did not complete within the timeout.

For comparing the maximum memory usage of SatImp with that of OtfSat, we have chosen the largest completed instances of each model from the results in Table 1. While matrix diagrams consume memory on megabytes scale, implicit relations manage to store the exact information in much lesser space. The figures in Table 2 provide substantial proof of improvement in memory usage.

Our results include metrics that are typical for efficiency comparisons between two approaches, and illustrate the efficiency of using SatImp in terms of both computational and storage requirements. In practice, the use of implicit relations allow for reachability analysis of much larger systems as compared to that with MxDs.

Inter-Tool Comparative Performance Analysis

This section presents the performance comparison between the state-space generation algorithms of SMART and ITSTools, where the former tool uses implicit relations and MDD and the latter is based on homomorphisms, DDD and SDD [18]. The goal of this comparative analysis is to only gauge the efficiency of SatImp by using the well-established technique of homomorphism-based saturation as a benchmark.

A suite of 67 Petri net models from *known models* section of MCC 2018 is used in the experiments to compare the tools based on the runtime of state-space generation process. The largest instance of each model that could complete state-space generation with both SMART and ITSTools in MCC 2018, is chosen for this experiment. Table 3 shows only a subset of these experiments due to space constraint. Since the tools are based on decision diagrams and variable order is critical for efficiency of the state-space generation, identical static variable orders are used in both tools for each experimental run. SMART is tuned to run on implementation settings similar to that of MCC 2018 settings of ITS-Tools. For example, with reference to Section 3.3, the MDD variable u_i is adapted to be equal to the number of tokens in place p_i. However, the use of SDD in ITS-tools is omitted from the experiments to ensure fair comparison, because the construction of an SDD in ITS-tools requires auxiliary information about hierarchy of model variables and is not inferred directly from the model. All experiments are run in the same environment as described in the previous section.

In Table 3, it is observed that SMART is faster than ITSTools for 41 out of 67 models by an average of 3.125 times. These models include Kanban, Flexible Manufacturing System, House Construction and Philosophers where SMART is 235, 30, 10 and 16 times faster respectively. For 10 of the models, where Angiogenesis, SharedMemory etc are few of them, SMART is about 0.56 times faster than ITSTools. For the remaining 16 models, ITSTools is 4.69 times faster than SMART on an average. The experimental results allow us to surmise that the performance of SMART when using implicit relations on MDD-based storage complements the performance of ITSTools that use inductive homorphisms with DDD-based storage for saturation.

6 Conclusions and future work

Reachability set generation using on-the-fly saturation with MxDs is quite an improvement over explicit techniques, as it is often able to handle extremely large sets with less time. However, the computational cost for building such transition relations repeatedly during reachability set generation, creates additional focus towards handling the relations along with the state space. The transition relations undergo manipulations for the construction of next-state functions necessary for state space generation, while the underlying functions to generate the next state is available in the model itself. Implicit relations manage to encapsulate and exploit these properties. Hence, we adapted the saturation algorithm to work with implicit relations and showed how additional computations can be saved during the reachability set generation.

Saturation algorithm using implicit relations for the reachability set generation provides promising results for the defined class of models. Experimental results indicate that the costs are improved for a large set of models across different sizes, though the approach is not adapted to handle marking dependent events as discussed in Section 2.1. The improvement is mainly due to the

essence of implicit relations to encase the properties of the system in a simple straight-forward approach.

Future work should investigate modification of the implicit relations to represent Petri nets with marking-dependent arcs by disclosing value of each variable to the underlying variables in the implicit relation. We intend to create a merger between implicit relations and MxDs that will exploit their respective static and dynamic ingredients in the fusion. Allowing implicit nodes inside an MxD forest would allow us to handle models that are not "Kronecker consistent", but still get benefits for events that affect the participant variables independent of each other.

Acknowledgment

This work was supported in part by the National Science Foundation under grant ACI-1642397.

