UNIVERSITY OF TWENTE.

Master Thesis

Improving Reachability Analysis in LTSmin

Guards, Read, Write and Copy Dependencies for mCRL2, Promela and Dve

Author:
Jeroen Meijer

Committee:
Prof. Dr. Jaco van de Pol
Dr. Stefan Blom
Gijs Kant, MSc.

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Formal Methods and Tools group,
department of Computer Science,
faculty of Electrical Engineering, Mathematics and Computer Science.

March 27, 2014

Abstract

To improve symbolic reachability analysis in the model checking toolset LTSmin, we present two improvements to existing reachability algorithms. LTSmin uses a disjunctive partitioning scheme to efficiently analyze models of asynchronous systems. In these models transitions can be partitioned into groups, which modify only a small part of the state vector. Currently, there are no well defined notions, for whether a transition group reads and/or writes to an element in the state vector, which can be used in symbolic algorithms. Therefore, we present new definitions for read and write dependencies and show how algorithms can exploit these. This improvement always results in faster state space generation and many models such as 1-safe Petri nets highly benefit of these changes. A major issue we solved to separate dependencies correctly is that we had to cope with copying values. We provide examples that were intractable for LTSmin before, but can now be computed in a matter of minutes.

A transition in a model specification is often of the form "if condition(x1,...,xn) $\implies$ A . X(x1',...,xn')". Our second improvement divides the condition into multiple conjuncts. These conjuncts can then be evaluated separately. Symbolic algorithms can exploit this information and prevent computing successors for large sets of states for only one conjunct evaluation. This greatly speeds up state space generation for models such as Sokoban or dining philosophers. We provide examples that show a speedup ranging from twice as fast to hundreds of times faster. An important issue in algorithms that exploit guard-splitting is that they need to support a ternary logic for the evaluation of guards. This is due to the fact that a guard can not only evaluate to true or false, but also to 'maybe'. Consider for example a term rewriting system that splits the term `someterm` $\wedge$ `false` and rewrites both conjuncts individually.

We present clear benchmarking results in the form of scatter plots with time and memory usage as well as tables with detailed information about the size of the data structures. We make some important assumptions such as on the completeness of model specifications. We use these assumptions and the benchmarking results to validate our work.

Contents

Introduction

Testing software and hardware for correctness is traditionally a hard problem. To meet this problem formal methods, such as model checking have been developed. Model checking suffers from the state space explosion which is caused by the fact that modeling many parallel processes results in an exponential increase in the amount of states. To cope with this problem one approach is to store states symbolically. For some modeling languages such as mCRL2 improvements to symbolic reachability algorithms can be made, such as analyzing read and write dependencies between transitions and state variables. Another improvement we make is to split the guards (or enabledness) from the transition relation.

One model checking tool set the FMT chair at the University of Twente has chosen to develop is LTSmin [3, 6]. This tool incorporates much research with regard to this topic, developed either in house or by other research organizations such as the Technische Universiteit Eindhoven.

A key component in LTSmin is the Partitioned Interface to the Next State (PINS) interface. This interface allows for generic model checking algorithms such as symbolic, distributed or multi-core for modeling languages such as mCRL2, Promela, DiVinE, Uppaal and other. This is separated into a back-end for the algorithms and a front-end for the languages. The Pins interface partitions the state vector into individual state slots as well as the transition relation into multiple transition groups. With this partitioning LTSmin can exploit the notion of event locality by letting every transition group modify only a small part of the state vector.

We contribute two improvements to existing reachability algorithms. The first involves precise definitions for dependencies between transition groups and state slots and algorithms which benefit hereof. These precise definitions allow us to create smaller decision diagrams and less computations for successor states. This first improvement speeds up symbolic reachability analysis for models such as 1-safe Petri nets. The second improvement is a guard-splitting algorithm for symbolic reachability analysis. Guard-splitting can improve the computation time for models of Sokoban games. In Sokoban games computing whether the game is finished is expensive. With guard-splitting we can efficiently prevent computing successor states for states that which do not satisfy the condition of a transition. We do this by implementing a join (like in relational databases) operation on symbolic sets.

One major issue in separating dependencies is writing to an index of an array. To this end we will give two notions of whether a transition group writes to a specific state slot. The first notion is may-write-independent. Like the name says this notion handles cases in which it is uncertain whether a value is purely written or copied. Suppose we have an array a of length 2 and we partition this array into two state slots a_0 and a_1. Also suppose we have an index i to write to a position in array a. If we have a statement $a[i] = 1$ then we do not know if we either write to a_0 and copy the value for a_1 or copy the value from a_0 and write to a_1. Our algorithms together with the notion of may-write-independent can handle this case. The second notion for writing to a state slot is must-write-dependent. This notion covers simple cases such as $b = 1$ where b is simply an integer. The last important definition we provide in the chapter about separating dependencies is read-independent. With this definition we can identify whether a transition group writes to a state slot or not. With the notions for reading and writing to state slots we can improve the projections used by symbolic algorithms to project to fewer short vectors and thus reduce the amount computations for successor states. Furthermore we can reduce the size of the transition relation, by removing read nodes if a state slot is not read and write nodes if a state slot is not written. Removing nodes from the transition relation is an important improvement, because it is a way of coping with bad variable orderings. Our results for separated dependencies are not only shown

in clear scatter plots with time and memory usage but also in tables with detailed information about for example the size of the transition relation. The scatter plots can be found in Sections 3.5 and 4.8. The tables with detailed information can be found in Appendix C. We have run experiments with three well known modeling languages; mCRL2, PROMELA and DVE.

The major issue in guard-splitting is the need for a ternary logic. Guard-splitting splits the condition of a transition group into multiple conjuncts (guards) so that an algorithm can evaluate guards for every state individually. This however introduces a problem for term rewriting systems such as mCRL2 and lazy evaluation in imperative programming languages such as Java. In mCRL2 we can have a condition of a transition group represented by the term `someterm` $\wedge$ `false`. Naturally, if we individually evaluate the guard `someterm` then this does term does not rewrite to true or false. We say that such a term rewrites to maybe. Algorithms that exploit guard-splitting can for efficiency reasons evaluate guards not in the same order as they are expressed in a condition. In a Java statement such as '`p != null && p.method()`' this is a problem when the pointer `p` is `null`. Because in that case `p.method()` neither evaluates to true nor false. To cope with this ternary logic we developed algorithms that check if a model specification is complete. That is, make sure a model specification does not have a term such `someterm` $\wedge$ `true` in the condition of a transition group. When our algorithms exploiting guard-splitting have checked that a model specification is correct symbolic set operations such as the join operation can be used to efficiently compute successor states. In Chapter 4 about guard-splitting we also show clear scatter plots for time and memory usage. Additionally, we show tables which indicating that removing the condition from the transition relation indeed produces smaller transition relations.

Our changes also affect other functionality in LTSMIN. For example we discuss how operations as row subsumption on the dependency matrix can be improved. Furthermore we will discuss how an advanced reachability algorithm such as saturation can be improved with our notions of separated dependencies and guard-splitting.

Acknowledgements

First of all I want to thank my committee. *Jaco van de Pol* for his excellent guidance; I feel I learned a great many things while working on my thesis from you. Second, I want to thank *Stefan Blom* for always making time to answer my questions. I really appreciate the time and patience you took to explain things to me. From you I learned there are actually a great many things I do not know yet. From my committee I would also like to thank *Gijs Kant* you were always available when I needed your help. Seeing you work on you PhD thesis I decided that I would also like to do a PhD too. *Alfons Laarman*, other than the fact that you are not on the front page of my thesis, I feel like you were part of my committee all the same. Without your work and time (update matrix for PROMELA and other guidance) my results would not have been as impressive as they are now.

List of Contributions

Along with this report we have contributed patches to several projects. We contributed to LTSMIN, DIVINE and mCRL2. For analysing the results of our contributions we provide some scripts and URLs to these results.

General fixes

LTSMIN (`git@github.com:Meijuh/ltsmin.git` next):

`8af5ae6e` a change to fix caching in the `mdd_next` operation.

DIVINE (`git@github.com:Meijuh/Divine2.git` master):

`1d42067c` a change to fix a bug in the dependency matrix were a read was incorrectly marked as a write.

Changes for separated dependencies

LTSmin (`git@github.com:Meijuh/ltsmin.git` rw):

`f4f24e32` add support for separated dependencies to LTSmin.

DiVinE (`git@github.com:Meijuh/Divine2.git` rw):

`eec89c58` add `-W` option to the DiVinE compiler to over-approximate `W` to `+`.

Changes for guard-splitting

LTSmin (`git@github.com:Meijuh/ltsmin.git` guard):

`37bf6c05` support for the update matrix for DiVinE.

`f91d722a` support for guards for mCRL2.

`a53bb4b3` symbolic reachability algorithms for LTSmin, including a fix for the state label matrix.

DiVinE (`git@github.com:Meijuh/Divine2.git` guard):

`8edcecd2` partially working update matrix.

A patch for guard-splitting for mCRL2 (version 2012-10): `https://gist.github.com/Meijuh/9617728`.

Scripts for benchmarking

Running experiments `https://gist.github.com/Meijuh/9712862` — shell script to run experiments. Place models in a folder and this script will run multiple reachability algorithms on these models.

Analyse experiments `https://gist.github.com/Meijuh/9712843` — PHP script to analyze the standard out output from the experiments. Creates CSV files and LaTeX code.

