Symbolic Reachability using LDDs

Wieger Wesselink and Maurice Laveaux

April 26, 2024

1 Introduction

In this document we describe a symbolic reachability algorithm that uses list decision diagrams (LDDs) to store states and transitions. It is based on the work in [**?**] and [**?**].

1.1 Definitions

Let S be a set of states, and $R \subseteq S \times S$ a transition relation. There is a transition from x to y if $(x, y) \in R$. We assume that the set of states S is a Cartesian product

$$S = S_1 \times \ldots \times S_m$$

In other words, states are vectors of m elements.

Definition 1. *The domain of a relation R is defined as*

$$\mathit{domain}(R) = \{x \in S \mid \exists y \in S : (x, y) \in R\}$$

Definition 2. *The function next returns the successors of an element $x \in S$:*

$$\mathit{next}(R, x) = \{y \in S \mid (x, y) \in R\}$$

It can be lifted to a subset $X \subseteq S$ using

$$\mathit{next}(R, X) = \cup\{\mathit{next}(R, x) \mid x \in X\}$$

Definition 3. *The set of reachable states that can be reached from an initial state $x \in S$ is defined as*

$$\mathit{reachable\_states}(R, x) = \{y \in S \mid \exists n \geq 0 : (x, y) \in R^n\}$$

2 Read, write and copy parameters

In [**?**] three types of dependencies for the parameters of a relation are distinguished: *read dependence* (whether the value of a parameter influences transitions), *must-write dependence* (whether a parameter is written to), and *may-write dependence* (whether a parameter may be written to, depending on the value of some other parameter). The may-write versus must-write distinction is introduced for arrays, that do not exist in mCRL2. So for our use cases may- and must-write dependence coincide, and we will refer to it as *write dependence* instead.

Before we formalize the notions *read independent* and *write independent*, it is important to understand the application that they are used for. We assume that we have a sparse relation R that is defined as a set of

pairs (x, y) with $x, y \in S$. Our goal is to define the notions read and write independent such that the values of read and write independent parameters are not needed for the successor computation of R. So for the values x_i, y_i corresponding to a read independent parameter d_i we do not need to store value x_i, and for the values corresponding to a write independent parameter we do not need to store the value y_i.

In order to satisfy these requirements we define a read independent parameter as a parameter whose value is always copied ($y_i = x_i$), or mapped to a constant value ($y_i = c$). However, this is not enough for being able to discard the value of such a parameter. In addition, the corresponding transition has to be enabled for any value in its domain. We define a write independent parameter as a parameter whose value is always copied ($y_i = x_i$).

Definition 4. *For a vector $x = [x_1, \ldots, x_m] \in S$ we define the following notation for updating the element at position i with value $y \in S_i$:*

$$x[i = y] = [x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_m]$$

We lift this definition to a set as follows. Let $I = \{i_1, \ldots i_k\}$ with $1 \leq i_1 < \ldots < i_k \leq m$ be a set of parameter indices, and $y \in S_{i_1} \times \ldots \times S_{i_k}$. Then

$$x[I = y] = z \text{ with } z_r = \begin{cases} x_r & \text{if } r \notin I \\ y_r & \text{otherwise} \end{cases}$$

Definition 5.

$$\mathit{always\_copy}(R, i) = (\forall (x, y) \in R : x_i = y_i) \wedge (\forall (x, y) \in R, d \in S_i : (x[i = d], y[i = d]) \in R)$$

Definition 6.

$$\mathit{always\_constant}(R, i) = (\exists d \in S_i : \forall (x, y) \in R : y_i = d) \wedge (\forall (x, y) \in R, d \in S_i : (x[i = d], y) \in R)$$

Definition 7.

$$\mathit{read\_independent}(R, i) = \mathit{always\_copy}(R, i) \vee \mathit{always\_constant}(R, i)$$

Definition 8.

$$\mathit{write\_independent}(R, i) = \forall (x, y) \in R : x_i = y_i$$

Note that our definitions differ slightly from the ones in [**?**], but they are equivalent. To illustrate these definitions, consider the following example.

Example 9. *Let $S = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ and let R be a relation on the variables x_1, x_2 and x_3 defined by the statement*

$$\textit{if } x_3 > 4 \textit{ then begin } x_2 := 3 \textit{ end}$$

In this case x_1 is both read and write independent, x_2 is read independent, and x_3 is write independent. Even though the value of x_3 is always copied, we do not consider it read independent. This is because for values $x_3 \leq 4$ the transition is not enabled, and this information would be lost if we discard those values from the transition relation. Now instead of storing the transition $((1, 2, 5), (1, 3, 5))$, we only need to store $((5), (3))$ to be able to derive that a transition from $(1, 2, 5)$ to $(1, 3, 5)$ is possible.

Read and write parameters are defined using

Definition 10.

$$\mathit{read}(R, i) = \neg \mathit{read\_independent}(R, i)$$

Definition 11.

$$\mathit{write}(R, i) = \neg \mathit{write\_independent}(R, i)$$

Definition 12.

$$\mathit{read\_parameters}(R) = \{i \mid \mathit{read}(R, i)\}$$

Definition 13.

$$\mathit{write\_parameters}(R) = \{i \mid \mathit{write}(R, i)\}$$

Now let us consider an mCRL2 summand P of the following shape:

$$P(d) = c(d) \to a(f(d)).P(g(d))$$

For such a summand [**?**] defines the following approximations of read and write parameters:

$$\mathsf{read}_s(P, i) = \begin{array}{ll} d_i \in \mathsf{freevars}(c(d)) & \vee \\ (d_i \in \mathsf{freevars}(g_i(d)) \wedge d_i \neq g_i(d)) & \vee \\ \exists 1 \leq k \leq m : (d_i \in \mathsf{freevars}(g_k(d)) \wedge i \neq k) & \end{array}$$

$$\mathsf{write}_s(P, i) = (d_i \neq g_i)$$

Indeed we have $\mathsf{read}_s(P, i) \Rightarrow \mathsf{read}(P, i)$ and $\mathsf{write}_s(P, i) \Rightarrow \mathsf{write}(P, i)$.

