Read, Write and Copy Dependencies for Symbolic Model Checking

Jeroen Meijer, Gijs Kant, Stefan Blom, and Jaco van de Pol

Formal Methods and Tools, University of Twente, The Netherlands
{j.j.g.meijer,g.kant,s.c.c.blom,j.c.vandepol}@utwente.nl

Abstract. This paper aims at improving symbolic model checking for explicit state modeling languages, e.g., Promela, Dve and mcrl2. The modular Pins architecture of LTSmin supports a notion of event locality, by merely indicating for each event on which variables it depends. However, one could distinguish four separate dependencies: *read*, *may-write*, *must-write* and *copy*. In this paper, we introduce these notions in a language-independent manner. In particular, models with arrays need to distinguish overwriting and copying of values.

We also adapt the symbolic model checking algorithms to exploit the refined dependency information. We have implemented refined dependency matrices for Promela, Dve and mcrl2, in order to compare our new algorithms to the original version of LTSmin. The results show that the amount of successor computations and memory footprint are greatly reduced. Finally, the optimal variable ordering is also affected by the refined dependencies: We determined experimentally that variables with a read dependency should occur at a higher BDD level than variables with a write dependency.

1 Introduction

Model checking [11] is a technique to verify the correctness of systems. Often these systems are made up of several processes running in parallel. Examining all possible execution paths of the system is hard, because of the well known state space explosion problem: because of the interleaving of the processes, the possible number of states is exponential in the number of processes. Symbolic model checking [6, 13] has proven to be very effective in dealing with that problem. Symbolic here means storing sets of vectors and relations between vectors as decision diagrams, such as Binary Decision Diagrams (BDDs) or Multi-Value Decision Diagrams (MDDs). A well known symbolic model checker is nusmv [9], where systems are specified in the SMV language, directly describing transition relations.

We use the LTSmin toolset [4], which also provides a symbolic model checker, but is different from nusmv in several ways. LTSmin provides a language independent interface, called Pins, to communicate states and transitions, and learns the partitioned transition relation on-the-fly, as in, e.g., [2,7]. New transitions are learned through an explicit Next-state function, which is the language specific

E. Yahav (Ed.): HVC 2014, LNCS 8855, pp. 204–219, 2014.

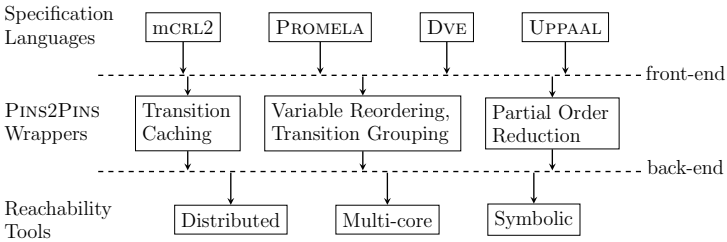

Fig. 1. Modular Pins architecture of LTSmin

part of LTSmin. Currently the languages that have been implemented on top of Pins include Promela [1], DiVinE, mCRL2, and Uppaal. In [2] and [3], the Pins interface and underlying symbolic core algorithms of LTSmin are described. An overview of the architecture is in Figure 1.

In Pins, states are represented as fixed-length *vectors* of values. Transitions are distinguished in separate disjunctive *transition groups*. A generalized definition of systems that is compatible with Pins is given in Section 2. Between slots of the state vector and the transition groups there can be dependencies, i.e., a transition group can be dependent on the value of a state variable for a condition to be true, or a transition may change the value of a state variable. The dependencies between transition groups and state slots are captured in a *dependency matrix*, which can be determined by static analysis of the model. Often it is the case that a transition group depends on a limited number of slots, which is known as *event locality*. This is the basis of many optimisations in symbolic model checking, as presented in, e.g., [5,8,10]. For symbolic state space generation it is best when the dependency matrix is sparse, i.e., when transition groups have a relatively local footprint, for the following reasons. First, a sparse matrix means that the transition relations for the transition groups depend on few variables and can be quite small. Also, because of the on-the-fly nature of LTSmin, there will be fewer redundant calls to Next-state.

To further benefit from dependencies in the input models, in this paper we refine the notion of dependency and distinguish three types of dependencies: *read dependence* and two types of *write dependence*, *must-write* and *may-write*. To illustrate read and write dependence, we use a simple system with three variables $\langle x, y, z \rangle$ and two transitions:

$$\begin{array}{ll} 1 : x = 1 \vee z = 0 & \rightarrow y := 1, x := 0 \\ 2 : y = 1 & \rightarrow z := 0, x := 1 \end{array}$$

a. Transitions

$$\begin{array}{c} \\ 1 \\ 2 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \end{array}$$

b. Dep. matrix

$$\begin{array}{c} \\ 1 \\ 2 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \end{array}$$

c. Read matrix

$$\begin{array}{c} \\ 1 \\ 2 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \end{array}$$

d. Write matrix

In b), the dependency matrix indicates no event locality, but if read and write dependencies are considered separately, as in c) and d), then we see the transition groups depend on most variables only for reading or only for writing. Separating reads and writes helps in reducing the number of Next-state calls, the size of

the transition relation, and the size of its symbolic representation. For instance, when a transition group is not read dependent on variable y, then the previous value of y can be discarded in computing the successors of a state.

However, it is not trivial to statically determine whether a transition group writes to a state slot. In the case of dynamic addressing of variables, e.g., writing a position in an array, it may be needed to mark an entire array as write dependent, even if only one position is changed. This problem is resolved by using two types of write dependence: *may-write*, which allows copying of values, and *must-write*, which does not.

In [8], a similar distinction is made between types of dependence. The main difference with this work is that we deal with dynamic variable addressing, both in the definitions of dependency and in the symbolic algorithms, where we use a special symbol in transition relations to mark that a variable should be copied.

The dependencies and the associated matrices are described in detail in Section 3. There also the *row subsumption* in dependency matrices and *variable reordering* are discussed. These two techniques improve the effect of the read-write distinction. In Section 4, we provide an adapted symbolic reachability algorithm that exploits the read and write dependencies.

We have benchmarked our work with the whole Beem database and many Promela and mCRL2 models. There are many models that benefit from the distinction between read and write dependencies, but also several that do not. In Section 5, we highlight the results for six models. For mCRL2, performance is improved, because many calls to Next-state can be avoided. The Next-state function for mCRL2 is relatively slow, due to the term rewriter that was introduced to provide very expressive datatypes. For Beem and Promela models, we find an improvement when a good variable ordering (a good reordering strategy) is chosen.

This work is based on Meijer's MSc thesis [14] and extends it with an extension to the transition relation to support copying values, and an analysis of the effect of variable ordering in the context of distinct read and write dependencies.

2 The Partioned Next-State Interface (Pins)

The starting point of our approach is a generalised model of systems, called *Partioned Next-State Interface* (Pins), which allows supporting several modeling languages within a single framework, without exposing language details to the underlying algorithms.