Scatter plots `https://gist.github.com/Meijuh/9712867` — creates scatter plots for time and memory usage from the generated CSV files.

Open online access to our benchmarking results

Zip file of our results `https://drive.google.com/file/d/0B98nrOHB7d4Ab1UxZFNpSGJDcjg/edit?usp=sharing`

Chapter 1

Current Situation

LTSmin is a model checking tool set developed at the Formal Methods and Tools group at the University of Twente. Over the last several years much research on model checking is incorporated into this software package, making LTSmin a high performance model checker. In this chapter we will discuss the current state of LTSmin. This background information allows us to accurately describe two improvements we make to the tool set in this master thesis. First we will discuss the traditional notion of transition systems. Then we will describe how language front-ends, which allow specifying these transition systems are connected to LTSmin. The last part of this chapter shows how we can efficiently perform reachability analysis on these transition systems.

In reachability analysis simply calculating the next state for each state in the entire state space is not efficient because transitions may only affect a small part of the state space. This is also referred to as *event locality* [2]. Therefore instead of a simple *monolithic* interface to the next-state function LTSmin exploits the notion of event locality by partitioning the next-state function. Of particular interest in the LTSmin tool set is the Pins interface, illustrated in Figure 1.1.

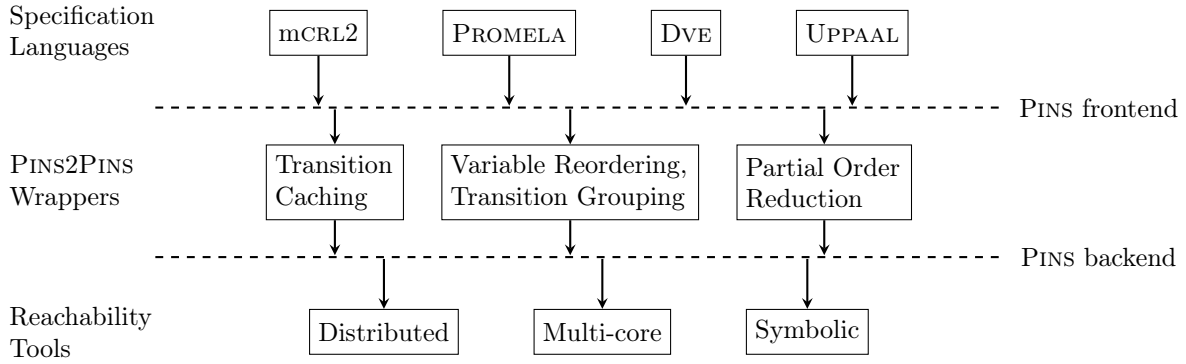


Figure 1.1: Modular Pins architecture of LTSmin

The Pins interface allows for a partioned interface to the next-state function. Key components of the Pins interface are the front-end, the back-end and the Pins2Pins wrappers. The front-end allows one to implement the next-state function for modeling languages such as Promela, mCRL2 and many more. The Pins2Pins wrappers allow for *caching* transition group vectors, *reordering* of variables and transitions and *partial order reduction*. The back-end allows for storage in main memory and reachability algorithms of the state space. Currently there exist three ways for storing the space space and performing reachability, namely *distributed*, *multi-core* and *symbolic*. Our research focuses on improving symbolic reachability algorithms for all modeling languages and enabling partial order reduction for mCRL2.

1.1 The Monolithic Next-State Interface

The monolithic next-state interface is not very different from the conventional notation for transition systems and its transition relation.

Definition 1.1 (Transition System)**.** A Transition System (TS) is a structure $\langle S, \rightarrow, s^0 \rangle$, where S is a set of states, $\rightarrow \subseteq S \times S$ is a transition relation and s^0 is the initial state. ■

In order to understand our notion for TSs consider the example 1-safe Petri net in Figure 1.2.

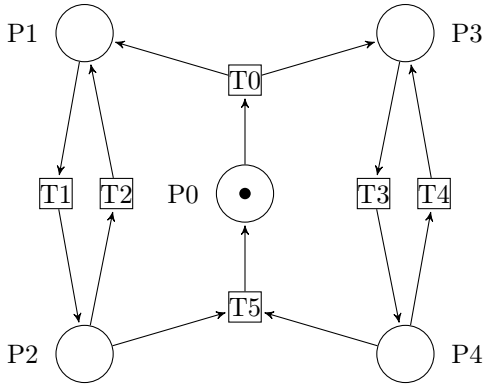


Figure 1.2: Example 1-safe Petri net

In this Petri net, if transition T0 fires then the token from place P0 moves to both place P1 and P3. Firing transitions (T1..T4) move the token between P1 and P2 or P3 and P4. Firing transition T5 can be done when there is a token in both P2 and P4. T5 removes the tokens from P2 and P4 and places one in P0. Figure 1.3 on Page 10 illustrates the state space of the Petri net.

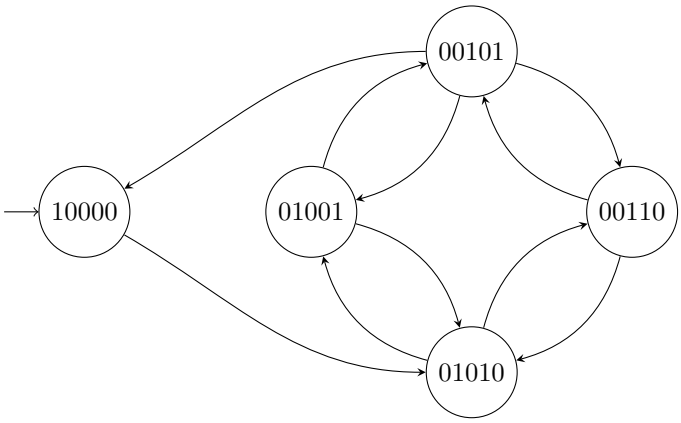


Figure 1.3: State space of the Petri net in Figure 1.2

Example 1.2 (transition system)**.** A TS for Figure 1.2 may be denoted as

$$TS = \langle \{10000, 01010, 00110, 01001, 00101\}, \{(10000, 01010), \ldots\}, 10000 \rangle$$

, with $|S| = 5, |\rightarrow| = 10$. Note that a string such as 10000 encodes the initial state with a token at place P0. ▲

In a monolithic interface, for a TS the initial state may be obtained with the function INITIAL-STATE() $= s^0$. In the monolithic next-state interface successor states can be obtained with the NEXT-STATE (s) function for a TS where s is defined as $s \in S$, NEXT-STATE$(s) = \{s' \mid s \rightarrow s'\}$.

The monolithic next-state interface allows explicit state algorithms to be implemented for any language, but it has very limited use in combination with symbolic techniques due to the fact that we need to call the next-state function for every state.

1.1.1 Explicit Reachability for Transition Systems

We first give a set of definitions and an algorithm for explicit reachability in order to provide the necessary background information on symbolic reachability analysis for Partitioned Transition System (PTS)s.

Definition 1.3 (Reachable states [2])**.** Given a transition system $ts = \langle S, \rightarrow, s^0 \rangle$. The set of reachable states is

$$R = \{s \in S \mid s^0 \rightarrow^* s\}$$

a state $s \in S$ is reachable if $s \in R$. The relation $\rightarrow^*$ is recursively defined as

$$\begin{cases} s \rightarrow^* s & \text{(symmetric closure)} \\ s \rightarrow^* t \wedge t \rightarrow t' \implies s \rightarrow^* t' & \text{(transitive closure)} \end{cases}$$

The set of reachable states for a PTS is given by the set of reachable states of its TS. ■

The set of reachable states is usually built using a Breadth First Search (BFS) strategy. Using BFS we define a level l, denoted L_l such that the set of states with a distance l to s^0 are in L_l. States with a distance *less* or *equal* to l are in R_l.

Definition 1.4 (Explicit Reachability [2])**.** Given a transition system $ts = \langle S, \rightarrow, s^0 \rangle$. Let

$$L_0 = \{s^0\} \qquad R_0 = \{s^0\}$$
$$L_{l+1} = \{s' \in S \mid \exists s \in L_l : s \rightarrow s' \wedge s' \notin R_l\} \qquad R_{l+1} = R_l \cup L_{l+1}$$

then

$$R = \bigcup_{l=0}^{\infty} R_l.$$

■

An algorithm to calculate the set R is presented in Algorithm 1. The algorithm calculates for each level the next states until the set of states does not grow anymore, i.e. a greatest fixed point is reached. Note however that sometimes state spaces may be infinite and a greatest fixed point is never reached and thus the algorithm never terminates.