3 List Decision Diagrams

A List Decision Diagram (LDD) is a DAG. It has two types of leaf nodes, false and true, or 0 and 1. The third type of node has a label a and two successors *down* and *right*, or $=$ and $>$. An LDD represents a set of lists, as follows:

$$\begin{array}{lcl} [\![\mathsf{false}]\!] & = & \emptyset \\ [\![\mathsf{true}]\!] & = & \{[\,]\} \\ [\![\mathsf{node}(v, down, right)]\!] & = & \{vx \mid x \in [\![down]\!]\} \cup [\![right]\!] \end{array}$$

In [**?**] an LDD is defined as

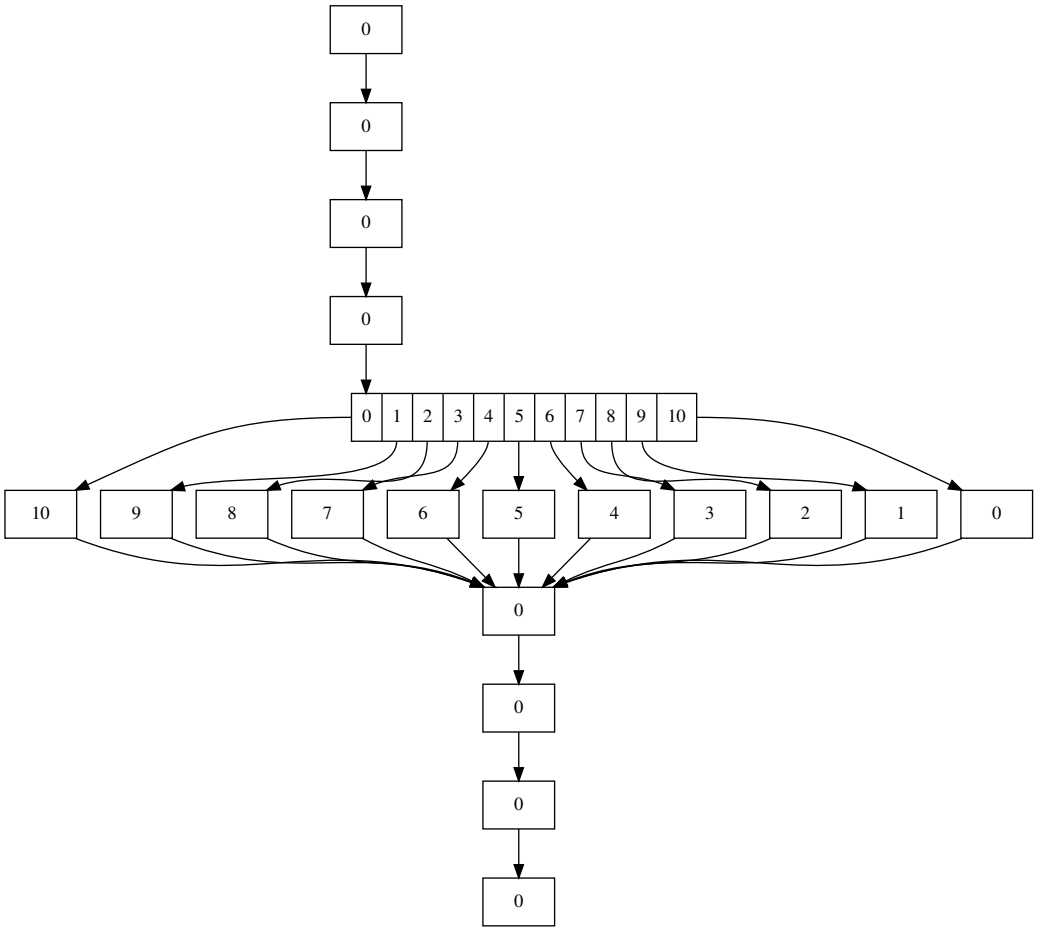
Definition 14. *A List decision diagram (LDD) is a directed acyclic graph with the following properties:*

1. *There is a single root node and two terminal nodes 0 and 1.*
2. *Each non-terminal node p is labeled with a value v, denoted by $val(p) = v$, and has two outgoing edges labeled $=$ and $>$ that point to nodes denoted by $p[x_i = v]$ and $p[x_i > v]$.*
3. *For all non-terminal nodes p, $p[x_i = v] \neq 0$ and $p[x_i > v] \neq 1$.*
4. *For all non-terminal nodes p, $val(p[x_i > v]) > v$.*
5. *There are no duplicate nodes.*

LDDs are well suited to store lists that differ in only a few positions. Consider the transition relation R on $S = \mathbb{N}^{10}$ with initial state $x = [0, 0, 0, 0, 10, 0, 0, 0, 0, 0]$, that is defined by

$$\mathsf{if}\ x_5 > 0\ \mathsf{then\ begin}\ x_5 := x_5 - 1; x_6 := x_6 + 1\ \mathsf{end}$$

Clearly this is a sparse relation with $\mathsf{used}(R) = \{5, 6\}$. The state space consists of 11 states that differ only in the 5th and 6th parameter. It can be compactly represented using an LDD, see the figure below. For our applications we use the LDD implementation that is part of the Sylvan multi-core framework for decision diagrams, see [**?**].


0
0
0
0
0 1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1 0
0
0
0
0

4 Reachability

4.1 Computing the set of reachable states

A straightforward algorithm to compute the set of reachable states from an initial state $x \in S$ is the following.

Algorithm 1 Reachability

```
1: function ReachableStates(R, x)
2:     visited := {x}
3:     todo := {x}
4:     while todo ≠ ∅ do
5:         todo := next(R, todo) \ visited
6:         visited := visited ∪ todo
7:     return visited
```

There are two bottlenecks in this algorithm. First of all the set *visited* may be large and therefore consume a lot of memory. Second, the computation of next(R, *todo*) may become expensive once *todo* becomes large.

4.2 Reachability with learning

To reduce the memory usage, we store the sets *visited* and *todo* using LDDs. It is not always feasible to store the entire transition relation R using LDDs, because it may be huge or even have infinite size. To deal with this, we only store the subset L of R that is necessary for the reachability computation in an LDD. The relation L is computed (learned) on the fly.

Algorithm 2 Reachability with learning

```
1: function ReachableStates(R, x)
2:     visited := {x}
3:     todo := {x}
4:     L := ∅                                          ▷ L is the learned relation
5:     while todo ≠ ∅ do
6:         L := L ∪ {(x, y) ∈ R | x ∈ todo}             ▷ This operation is expensive
7:         todo := next(L, todo) \ visited              ▷ This operation is cheap
8:         visited := visited ∪ todo
9:     return visited
```

4.3 Reachability of a sparse relation

Suppose that we have a sparse relation R, i.e. the number of read and write parameters is small. In that case we can use projections to increase the efficiency.