In Pins, states are vectors of N values. We write $\langle x_1, \ldots, x_N \rangle$, or simply $\boldsymbol{x}$, for vector variables. Each slot of the vector has a unique identifier, which is used in the language front ends to specify conditions and updates. Every language module, furthermore, has a Next-state function, which computes the successor of a state. This function is partitioned in K transition groups, such that $\text{Next-state}(\boldsymbol{x}) = \bigcup_{1 \leq i \leq K} \text{Next-state}_i(\boldsymbol{x})$. A model, available through Pins, gives rise to a partitioned transition system, defined as follows.

Definition 1. *A* Partitioned Transition System *(PTS) [3] is a structure* $\mathcal{P} = \langle \langle S_1, \ldots, S_N \rangle, \langle \rightarrow_1, \ldots, \rightarrow_K \rangle, \langle s_1^0, \ldots, s_N^0 \rangle \rangle$. *The tuple* $\langle S_1, \ldots, S_N \rangle$ *defines the*

set of states $S_{\mathcal{P}} = S_1 \times \ldots \times S_N$*, i.e., we assume that the set of states is a* Cartesian product*. The* transition groups $\rightarrow_i \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$ *(for* $1 \leq i \leq K$*) define the transition relation* $\rightarrow_{\mathcal{P}} = \bigcup_{i=1}^{K} \rightarrow_i$*. The initial state is* $\boldsymbol{s}^0 = \langle s_1^0, \ldots, s_N^0 \rangle \in S_{\mathcal{P}}$*. We write* $\boldsymbol{s} \rightarrow_i \boldsymbol{t}$ *when* $(\boldsymbol{s}, \boldsymbol{t}) \in \rightarrow_i$ *for some* $1 \leq i \leq K$*. Also we write* $\boldsymbol{s} \rightarrow_{\mathcal{P}} \boldsymbol{t}$ *when* $(\boldsymbol{s}, \boldsymbol{t}) \in \rightarrow_{\mathcal{P}}$*.*

The partitioning of the state vector into *slots* and of the transition relations into *transition groups*, enables to specify the *dependencies* between the two, i.e., which transition groups touch which slots of the vector. The definition of these dependencies will be given in Section 3. Here we give an abstract description of how the variables in the state vector are read from and written to by the transition groups.

For every language module this is different, but there is a common pattern. In all of the supported languages, the specification of a transition is in the shape $\textsc{Next-state}_i(\boldsymbol{x}) = \mathit{cond}_i \rightarrow \mathrm{action}_i \,.\, \mathit{update}_i(\boldsymbol{x})$.

The expression cond_i is the condition that guards an action and may read variables from $\boldsymbol{x}$. The symbol 'action_i' specifies the name of the action that is performed, i.e., the transition label. The expression $\mathit{update}_i(\boldsymbol{x})$ defines the state after the action. The update is a parallel assignment to the variables in the vector. However, these variables may be defined dynamically, e.g., they may be references to a location in an array.

Example 1. Given a state vector with variables $\langle c, a_0, a_1, i \rangle$, valid assignments would be, e.g., $c \coloneqq c + 1$, $a_i \coloneqq a_{1-i}$ and $i \coloneqq c$.

We define the state updates more formally, abstracting away from the specific input languages of LTSmin.

Definition 2 (State Update Specification). *The syntax of a state update of transition group* i *is as follows:* $\sigma_i ::= c_i \rightarrow \mathrm{a}_i \,.\, \langle v_{i,1} \coloneqq t_{i,1}, \ldots, v_{i,L_i} \coloneqq t_{i,L_i} \rangle$*, where* $L_i \leq N$ *and* c_i*,* $v_{i,j}$*, and* $t_{i,j}$ *are expressions over* $x_1, \ldots, x_N$*. The conditions* c_i *are Boolean expressions and the left hand sides* $v_{i,j}$ *evaluate to variables in* $\{x_1, \ldots, x_N\}$*. The semantics of this state update is defined as the successor states after applying the update:*

$$\boldsymbol{s} \rightarrow_i \boldsymbol{t} \iff [\![c_i]\!]_{\boldsymbol{s}} \wedge \boldsymbol{t} = \boldsymbol{s}[[\![v_{i,1}]\!]_{\boldsymbol{s}} \coloneqq [\![t_{i,1}]\!]_{\boldsymbol{s}}, \ldots, [\![v_{i,L_i}]\!]_{\boldsymbol{s}} \coloneqq [\![t_{i,L_i}]\!]_{\boldsymbol{s}}] \ .$$

A State Update Specification *(SUS) is a triple* $\mathcal{U} = \left\langle \langle x_1, \ldots, x_N \rangle, \{\sigma_1, \ldots, \sigma_K\}, \boldsymbol{s}^0 \right\rangle$*, containing a vector of state variables* x_j *for* $1 \leq j \leq N$*, a set of* state updates σ_i *with* $1 \leq i \leq K$*, and an initial state* $\boldsymbol{s}^0$*.*

Example 2 (1-safe Petri net). An example model is the Petri net in Figure 2, of which a specification is given in Listing 1.1. The behavior of this 1-safe Petri net is as follows. Initially, there is only one token in p_0. If transition t_0 fires then the token is moved from place p_0 to both p_1 and p_3. Transitions $t_{1..4}$ move the tokens between places $p_{1..4}$ independently. If the token is in both p_2 and p_4 then transition t_5 can fire to move the token to p_0. There are 5 reachable states for this Petri net. With booleans represented as $0, 1$, the states are: $\{\langle 1,0,0,0,0 \rangle, \langle 0,1,0,1,0 \rangle, \langle 0,0,1,1,0 \rangle, \langle 0,1,0,0,1 \rangle, \langle 0,0,1,0,1 \rangle\}$.

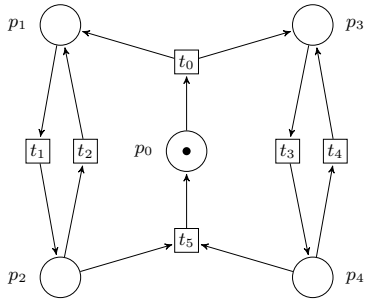

Fig. 2. Example 1-safe Petri net

Listing 1.1. State Update Specification (SUS) for a 1-safe Petri net

```
1  init ⟨true, false, false, false, false⟩
2  sus ⟨p0, p1, p2, p3, p4 ∈ 𝔹⟩ = ⟨
3      p0 →
4          t0. ⟨p0 := false, p1 := true, p3 := true⟩,
5      p1 →
6          t1. ⟨p1 := false, p2 := true⟩,
7      p2 →
8          t2. ⟨p1 := true, p2 := false⟩,
9      p3 →
10         t3. ⟨p3 := false, p4 := true⟩,
11     p4 →
12         t4. ⟨p3 := true, p4 := false⟩,
13     (p2 ∧ p4) →
14         t5. ⟨p0 := true, p2 := false, p4 := false⟩⟩
```

3 State Slot Dependencies

We exploit the notion of event locality by statically (a priori, before exploring any states) approximating dependencies between transition groups and state slots. We distinguish three types of dependencies: *read dependence* (whether the value of a state slot influences transitions), *must-write dependence* (whether a state slot is written to), and *may-write dependence* (whether a state slot may be written to, depending on the value of some other state slot). We provide formal definitions for the dependencies and dependency matrices for state update specifications.