Algorithm 1: Reachability for TSs [2]

```
   Data: →, s^0
   Result: R
1  R ← {s^0};
2  L ← R;
3  while L ≠ ∅ do
4  |  L ← {y | ∃x ∈ L : x → y};
5  |  L ← L \ R;
6  |  R ← R ∪ L;
7  end
```

1.1.2 Symbolic Reachability for Transition Systems

The advantage of using symbolic techniques to represent the state space is that the memory footprint is very small. Thus instead of storing each element of the transition relation and state space explicitly, we can somehow represent them in a boolean expression. Given a $TS = \langle S, \rightarrow, s^0 \rangle$ we can represent a set $S' \subseteq S$ by a boolean expression $\mathcal{S}'(\boldsymbol{x})$ such that

$$\boldsymbol{x} \in S' \Leftrightarrow \mathcal{S}'(\boldsymbol{x})$$

where the expression is stored as a decision diagram and $\boldsymbol{x}$ stands for the vector $x_1, \ldots, x_N$, i.e. we assume the state space a cartesian product ($S = S_1 \times \ldots \times S_N$). The transition relation is stored as a boolean expression $\hookrightarrow(\boldsymbol{x}, \boldsymbol{x'})$, such that

$$\boldsymbol{x} \rightarrow \boldsymbol{x'} \Leftrightarrow \hookrightarrow(\boldsymbol{x}, \boldsymbol{x'}).$$

Given a level as a formula $\mathcal{L}(\boldsymbol{x})$, we can compute the next level using the expression

$$(\exists \boldsymbol{x}.(\mathcal{L}(\boldsymbol{x}) \wedge \hookrightarrow(\boldsymbol{x}, \boldsymbol{x'})))[\boldsymbol{x'} := \boldsymbol{x}].$$

Which provides us the symbolic implementation of Line 4 of Algorithm 1. However, using this framework we can not yet exploit the notion of *event locality*.

1.2 The Pins front-end

The Pins front-end allows one to implement a modeling language such as the process algebraic language mCRL2. Other languages such as the state-based specification languages Promela and DiVinE are also supported.

The mCRL2 tool set is an implemented front-end for the Pins interface. The concept of the tool set is to take a process algebra specification and compile it into a Linear Process System (LPS). An LPS is a single recursive process.

Definition 1.5 (Linear Process System [1])**.**

$$X(x_1, \ldots, x_N) = \sum_{i=1}^{K} \underbrace{\sum_{e_i \in E_i} C_i \implies a(t_{i,0}).X(t_{i,1}, \ldots, t_{i,N})}_{\text{summand } i}$$

Where C_i and $t_{i,j}$ are expressions over $e_i, x_1, \ldots, x_N$. The intended meaning of this equation is that to perform a step we first non-deterministically select $1 \leq i \leq K$ (determining a summand), then non-deterministically select some $e_i \in E_i$, evaluate the condition C_i to see if the transition is enabled and if it is enabled then the label of the transition is the result of the expression $a(t_{i,0})$ and the next state is $t_{i,1}, \ldots, t_{i,N}$. ■

1.3 The Partitioned Next-State Interface

In symbolic techniques *event locality* can be exploited. Consider a TS consisting of several processes that communicate using shared variables. While the behaviour of the entire TS depends on the whole state, the behaviour of a single process depends only on its local variables and the relevant shared variables. To exploit this the TS can be partitioned into groups.

Definition 1.6 (Partitioned Transition System [7])**.** A PTS 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 $(1 \leq i \leq K)$ defines the transition

relation $\rightarrow_{\mathcal{P}} = \bigcup_{i=1}^{K} \rightarrow_i$. The initial state is $s^0 := \langle s_1^0, \ldots, s_N^0 \rangle \in S_{\mathcal{P}}$. We write $s \rightarrow_i t$ when $(s,t) \in \rightarrow_i$ for some $1 \leq i \leq K$. Also we write $s \rightarrow_{\mathcal{P}} t$ when $(s,t) \in \rightarrow_{\mathcal{P}}$ and for single states we may also write the vector $(\boldsymbol{x})$ or the explicit vector $(\langle s_1, \ldots, s_N \rangle)$. The defined TS of $\mathcal{P}$ is $\langle S_{\mathcal{P}}, \rightarrow_{\mathcal{P}}, s^0 \rangle$.

■

A natural way to partition the transition groups is introducing a group for each transition. This is for example done in the following Figure.

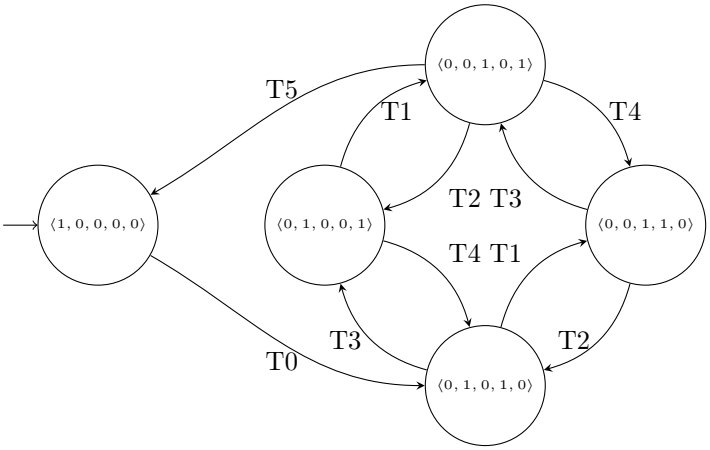


Figure 1.4: Example partitioning of the Petri net in Figure 1.2

Example 1.7. For the 1-safe Petri net in Figure 1.2 on Page 10 the PTS $\mathcal{P}$ for this model is

$$\langle \{\langle 1,0,0,0,0\rangle, \langle 0,1,0,1,0\rangle, \langle 0,0,1,1,0\rangle, \langle 0,1,0,0,1\rangle, \langle 0,0,1,0,1\rangle\},$$
$$\{(\langle 1,0,0,0,0\rangle, \langle 0,1,0,1,0\rangle), \ldots\}, \langle 1,0,0,0,0\rangle\rangle,$$
with $N = 5$ and $K = 6$.

▲

1.3.1 State slot dependencies

In a PTS a transition group i is independent of slot j if none of the transitions in the transition group can change the value of slot j and any transition in the group is enabled or disabled, regardless of the value of slot j. Formally, this can be stated as follows.

Definition 1.8 (indepedent [1])**.** Given a PTS $\mathcal{P} = \langle S_{\mathcal{P}}, \rightarrow_{\mathcal{P}}, s^0 \rangle$ transition group i is *independent* on state slot j if for all $\langle s_1, \ldots, s_N \rangle, \langle t_1, \ldots, t_N \rangle \in S_{\mathcal{P}}$, whenever $\langle s_1, \ldots, s_j, \ldots, s_N \rangle \rightarrow_i \langle t_1, \ldots, t_j, \ldots, t_N \rangle \in \rightarrow_{\mathcal{P}}$, then

1. $s_j = t_j$ (i.e., state slot j is not modified in transition i)
2. for all $r_j \in S_j$, we also have $\langle s_1, \ldots, r_j, \ldots, s_N \rangle \rightarrow_i \langle t_1, \ldots, r_j, \ldots, t_N \rangle$. (I.e., the value of state slot j is not relevant in transition group i.)

Whether some transition group i and state slot j is *dependent* may be over approximated as shown in Example 1.11. ■

Now, a formal definition of the Dependency Matrix (DM) is as follows.

Definition 1.9 (dependency matrix)**.** A *dependency matrix* $DM_{K \times N}$ for PTS $\mathcal{P}$ is a matrix with K rows and N columns containing $\{0,1\}$ such that if $DM_{i,j} = 0$ then transition group i is *independent* on state slot j.

For any transition group $1 \leq i \leq K$ we define π_i as the projection $\pi_i : S \rightarrow \Pi_{\{1 \leq j \leq N | DM_{i,j}=1\}} S_j$. ■

To be able to perform reachability analysis on our Petri net we need to translate the Petri net to for example mCRL2. Listing 1.1 contains the Petri net specified in the mCRL2 modeling language.

Listing 1.1: Petri net in mCRL2

```
act T0, T1, T2, T3, T4, T5;
init X(true, false, false, false, false);
proc X(P0, P1, P2, P3, P4: Bool) =
    P0 ⟹
        T0. X(P0=false, P1=true, P3=true) +
    P1 ⟹
        T1. X(P1=false, P2=true ) +
    P2 ⟹
        T2. X(P1=true, P2=false) +
    P3 ⟹
        T3. X(P3=false, P4=true ) +
    P4 ⟹
        T4. X(P3=true, P4=false) +
    (P2 && P4) ⟹
        T5. X(P0=true, P2=false, P4=false);
```

1.3.2 A Dependency Matrix for mCRL2

An LPS admits a natural partitioning by assigning each summand its own group. Using this partitioning we define the contents of the PINS DM for LPS X $DM(X) = DM^X_{K \times N}$ as follows.

$$DM^X_{i,j} = \begin{cases} 1 & \text{if } t_{i,j} \neq x_j \vee \exists 0 \leq k \leq N, k \neq j : x_j \text{ occurs in } C_i \text{ or } t_{i,k} \\ 0 & \text{otherwise.} \end{cases}$$

We can now analyze the dependencies for the specification in Listing 1.1.

Example 1.10 (Dependency Matrix)**.** The DM for the PTS of Listing 1.1 on Page 14 is:

$$\begin{array}{c} \\ T0 \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{array} \begin{array}{c} \begin{array}{ccccc} P0 & P1 & P2 & P3 & P4 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \end{array}.$$

Using this DM, reachability tools can use the fact that transition group T1 does not depend on all state slots, e.g.

$$\begin{aligned} \pi_{T1}(\{\langle 1,0,0,0,0\rangle, \langle 0,1,0,1,0\rangle, \langle 0,0,1,1,0\rangle, \\ \langle 0,1,0,0,1\rangle, \langle 0,0,1,0,1\rangle\}) = \\ \{\langle 0,0\rangle, \langle 1,0\rangle, \langle 0,1\rangle\}. \end{aligned}$$

▲

Note that we do not define the elements of the matrix in terms of ones (1), because these values may be approximated. I.e. if an element in a row contains a 1 it may actually be independent, but this can not be determined in general with static analysis.