Definition 15. *The projection of a state $x \in S$ with respect to a set of parameter indices $\{i_1, \ldots i_k\}$ with $1 \leq i_1 < \ldots < i_k \leq m$ is defined as*

$$\mathit{project}(x, \{i_1, \ldots, i_k\}) = (x_{i_1}, \ldots, x_{i_k})$$

We lift this to a relation R with read parameter indices I_r and write parameter indices I_w as follows:

$$\mathit{project}(R, I_r, I_w) = \{(\mathit{project}(x, I_r), \mathit{project}(y, I_w)) \mid (x, y) \in R\}$$

The application of a projected relation to an unprojected state is defined using the function relprod. The Sylvan function relprod implements this, or something similar.

Definition 16. *Let R be a relation with read parameter indices I_r and write parameter indices I_w, let $x \in S$ and let $\hat{R} = \mathit{project}(R, I_r, I_w)$. Then we define*

$$\mathit{relprod}(\hat{R}, x, I_r, I_w) = \{x[I_w = \hat{y}] \mid \mathit{project}(x, I_r) = \hat{x} \wedge (\hat{x}, \hat{y}) \in \hat{R}\}$$

$$\mathit{relprev}(\hat{R}, y, I_r, I_w, X) = \{x \in X \mid y \in \mathit{relprod}(\hat{R}, x, I_r, I_w)\}$$

Algorithm 3 Reachability of a sparse relation using projections

```
1:  function ReachableStates(R, x)
2:      visited := {x}
3:      todo := {x}
4:      L := ∅                                          ▷ L is a projected relation
5:      I_r, I_w := read_parameters(R), write_parameters(R)
6:      while todo ≠ ∅ do
7:          L := L ∪ project({(x, y) ∈ R | x ∈ todo}, I_r, I_w)
8:          todo := relprod(L, todo, I_r, I_w) \ visited
9:          visited := visited ∪ todo
10:     return visited
```

In this new version of the algorithm, the computation of the successors in line 7 is still the bottleneck:

$$L := L \cup \mathsf{project}(\{(x, y) \in R \mid x \in \mathit{todo}\}, I_r, I_w)$$

An important observation is that the same result can be obtained by applying the projected relation to the projected arguments:

$$L := L \cup \{(x, y) \in \mathsf{project}(R, I_r, I_w) \mid x \in \mathsf{project}(\mathit{todo}, I_r)\}$$

For our applications, the set $\mathsf{project}(\mathit{todo}, I_r \cup I_w)\}$ is typically much smaller than *todo*, which means that a lot of duplicate successor computations in line 7 are avoided.

4.4 Reachability of a union of sparse relations

We now consider a relation R that is the union of a number of sparse relations. Examples of these sparse relations are the summands of an LPS or of a PBES in SRF format.

$$R = \bigcup_{i=1}^{n} R_i$$

Algorithm 4 Reachability of a union of sparse relations

1: **function** ReachableStates($\{R_1, \ldots, R_n\}$, x)
2: $\quad$ *visited* $:= \{x\}$
3: $\quad$ *todo* $:= \{x\}$
4: $\quad$ **for** $1 \leq i \leq n$ **do**
5: $\quad\quad$ $L_i := \emptyset$
6: $\quad\quad$ $I_{r,i}, I_{w,i} :=$ read_parameters(R_i), write_parameters(R_i)
7: $\quad$ **while** *todo* $\neq \emptyset$ **do**
8: $\quad\quad$ **for** $1 \leq i \leq n$ **do**
9: $\quad\quad\quad$ $L_i := L_i \cup \{(x, y) \in \mathsf{project}(R_i, I_{r,i}, I_{w,i}) \mid x \in \mathsf{project}(todo, I_{r,i}) \setminus \mathsf{domain}(L_i)\}$
10: $\quad\quad$ $todo := \left(\bigcup_{i=1}^{n} \mathsf{relprod}(L_i, todo, I_r, I_w) \right) \setminus visited$
11: $\quad\quad$ $visited := visited \cup todo$
12: $\quad$ **return** *visited*

Note that in line 9 another optimization has been applied, by excluding elements in the projected todo list that are already in the domain of L_i. For all values in $\mathsf{domain}(L_i)$ the outgoing transitions have already been determined. We can also replace $\mathsf{domain}(L_i)$ by a set X that keeps track of all values of x that have already been processed. Let X be initially the empty set and updated to $X \leftarrow X \cup \{x \in \mathsf{project}(todo, I_{r,i})\}$ at line 9. The set X contains all elements in $\mathsf{domain}(L_i)$ but also all x with no outgoing transitions. In practice, the transitions of R_i are computed on-the-fly and this additional caching can avoid these computations with the downside that it requires more memory.

4.5 Reachability with chaining

For symbolic reachability it can be useful to reduce the amount of breadth-first search iterations because this also reduces the amount of symbolic operations that have to be applied. Updating the todo set after applying each sparse relation can potentially increase the amount of states that are visited and thus reduce the amount of breadth-first search iterations necessary. Note that we only add the states for which all sparse relations have been applied to visited.

Algorithm 5 Reachability of a union of sparse relations

1: **function** ReachableStates($\{R_1, \ldots, R_n\}$, x)
2: $\quad$ $visited := \{x\}$
3: $\quad$ $todo := \{x\}$
4: $\quad$ **for** $1 \leq i \leq n$ **do**
5: $\quad\quad$ $L_i := \emptyset$
6: $\quad\quad$ $I_{r,i}, I_{w,i} := \mathsf{read\_parameters}(R_i), \mathsf{write\_parameters}(R_i)$
7: $\quad$ **while** $todo \neq \emptyset$ **do**
8: $\quad\quad$ $todo1 := todo$
9: $\quad\quad$ **for** $1 \leq i \leq n$ **do**
10: $\quad\quad\quad$ $L_i := L_i \cup \{(x, y) \in \mathsf{project}(R_i, I_{r,i}, I_{w,i}) \mid x \in \mathsf{project}(todo1, I_{r,i}) \setminus \mathsf{domain}(L_i)\}$
11: $\quad\quad\quad$ $todo1 := todo1 \cup \mathsf{relprod}(L_i, todo1, I_r, I_w)$
12: $\quad\quad$ $visited := visited \cup todo$
13: $\quad\quad$ $todo := todo1 \setminus visited$
14: $\quad$ **return** $visited$

4.6 Reachability with deadlock detection

To detect *deadlocks*, *i.e.*, states with no outgoing transitions, during reachability we have to determine which states in the todo sets have no outgoing transitions after applying all the transitions groups. This can be achieved as follows by considering the predecessors.