Definition 3 (Read Independence). *Given a Partitioned Transition System (*PTS*)* $\mathcal{P} = \langle S_\mathcal{P}, \rightarrow_\mathcal{P}, \boldsymbol{s}^0 \rangle$*, transition group* i *is* read independent *on state slot* j *if: for all* $\boldsymbol{s}, \boldsymbol{t} \in S_\mathcal{P}$*: whenever* $\langle s_1, \ldots, s_j, \ldots, s_N \rangle \rightarrow_i \langle t_1, \ldots, t_j, \ldots, t_N \rangle$*, it holds that*

- *either always* $(s_j = t_j) \wedge \forall r_j \in S_j$*:* $\langle s_1, \ldots, r_j, \ldots, s_N \rangle \rightarrow_i \langle t_1, \ldots, r_j, \ldots, t_N \rangle$*;*
- *or* $\forall r_j \in S_j$*:* $\langle s_1, \ldots, r_j, \ldots, s_N \rangle \rightarrow_i \langle t_1, \ldots, t_j, \ldots, t_N \rangle$*,*

i.e., the values t_k *for* $k \neq j$ *do not depend on the value of slot* j *and either the value of state slot* j *is always copied, or always the value* t_j *is written, regardless of the value of* s_j*. In both cases the specific value of* s_j *is not relevant in transition group* i*.*

Definition 4 (Read Dependency Matrix). *For a* PTS $\mathcal{P}$*, the* Read Dependency Matrix *(RDM) is a* $K \times N$ *matrix* $RDM(\mathcal{P}) = RM^\mathcal{P}_{K \times N} \in \{0,1\}^{K \times N}$ *such that* $(RM_{i,j} = 0) \Rightarrow$ *transition group* i *is* read independent *on state slot* j*. For a State Update Specification (*SUS*)* $\mathcal{U}$*, the read dependency matrix* $RDM(\mathcal{U})$ *is defined as* $RM^\mathcal{U}_{K \times N}$ *with:*

$$RM^\mathcal{U}_{i,j} = \begin{cases} 1 \text{ if } (x_j \text{ occurs in } c_i); \text{ or} & \text{(a)} \\ \quad \text{if } \exists_{1 \leq k \leq L_i} : (x_j \text{ occurs in } t_{i,k}) \wedge (v_{i,k} \neq x_j \vee t_{i,j} \neq x_j); \text{ or} & \text{(b)} \\ \quad \text{if } \exists_{1 \leq k \leq L_i} : (x_j \text{ occurs in } v_{i,k}) \wedge (v_{i,k} \neq x_j); & \text{(c)} \\ 0 \text{ otherwise.} & \text{(d)} \end{cases}$$

In case (a), the condition c_i depends on x_j. In case (b), the right hand side of the update depends on x_j, but the assignment is not merely a copy. In case (c), variable x_j is used to dynamically determine a state slot for an assignment, but x_j is not directly used as left hand side, as in, e.g., the assignment $a_{x_j} := 1$. In that case, x_j is marked as read dependent, because it influences how the state vector is updated. In all other cases, x_j is read independent.

We say that a transition group is *must-write dependent* for variable x, if it modifies x definitely, i.e. by a static assignment. For instance, the assignment $x := 2y$ is must-write dependent for variable x, because the right-hand side does not depend on x. The assignment $x := 2x$ is must-write dependent on variable x because, independent of any other variables, it can modify the value of x. However, the assignment $x_i := 3$ is not must-write dependent on variable x_0, because for $i = 1$, the value of x_0 is never modified.

Definition 5 (Must-Write Dependency Matrix). *For a* PTS $\mathcal{P}$*, the* Must-write Dependency Matrix *(WDM) is a* $K \times N$ *matrix* $WDM(\mathcal{P}) = WM^{\mathcal{P}}_{K\times N} \in \{0,1\}^{K\times N}$ *such that* $(WM_{i,j} = 1) \Rightarrow$ *transition group* i *is* must-write dependent *on state slot* j*.*
For a SUS $\mathcal{U}$*, the must-write dependency matrix* $WDM(\mathcal{U})$ *is defined as* $WM^{\mathcal{U}}_{K\times N}$ *with:*

$$WM^{\mathcal{U}}_{i,j} = \begin{cases} 1 \text{ if } \exists_{1\le k\le L_i} : (v_{i,k} = x_j) \wedge (v_{i,k} \neq t_{i,k}) & \text{(a)} \\ 0 \text{ otherwise.} & \text{(b)} \end{cases}$$

In case (a), x_j is the left hand side of an assignment $v_{i,k} := t_{i,k}$. If the right hand side $t_{i,k}$ is the same, there is no must-write dependency, but instead the value is copied. If they are different, x_j is marked as must-write dependent. E.g., the assignment $x := x + 1$, x is marked both as must-write dependent and as read dependent.

Consider the case of an array assignment $a_i := c$. Then a_0 cannot be marked as must-write dependent. Still, we know that a_0 is either copied, or replaced by a constant. To exploit this knowledge for dynamic assignments, we introduce a third notion of independence.

3.1 The May-Write Dependency

In the case of assignment to a dynamically defined variable, using only read and must-write dependencies is not optimal, as is explained in the following example.

Example 3. Suppose we extend the specification of the 1-safe Petri net specification in Listing 1.1 by adding some data. We extend the state vector with variables $b_0, b_1 \in \mathbb{B}, i \in \{0,1\}$. The initial state is extended with the values $\langle \text{false}, \text{false}, 0\rangle$. We add two state updates: "$p_1 \to \text{w} \;.\; \langle i := 1\rangle$" and "$\text{true} \to \text{W} \;.\; \langle b_i := \text{true}\rangle$". For the second assignment it cannot be statically determined if b_0 or b_1 is written to. This depends on the value of i. Therefore, b_0 and b_1 are marked as must-write independent. However, one of both may be changed, so our definition is not sufficient in this case. Changing it in a way that marks both b_0 and b_1 is safe,

but requires that both are also marked as read dependent: one of the variables is copied and requires a read, but it cannot a priori be determined which one. Ideally, both variables are marked as write dependent, while allowing to indicate which variables are copied. Then they do not need to be read dependent.

To address the problem of dynamic resolution of variables, we introduce a weaker notion of write dependence: *may-write independence*.

Definition 6 (May-Write Independence). *Given a* PTS $\mathcal{P} = \langle S_\mathcal{P}, \to_\mathcal{P}, \boldsymbol{s}^0 \rangle$, *transition group* i *is* may-write independent *on state slot* j *if:* $\forall \boldsymbol{s}, \boldsymbol{t} \in S_\mathcal{P}$, $\langle s_1, \ldots, s_j, \ldots, s_N \rangle \to_i \langle t_1, \ldots, t_j, \ldots, t_N \rangle \Rightarrow (s_j = t_j)$, *i.e., state slot* j *is never modified in transition group* i.