Example 1.11 (dependency approximation)**.** In the simple recursive process

$X(x, y) :=$ if $x \vee y$ then a.$X(y \leftarrow false)$ else b.$X(y \leftarrow true)$, with initial state $X(false, true)$

static analysis – like in LTSMIN – will notice that x is not independent. However good heuristics may notice that the truth value of the condition $(x \vee y)$ does not depend on variable x, because the variable x is always assigned $false$. Heuristics such as these are currently not built into LTSMIN. ▲

Using the projection function π_i we can redefine the *next-state* function for a PTS. That is we extend the definition of the function to get the next states for only the a particular transition group i.

Definition 1.12 (Partitioned Next-State function)**.**

$$\text{NEXT-STATE}_i(s \in \pi_i(S_{\mathcal{P}})) = \{s' \in \pi_i(S_{\mathcal{P}}) \mid s \rightarrow_{\mathrm{i}} s'\}$$

■

1.4 The PINS back-end

The PINS back-end provide algorithms for state space exploration, including capabilities for checking properties. The back-end provides two main ways of performing reachability. The first is *symbolic* and the second is *explicit*. The latter can be performed *distributed*, *multi-core* or *sequential*. Currently symbolic reachability can only be performed single-core, but attempts are currently being made to parallelize symbolic algorithms [12]. In symbolic model-checking both the transition relation and the state space is stored in some form of decision diagram. Our research focuses on improving symbolic model-checking.

1.4.1 Symbolic reachability for Partitioned Transition Systems

The symbolic PINS back-end uses by default an List Decision Diagram (LDD) as an implementation for a Multi-way Decision Diagram (MDD) to store sets of states and transition relations. It builds a symbolic transition relation for each transition group and the set of reachable states in parallel. Transition relations can not be built in advance, because the size of the domain of a state slot is often infinite. The transition relation is instead extended at each level.

The transition relations are built by calling the NEXT-STATE function for every unique combination of relevant state slots and adding the transition with the identity relation for all irrelevant state slots implicitly.

In terms of symbolic algorithms, we can partition the transition relation according to the transition groups of the DM. That is we partition the transition relation into a disjunction – over separate groups of conjunctions (collections of local transitions) as follows

$$\hookrightarrow(\boldsymbol{x}, \boldsymbol{x'}) = \bigvee_{i=1}^{K} \mathcal{D}_i(\boldsymbol{x}, \boldsymbol{x'}) = \bigvee_{i=1}^{K} (\hookrightarrow_i(\pi_i(\boldsymbol{x}), \pi_i(\boldsymbol{x'})) \wedge \bigwedge_{\{1 \leq j \leq N \mid D_{i,j}=0\}} [x_j = x'_j]).$$

Heuristically conjunctively partitioned transition relations have been shown effective for synchronous systems while disjunctively partitioned transition relations have been shown effective for asynchronous systems [4].

For symbolic vector set operations we need to define a one step operation for a sub-vector.

$$\text{STEP}(\mathcal{S}(\boldsymbol{x}), \hookrightarrow((x_i)_{X_i=1}, (x'_i)_{X_i=1}), X) = (\exists(x_i)_{X_i=1}(\mathcal{S}(\boldsymbol{x}) \wedge \hookrightarrow((x_i)_{X_i=1}, (x'_i)_{X_i=1})))[x'_i := x_i \mid X_i = 1]$$

Using these redefined functions we can show an algorithm for symbolic reachability for PTSs, which is presented in Algorithm 2 on Page 16. In the algorithm presented we use a calligraphic font for variables which are stored symbolically, such as the set of reachable states ($\mathcal{R}$) and a normal font for variables

which are stored explicitly, such as the number of rows of the DM (K). The algorithm first initializes two variables $\mathcal{R}$ and $\mathcal{L}$ which will contain the reachable states for the entire model and for a particular level respectively. Then for each transition group a variable is initialized which contains the reachable states ($\mathcal{R}_i^p$) for that transition group and a variable which contains the transition relation for that transition group ($\hookrightarrow_i^p$). After these initialization steps a loop starts which ends when no new states arise in a new level. In this main loop the transition relation is built. The transition relation is built independently for each transition group. First a state vector is initialized which only contains read and write dependent state slots for transition group i. This state vector $\mathcal{L}^p$ is assigned the states of the last calculated level. Then for every *new* state of this level the transition relation is extended, i.e. the successor states for the new states are put in the transition relation. Next, the reachable states for the current transition group extended with the states in the current level. Next the state space is built – again independently for each transition group. The variable $\mathcal{N}$ is extended with all successor states using the one step function STEP. Lastly all new states are assigned to $\mathcal{L}$ and $\mathcal{R}$.

Algorithm 2: REACH-BFS-PREV [2]

```
   Data: DM, K, s^0
   Result: 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 |   for 1 ≤ i ≤ K do
 9 |   |   L^p ← π_i(L);
10 |   |   for s^p ∈ L^p \ R_i^p do
11 |   |   |   ↪_i^p ← ↪_i^p ∪ {(s^p, d^p) | d^p ∈ NEXT-STATE_i(s^p)};
12 |   |   end
13 |   |   R_i^p ← R_i^p ∪ L^p ;
14 |   end                                   } Build the transition relation
15 |   N ← ∅;
16 |   for 1 ≤ i ≤ K do
17 |   |   N ← N ∪ STEP(L, ↪_i^p, DM_i) ;
18 |   end
19 |   L ← N \ R;
20 |   R ← R ∪ N;                           } Build the state space
21 end
22
```

An alternative for REACH-BFS-PREV is an algorithm that learns the transition relation for transition group i from the current level plus all the states in the transition groups $< i$. This is called chaining and

therefore the algorithm is called REACH-CHAIN-PREV.

Algorithm 3: REACH-CHAIN-PREV

Data: DM, K, s^0
Result: $\mathcal{R}$

```
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  |   for 1 ≤ i ≤ K do
9  |   |   L^p ← π_i(L);
10 |   |   for s^p ∈ L^p \ R_i^p do
11 |   |   |   ↪_i^p ← ↪_i^p ∪ {(s^p, d^p) | d^p ∈ NEXT-STATE_i(s^p)};
12 |   |   end
13 |   |   R_i^p ← R_i^p ∪ L^p ;
14 |   |   L ← L ∪ STEP(L, ↪_i^p, DM_i) ;
15 |   end
16 |   R ← R ∪ L;
17 |   L ← L \ R;
18 end
```

A projection function on symbolic set is implemented as follows.

$$\text{PROJECT}(\mathcal{S}(\boldsymbol{x}), X) = \exists(x_i)_{X_i=0}.\mathcal{S}(\boldsymbol{x}),$$

where X is a row from a dependency matrix, e.g. $\pi_i(\mathcal{S}(\boldsymbol{x})) = \text{PROJECT}(\mathcal{S}(\boldsymbol{x}), DM_i)$.

1.4.2 A data structure for symbolic algorithms

For symbolic storage of the transition relation and the state space a form of Binary Decision Diagram (BDD) is used, namely a LDD, which is defined as follows.

Definition 1.13 (List Decision Diagram [2])**.** An LDD is a Directed A-cyclic Graph (DAG). A DAG has three types of nodes.

- $\{\epsilon\}$: meaning *true* and does not have successors.
- $\varnothing$: meaning *false* and does not have successors.
- a node with label v and two successors (down, right) and is written as node(v, down, right).

The semantics $[\![S]\!]$ of an LDD S is as follows.

$$\begin{aligned}
[\![\{\epsilon\}]\!] &= \{\epsilon\} \\
[\![\varnothing]\!] &= \varnothing \\
[\![\text{node(v, down, right)}]\!] &= \{\text{vw} \mid \text{w} \in [\![\text{down}]\!]\} \cup \{[\![\text{right}]\!]\}
\end{aligned}$$

■

1.4.3 Implementations of Decision Diagrams

The symbolic PINS back-end supports multiple implementations for decision diagrams, such as BDDs, MDDs and other. All supported implementations are listed in Table B.2 on Page 74.

1.4.4 Symbolic Reachability Implementation in LTSmin

Currently the Pins back-end supports various reachability algorithms. One key variant is described in Algorithm 2 is REACH-BFS-PREV. This algorithm can be found in `pins2lts-sym.c`.Algorithms 4, 5, 6, 7 and 8 on Page 18 show how actual vector set operations are implemented in the LTSmin back-end.

The way LTSmin performs reachability with LDDs is by first initializing the cache with INIT-CACHE and forwarding a call to STEP() to STEP-LDD(). To initialize the cache we define a vector $P = \langle j \mid DM_{i,j} = 1 \rangle$. The algorithm INIT-CACHE creates a list of nodes – bottom up – with values indicating the projected element. The recursive algorithm STEP-LDD first evaluates the base case – that is if the algorithm is done. If not done then it first matches the values of the nodes in the set and relation. Then it tries to look up the result in the cache for these values. If there is a cache miss then if the relation contains an update for the current node then it writes the correct value of every linked node on the right. If the relation does not contain an update than it simply copies the values already in the set – that is it recursively calls STEP-LDD on the nodes below and to the right.