Algorithm 6 Reachability of a union of sparse relations

1: **function** ReachableStates($\{R_1, \ldots, R_n\}$, x)
2: $\quad$ $visited := \{x\}$
3: $\quad$ $todo := \{x\}$
4: $\quad$ $deadlocks := \emptyset$
5: $\quad$ **for** $1 \leq i \leq n$ **do**
6: $\quad\quad$ $L_i := \emptyset$
7: $\quad\quad$ $I_{r,i}, I_{w,i} := \mathsf{read\_parameters}(R_i), \mathsf{write\_parameters}(R_i)$
8: $\quad$ **while** $todo \neq \emptyset$ **do**
9: $\quad\quad$ $potential\_deadlocks := todo$
10: $\quad\quad$ **for** $1 \leq i \leq n$ **do**
11: $\quad\quad\quad$ $L_i := L_i \cup \{(x, y) \in \mathsf{project}(R_i, I_{r,i}, I_{w,i}) \mid x \in \mathsf{project}(todo, I_{r,i}) \setminus \mathsf{domain}(L_i)\}$
12: $\quad\quad$ $todo := \left(\bigcup_{i=1}^{n} \mathsf{relprod}(L_i, todo, I_r, I_w)\right) \setminus visited$
13: $\quad\quad$ $potential\_deadlocks := potential\_deadlocks \setminus \left(\bigcup_{i=1}^{n} \mathsf{relprev}(L_i, todo, I_r, I_w, potential\_deadlocks)\right)$
14: $\quad\quad$ $visited := visited \cup todo$
15: $\quad\quad$ $deadlocks := deadlocks \cup potential\_deadlocks$
16: $\quad$ **return** $visited$

For the chaining strategy we remove predecessors from the potential deadlocks at the end of every transition group iteration (on line 11 of Algorithm 5) using *todo1* instead of *todo*.

4.7 Joining relations

If two or more of the relations R_i have approximately the same set of read and write parameters, it can be beneficial to join them into one relation. In [**?**] this is called combining transition groups. In order to determine how well two relations match, we define a bit pattern for a relation that contains the read and write information of the parameters.

Definition 17. *The read write pattern of a relation R is defined as*

$$\mathit{read\_write\_pattern}(R) = [r_1, w_1, \ldots, r_m, w_m]$$

with

$$r_i = \mathit{read}(R, i) \text{ and } w_i = \mathit{write}(R, i) \qquad (1 \leq i \leq m)$$

For two read write patterns p and q, we define $p \vee q$ as the bitwise or of both patterns. In other words, if $r = p \vee q$, then $r_i = p_i \vee q_i$ $(1 \leq i \leq 2m)$. Similarly we say that $p \leq q$ iff $p_i \leq q_i$ for $1 \leq i \leq 2m$.

Example 18. *Let $S = \mathbb{N} \times \mathbb{N}$ and let T and U be relations on the variables x, y. Let T be defined by $(x, y) \to (x + 1, x)$ and let U be defined by $x, y := x + 2, y$. In this case x is a read independent parameter in both T and U, but according to the definition x is not a read independent parameter of $T \cup U$. Hence $\mathit{read\_write\_pattern}(T) = 1101$, $\mathit{read\_write\_pattern}(U) = 1100$, and $\mathit{read\_write\_pattern}(T \cup U) = 1111$.*

4.7.1 Row subsumption

In [**?**] a notion called *row subsumption* is introduced for joining relations. This notion is based on an extension to the theory, which ensures that the following property holds for two relations T and U:

$$\mathsf{read\_write\_pattern}(T \cup U) = \mathsf{read\_write\_pattern}(T) \vee \mathsf{read\_write\_pattern}(U) \tag{1}$$

It works as follows. Suppose we have a transition $(x, y) \in T$, and let L be the projected transition relation corresponding to $T \cup U$. Then we insert the special value ▲ in L for all entries of y that correspond to a copy parameter of T (i.e. with read write values 00). The meaning of this special value is that the corresponding parameter will not be overwritten by L. The

This is achieved by using $\mathsf{relprod}^{\blacktriangle}$ instead of relprod, which is defined as follows:

$$x[I = y]^{\blacktriangle} = z \text{ with } z_r = \begin{cases} x_r & \text{if } r \notin I \text{ or } y_r = \blacktriangle \\ y_r & \text{otherwise} \end{cases}$$

$$\mathsf{relprod}^{\blacktriangle}(\hat{R}, x, I_r, I_w, X) = \{x[I_w = \hat{y}]^{\blacktriangle} \mid \mathsf{project}(x, I_r) = \hat{x} \wedge (\hat{x}, \hat{y}) \in \hat{R}\}$$

N.B. In the Sylvan relprod function this functionality is implemented in a slightly different way, using a concept called 'copy nodes'. It requires that the matrix L is assembled using the function union_cube_copy instead of union_cube.

4.8 An algorithm for partitioning a union of relations

In this section we sketch an approach for making a partition of a union of relations $R = \bigcup_{i=1}^{n} R_i$ that is based on property 1. The goal is to create groups of relations R_i that have approximately the same read write patterns. The main idea is that each group of the partition can be characterized by a read write pattern.

If we choose K read write patterns $p_1, \ldots, p_K$ such that

$$\bigvee_{k=1}^{K} p_k = \mathsf{read\_write\_pattern}(R)$$

and

$$\forall 1 \leq i \leq n : \exists 1 \leq k \leq K : \mathsf{read\_write\_pattern}(R_i) \leq p_k$$

then we can define K groups by assigning relation R_i to group k if $\mathsf{read\_write\_pattern}(R_i) \leq p_k$. Note that a relation R_i can match multiple read write patterns p_k, but in that case an arbitrary pattern may be chosen.

A heuristic for selecting suitable patterns p_k is to choose them such that both

$$\sum_{1 \leq k \leq K} \mathsf{bitcount}(p_k)$$

and

$$\sum_{1 \leq k < l \leq K} \mathsf{bitcount}(p_k \wedge p_l)$$

are small. For example an SMT solver might be used to determine them.