References

1. MCC : Model Checking Competition @ Petri Nets. https://mcc.lip6.fr.
2. MEDDLY webpage. https://sourceforge.net/projects/meddly/.
3. J. Babar and A. S. Miner. Meddly: Multi-terminal and Edge-valued Decision Diagram LibrarY. In *Proc. QEST*, pages 195–196. IEEE Computer Society, 2010.
4. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comp. Surv.*, 24(3):293–318, 1992.
5. M.-Y. Chung, G. Ciardo, and A. J. Yu. A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis. In *Proc. ATVA*, LNCS 4218, pages 51–66. Springer, 2006.
6. G. Ciardo, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31:63–100, 2007.
7. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342. Springer, 2001.
8. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. TACAS*, LNCS 2619, pages 379–393. Springer, 2003.
9. G. Ciardo and A. S. Miner. SMART: Simulation and Markovian Analyzer for Reliability and Timing. In *Proc. IEEE Int. Computer Performance and Dependability Symp. (IPDS'96)*, page 60. IEEE Comp. Soc. Press, 1996.
10. G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Proc. CHARME*, LNCS 3725, pages 146–161. Springer, 2005.
11. J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier. Data decision diagrams for petri net analysis. In *Proc. ATPN*, pages 101–120. Springer, 2002.
12. J.-M. Couvreur and Y. Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In *Proc. Formal Description Techniques, FORTE95*, volume 3731 of *LNCS*, pages 443–4572, 2005.

13. T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
14. A. S. Miner. Implicit GSPN reachability set generation using decision diagrams. *Performance Evaluation*, 56(1):145 – 165, 2004. Dependable Systems and Networks - Performance and Dependability Symposium (DSN-PDS) 2002: Selected Papers.
15. A. S. Miner. Saturation for a general class of models. In *Proc. QEST*, pages 282–291, Sept. 2004.
16. T. Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, Apr. 1989.
17. K. Strehl and L. Thiele. Interval diagram techniques for symbolic model checking of Petri nets. In *Proc. Design, Automation and Test in Europe (DATE'99)*, pages 756–757, Mar. 1999.
18. Y. Thierry-Mieg. Symbolic Model-Checking Using ITS-Tools. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 231–237, London, United Kingdom, Apr. 2015. Springer Berlin Heidelberg.
19. M. Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In *Proc. SOFSEM*, LNCS 5404, pages 582–594. Springer, 2009.
20. T. Yoneda, H. Hatori, A. Takahara, and S.-I. Minato. BDDs vs. zero-suppressed BDDs: for CTL symbolic model checking of Petri nets. In *Proc. FMCAD*, LNCS 1166, pages 435–449, 1996.

Table 1: Computational requirements of saturation for reachability set generation in Type-1 models and Type-2 models.