Algorithm 4: INIT-CACHE

Result: C
$C \leftarrow \{\epsilon\}$;
$p \leftarrow |P| - 1$;
while $p \geq 0$ **do**
  $C \leftarrow \text{NODE}(P_p, C, \varnothing)$;
  $p = p - 1$;
end

Algorithm 5: STEP-LDD

Data: C, S, R, s, r
Result: L
if $R = \varnothing \vee S = \varnothing$ **then return** $\varnothing$;
if $r = |P|$ **then return** S;
if $P_r = s$ **then**
  $S, R \leftarrow \text{MATCH}(S, R)$;
  if $R = \varnothing \vee S = \varnothing$ **then return** $\varnothing$;
end
if IN-OP-CACHE(C, S, R) **then return** CACHE-LOOKUP(C, S, R);
$L \leftarrow \varnothing$;
$O \leftarrow R$;
if $P_r = s$ **then**
  $L \leftarrow \text{STEP-LDD}(C, \text{RIGHT}(S), \text{RIGHT}(R), s, r)$;
  $R \leftarrow \text{DOWN}(R)$;
  $L \leftarrow \text{WRITE}(C, S, R, L, s, r)$;
else
  $L \leftarrow \text{COPY}(C, S, R, s, r)$;
end
ADD-TO-CACHE(C, S, O, L) ;

Algorithm 6: MATCH

Data: S,R
Result: S,R
while $\text{VAL}(S) \neq \text{VAL}(R)$ **do**
  if $\text{VAL}(S) < \text{VAL}(R)$ **then**
    $S \leftarrow \text{RIGHT}(S)$;
    if $S = \varnothing$ **then return** S, R;
  end
  if $\text{VAL}(R) < \text{VAL}(S)$ **then**
    $R \leftarrow \text{RIGHT}(R)$;
    if $R = \varnothing$ **then return** S, R;
  end
end

Algorithm 7: COPY

Data: C, S, R, s, r
Result: L
$T_r \leftarrow \text{STEP-LDD}(C, \text{RIGHT}(S), R, s, r)$;
$T_d \leftarrow \text{STEP-LDD}(C, \text{DOWN}(S), R, s, r)$;
$L \leftarrow \text{NODE}(\text{VAL}(S), T_d, T_r)$

Algorithm 8: WRITE

Data: C, S, R, L, s, r
Result: L
while $R \notin \{\{\epsilon\}\}$ **do**
  $T \leftarrow \text{STEP-LDD}(\text{DOWN}(C), \text{DOWN}(S), \text{DOWN}(R), s+1, r+1)$;
  $T \leftarrow \text{NODE}(\text{VAL}(R), T, \varnothing)$;
  $L \leftarrow \text{UNION}(L, T)$;
  $R \leftarrow \text{RIGHT}(R)$;
end

1.5 Pins2Pins wrappers

Due to the nice architecture of the Pins interface LTSmin allows many front-end and back-end independent optimizations for model-checking which are shown as Pins2Pins *wrappers* in Figure 1.1 on Page 9. Three key wrappers are *local transition caching*, *variable reordering, transition regrouping* and *partial order reduction*[7].

1.5.1 local transition caching

If a language module is slow the back-ends may benefit from caching the transition group vectors. The cache uses the DM to only cache the short-vectors used by each transition group.

1.5.2 variable reordering, transition regrouping

This wrapper allows the reordering of transition groups and slots in the state vector. Optimizing both may greatly speed up symbolic state space generation. For more detailed information about reordering and regrouping please refer to [13]. The following reorderings and regroupings are implemented in PINS. Note that some algorithms below use a `cost` function for heuristics.

column sort	sorts columns such that 1s are placed leftmost.
column nub	merges columns which have the same dependencies.
column swap with simulated annealing	swaps columns based using simulated annealing [10] using the `cost` function.
column swaps	swaps columns using the `cost` function.
column all permutations	swaps columns using the `cost` function; tries every permutation.
row sort	sorts rows such that 1s are placed leftmost.
row nub	merges rows which have the same dependencies.
row subsume	merges row i in row j if the dependencies of row i is a subset of the dependencies of row j.

In our next chapters *row subsumption* poses some problems, we therefore provide a formal notion of row subsumption.

Definition 1.14 (row subsumption)**.** The row subsumption operator $\sqsubseteq$ is a binary operator and is defined in terms of two different rows from the DM $DM_{K\times N}$. We put an ordering on the elements of the dependency matrix as $0 < 1$.

$$DM_{i'} \sqsubseteq DM_i = \begin{cases} DM_i & \text{if } \forall 0 \leq j \leq N\colon DM_{i',j} \leq DM_{i,j}, \\ DM_{i'},\ DM_i & \text{otherwise.} \end{cases}$$

$DM_{i'} \sqsubseteq DM_i$ reads as "row $DM_{i'}$ is subsumed by row DM_i" or "row DM_i subsumes row $DM_{i'}$".

■

1.5.3 partial order reduction

Partial Order Reduction (POR) is used to explore a subset of the state space. In symbolic state space exploration POR is not supported. Also for languages which do not have a guard-splitting implementation POR is not available. In this research we implement guard-splitting for MCRL2 in order to make POR available. Guard-splitting is however one step towards POR for a language front-end. In Chapter 4 we will explain what is also necessary to enable POR.

1.6 Implementation in LTSMIN

Our work is done on all three main components of LTSMIN. Namely the PINS front-end, the PINS2PINS wrappers and the PINS back-end.

1.6.1 The PINS front-end

A language front-end in the PINS interface only has to implement GreyBox interface (GB) methods. These methods are always prefixed with the string GB and are defined in the C header file `pins.h`. Table B.1 show a list of GreyBox methods and how they relate to the algorithms used in this Chapter. The `GBgetTransitionsShort` and `GBgetTransitionsLong` relate closely together. Symbolic algorithms such as the REACH-BFS-PREV use `GBgetTransitionsShort` to compute successor states. However, most language front-ends only support computing successor states for a long vector. Therefore `GBgetTransitionsShort` expands a short vector to a long vector by using values from the initial state. `GBgetTransitionsShort` accepts only states projected using π. In Chapter 3 we will explain how we change these methods to improve reachability analysis. In chapter 4 we will mainly show how we added functions to the PINS interface to support guard-splitting.

1.6.2 The PINS2PINS wrappers

Table B.1 also shows the regrouping function on the DM. This function takes a model and a regrouping specification to apply transformations on the DM. We will show in Chapter 3 how we made modifications to this function.

1.6.3 The PINS back-end

The variables used in algorithms such as REACH-BFS-PREV are shown in Table B.3. Furthermore the operations on sets are listed in tables B.5 and B.6. In Chapter 3 we will discuss how the `vset_next` function is changed to support separating read and write dependencies. In Chapter 4 we will add a join operation on two sets. We thus get a new `vset_join` function in the symbolic back-end.

Every vector set implementation in a symbolic back-end contains not only the nodes which make up the set, but also the projection it uses. Thus one key function call is `init_domain` to initialize the domain of a set by using the projection information. After a set is initialized one can use all vector set operations such as `vset_minus`. Initialization of sets with some projection is something we are going to change in Chapter 3.

Chapter 2

Research Method

Separating read and write dependencies and guard-splitting are two distinct changes to existing reachability algorithms. We thus have worked on these improvements separately. Both improvements have separate chapters in this thesis and our approach will therefore also be discussed separately. Both problems will be discussed with the expected results and goals. To accurately conclude our work we will state research questions and challenges we will face. Lastly will will make important assumptions to be able to validate our work.

Comparing sizes of state spaces of a large amount of models is part of the validation of our work. This means that instead of tweaking options to reachability algorithms such as the size of node tables and cache sizes to get the best results for each model, we instead produce benchmarks for a large set of models. There is one other decision that contributed to our choice to produce many results. Before starting on the benchmarks we did not know in advance which models could benefit of our work. Only clear scatter plots (such as Figure 3.4 on Page 43) helped us to see which models benefit of separating dependencies and guard-splitting.

Our experiments are done on machines with Intel Xeon E5335 processors running at 2 GHz. The machines have 24 GB of memory and run an up to date Scientific Linux distribution. Every experiment we do is repeated three times and in the results we will show the standard deviation of both the time and memory usage to show that our measurements are precise. The scatter plots of our experiments always show the result without our changes on the horizontal axis and the result with our changes on the vertical axis. That means if a result is plotted on the right half of the $y = x$ line then the experiment with our changes runs faster or uses less memory.

2.1 Separating Read and Write Dependencies

Separating read and write dependencies involves providing a more fine-grained view on dependencies between transition groups and state slots. We will first describe this problem and then show how we approach this problem. Separating read and write dependencies entail two key improvements. We always have transition relations that are at least as large as without separating read and write dependencies. Thus we can cope better with the explosion of nodes when using a bad variable ordering. Also, we can perform less NEXT-STATE calls if we only compute successor states for short vectors of read dependent state slots. In order to validate our work we will make assumptions with relation to existing work on for example dependency matrices and existing algorithms.

2.1.1 Problem

Currently the projection π_i as in Definition 1.9 is defined as $\pi_i : S \to \Pi_{\{1 \leq j \leq N | D_{i,j} \neq -\}} S_j$. This means that when this projection is used every state vector of some transition group is projected to a *short vector*

where variables are read, written or both, e.g. on Line 9 of Algorithm 2. When building the transition relation (Line 8 to 14 of Algorithm 2)

1. the value for the next level for a state slot that is only read is also calculated. Naturally this gives much overhead, because the value of this state slot will not change.
2. the value for the current level for a state slot that is only written is also calculated. This gives much overhead, because the value of the state slot in the current level is not relevant.