Thus, if transition group i is may-write-dependent on state slot j, then there are some states $\boldsymbol{s}, \boldsymbol{t}$ and a transition $\boldsymbol{s} \to_i \boldsymbol{t}$, where the value in state slot j is changed: $s_j \neq t_j$.

Definition 7 (May-Write Dependency Matrix). *For a* PTS $\mathcal{P}$, *the* May-write Dependency Matrix *(MDM) is a* $K \times N$ *matrix* $MDM(\mathcal{P}) = MM^\mathcal{P}_{K\times N} \in \{0,1\}^{K\times N}$ *such that* $(MM_{i,j} = 0) \Rightarrow$ *transition group* i *is* may-write independent *on state slot* j.
For a SUS $\mathcal{U}$, *the may-write dependency matrix* $MDM(\mathcal{U})$ *is defined as* $MM^\mathcal{U}_{K\times N}$ *with:*

$$MM^\mathcal{U}_{i,j} = \begin{cases} 0 \text{ if } \forall 1 \leq k \leq L_i \colon \forall \boldsymbol{s} \colon (\llbracket v_{i,k} \rrbracket_{\boldsymbol{s}} = x_j) \Rightarrow (v_{i,k} = t_{i,k}) & \text{(a)} \\ 1 \text{ otherwise.} & \text{(b)} \end{cases}$$

In case (a), if $v_{i,k}$ evaluates to x_j for some state $\boldsymbol{s}$, i.e., an assignment to x_j is possible, then the assignment is a direct copy, i.e., the left hand side and right hand side are syntactically the same: $v_{i,k} = t_{i,k}$. This is determined statically by the language front-end before generation.

Example 4. In the extended Petri net example (Example 3), transition W is may-write dependent on both variables b_0 and b_1, because there exists both a state in which $i = 0$ and a state in which $i = 1$. Hence, both variables can be written to by the assignment $b_i :=$ true.

Definition 8 (Combined Dependency Matrix). *For a* PTS $\mathcal{P}$, *the* dependency matrix *(DM) is a* $K \times N$ *matrix* $DM^\mathcal{P}_{K\times N} \in \{0,1\}^{K\times N}$ *with the elements* $DM^\mathcal{P}_{i,j}$ *as specified in Table 1. Note that* $WM_{i,j} \Rightarrow MM_{i,j}$.

Example 5. The combined dependency matrix for the extended 1-safe Petri net (Example 3) is shown in Table 2.

Note that here we say a transition group has a may-write dependency on a state slot if it is not must-write dependent. This differs from the definition of may-write independence. The definition of must-write dependence may seem superfluous, but it is necessary for language front-ends and symbolic back-ends which do not support copying values. So we have to take must-write dependence into account when applying transformations on the combined dependency matrix.

Table 1. Combined DM $DM^{\mathcal{P}}$

$DM^{\mathcal{P}}_{i,j}$		$RM^{\mathcal{P}}_{i,j}$	$WM^{\mathcal{P}}_{i,j}$	$MM^{\mathcal{P}}_{i,j}$
-	(copy)	0	0	0
r	(read)	1	0	0
W	(may-write)	0	0	1
w	(must-write)	0	1	1
+	(read/write)	1	{0, 1}	1

Table 2. DM for the Petri net

	p_0	p_1	p_2	p_3	p_4	i	b_0	b_1
t_0	+	w	-	w	-	-	-	-
t_1	-	+	w	-	-	-	-	-
t_2	-	w	+	-	-	-	-	-
t_3	-	-	-	+	w	-	-	-
t_4	-	-	-	w	+	-	-	-
w	-	r	-	-	-	w	-	-
W	-	-	-	-	-	r	W	W

3.2 Optimisation Operations on the Dependency Matrix

Combining transition groups and reordering variables are two techniques to enhance symbolic state space exploration. We adapted these techniques to benefit from read and write dependencies.

Definition 9 (Row Subsumption). *For matrix rows* $\boldsymbol{m}, \boldsymbol{m'} \in M^N$ *of length* N*, the* row subsumption *operator* $\sqsubseteq\colon M^N \times M^N \to \mathbb{B}$ *is defined as follows:*

$$\boldsymbol{m} \sqsubseteq \boldsymbol{m'} \iff \forall_{1 \le j \le N}\colon m_j \le m'_j \ .$$

If a row $\boldsymbol{m}_i$ is *subsumed* by row $\boldsymbol{m}_k$, ($\boldsymbol{m}_i \sqsubseteq \boldsymbol{m}_k$), the corresponding transition groups i and k can be merged and the combined matrix row becomes the larger one, $\boldsymbol{m}_k$. In general, any two rows could be merged by taking their pointwise least upperbound. The result is that there are fewer transition groups and less applications of a transition relation, but the transition relations will be larger. For this to work, we need a correct definition of $\le$, i.e., a partial order on dependencies, which is given in Figure 3. Note that *may-write* dependency (W) may subsume a copy dependency (-), but *must-write* dependency (w) does not, because only W supports copying values.

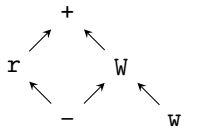

Fig. 3. Partial order on dependencies $\le$

Variable reordering is widely used to reduce the size of decision diagrams [16]. When using separate read and write dependencies, the order of read and write variables needs to be taken into account. In general it is a good idea to move variables that are read before variables that are written. Algorithm 1 uses the heuristic that every read which occurs after a write is increasingly expensive. Algorithm 2 shows a naive way to compute the cost of every column permutation of the Dependency Matrix (DM). The algorithm will choose the matrix with the lowest cost. Naturally, trying every permutation is exponentially expensive in terms of number of columns. LTSmin implements more advanced column swap algorithms, for instance based on simulated annealing from [17]).

The dependency matrices have been implemented for the mCRL2, Dve and Promela input languages. For mCRL2, may-write dependencies are not needed,

Algorithm 1. COST

```
   Input: DM
1  cost ← 0;
2  for 0 ≤ i < K do
3  |  writes ← 0;
4  |  for 0 ≤ j < N do
5  |  |  if DM_{i,j} ∈ {W, w} then
   |  |  |  writes ← writes + 1;
6  |  |  if DM_{i,j} = r then
   |  |  |  cost ← cost + writes;
7  |  end
8  end
9  return cost;
```

Algorithm 2. DM-OPTIMIZE

```
   Input: DM
1  best ← DM;
2  for 0 ≤ i < N do
3  |  for i < j < N do
4  |  |  test ← SWAP-COLUMNS(DM, i, j);
5  |  |  if COST(test) < COST(best) then
   |  |  |  best ← test;
6  |  end
7  end
8  return best;
```

because assignment to dynamic variables is not supported in the language. Still, may-write dependencies can arise by row subsumption.