5 Application: LPS Reachability

Consider the following untimed linear process specification P, with initial state d_0.

$$P(d : D) = \sum_{i \in I} \sum_{e_i} c_i(d, e_i) \to a_i(f_i(d, e_i)) \cdot P(g_i(d, e_i))$$

Each summand $i \in I$ defines a transition relation R_i characterized by

$$\begin{cases} e_i & \text{a sequence of summation variables} \\ c_i(d, e_i) & \text{a condition} \\ a_i(f_i(d, e_i)) & \text{a transition label} \\ g_i(d, e_i) & \text{the successor states} \end{cases}$$

The set of states is D, which is naturally defined as a Cartesian product

$$D = D_1 \times \ldots \times D_m$$

In order to apply the reachability algorithm, we need the following ingredients:

$\{(x, y) \in R_i \mid x \in todo\}$	enumerate solutions of condition
conversion between states and LDDs	map values to integers
$\mathsf{read\_parameters}(R_i)$	syntactic analysis
$\mathsf{write\_parameters}(R_i)$	syntactic analysis
$\mathsf{project}(R_i, I_{r,i}, I_{w,i})$	discard unused parameters
$\mathsf{project}(todo, I_{r,i} \cup I_{w,i})$	project (Sylvan function)
$\mathsf{relprod}(L_i, todo, I_r, I_w)$	relprod (Sylvan function)

For a summand of the shape $R_i = c_i \to a_i(f_i) \cdot P(g_i)$ we will now define the read and write dependencies.

$$\mathsf{read}(R_i, j) = d_j \in \mathsf{freevars}(c_i) \vee \exists 1 \leq k \leq m : (d_j \in \mathsf{freevars}(g_{i,k}) \wedge (d_k \neq d_j \vee g_{i,j} \neq d_j))$$

$$\mathsf{write}(R_i, j) = (d_j \neq g_{i,j})$$

6 Application: PBES Reachability

Definition 19. *A parameterised Boolean equation system (PBES) is a sequence of equations as defined by the following grammar:*

$$\mathcal{E} ::= \emptyset \mid (\mu X(d : D) = \varphi)\mathcal{E} \mid (\mu X(d : D) = \varphi)\mathcal{E}$$

where $\emptyset$ is the empty PBES, μ and ν denote the least and greatest fixpoint operator, respectively, and $X \in \chi$ is a predicate variable of sort $D \to B$. The right-hand side φ is a syntactically monotone predicate formula. Lastly, $d \in V$ is a parameter of sort D.

Definition 20. *Let $\mathcal{E}$ be a PBES. Then $\mathcal{E}$ is in standard recursive form (SRF) iff for all $\sigma_i X_i(d : D) = \varphi) \in \mathcal{E}$, where φ is either disjunctive or conjunctive, i.e., the equation for X_i has the shape*

$$\sigma_i X_i(d : D) = \bigvee_{j \in J_i} \exists e_j : E_j . f_{ij}(d, e_j) \wedge X_{g_{ij}}(h_{ij}(d, e_j))$$

or

$$\sigma_i X_i(d : D) = \bigwedge_{j \in J_i} \forall e_j : E_j . f_{ij}(d, e_j) \implies X_{g_{ij}}(h_{ij}(d, e_j)),$$

where $d = (d_1, \ldots, d_m)$.

6.1 PBES Reachability

We will now define reachability for a PBES. The set of states is $S = \{X_i(d) \mid 1 \leq i \leq n \wedge d \in D\}$. Each summand (i, j) with $j \in J_i$ defines a transition relation R_{ij} that is characterized by the parameters

$$\begin{cases} \sigma_i & \text{a fixpoint symbol} \\ e_j & \text{a sequence of quantifier variables} \\ (X = X_i) \wedge f_{ij}(d, e_j) & \text{a condition} \\ X_{g_{ij}}(h_{ij}(d, e_j)) & \text{the successor states,} \end{cases}$$

where $X(d)$ is the current state.

In order to apply the reachability algorithm, we need the following ingredients (exactly the same as for LPSs):

$\{(x, y) \in R_i \mid x \in todo\}$	enumerate solutions of condition
conversion between states and LDDs	map values to integers
read_parameters(R_i)	syntactic analysis
write_parameters(R_i)	syntactic analysis
project$(R_i, I_{r,i}, I_{w,i})$	discard unused parameters
project$(todo, I_{r,i} \cup I_{w,i})$	project (Sylvan function)
relprod$(L_i, todo, I_r, I_w)$	relprod (Sylvan function)

6.2 Splitting Conditions

Symbolic reachability is most effective when the number of read/write parameters and the number of transitions per transition group are small. We can split the disjunctive conditions in a conjunctive equation as follows: Consider a PBES $\mathcal{E}$ in SRF with an equation $\sigma_i X_i(d : D) \in \mathcal{E}$ of the shape:

$$\sigma_i X_i(d : D) = \bigwedge_{j \in J_i} \forall e_j : E_j . (\bigvee_{k \in K_{ij}} f_{ijk}(d, e_j)) \implies X_{g_{ij}}(h_{ij}(d, e_j))$$

$$\sigma_i X_i(d : D) = \bigwedge_{j \in J_i} \forall e_j : E_j . \bigwedge_{k \in K_{ij}} f_{ijk}(d, e_j) \implies X_{g_{ij}}(h_{ij}(d, e_j))$$

$$\sigma_i X_i(d : D) = \bigwedge_{j \in J_i} \bigwedge_{k \in K_{ij}} \forall e_j : E_{ij} . f_{ijk}(d, e_j) \implies X_{g_{ij}}(h_{ij}(d, e_j))$$

This transformation is safe and allows us to consider every $f_{ijk}(d, e_j))$ as its own transition group. A similar transformation can be applied to conjunctions in a disjunctive equation.

We can apply another transformation to conjunctive conditions inside conjunctive equations. By introducing a new equation for every conjunctive clause $f_{ijk}(d, e_j))$, for $k \in K_j$, we can remain in SRF by the following transformation:

$$\sigma_i X_i(d : D) = \bigwedge_{j \in J_i} \forall e_j : E_j.(\bigwedge_{k \in K_{ij}} f_{ijk}(d, e_j)) \implies X_{g_{ij}}(h_{ij}(d, e_j))$$

$$\sigma_i X_i(d : D) = \bigwedge_{j \in J_i} \forall e_j : E_j. \bigvee_{k \in K_{ij}} (\neg f_{ijk}(d, e_j) \vee X_{g_{ij}}(h_{ij}(d, e_j)))$$

$$\sigma_i X_i(d : D) = \bigwedge_{j \in J_i} \forall e_j : E_j.true \implies Y_j(h_{ij}(d, e_j), e_j)$$

Where Y_j is a fresh name and its equation defined as follows:

$$\nu Y_j(d : D, e_j : E_j) = (\bigvee_{k \in K_{ij}} \neg f_{ijk}(d, e_j) \wedge Y_j(d, e_j)) \vee (true \wedge X_{g_{ij}}(d, e_j))$$

Note that this transformations makes the variable e_j visible in the state space. Furthermore, this parameter is also quantified without a condition. In practice, this means that the latter transformation is probably unwanted in the presence of quantifiers. Note that every $j \in J_i$ can be transformed according to whether the condition is conjunctive or disjunctive respectively, or not transformed at all. The cases for a disjunctive or conjunctive condition within a disjunctive equation are completely dual.