Example 2.1. Suppose we have a very simple recursive process

$$X(x, y, z) := x < 5 \wedge z > 0 \Rightarrow X(x + 1, x + 1, z)$$

Then we have a DM with three state slots and one transition group.

$$\begin{array}{c} \quad x \quad y \quad z \\ g_1 \begin{bmatrix} \texttt{+} & \texttt{w} & \texttt{r} \end{bmatrix}, \end{array}$$

where `+` means that a variable is both read and written, `w` means that a variable is only written and `r` means that a variable is only read. Then the transition relation for transition group g_1 will – according to π_{g_1} – contain six variables: $\langle x, x', y, y', z, z' \rangle$, in which the primed variables are variables for the next state. But since we know y is only written and z is only read we can reduce the transition relation to the following variables: $\langle x, x', y', z \rangle$. ▲

2.1.2 Results & Quantitative Goals

Separating read and write dependencies entail two key improvements. The first improvement involves obtaining a smaller transition relation. Especially useful is, when a transition relation uses a bad variable ordering removing any node from the transition relation may significantly reduce the size of the transition relation. The second improvement comes from using the new projection functions on the new dependency matrices.]The set that a projection gives is used to compute successor states. So any improvement on this projection such that it gives a smaller set greatly speeds up computing successors. On a dependency matrix for read dependencies we can define a new projection which only projects to state slots which are read. The set that the new projection gives is only as large as the set the current π projection gives. When a row in the dependency matrix has many write dependencies and few read dependencies the set by π can be significantly larger. Thus a smaller set given by the new projection will greatly improve reachability analysis in terms of time. Removing nodes from the transition relation obviously improves reachability in terms of space.

2.1.3 Qualitative Goals

In the part of separating dependencies in our research we have some concrete goals. We give precise notions for read and write dependencies which can also cope with copying values. Also we will present a precise notion for the transition relation which exploits these dependencies. This notion can be used in general by all symbolic implementations (BuDDy: A BDD package (BUDDY), LDD, etc.). Reachability algorithms need to exploit the definitions of read and write dependencies, so we will also present abstract forms of these algorithms (improvements on Algorithms 2, 3) in our thesis. To benchmark our work against existing reachability algorithms we need to improve STEP-LDD (Algorithm 5). For better results we will also show new algorithms on the dependency matrix and one improvement. The results of the benchmarks will be given in scatter plots to quickly identify which models really benefit of our changes. Next one can then look up detailed information of the benchmarks in tables with information about memory usage etc. To be able to show benchmarks for DVE with our changes we will also fix a bug in the dependency matrix of the DVE compiler for LTSMIN. Lastly we will give a list of future work to be done and discuss why — in some parts we did more work than described in our research proposal and in other parts did less work.

2.1.4 Research Questions

We can conclude our research by answering the main research question.

How can precise notions for read and write dependencies improve symbolic reachability analysis?

Interesting sub questions include the following.

- How can we optimize how the transition relation is built?
- How can we improve projections with the notion of read and write dependencies?
- How much time and space do we save with this technique?
- To what extent can advanced reachability strategies such as saturation profit from our technique?

2.1.5 Challenges

For implementing the separation of read and write dependencies we will face four main challenges. The first is to come up with precise notions for read and write dependencies and letting algorithms exploit these. The second challenge is to make sure read and write dependency matrices for mCRL2, DVE and PROMELA are correct. The third challenge is to make sure the read and write dependencies are in line with the definitions for explicit reachability analysis with Partial Order Reduction. Lastly we need to find models which can be used in benchmarking. Finding good models is hard because firstly, we do not know models of the mentioned modeling languages very well. Let alone knowing whether or not they benefit of our changes. Simply using all known models in for example the BEEM [8] database can not be done without precise measurements. We need to repeat experiments multiple times to see if standard deviation of our measurements is not to large.

2.1.6 Validation & Assumptions

In our research we will not prove our algorithms correct using our definitions. This is because this is also not done for what is already there in the current situation. We will however precisely explain how our new definitions and algorithms work. But most importantly, we will provide an exhaustive list of tables with many details of our experiments done. With this information one can see that all of our experiments compute the same state spaces as without our changes. Naturally, wrong assumptions on either the models or the environment in which we do our experiments will probably lead to false results if we compare them to results which are known to be correct. Nevertheless, we assume that the work which is already done and described in Chapter 1 is correct. This means that for example we assume dependency matrices for mCRL2, DVE and PROMELA are correct. Moreover we assume that reachability algorithms (2, 3) and their implementations in LTSMIN produce the correct state space.

2.2 Guard-based Symbolic Reachability

Our work on guard-splitting uses the same approach as separating read and write dependencies. That is, describing definitions for dependencies, dependency matrices and algorithms. Also both chapters will have a discussion about the results. To validate our work we make one additional assumption on the logic of term rewriting systems and the completeness of model specifications.

2.2.1 Problem

Although symbolic model checking works fine for many models, it highly depends on the so-called locality of transitions. It fails for systems with transitions that read or write many state variables in one transition. Consider a transition of the following form.

$$Condition(x_j \dots x_k) => x_i := Expression,$$
$$\text{where } 1 \leq j \leq k \leq |x|.$$

It reads the whole state vector, but only modifies one variable x_i. This is very inconvenient for symbolic model checking. Often, the Condition can be split in many guards. Assume that the transition above is of the following form.

$$G_1(x_j \dots x_k) \wedge \dots \wedge G_m(x_o \dots x_p) => x_i := Expression,$$
$$\text{where } 1 \leq j \leq k \leq |x| \text{ and } 1 \leq o \leq p \leq |x|.$$

The main research question is how to improve symbolic reachability, so that it can profit from the guard structure. We expect big improvements in the efficiency, especially for synchronous models like hardware models and combinatorial puzzles.

2.2.2 Results & Quantitative Goals

There are two main benefits to guard-splitting. The first one is that for some models we can greatly reduce the amount of NEXT-STATE calls. This is due to the fact that we also store which states satisfy which guards separately from what the successor states are. We keep this information throughout the computation of each level. This means if one guard in the condition evaluates to false we already know for which states we do not have to compute successor states. This can not be done without guard-splitting. The second benefit involves just like with separating read and write dependencies much smaller decision diagrams. This is because instead of one transition relation for one transition group we have more smaller ones for each guard. Again — if we have a bad variable ordering removing nodes from a single decision diagram greatly reduces the bad effects of this ordering.

In symbolic reachability analysis we save time because operations on decision diagrams take less time if we have smaller decision diagrams. Also, reducing the amount of NEXT-STATE calls reduces the time needed to complete reachability analysis. Since we now store extra decision diagrams, namely for the guards we might not always save space, especially when there are many transitions which always satisfy all guards for a transition group.

2.2.3 Qualitative Goals

Like with separating read and write dependencies we need to give formal definitions for independence of guards. We will also give a precise notion of independence for only the update part of a transition group. Along with the definitions for independence we will provide notions for dependency matrices and their respective projections. For DVE and PROMELA guard-splitting is already (partially) implemented. So we will look at how we can implement guard-splitting for MCRL2. This is especially interesting because MCRL2 is a term rewriting system, unlike DVE and PROMELA. For MCRL2 we will thus illustrate how to perform guard-splitting and how to evaluate guards. For reachability algorithms, to use guards we have to show how we can reduce sets of states in the current level. This reduction can be efficiently performed by a join operation on sets. We will thus give a precise notion for the join operation which multiple symbolic back-ends can use. In order to benchmark our guard-splitting algorithm with this join operation we implement the join operation in LDD. We tried to also benchmark guard-splitting for DVE but due to time restrictions we are not able to show the results hereof. We will however discuss how far we got benchmarking guard-splitting for DVE. The result of our experiments will also be given in scatter plots and an exhaustive list of tables with detailed information. Guard-splitting poses problems to dependency matrix operations because the knowledge of whether the front-end or back-end knows which states satisfies which condition is switched. We will thus discuss this in a section about dependency

matrix operations. Like with our approach for separating read and write dependencies we will give a list of future work and discuss why we did more or less work than described in our research proposal. For example why we can not do Partial Order Reduction with our work as was expected when writing our research proposal.

2.2.4 Research Questions

Analogous to our other research question we will answer the following.

To what extent can guard-splitting improve symbolic reachability analysis?

Interesting sub questions include the following.

- What is a good notion for guards usable in symbolic reachability analysis?
- How can guard-splitting be implemented for mCRL2?
- To what extent can saturation profit from guards?
- How much time and space do we save with our technique of guard-splitting?

2.2.5 Challenges

There is one main challenge in performing guard-splitting. That is, an individual guard may evaluate to 'maybe', i.e. not true or false. We give detailed information in Section 4.3 what is involved here. Unlike in our research proposal we thought that guard-splitting would be much harder than separating read and write dependencies. This is however not the case because algorithms involving guard-splitting can be drafted quite easily on paper. Furthermore we experienced that guard-splitting for term rewriting systems is rather easy; Section 4.4.