4 Symbolic Reachability Analysis

To allow symbolic reachability analysis with read, write and copy dependencies we provide three definitions. The first contains projections on dependency matrices. Secondly, we provide a definition of the restricted NEXT-STATE function from read-projected states to their write-projected successors according to transition group $\rightarrow_i$, as it is used in PINS for on-the-fly reachability analysis. This technique is language independent, but depends essentially on the PINS-architecture, based on state vectors, disjunctive transition partitioning and read and write dependency matrices. Lastly, we provide a symbolic definition of NEXT that formalizes the transition relation and the application of the transition relation on a set of states.

Notation. For convenience, we introduce the function *ind*, the column indices of the cells that contain a '1' in row M_i: $ind(M_i) = \{j \mid 1 \leq j \leq |M_i| \wedge M_{i,j} = 1\}$. Given a vector $\boldsymbol{s}$ and a set of indices I, the notation $(s_j)_{j \in I}$ is used to represent the *subvector* $(s_{\bar{I}_1}, \ldots, s_{\bar{I}_\ell})$ of length $\ell = |I|$, where $\bar{I}$ is the sorted list of elements from I.

Definition 10 (Projections). *For any vector set $S = \prod_{1 \leq j \leq N} S_j$, transition group $1 \leq i \leq K$ and $K \times N$ matrix M, we define the projection $\pi_i^M \colon S \to \prod_{j \in ind(M_i)} S_j$ as $\pi_i^M(\boldsymbol{x}) = (x_j)_{j \in ind(M_i)}$, i.e., the subvector of $\boldsymbol{x}$ that contains the elements at indices in $ind(M_i)$, the indices that are marked in row i of matrix M. The projection function is extended to apply to sets in a straightforward way: $\pi_i^M(S) = \{\pi_i^M(\boldsymbol{x}) \mid \boldsymbol{x} \in S\}$. We also write $\pi_i^{\mathtt{r}}$ for π_i^{RM} and $\pi_i^{\mathtt{W}}$ for π_i^{MM}.*

Using these read and write projections, we can define how the read and write dependency matrices can be used to compute the successor states for a transition group, using only the dependent variables. We define the function NEXT-STATE$_i^{\mathrm{p}}$ that takes as input a read projected vector, and computes for transition group i

the set of may-write projected successor vectors. The input read projected vector may match a set of input states, and each of the output projected successor vectors may represent a set of successor states. In the case a variable is may-write dependent, but not changed, the symbol ▲ is used to mark that the variable should be copied from the input vector. This can occur, e.g., in the case an entire array $a_{1..10}$ is marked may-write dependent, because of an assignment $a_z := e$. If $z = 5$, the position a_5 is written to and all positions a_j with $j \neq 5$ are marked with ▲. We use $S_j^{\blacktriangle}$ for $S_j \cup \{\blacktriangle\}$ and $S_{\mathcal{P}}^{\blacktriangle}$ for the set $S_1^{\blacktriangle} \times \cdots \times S_N^{\blacktriangle}$.

Definition 11 (Partitioned Next-State Function). $\textsc{Next-state}_i^{\mathrm{p}} : \pi_i^{\mathtt{r}}(S_{\mathcal{P}}) \to \wp(\pi_i^{\mathtt{W}}(S_{\mathcal{P}}^{\blacktriangle}))$. *Given a read projected state* $(s_j)_{j \in ind(RM_i)}$,

$$\textsc{Next-state}_i^{\mathrm{p}}((s_j)_{j\in ind(RM_i)}) = \Big\{ \pi_i^{\mathtt{W}}(\boldsymbol{t}) \mid \exists \boldsymbol{s'}, \boldsymbol{t'}, \boldsymbol{t} \in S_{\mathcal{P}} : \pi_i^{\mathtt{r}}(\boldsymbol{s'}) = (s_j)_{j\in ind(RM_i)} \wedge \boldsymbol{s'} \rightarrow_i \boldsymbol{t'} \wedge \forall_{1\le j\le N} : t_j = \begin{cases} \blacktriangle \text{ if } (j \notin ind(MM_i) \vee s_j = t_j), \\ t'_j \text{ otherwise} \end{cases} \Big\} .$$

The result vectors $(t_j)_{j\in ind(MM_i)}$, combined with the input vectors $(s_j)_{j\in ind(RM_i)}$ are stored in a symbolic transition relation $\hookrightarrow_i^{\mathrm{p}}$.

Definition 12 (Next). *We define the function* $\textsc{Next} : \wp(S_{\mathcal{P}}) \times (\pi^{\mathtt{r}}(S_{\mathcal{P}}) \times \pi^{\mathtt{W}}(S_{\mathcal{P}}^{\blacktriangle}) \to \mathbb{B}) \times M_N \times M_N \to \wp(S_{\mathcal{P}})$, *which applies a partial transition relation to a set of states, as follows. Given a set* S, *a partial transition relation* $\hookrightarrow^{\mathrm{p}}$, *a read matrix row* $\boldsymbol{r}$ *and a may-write matrix row* $\boldsymbol{w}$,

$$\textsc{Next}(S, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}) = \Big\{ \boldsymbol{y} \in S_{\mathcal{P}} \mid \exists \boldsymbol{x} \in S, \boldsymbol{z} \in S_{\mathcal{P}}^{\blacktriangle} : \hookrightarrow^{\mathrm{p}}(\pi^{\boldsymbol{r}}(\boldsymbol{x}), \pi^{\boldsymbol{w}}(\boldsymbol{z})) \wedge \bigwedge_{j\in ind(\boldsymbol{w})} \left(y_j = \begin{cases} x_j \text{ if } z_j = \blacktriangle, \\ z_j \text{ otherwise} \end{cases} \right) \wedge \bigwedge_{j\notin ind(\boldsymbol{w})} (y_j = z_j) \Big\} .$$

The symbolic reachability algorithm that uses the functions Next-state and Next is in Algorithm 3. The algorithm is an extension of the symbolic reachability algorithm in [2, Table 6].

Variable $\mathcal{R}$ maintains the set of reachable states so far, while $\mathcal{L}$ stores the current level. After initialisation (lines 1–6), the next level $\mathcal{N}$ will be continuously computed and added, until the current level is empty (lines 7–15). In each iteration, first the new transitions must be learned (Algorithm 4). The next level is computed by calling Next for each transition group (line 11).