Model	$\lvert\mathcal{S}\rvert$	OtfSat			SatImp			Additional MxD CT in OtfSat	
		Time (sec)	Pings $\times10^5$	Hits $\times10^5$	Time (sec)	Pings $\times10^5$	Hits $\times10^5$	Pings $\times10^3$	Hits $\times10^3$
Type-1:									
BridgeAndVehicles V50P50N10	3.48×10^8	87.05	444	168	51.53	408	150	7790	7680
BridgeAndVehicles V80P20N10	2.43×10^9	473.05	2051	904	239.93	1906	827	50010	49791
BridgeAndVehicles V80P50N10	2.43×10^9	475.24	2051	904	253.12	1979	855	50010	49791
DES 30a	1.92×10^{13}	22.60	175	34	23.91	176	34	24	16
DES 30b	1.97×10^{22}	103.97	719	201	102.74	745	203	18	11
DES 40a	3.52×10^{13}	49.27	341	62	49.54	344	63	28	18
DES 40b	3.60×10^{22}	149.64	995	256	152.69	1041	258	20	12
DNAwalker 14ringLRLarge	1.86×10^9	11.38	90	66	12.23	90	66	9	6
DNAwalker 15ringRRLarge	1.86×10^9	10.11	78	58	12.29	78	58	9	6
RWmutex r10w100	1.12×10^3	2.29	26	0	2.48	25	0	102	69
RWmutex r10w500	1.52×10^3	38.01	432	30	38.75	431	30	590	422
Type 2 :									
Angiogenesis 15	1.12×10^{15}	324.79	4873	4340	140.33	2973	2671	320	310
CircadianClock 1000	4.02×10^{15}	3334.53	6834	6753	78.02	186	136	88016	88014
FMS 100	2.70×10^{21}	8.90	340	332	4.12	220	214	341	50
FMS 200	1.95×10^{25}	78.28	3331	3294	33.50	1737	1714	50	50
GPPP C1000N10	1.42×10^{10}	1.19	5	3	0.21	3	2	43	43
GPPP C1000N100	1.14×10^{15}	440.35	1560	1413	114.88	654	543	18072	18071
Kanban 500	$7.09\times10^{2}6$	458.66	2558	2542	12.14	756	752	12704	12703
Kanban 1000	1.42×10^{30}	2347.18	20231	20170	72.50	5909	5891	50677	50436
Robot Manipulation 20	4.11×10^9	6.91	172	153	1.22	34	31	18	17
Robot Manipulation 50	8.53×10^{12}	176.05	3941	3659	33.15	871	833	253	253
SmallOS MT1024DC256	3.27×10^{12}	971.77	7506	7475	66.13	3198	3184	24522	24521
SmallOS MT2048DC0512	1.04×10^{14}	–	–	–	620.61	25184	25118	–	–
SmallOS MT2048DC1024	2.46×10^{14}	–	–	–	1105.18	38082	37945	–	–
SwimmingPool 9	1.81×10^{10}	11.73	116	97	7.28	116	97	70	70
SwimmingPool 10	3.36×10^{10}	15.81	163	137	10.98	161	135	95	95

Table 2: Storage requirements by saturation algorithm for transition encodings.

Model	Memory for OtfSat (KB)	Memory for SatImp (KB)
DES 40b	524.00	7.55
DNAwalker 15ringRRLarge	168.93	5.84
Angiogenesis 15	1133.90	9.28
CircadianClock 1000	959318.00	12.00
FMS 200	10175.00	26.59
GPPP 100 100	1470591.00	48.56
Kanban 1000	464342.00	16.00
Robot Manipulation 50	18020.00	14.12
Small OS 1024 256	242091.00	106.50
Swimming Pool 10	1823.60	10.88

Table 3: Performance comparison between SMART and ITS-tools.

Model	Instance	Reachable States (#)	Runtime (sec)	
			ITS Tools	SMART
Kanban	100	1.7263E+19	3.37E+03	1.42E+01
FMS	200	1.9536E+25	4.64E+02	1.42E+01
SwimmingPool	6	1.6974E+09	5.26E+01	2.89E+00
Philosophers	500	3.6300E+238	4.03E+00	2.24E-01
HouseConstruction	10	1.6636E+09	6.11E+00	5.99E-01
ClientsAndServers	5	1.2551E+11	7.11E+01	8.55E+00
CircadianClock	100	4.2040E+10	1.74E+00	4.60E-01
IBMB2S565S3960	none	1.5511E+16	1.60E+01	8.65E+00
Ring	none	9.0265E+11	1.72E-01	9.34E-02
TokenRing	15	3.5358E+07	1.57E+01	1.26E+01
Referendum	100	5.1537E+47	1.07E+00	8.96E-01
SharedMemory	20	4.4515E+11	5.04E+00	7.76E+00
EnergyBus	none	2.1318E+12	5.34E+01	9.00E+01
Angiogenesis	5	4.2735E+07	5.20E-01	9.58E-01
FlexibleBarrier	4a	2.0737E+04	6.11E-02	1.51E-01
Railroad	10	2.0382E+06	2.34E+00	5.94E+00
Peterson	3	3.4079E+06	2.70E+01	7.45E+01
CSRepetitions	3	1.3407E+08	6.01E-01	2.47E+00
UtahNoC	none	4.7599E+09	5.29E+00	3.44E+01
PaceMaker	none	3.6803E+17	2.47E-01	3.07E+00