We can also observe that in the equation for Y_j there is always a dependency on $X_{g_{ij}}(h_{ij}(d, e_j))$, whereas, in the original equation this was guarded by $(\bigwedge_{k \in K_j} f_{ijk}(d, e_j))$. Alternatively, we can also consider the following equation for Y_j such that $X_{g_{ij}}(h_{ij}(d, e_j))$ still has (weaker) guards.

$$\nu Y_j(d : D, e_j : E_j) = \bigvee_{k \in K_j} true \wedge Y_{jk}(d, e_j)$$

$$\nu Y_{jk}(d : D, e_j : E_j) = f_{ijk}(d, e_j) \implies X_{g_{ij}}(h_{ij}(d, e_j))) \qquad \text{for } k \in K_i$$

7 Parity Game Solving

Let $G = (V, E, r, (V_0, V_1))$ be a parity game where V_0 and V_1 are disjoint sets of vertices in V owned by 0 (even) and 1 (odd) respectively. Furthermore, $E \subseteq V \times V$ is the edge relation, and we denote elements of it by $v \to u$ iff $(v, u) \in E$. Finally, $r(v)$ is the priority function assigning a priority to every vertex $v \in V$. For a non-empty set $U \subseteq V$ and a player α, the control predecessor of set U contains the vertices for which player α can force entering U in one step. Let $\mathsf{pre}(U, V) = \{u \in U \mid \exists v \in V : u \to v\}$ then it is defined as follows:

$$\mathsf{cpre}_\alpha(G, U) = (V_\alpha \cap \mathsf{pre}(G, U)) \cup (V_{1-\alpha} \setminus \mathsf{pre}(G, V \setminus U))$$

The α-attractor into U, denoted $Attr_\alpha(U, V)$, is the set of vertices for which player α can force any play into U. We define $Attr_\alpha(U, V)$ as $\bigcup_{i \geq 0} Attr^i_\alpha(U, V)$, the limit of approximations of the sets $Attr^n_\alpha(U, V)$, which are inductively defined as follows:

$$\begin{array}{lcl} Attr^0_\alpha(G, U) & = & U \\ Attr^{n+1}_\alpha(G, U) & = & Attr^n_\alpha(G, U) \\ & \cup & \mathsf{cpre}_\alpha(G, Attr^n_\alpha(G, U)) \end{array}$$

We reformulate this into

$$\begin{array}{lcl} Attr^0_\alpha(G, U) & = & U \\ Attr^{n+1}_\alpha(G, U) & = & Attr^n_\alpha(G, U) \\ & \cup & (V_\alpha \cap \mathsf{pre}(V, Attr^n_\alpha(G, U))) \\ & \cup & (V_{1-\alpha} \cap (V \setminus \mathsf{pre}(V, V \setminus Attr^n_\alpha(G, U)))) \end{array}$$

This can be further optimised by avoiding two intersections and only computing predecessors within $V_{1-\alpha}$. Furthermore, for a union of sparse relations $\to = \bigcup_{i=1}^n \to_i$ we can also apply the transition relations $\to_i$ directly instead of determining $\to$ first. For $pred_i(U, V) = \{u \in U \mid \exists v \in V : u \to_i v\}$ we define:

$$\begin{array}{lcl} Attr^0_\alpha(G, U) & = & U \\ Attr^{n+1}_\alpha(G, U) & = & Attr^n_\alpha(G, U) \\ & \cup & \bigcup_{i=1}^n \mathsf{pre}_i(V_\alpha, Attr^n_\alpha(G, U)) \\ & \cup & (V_{1-\alpha} \setminus \bigcup_{i=1}^n \mathsf{pre}_i(V_{1-\alpha}, V \setminus Attr^n_\alpha(G, U)) \end{array}$$

First of all, we implement a variant of $\mathsf{cpre}_\alpha(G, U)$ that uses the union of sparse relations. The parameter $Z_{outside}$ can be used as an optimisation to reduce the amount of states considered, but can always be equal to V. In the attractor set computation this will be equal to $V \setminus Attr^n_\alpha(G, U))$.

Algorithm 7 Control predecessor set computation for a union of sparse relations

```
1: function CPRE_α(G = (V, E, r, (V_0, V_1)), U, Z_outside)
2:     P := pre(G, U)
3:     P_α := P ∩ V_α
4:     P_forced := P ∩ V_{1−α}
5:     for 1 ≤ i ≤ n do
6:         P_forced := P_forced \ pre_i(P_forced, Z_outside)
7:     return P_α ∪ P_forced
```

For the actual implementation of the attractor set scheme we make two additional observations. Instead of determining all predecessors for states in $V \setminus Attr^n_\alpha(U, V))$ in the final step we only have to determine predecessors that can actually reach $V \setminus Attr^n_\alpha(U, V))$ in one step. Furthermore, we can actually keep track of the states that have been added in the last iteration and were not yet part of the attractor set. We then only have to compute predecessors with respect to this todo set.

Algorithm 8 Attractor set computation for a union of sparse relations

function $\textsc{Attr}_\alpha(G = (V, E, r, (V_0, V_1),\ U)$
  $todo := U$
  $Z := U$
  $Z_{outside} := V \setminus X$
  while $todo \neq \emptyset$ **do**
    $todo := \mathsf{CPre}_\alpha(G, Z, Z_{outside})$
    $Z := Z \cup todo$
    $Z_{outside} := Z_{outside} \setminus todo$
  return Z

7.1 Zielonka

The standard Zielonka solving algorithm, defined in Algorithm 10 requires that every vertex has an outgoing edge. If this is the case then the graph is called *total*. We can achieve this by extending every disjunctive PBES equation with $\mathsf{true} \wedge X_{\mathsf{false}}$ where X_{false} is defined as $\mu X_{\mathsf{false}} = \mathsf{true} \wedge X_{\mathsf{false}}$ and similarly for the conjunctive PBES equations with X_{true}. However, adding these unnecessary transitions can be costly.

Therefore, if we perform the deadlock detection we can avoid extending the PBES and obtain a total graph by performing the preprocessing step described in Algorithm 9. Every disjunctive vertex that is a deadlock is won by player odd (previously indicated by a transition to X_{false}) and every conjunctive vertex that is a deadlock by player even. If we compute the attractors to these won vertices then the resulting graph is total and can be used in the Zielonka algorithm as follows $\textsc{zielonka}(\textsc{preprocess}(V, D, \emptyset, \emptyset))$.