Our extension includes three subtle modifications compared to [2, Table 6], when growing the transition relations on-the-fly (Algorithm 4). *First*, the state is read-projected in line 2. The benefit being that fewer calls to Next-state are needed. *Secondly*, the tuples added to the partial transition relation in line 4 may contain the special value ▲. This allows dynamic assignments to be resolved

Algorithm 3. REACH-BFS-PREV

```
   Input  : s^0 ∈ S_P, K ∈ ℕ, RM, MM
   Output: The set of reachable states R
1  R ← {s^0};
2  L ← R;
3  for 1 ≤ i ≤ K do
4  |  R_i^P ← ∅;
5  |  ↪_i^P ← ∅;
6  end
7  while L ≠ ∅ do
8  |  LEARN-TRANS();
9  |  N ← ∅;
10 |  for 1 ≤ i ≤ K do
11 |  |  N ← N ∪ NEXT(L, ↪_i^P, RM_i, MM_i);
12 |  end
13 |  L ← N \ R;
14 |  R ← R ∪ N
15 end
16 return R
```

Algorithm 4. LEARN-TRANS

```
   Output: Extends ↪_i^P with new
           transitions on-the-fly
1  for 1 ≤ i ≤ K do
2  |  L^P ← π_i^r(L);
3  |  for s^P ∈ L^P \ R_i^P do
4  |  |  ↪_i^P ← ↪_i^P ∪ {⟨s^P, d^P⟩ |
5  |  |  d^P ∈ NEXT-STATE_i(s^P)};
6  |  end
7  |  R_i^P ← R_i^P ∪ L^P;
8  end
```

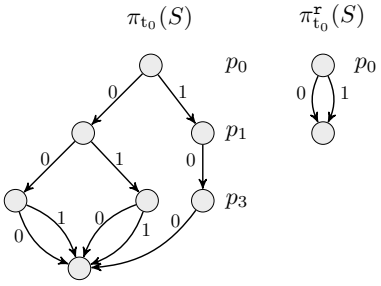

Fig. 4. Projection without and with read-separation for Ex. 2

efficiently at a lower (symbolic) level. *Thirdly*, the transition relation is applied using both the read and the may-write dependency matrices (line 11). That way, fewer levels of the underlying decision diagrams are affected.

Figure 4 clearly shows the difference between using the previously used projections (to the left) and using read-projections (to the right). Both can be used to compute successors for the states in Example 2, but when using read-projections, the function $\rightarrow_i$ is applied to only one of the four states with $p_0 = 0$, instead of to all.

4.1 Implementation

To investigate the effects of separating dependencies, we have implemented the transition relation and its application from Definition 12 in LTSMIN's native List Decision Diagram (LDD) library. An LDD is a form of Multi-way Decision Diagram MDD which was initially described in [2, Sect. 5]. The definition is as follows.

Definition 13 (List Decision Diagram). *A* List Decision Diagram *(LDD) is a* Directed A-cyclic Graph *(DAG) with three types of nodes:*

- $\{\epsilon\}$*, which encodes* true *and has no successors,*
- $\varnothing$*, which encodes* false *and has no successors,*
- $\langle v, down, right\rangle$*, a tuple with label* v *and two successors:* $down$ *and* $right$*.*

The semantics $[\![s]\!]$ *of an* LDD *node* s *is a set of vectors, as follows:*

$$[\![\{\epsilon\}]\!] = \{\epsilon\}\,, \quad [\![\varnothing]\!] = \varnothing, \quad [\![\langle v, down, right\rangle]\!] = \{vw \mid w \in [\![down]\!]\} \cup [\![right]\!].$$

Algorithm 5. LDD-NEXT

Input: LDD nodes $s, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}$ and level $l \in \mathbb{N}$

if $s = \varnothing \vee \hookrightarrow^{\mathrm{p}} = \varnothing$ **then return** $\varnothing$;
if $|\boldsymbol{r}| = 0 \wedge |\boldsymbol{w}| = 0$ **then return** s ;
if $s = \{\epsilon\} \vee \hookrightarrow^{\mathrm{p}} = \{\epsilon\}$ **then** ERROR ;
if $\boldsymbol{r}_0 = l \wedge \boldsymbol{w}_0 = l$ **then** // Read and write dependent
  if $\hookrightarrow^{\mathrm{p}}_{\mathrm{v}} < s_{\mathrm{v}}$ **then return** LDD-NEXT$(s, \hookrightarrow^{\mathrm{p}}_{\mathrm{r}}, \boldsymbol{r}, \boldsymbol{w}, l)$;
  else if $\hookrightarrow^{\mathrm{p}}_{\mathrm{v}} > s_{\mathrm{v}}$ **then return** LDD-NEXT$(s_{\mathrm{r}}, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}, l)$;
  else return LDD-WRITE$(s, \hookrightarrow^{\mathrm{p}}_{\mathrm{d}}, \boldsymbol{r}_{\mathrm{d}}, \boldsymbol{w}, l) \cup$ LDD-NEXT$(s_{\mathrm{r}}, \hookrightarrow^{\mathrm{p}}_{\mathrm{r}}, \boldsymbol{r}, \boldsymbol{w}, l)$;
else if $\boldsymbol{r}_0 = l$ **then** // Only read dependent
  if $\hookrightarrow^{\mathrm{p}}_{\mathrm{v}} < s_{\mathrm{v}}$ **then return** LDD-NEXT$(s, \hookrightarrow^{\mathrm{p}}_{\mathrm{r}}, \boldsymbol{r}, \boldsymbol{w}, l)$;
  else if $\hookrightarrow^{\mathrm{p}}_{\mathrm{v}} > s_{\mathrm{v}}$ **then return** LDD-NEXT$(s_{\mathrm{r}}, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}, l)$;
  else return $\langle s_{\mathrm{v}}$, LDD-NEXT$(s_{\mathrm{d}}, \hookrightarrow^{\mathrm{p}}_{\mathrm{d}}, \boldsymbol{r}_{\mathrm{d}}, \boldsymbol{w}, l+1)$, LDD-NEXT$(s_{\mathrm{r}}, \hookrightarrow^{\mathrm{p}}_{\mathrm{r}}, \boldsymbol{r}, \boldsymbol{w}, l)\rangle$;
else if $\boldsymbol{w}_0 = l$ **then** // Must-write or may-write dependent
  if $\hookrightarrow^{\mathrm{p}}_{\mathrm{v}} = \blacktriangle$ **then return** LDD-COPY$(s, \hookrightarrow^{\mathrm{p}}_{\mathrm{d}}, \boldsymbol{r}, \boldsymbol{w}_{\mathrm{d}}, l) \cup$ LDD-WRITE$(s, \hookrightarrow^{\mathrm{p}}_{\mathrm{r}}, \boldsymbol{r}, \boldsymbol{w}, l)$;
  else return LDD-WRITE$(s, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}, l) \cup$ LDD-WRITE$(s_{\mathrm{r}}, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}, l)$;
else return LDD-COPY$(s, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}, l)$; // Copy

Algorithm 6. LDD-WRITE

Input: LDD nodes $s, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}$ and $l \in \mathbb{N}$

if $\hookrightarrow^{\mathrm{p}} = \varnothing$ **then return** $\varnothing$;
return $\langle \hookrightarrow^{\mathrm{p}}_{\mathrm{v}}$,
  LDD-NEXT$(s_{\mathrm{d}}, \hookrightarrow^{\mathrm{p}}_{\mathrm{d}}, \boldsymbol{r}, \boldsymbol{w}_{\mathrm{d}}, l+1), \varnothing\rangle$
  $\cup$ LDD-WRITE$(s, \hookrightarrow^{\mathrm{p}}_{\mathrm{r}}, \boldsymbol{r}, \boldsymbol{w}, l)$

Algorithm 7. LDD-COPY

Input: LDD nodes $s, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}$ and $l \in \mathbb{N}$

if $s = \varnothing$ **then return** $\varnothing$;
return $\langle s_{\mathrm{v}}$,
  LDD-NEXT$(s_{\mathrm{d}}, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}, l+1)$,
  LDD-COPY$(s_{\mathrm{r}}, \hookrightarrow^{\mathrm{p}}, \boldsymbol{r}, \boldsymbol{w}, l)\rangle$

For some node $n = \langle v, down, right\rangle$, we use n_{v}, n_{d} and n_{r} to denote its elements v, down and right, respectively.