2.2.6 Validation & Assumptions

Validation of our work on guard-splitting is done much the same way as for separating read and write dependencies (Section 2.1.6). We will give an abstract description of extensions to existing reachability algorithms (Algorithms 2, 3). From these descriptions it should not be hard to identify whether or not these algorithms are correct. Since — as a result we will have an exhaustive list of benchmarks we can compare state spaces of experiments with and without our changes. So we can conclude that our work is valid. We will however make some assumptions on existing work and on a specific element in mCRL2. Our work with relation to guard-splitting will be benchmarked against our changes with read and write separation, we thus assume that our work there is correct. Furthermore we assume that both matrices which already exists in DVE and PROMELA for guard-splitting are correct. For mCRL2 we do one important assumption which relates to the fact that a guard can evaluate to 'maybe'. The assumption is that we assume that either if a guard evaluates to maybe than the original condition evaluates to false. Or, if all required guards for a transition group evaluate to 'true' than the original condition also evaluates to 'true'. Since we leave the 'maybe check' to future work — that is, checking whether if one guard evaluates to 'maybe' there is another guard which evaluates to false. We assume that every model with which we test has a complete specification.

2.3 Related Work

Our work presented in this thesis is based on some earlier work done at the University of Twente on PROMELA and Partial Order Reduction [5, 7]. The work done on guards for Partial Order Reduction is not completely compatible with symbolic algorithms and mCRL2 as a language front-end. The definition for write dependent used in the work by Pater et al. [7] is inconvenient for symbolic algorithms, we will

thus extend this definition. In mCRL2 individual conjuncts of the condition (guards) may not evaluate to true or false. Because Definition 2.2 does assume this we need to extend this definition.

LTSmin supports many front-ends. In our benchmarks we will measure if some of these front-ends benefit of our changes. We will perform benchmarks for mCRL2, DiVinE and Promela [11]. Guards are already implemented for DiVinE and Promela so benchmarking our work with these front-ends is easy and it may give extra insights in whether our work is correct and whether it really speeds up the computation of state spaces.

Ciardo et al. [4] implemented a partitioning for a saturation algorithm which can be applied to completely general asynchronous systems. This is implemented in the SMART model checker. Saturation is a technique which also recognizes and exploits the presence of event locality but recursively applies multiple 'local' fixed-point iterations instead of a single fixed-point in our 'traditional' reachability algorithm. Saturation is a hard algorithm but results in peak memory requirements and consequently run times often several orders of magnitude smaller than the traditional algorithms. Saturation is also implemented in LTSmin by Tien Loong Siaw [9] in LTSmin. The saturation algorithm uses traditional reachability algorithms such as described in Listing 2 to compute these local fixed-points. In our research it is relevant to see how separating dependencies and implementing guard-splitting can improve saturation.

2.3.1 Guards for Pins

Typically a specification written in mCRL2 contains multiple guarded transitions; see Definition 1.5. A guarded transition consists of a guard (or condition) and an assignment. When POR was implemented as a Pins2Pins wrapper guards were specifically designed to be used for POR. The drawback of the following definitions is that they can only be used for POR and not for symbolic reachability. The reason for this is that the definitions assume every state only has one successor state, like an LPS without Σ. Another drawback is that the write set in Definition 2.5 is only usable for POR and not for symbolic algorithms. In symbolic algorithms we must separate copying values from simply writing values. To the best of our knowledge the need for two different notions for write dependent is quite unique. The need arises from the fact that we have many front-ends in LTSmin for which some support arrays and other do not. Also LTSmin supports both explicit reachability analysis with partial order reduction and symbolic reachability analysis.

Definition 2.2 (guard [5])**.** A guard $g\colon S_{\mathcal{P}} \mapsto \mathbb{B}$ is a total function that maps each state in a TS to a boolean value, $\mathbb{B} = \{true, false\}$. We write $g(s)$ or $\neg g(s)$ to denote that guard g is $true$ or $false$ in state $s \in S_{\mathcal{P}}$. We also say that g is *enabled* or *disabled*. ■

Definition 2.3 (structural transition [5])**.** A structural transition $t \in T$ is a tuple $(\mathcal{G}, a)$ such that a is an assignment $a\colon S_{\mathcal{P}} \mapsto S_{\mathcal{P}}$ and $\mathcal{G}$ is a set of guards. We denote the set of enabled transitions by $en(s) := \{t \in T \mid \bigwedge_{g\in\mathcal{G}} g(s)\}$. We write $s \xrightarrow{t}$ when $t \in en(s)$, $s \xrightarrow{t} s'$ when $s' = a(s)$ and $s \xrightarrow{t_1 t_2 \ldots t_k} s_k$ when $\exists s_1, \ldots, s_k \in S_{\mathcal{P}}\colon s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_k} s_k$. ■

Besides guarded transitions, structural information is required on the exact involvement of state variables in a transition. To define some guard g depends on index i, we test whether $g(s)$ is different from $g(s')$ for some state s and s' that only differ at index i.

Definition 2.4 (disagree sets [5])**.** Given states $s, s' \in S_{\mathcal{P}}$, for $1 \leq i \leq N$, we define the set of indices on which s and s' disagree as $\delta(s, s') := \{i \mid s_i \neq s'_i\}$. ■

Definition 2.5 (affect sets [5])**.** For $t = (\mathcal{G}, a) \in T$, the set of reachable states R and $g \in \mathcal{G}$, we define

1. the test set of g is $Ts(g) \supseteq \{i \mid \exists r, r' \in R\colon \delta(r, r') = \{i\} \wedge g(r) \neq g(r')\}$,
2. the test set of t is $Ts(t) := \bigcup_{g\in\mathcal{G}} Ts(g)$,
3. the write set of t is $Ws(t) \supseteq \bigcup_{r\in R}\{\delta(r, a(r)) \mid r \xrightarrow{t} a(r)\}$,
4. the read set of t is $Rs(t) \supseteq \{i \mid \exists r, r' \in R\colon \delta(r, r') = \{i\} \wedge r \xrightarrow{t} \wedge r' \xrightarrow{t} \wedge Ws(t) \cap \delta(a(r), a(r')) \neq \varnothing\}$ and

5. the variable set of t is $Vs(t) := Ts(t) \cup Rs(t) \cup Ws(t)$.

■

Just like over approximating variable dependencies these *affect sets* may be over approximated ($\supseteq$) by the language front-end.

Example 2.6 (Guards)**.** If we ignore the assumption that every state only has one successor state the guard information of process `p1()` of Example 1.1 on Page 14 is as follows. The guard is $g_1 =$ (`x==true`), the affect sets of the first transition group are $Ts(g_1) = \{1\}$, $Ts(t) = \{1\}$, $Ws(t) = \{1, 3\}$, $Vs(t) = \{1, 3\}$ and $Rs(t) = \{1\}$ because it x is both read and written. ▲

Using conjunctive partitioning schemes in symbolic model checking is not new. Ciardo actually used this scheme for saturation. In our work we use guards to reduce sets of states to prevent computing successor states. We can not however take the intersection ($\cap$) of a set of states that satisfy a guard and the set of states for which we want to compute successors for. This is due to the fact that a guard often uses only a small subset of state slots and the set of states for which we want to compute successors for uses all state slots. We thus define a join like in relational databases. The join for symbolic sets is defined in Section 4.5 on Page 54. The join is a conjunction between sets with different projections. It is thus a special case of a conjunction.

Chapter 3

Symbolic Reachability Using Separate Read, Write and Copy Dependencies

In Chapter 1 we saw that partitioning the NEXT-STATE interface and defining dependencies between transition groups and state slots can greatly speed up reachability analysis. We can however put an even more fine-grained view on these dependencies. If we take into account that a transition group can either *read*, *write* and *copy* values we can speed up reachability analysis even more. We first extend the PINS interface with definitions for reading, writing and copying values. Then we show how the symbolic back-end can profit from these definitions. Also, we show how PINS2PINS wrappers are updated to cope with these new definitions.

3.1 The Partitioned Next-State Interface

In general a transition group can read, write and copy values in state slots. This is a more fine-grained view on dependencies in Chapter 1. Here — a dependency means that a transition group both reads and writes from and to a state slot. One can imagine that this is naturally not always the case. It is however not easy to identify whether a transition group only reads or writes to a state slot. Even more non evident is that a transition group can copy values in case a state slot represents an element in an array. We provide three definitions; *read-independent*, *must-write-dependent* and *may-write-independent*. With these definitions we can precisely identify the behavior of transition groups.

Our first definition can be used to state that a transition group does not read a state slot.

Definition 3.1 (read-independent)**.** Given a PTS $\mathcal{P} = \langle S_\mathcal{P}, \rightarrow_\mathcal{P}, s^0 \rangle$ transition group i is *read-independent* on state slot j if for all $\langle s_1, \ldots, s_N \rangle, \langle t_1, \ldots, t_N \rangle \in S_\mathcal{P}$, whenever $\langle s_1, \ldots, s_j, \ldots, s_N \rangle \rightarrow_\mathrm{i} \langle t_1, \ldots, t_j, \ldots, t_N \rangle \in \rightarrow_\mathcal{P}$ it holds that $(s_j = t_j \wedge \forall r_j \in S_j \colon \langle s_1, \ldots, r_j, \ldots, s_N \rangle \rightarrow_\mathrm{i} \langle t_1, \ldots, r_j, \ldots, t_N \rangle) \vee (\forall r_j \in S_j \colon \langle s_1, \ldots, r_j, \ldots, s_N \rangle \rightarrow_\mathrm{i} \langle t_1, \ldots, t_j, \ldots, t_N \rangle)$. I.e. the value of state slot j is not relevant in transition group i. Whether some transition group i and state slot j is not read-independent may also be over-approximated. ■

This definition says that if there is a transition between two states s and t, then if state slot j is read-independent then transition group i must always copy the value for state slot j, because it ignores state slot j. Or, the transition group must for any value in state slot j, in state s go to state t, such that it does not have to read state slot j. Note that this definition also correctly marks y as read-dependent in the following nondeterministic case.

$$y = 1 \implies y \leftarrow \{0, 1\}.$$

This is because the definition handles both cases $y = 1 \implies y \leftarrow 0$ and $y = 1 \implies y \leftarrow 1$ separately. In practice one can often easily see whether or not a state slot is read-independent or not. Suppose that state slot j represents a variable x in a modeling language. Then x is often read-independent if it is not

on the right hand side of an assignment or in the condition of a transition group. This is also how we fill the following matrix for an LPS.