Algorithm 9 Preprocess the graph to be total and remove deadlocks D already solved vertices W_0, W_1 where W_0 is won by even and W_1 by odd.

function $\textsc{Preprocess}(V, D, W_0, W_1)$
  $W_0' \leftarrow W_0 \cup (D \cap V_1)$
  $W_1' \leftarrow W_1 \cup (D \cap V_0)$
  $W_0' \leftarrow Attr_0(W_0', V)$
  $W_1' \leftarrow Attr_1(W_1', V)$
  return $V \setminus (W_0' \cup W_1')$

Algorithm 10 Zielonka

function $\textsc{Zielonka}(V)$
  if $V = \emptyset$ **then**
    return $\emptyset, \emptyset$
  $m := \min(\{r(v) \mid v \in V\})$
  $\alpha := m \bmod 2$
  $U := \{v \in V \mid r(v) = m\}$
  $A := Attr_\alpha(U, V)$
  $W_0', W_1' := \textsc{Zielonka}(V \setminus A)$
  if $W_{1-\alpha}' = \emptyset$ **then**
    $W_\alpha, W_{1-\alpha} := A \cup W_\alpha', \emptyset$
  else
    $B := Attr_{1-\alpha}(W_{1-\alpha}', V)$
    $W_0, W_1 := \textsc{Zielonka}(V \setminus B)$
    $W_{1-\alpha} := W_{1-\alpha} \cup B$
  return W_0, W_1

7.2 Partial Solving and Safe (Chaining) Attractors

We recall some of the definitions necessary for the purpose for partial solving (incomplete) parity games that is introduced in [**?**], and provide pseudocode for these algorithms as well. First of all, we introduce the notion of an incomplete parity game $\eth = (G, I)$, where $I \subseteq V$ is a set of incomplete vertices.

$$\mathsf{spre}_\alpha(G, U) = (V_\alpha \cap \mathsf{pre}(G, U)) \cup (V_{1-\alpha} \setminus (\mathsf{pre}(G, V \setminus U) \cup I))$$

Secondly, we extend our attractor set computation to allow chaining through the predecessors. For this purpose we introduce a chaining predecessor function $\mathsf{chained\_pre}(G, U, W)$ that has a parameter W which defines the set of vertices where chaining is allowed. A function is a chaining predecessor iff it returns a set $P = \mathsf{chained\_pre}(G, U, W)$ such that $\mathsf{pre}(G, U) \subseteq P$ and $P \subseteq \{u \in U \mid \exists v \in V : u \to_W^* v\}$ such that $u \to_W^* v$ iff there is a sequence of vertices $s \to u_0 \to u_1 \to \ldots \to_v$ for which $u_0, u_1, \ldots \in W$.

Algorithm 11 Safe control predecessor set computation for a union of sparse relations

```
1: function SPRE_α(G = (V, E, r, (V_0, V_1)), U, Z_outside, I, W)
2:     P := chained_pre(G, U, V_α ∩ W)
3:     P_α := P ∩ V_α
4:     P_forced := (P ∩ V_{1−α}) \ I
5:     for 1 ≤ i ≤ n do
6:         P_forced := P_forced \ pre_i(P_forced, Z_outside)
7:     return P_α ∪ P_forced
```

7.2.1 Winning Cycle Detection

First, we implement cycle detection for a set of vertices V by determining the largest subset U of V such that every vertex in U has a predecessor in U, as presented in Algorithm 12. If U satisfies this condition then every vertex in U is part of a cycle.

Algorithm 12 Cycle detection

```
1: function DETECT-CYCLES(V)
2:     U := ∅
3:     U' := V
4:     while U ≠ U' do
5:         U := U'
6:         U' := U ∩ pred(U, U)
7:     return U
```

If one of the players can force a play through a cycle of its own priority then these vertices (and all vertices in a attractor to that set) are won by that player. Therefore, for every priority p let $\alpha = p \mod 2$ be a player and let $P = \{v \in V_\alpha \mid r(v) = p\}$ be a subset of vertices owned by player of parity p. Then the set of vertices resulting from $Attr_\alpha(\mathsf{detect\text{-}cycles}(V), V)$ are won by player α.

8 LDD Operations

For an LDD A we use $\mathit{down}(A)$ to denote $A[x_i = v]$ and $\mathit{right}(A)$ to denote $A[x_i > v]$. We use $|A|$ to denote the size of the LDD, determined by the number of nodes. Given a vector $v = x_0\, x_1 \cdots x_n$ we define its length, denoted by $|v|$, as $n + 1$. Note that an LDD can represent (some) sets where two vectors have different lengths. For example the set $\{1\,1, 0\}$ can be represented by an LDD. However, the set $\{1\,1, 1\}$ cannot be represented since the root node can either have true or $\mathit{node}(1, \mathsf{true}, \mathsf{false})$ as the down node and true has no down or right nodes. In general, we cannot represent a set with vectors that are strict prefixes (any vector $x_0 \cdots x_m$ with $m < n$ is a strict prefix of v) of other vectors in the set. In practice, this means that we only consider LDDs where every vector in the represented set has the same length. We refer to this length as the height of the LDD. We often require that the input LDDs have equal height since not every output can be represented. For example the union of $\{1\,1\}$ and $\{1\}$ cannot be represented as previously shown.

8.1 Union

We define the union operator on two equal height LDDs A and B. This computes the union of the represented sets of vectors.

Algorithm 13 Union of two equal height LDDs A and B

```
1:  function UNION(A, B)
2:      if A = B then
3:          return a
4:      else if A = false then
5:          return b
6:      else if B = false then
7:          return a
8:      if val(A) < val(B) then
9:          return node(val(A), down(A), UNION(right(A), B))
10:     else if val(A) = val(B) then
11:         return node(val(A), UNION(down(A), down(B)), UNION(right(A), right(B)))
12:     else if val(A) > val(B) then
13:         return node(val(B), down(B), UNION(A, right(B)))
```

Lemma 21. *For all LDDs A and B it holds that $[\![\textsc{union}(A, B)]\!] = [\![A]\!] \cup [\![B]\!]$.*

Proof. Pick arbitrary LDDs A and B. Proof by induction on the structure of A and B. For all LDDs A' and B' we assume that $[\![\textsc{union}(A', \mathit{right}(B'))]\!] = [\![A']\!] \cup [\![\mathit{right}(B')]\!]$ and $[\![\textsc{union}(A', \mathit{down}(B'))]\!] = [\![A']\!] \cup [\![\mathit{down}(B')]\!]$. Furthermore, $[\![\textsc{union}(\mathit{right}(A'), B')]\!] = [\![\mathit{right}(A')]\!] \cup [\![B']\!]$ and $[\![\textsc{union}(\mathit{down}(A'), B')]\!] = [\![\mathit{down}(A')]\!] \cup [\![B']\!]$.