We assume (and enforce) in the implementation that the sequence of values in a level is ordered from small to large. E.g., $\langle 0, \ldots, \langle 1, \ldots, \varnothing\rangle\rangle$ is a valid node, but $\langle 1, \ldots, \langle 0, \ldots, \varnothing\rangle\rangle$ is not. We define $\blacktriangle$ to always be smallest.

A single vector $\boldsymbol{x} = \langle x_1, \ldots, x_N\rangle$ (or singleton set $\{\boldsymbol{x}\}$) can be represented as an LDD node as $\langle x_1, \langle x_2, \ldots, \varnothing\rangle, \varnothing\rangle$. Note that for vector $\boldsymbol{x}$, encoded as LDD node x, the LDD node of the subvector $\boldsymbol{x}_{1<j\leq|\boldsymbol{x}|}$, i.e., the vector $\boldsymbol{x}$ with the first element removed, equals x_{d}. An example LDD is given in Figure 5. This LDD encodes the set $\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\}$ $(= \{0,1\} \times \{0,1\})$ with two variables x_0 and x_1.

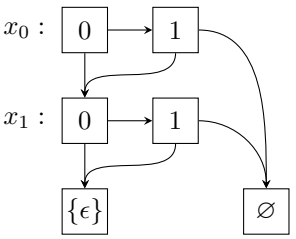

Fig. 5. $\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\}$ as LDD

In the implemention of the application of the transition relation NEXT in Def. 12, we use LDD's to encode the set $S_{\mathcal{P}}$, the relation $\hookrightarrow^{\mathrm{p}}$ and the matrix rows $\boldsymbol{r}$ and $\boldsymbol{w}$. Here, $S_{\mathcal{P}}$ is encoded as an LDD of depth N and $\hookrightarrow^{\mathrm{p}}$ as an LDD of depth $|\boldsymbol{r}| + |\boldsymbol{w}|$. The rows $\boldsymbol{r}$ and $\boldsymbol{w}$ are actually encoded as LDD representations of the sorted vectors with the indices of dependent variables $ind(\boldsymbol{r})$ and $ind(\boldsymbol{w})$, respectively. The algorithm using LDD's, given in Algorithm 5, recursively traverses the LDD's level by level, maintaining a counter l to keep track of the current level, initially 0.

Lines 1–3 handle a few base cases. In the case the current level l (variable x_l) is both *read and may-write dependent* (lines 4–7), first a (read) value is matched (s_v and $\hookrightarrow_v^{\mathrm{p}}$) and then each value from the next level of the relation is written using LDD-Write. The resulting node is united with all other values we may need to write. If the level is *read dependent* only (lines 8–11), then we first find a matching value and then create a new node with two recursive elements: downward matching the other levels, and, to the right, other nodes on the current level that may match the relation. If the level is *must-write or may-write dependent* only (lines 12–14), then for each value in the set we create a new node, where we either copy the value s_v or write the value from the relation. If the level has no read or write dependency (line 15), then a new node is created with the *down* and *right* nodes computed recursively with LDD-Copy. LDD-Write writes all values from the relation reachable on the current level. However, it needs to unite all the nodes with these values because they may occur in the wrong order. The unions are computed in the standard way for decision diagrams.

5 Results

To evaluate our work we have benchmarked with 266 Dve models from the Beem database [15], 38 mcrl2 models, mostly from mcrl2's distribution, and 60 Promela models from different sources[1]. To compare our results to both the current version of LTSmin and the effect of variable orderings we implemented the options `w2W` and `W2+`. These two options over-approximate must-write to may-write, and may-write to read and write, thus simulating the situation without read-write separation. Every experiment is run three times in both setups to determine the effect of our work. The machine we used has an Intel Xeon E5520 CPU, with 24 GB of memory. We have restricted the runtime of each experiment to 30 minutes.

Overall, we see that the mcrl2 models benefit from read-write separation, because of the reduced amount of Next-state calls. This is due to the fact that a Next-state call for mcrl2 is rather time consuming because of the term rewriting involved. The Dve and Promela front-ends run optimized C code. For these languages, the overhead of many unnecessary Next-state calls in the current version of LTSmin is less noticeable. We see however that the runtime of Dve models is improved when we use a good variable ordering, which reduces the number of symbolic operations. We have highlighted six interesting experiments with relevant information in Table 4, of which a legend can be found in Table 3. Of all experiments which have a run time longer than one second, 101 out of 167 are faster. With optimized dependency matrices, 125 out of 160 experiments are faster.

With Dve models we see speedups mainly when the amount of symbolic (LDD) operations is reduced, such as in `telephony.7`. We were less successful in this for `anderson.6`. However, the runtime for `blocks.3` is greatly reduced.

[1] Instructions to reproduce or obtain a copy of all models/results can be found at `http://pm.jmeijer.nl/32ae74f74e`

Table 3. Symbols used in Table 4

model	name of the model	**cs**	Column Sort, sorts columns such that writes are put on a diagonal
dm	dependency matrix operations		
$\overline{rt}$	average reachability time in seconds		
$\overline{mem}$	average peak memory usage in kilobytes	**rs**	Row Sort, sorts rows such that writes are put on a diagonal
#NS	number of NEXT-STATE calls		
#LDD	number of calls to LDD-NEXT (Alg. 5)	**cw**	Column sWap, minimizes distance between columns and puts reads before writes heuristically (Algs. 1 and 2)
$\lvert\mathcal{R}\rvert$	number of nodes of the set of reachable states		
$\lvert\pi(\mathcal{R})\rvert$	approx. number of nodes in the projections		
$\lvert\hookrightarrow\rvert$	approx. number of nodes of the whole transition relation		

The `anderson.6` model has 18,206,917 states, 36 groups and 19 state slots. In this model we see no speedup, because it is hard to find a good variable ordering. The bad ordering results in more recursive LDD-NEXT calls which slows down the reachability analysis. `blocks.3` is a model where we obtained very good results. The state space of this contains 695,418 states, there are 26 groups and 18 state slots. Because `blocks.3` contains many may-write dependencies we are able to greatly reduce the amount of NEXT-STATE calls. Furthermore the amount of nodes in the node table is reduced significantly. Telephony.7.dve is a model with 21,960,308 states, 24 state slots and 120 groups. Similar to `anderson.6` we see a slow down when we use separated dependencies. This slow down is the result of many more symbolic operations. However, opposed to `anderson.6` we are able to slightly speed up the reachability analysis by transforming the dependency matrix. We can reduce the amount of NEXT-STATE calls while only slightly increasing the amount of recursive LDD-NEXT calls.

In the first two MCRL2 models (`1394-fin` and `lift3-final`) we can see that the amount of reduced NEXT-STATE calls corresponds closely to the speedup attained. The model `1394-fin` has 188,596 states, 34 state slots and 1,069 transition groups. The second, `lift3-final`, has 4,312 states, 30 state slots and 60 transition groups. We obtained the most interesting result with the `vasy` model, a 1-safe Petri net submitted to the Petri net mailing list in 2003 [12] by Hubert Garavel. The model has 9.79×10^{21} states, 776 transition groups and 485 state slots. With our work we have managed to make reachability analysis for this model tractable for LTSMIN. Special about this model is the first transition, which removes the token from the initial place to several other places (like in Figure 2). Without read-write separations, this required exponentially many NEXT-STATE calls for this transition: $\leq 2^{61}$ calls, because there are 61 dependent state slots of boolean type. With our improvements it is identified that only one state slot is read, resulting in only 2^1 NEXT-STATE calls.

Table 4. Highlighted experiment results

model	dm	$\overline{\text{rt}}$	$\overline{\text{mem}}$	#NS	#LDD	$\lvert\mathcal{R}\rvert$	$\lvert\pi(\mathcal{R})\rvert$	$\lvert\hookrightarrow\rvert$
anderson.6.dve		25.4	439,076	7,464	64,034,383	50,120	2,442	2,064
anderson.6.dve		34.6	439,076	4,080	127,725,604	50,120	1,470	1,386
anderson.6.dve	cs;rs;cw;rs	27.6	144,216	7,464	84,028,747	41,079	2,568	1,884
anderson.6.dve	cs;rs;cw;rs	29.9	144,216	4,080	109,711,771	41,079	1,533	1,386
blocks.3.dve		31.0	239,293	6,559,927	69,695,086	39,522	375,603	269,996
blocks.3.dve		10.9	144,064	262,543	62,467,909	39,522	12,314	1,604
blocks.3.dve	cs;rs;cw;rs	25.7	280,344	6,559,927	25,021,658	49,685	464,916	325,763
blocks.3.dve	cs;rs;cw;rs	4.6	144,196	262,543	12,281,723	49,685	12,076	1,478
telephony.7.dve		107.6	1,111,840	918,817	231,808,995	284,449	36,951	6,038
telephony.7.dve		123.7	696,188	730,841	393,099,843	284,449	31,473	5,337
telephony.7.dve	cs;rs;cw;rs	26.8	144,656	918,817	62,889,960	18,479	39,410	6,263
telephony.7.dve	cs;rs;cw;rs	25.4	144,656	730,841	63,110,689	18,479	33,144	5,478
1394-fin.mcrl2		22.6	208,084	3,372,554	1,995,202	7,384	870,142	12,505
1394-fin.mcrl2		3.4	188,944	443,813	1,800,912	7,384	229,251	8,399
lift3-final.mcrl2		5.3	184,624	190,347	313,868	5,452	162,956	7,023
lift3-final.mcrl2		2.5	181,372	79,941	378,179	5,452	54,249	5,496
vasy.mcrl2		-	-	-	-	-	-	-
vasy.mcrl2		152.6	1,149,592	2,694	241,432,226	9,387	4,340	5,444

6 Conclusion

Separating dependencies into read and write dependencies can speed-up symbolic model checking considerably. To do so, we had to solve two key problems. The first problem is that a *copy* dependency can in general not be over-approximated to a must-write dependency. Therefore we introduced the definition of may-write independence. This notion is used when it can not be statically determined whether a value needs to be written or copied. Separating dependencies introduced a second problem. Reachability algorithms that exploit our notions for read and write dependencies only work well with a good variable ordering. We have provided heuristics that try to put read dependencies before write dependencies.

Models for the Promela and Dve language front-ends for Pins are both highly optimized C programs. Thus a Next-state call is relatively fast compared to symbolic operations of the back-end. On the contrary, computing the state space of mcrl2 models involves the term rewriter of mcrl2. The increased expressiveness has a prize: term rewriting is a lot slower than the optimized C programs for Promela and Dve. So symbolic operations are relatively fast compared to a Next-state call to the mcrl2 language front-end.

Overall, we conclude that separating dependencies in the transition relation by default in LTSmin is a good idea. We have observed only a few cases with a slow-down, and this slow-down was minimal. The observed speed-ups on the other hand were considerable, and in some cases necessary to make problems tractable for LTSmin, e.g., for the `vasy` model.

Future work will split conditions into single guards, and consider their dependencies separately. Also, the distinction between read and write variables can be included in more advanced heuristics for static variable ordering strategies in the dependency matrix. We also recommend to implement our new definitions and algorithms for other modeling languages and connect them to LTSmin through Pins.

References

1. van der Berg, F.I., Laarman, A.W.: SpinS: Extending LTSmin with Promela through SpinJa. ENTCS 296(2012), 95–105 (2013); pASM/PDMC 2012
2. Blom, S., van de Pol, J.: Symbolic Reachability for Process Algebras with Recursive Data Types. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 81–95. Springer, Heidelberg (2008)
3. Blom, S., van de Pol, J., Weber, M.: Bridging the Gap between Enumerative and Symbolic Model Checkers. Technical Report CTIT, University of Twente, Enschede (2009), http://eprints.eemcs.utwente.nl/15703/
4. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and Symbolic Reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
5. Burch, J.R., Clarke, E.M., Long, D.E.: Symbolic model checking with partitioned transition relations. In: VLSI 1991 (1991)
6. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS 1990. IEEE (1990)
7. Ciardo, G., Marmorstein, R., Siminiceanu, R.I.: Saturation unbound. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 379–393. Springer, Heidelberg (2003)
8. Ciardo, G., Yu, A.J.: Saturation-Based Symbolic Reachability Analysis Using Conjunctive and Disjunctive Partitioning. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 146–161. Springer, Heidelberg (2005)
9. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
10. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. STTT 2(4) (2000)
11. Clarke, E.M.: The birth of model checking. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 1–26. Springer, Heidelberg (2008)
12. Kordon, F., Linard, A., Beccuti, M., Buchs, D., Fronc, L., Hillah, L.M., Hulin-Hubard, F., Legond-Aubry, F., Lohmann, N., Marechal, A.: et al.: Model Checking Contest @ Petri Nets, Report on the 2013 edition (2013), ArXiv: http://arxiv.org/abs/1309.2485v1
13. McMillan, K.L.: Symbolic model checking. Kluwer (1993)
14. Meijer, J.J.G.: Improving Reachability Analysis in LTSmin. Master's thesis, University of Twente (2014)
15. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
16. Rudell, R.: Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In: ICCAD 1993. IEEE (1993)
17. Skiena, S.S.: The Algorithm Design Manual. Springer (2008)