Base case. The LDD A is either true or false. Then B is either true or false due to the equal height assumption. In both cases t he terminal conditions ensure correctness. For example $[\![\textsc{union}(\mathsf{true}, \mathsf{false})]\!] = [\![\mathsf{true}]\!]$ and $\{[]\} \cup \emptyset = \{[]\}$. Similarly, for the case where B is either true or false.

Inductive step.

- Case $\mathit{val}(A) < \mathit{val}(B)$. Since A is an LDD we know that $\mathit{val}(A) < \mathit{val}(\mathit{right}(A))$. Therefore, we know that $[\![\mathit{node}(\mathit{val}(A), \mathit{down}(A), \textsc{union}(\mathit{right}(A), B))]\!]$ is equal to $\{\mathit{val}(A)\, x \mid x \in [\![\mathit{down}(A)]\!]\} \cup [\![\textsc{union}(\mathit{right}(A), B)]\!]$. It follows that $[\![\textsc{union}(\mathit{right}(A), B)]\!]$ is equal to $[\![\mathit{right}(A)]\!] \cup [\![B]\!]$. From which we can derive $[\![A]\!] \cup [\![B]\!]$.
- Case $\mathit{val}(A) = \mathit{val}(B)$. Since A is an LDD we know that $\mathit{val}(A) < \mathit{val}(\mathit{right}(A))$ and similarly because B is an LDD we know that $\mathit{val}(A) < \mathit{val}(\mathit{right}(B))$. Therefore, the following node is valid

and $[\![node(val(A), \text{UNION}(down(A), down(B)), \text{UNION}(right(A), right(B)))]\!]$ is equal to $\{val(A)\, x \mid x \in [\![\text{UNION}(down(A), down(B))]\!]\} \cup [\![\text{UNION}(right(A), right(B))]\!]$. It follows that the interpretation $\{val(A)\, x \mid x \in [\![\text{UNION}(down(A), down(B))]\!]$ is equal to $\{val(A)\, x \mid x \in [\![down(A)]\!]\} \cup \{val(A)\, x \mid x \in [\![down(B)]\!]\}$ and $[\![\text{UNION}(right(A), right(B))]\!]$ is equal to $[\![(right(A)]\!] \cup [\![right(B))]\!]$.

- Case $val(A) > val(B)$. Similar to the $val(A) < val(B)$ case. □

We can show that $|\text{UNION(A, B)}|$ for any LDDs A and B is at most $|A| + |B|$. The time complexity of UNION(A, B) is also of order $\mathcal{O}(|A| + |B|)$.

8.2 Project

Given a vector $x_0 \cdots x_n$ and a subset $I \subseteq \mathbb{N}$, we define the *projection*, denoted by $project(x_0 \cdots x_n, I)$, as the vector $x_{i_0}, \ldots, x_{i_l}$ for the largest $l \in \mathbb{N}$ such that $i_0 < i_1 < \ldots < i_l \leq n$ and $i_k \in I$ for $0 \leq k \leq l$. We define the projection operator for an LDD where every vector in the set is projected. For the LDD operator it is more convenient to specify the indices $I \subseteq \mathbb{N}$ by a vector $x_0 \cdots x_n$ such that for $0 \leq i \leq n$ variable x_i is one iff $i \in I$. The operator takes an LDD A of height n and a sequence of numbers $x_0\, x_1, \cdots x_n$.

Algorithm 14 Project vectors of an LDD A of height n using a sequence $x_0 \cdots x_n$

```
1:  function PROJECT(A, x_0 x_1 ··· x_n)
2:      if A = true then
3:          return true
4:      else if A = false then
5:          return false
6:      if x_0 = 1 then
7:          return node(val(A), PROJECT(down(A), x_1 ··· x_n), PROJECT(right(A), x_0 ··· x_n))
8:      else if x_0 = 0 then
9:          a ← A
10:         R ← false
11:         while a ≠ false do
12:             R ← UNION(R, PROJECT(down(a), x_1 ··· x_n))
13:             a ← right(a)
14:         return R
```

Lemma 22. *For all LDDs A and sequences $x_0\, x_1 \cdots x_n$ it holds that $[\![\text{PROJECT}(A, x_0 \cdots x_n)]\!]$ is equal to $\{project(v, x_0 \cdots x_n) \in [\![A]\!]\}$.*

Note that for a sequence of $|A|$ zeroes $\text{PROJECT}(A, 0 \cdots 0)$ is equal to true and for a sequence of $|A|$ ones $\text{PROJECT}(A, 1 \cdots 1)$ is equal to A.

8.3 Caching

We can speed up LDD operations at the cost of memory by using an operation cache. For every operation there will be a separate *global* cache, represented by a set, that stores a tuple of the inputs and the output. We use *global* to emphasize that every recursion sees the latest values stored in the cache. For example the $\text{UNION}(A, B)$ operation has a cache C_{UNION} and the pseudocode of UNION is changed such that after the terminal cases first we check whether $\exists R : (A, B, R) \in C_{\text{UNION}}$ and return the result R if that is the case. Otherwise, we perform the computation as before but store the result in C_{UNION} instead before returning it. Thus we obtain the following algorithm:

Algorithm 15 Union of two LDDs A and B. With a set C_{UNION} as operation cache

```
1:  function UNION(A, B, C_UNION)
2:      if A = B then
3:          return a
4:      else if A = false then
5:          return b
6:      else if B = false then
7:          return a
8:      if ∃R : (A, B, R) ∈ C_UNION then return R
9:      if val(A) < val(B) then
10:         R ← node(val(A), down(A), UNION(right(A), B, C_UNION))
11:     else if val(A) = val(B) then
12:         R ← node(val(A), union(down(A), down(B), union(right(A), right(B), C_UNION)))
13:     else if val(A) > val(B) then
14:         R ← node(val(B), down(B), UNION(A, right(B), C_UNION))
15:     C_UNION ← C_UNION ∪ {(A, B, R)}
16:     return R
```

8.4 Parallelism

In some cases we can improve the performance further by computing several results in parallel during the operation. For example in the union of two LDDs A and B in the case $val(A) = val(B)$ we could determine the result of both unions in parallel and only merge them after they have finished.