Definition 3.2 (read dependency matrix)**.** A Read Dependency Matrix (RDM) $RM_{K\times N} = RDM(\mathcal{P})$ for PTS $\mathcal{P}$ is a matrix with K rows and N columns containing $\{0,1\}$ such that if $RM_{i,j} = 0$ then transition group i is *read-independent* on state slot j. ■

Recall Definition 1.5; the definition of an LPS in mCRL2. We define the contents of the PINS RDM for LPS X $RDM(X) = RM^X_{K\times N}$ as follows.

$$RM^X_{i,j} = \begin{cases} 1 & \text{if } \exists 0 \leq k \leq N : x_j \text{ occurs in } t_{i,k} \wedge (j \neq k \vee t_{i,j} \neq x_j) \vee x_j \text{ occurs in } C_i \\ 0 & \text{otherwise.} \end{cases}$$

Analogous to whether or not a transition group reads a state slot we define whether or not a transition group writes to a state slot.

Definition 3.3 (must-write-dependent)**.** Given a PTS $\mathcal{P} = \langle S_\mathcal{P}, \rightarrow_\mathcal{P}, s^0\rangle$ transition group i is *must-write-dependent* on state slot j if for all $\langle s_1, \ldots, s_N\rangle, \langle t_1, \ldots, t_N\rangle \in S_\mathcal{P}$, whenever $\langle s_1, \ldots, s_j, \ldots, s_N\rangle \rightarrow_i \langle t_1, \ldots, t_j, \ldots, t_N\rangle \in \rightarrow_\mathcal{P}$ we actually have $\forall s'_j \in S_j \colon \langle s_1, \ldots, s'_j, \ldots, s_N\rangle \rightarrow_i \langle t_1, \ldots, t_j, \ldots, t_N\rangle$. Note that whether a transition group i is must-write-dependent;unlike read-independent can not be over-approximated. Transition group i either always writes the same value or there does not exist a transition. ■

This definition says that if we have a transition between two states s and t and state slot j is must-write-dependent then we must be able to assign any value to state slot j and still go to state t. In practice a state slot is not must-write-dependent if it is not on the left hand side of an assignment. With — however the main exception being writing to an element in an array. To cope with this exception we will give another write dependency definition in a moment. First we show the definition of a new matrix and how we can fill this matrix for an LPS.

Definition 3.4 (must-write dependency matrix)**.** A Must-write Dependency Matrix (WDM) $WM_{K\times N} = WDM(\mathcal{P})$ for PTS $\mathcal{P}$ is a matrix with K rows and N columns containing $\{0,1\}$ such that if $WM_{i,j} = 1$ then transition group i is *must-write-dependent* on state slot j. ■

The WDM $WDM(X) = WM^X_{K\times N}$ for LPS X is defined as follows.

$$WM^X_{i,j} = \begin{cases} 1 & \text{if } t_{i,j} \neq x_j \\ 0 & \text{otherwise.} \end{cases}$$

Using both the definition of read-independent and write-dependent we can not efficiently cope with arrays in modeling languages. Now, mCRL2 does not have arrays as a language construct, but for example PROMELA and DIVINE do. Consider the following PROMELA example.

Listing 3.1: Example producer-consumer Promela model

```
1  #define N 2
2  int b[N];
3  int i = 0;
4  proctype c() { // consumer
5    do
6      :: i > 0 -> atomic { i--; b[i] = false; }
7    od
8  }
9  proctype p() { // producer
10   do
11     :: i < N -> atomic { b[i] = true; i++; }
12   od
13 }
```

```
14 init {
15     run c();
16     run p();
17 }
```

The state space of the model can be illustrated as follows.

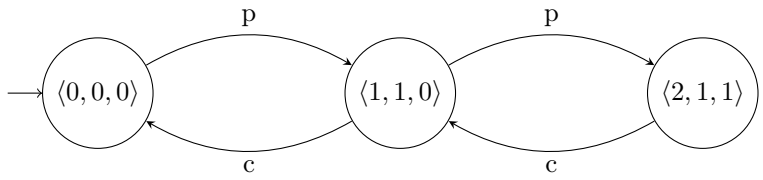


Figure 3.1: State space of model Listing 1.1. Edge labels are transition groups.

The model contains a buffer of length 2 and a variable for writing and reading an element in this buffer. The consumer writes false at index position i. The producer writes true at index position i. A natural way of partitioning this model is to introduce two transition groups. One for the consumer and one for the producer. The state vector is partitioned into three state slots; i, b_0, b_1. The key problem in this model is that if we write to one position in the buffer we must copy the other values in the buffer. Two major reasons to introduce a new definition for write independent are as follows.

1. There is a conflicting requirement for write dependent between POR in explicit reachability analysis and symbolic reachability analysis. I.e. Definition 3.3 does not match Definition 6 in [5]. Here the write set says a transition group is write dependent on a state slot if that transition group only supports a transition for *some* values, we will call this may-write dependent.

2. We do not want to over-approximate a may-write dependent state slot to read-dependent in symbolic reachability analysis, because this is not efficient. I.e. values only have to be copied, not read.

Now follows a definition in line with Definition 3.3 which can be used in both symbolic algorithms as well as POR in explicit algorithms.

Definition 3.5 (may-write-independent)**.** Given a PTS $\mathcal{P} = \langle S_\mathcal{P}, \rightarrow_\mathcal{P}, s^0 \rangle$ transition group i is *may-write-independent* on state slot j if for all $\langle s_1, \ldots, s_N \rangle, \langle t_1, \ldots, t_N \rangle \in S_\mathcal{P}$, whenever $\langle s_1, \ldots, s_j, \ldots, s_N \rangle \rightarrow_i \langle t_1, \ldots, t_j, \ldots, t_N \rangle \in \rightarrow_\mathcal{P}$ it holds that $s_j = t_j$, i.e. state slot j is not modified in transition group i. Whether some transition group i and state slot j is not may-write-independent may also be over approximated. ■

This definition can be best explained by discussing its negation. If transition group i is may-write-dependent on state slot j then there are some states s and t and a transition between them for which the value in state slot j is changed ($s_j \neq t_j$). This is thus exactly the case when we have some array and an index for which we write to state slot j. For all other values we do not do such a transition we must copy those values. Naturally we can not copy those values if we replace the value for state slot j with some other value.

For mCRL2 the May-write Dependency Matrix (MDM) is identical to the WDM, because mCRL2 does have features such as arrays.

Definition 3.6 (may-write dependency matrix)**.** A MDM $MM_{K \times N} = MDM(\mathcal{P})$ for PTS $\mathcal{P}$ is a matrix with K rows and N columns containing $\{0, 1\}$ such that if $MM_{i,j} = 0$ then transition group i is *may-write-independent* on state slot j. ■

There are two ways to use may-write dependencies in the projections. First we can reserve special values in the LDDs or we can handle a may-write dependent state slot as if it were read. The latter option is easier and we thus reserve the first options for future work. For the latter option we need to be able to 'merge' the may-write and the must-write matrix. This can be done with the logical $\vee$ operation on every element in both matrices.

Definition 3.7 ($\vee$ on matrices)**.** The logical or operation ($\vee$) on two matrices is defined as $M = M' \vee M'' = \forall i, j \colon M_{i,j} = M'_{i,j} \vee M''_{i,j}$. ■

We now have three new matrices for which we have to define a two projections.

Definition 3.8 (projections)**.** For any transition group $1 \leq i \leq K$, we define ρ_i as the projection $\rho_i : S \to \Pi_{\{1 \leq j \leq N | RM_{i,j}=1 \vee MM_{i,j}=1\}} S_j$ and ω_i as the projection $\omega_i : S \to \Pi_{\{1 \leq j \leq N | WM_{i,j}=1 \vee MM_{i,j}=1\}} S_j$. ■

In the following example we will give three dependency matrices for both the 1-safe Petri net in Figure 1.2 and the producer-consumer model in Listing 1.1. We omit giving a precise notion for dependency matrices for PROMELA, they can be found in [11].

	Petri net	**producer-consumer**
RDM	$\begin{array}{c} \\ T0 \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{array} \begin{array}{c} \begin{array}{ccccc} P0 & P1 & P2 & P3 & P4 \end{array} \\ \left[\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{array}\right] \end{array}$	$\begin{array}{c} \\ c \\ p \end{array} \begin{array}{c} \begin{array}{ccc} i & b_0 & b_1 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \end{array}\right] \end{array}$
WDM	$\begin{array}{c} \\ T0 \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{array} \begin{array}{c} \begin{array}{ccccc} P0 & P1 & P2 & P3 & P4 \end{array} \\ \left[\begin{array}{ccccc} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{array}\right] \end{array}$	$\begin{array}{c} \\ c \\ p \end{array} \begin{array}{c} \begin{array}{ccc} i & b_0 & b_1 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \end{array}\right] \end{array}$
MDM	$\begin{array}{c} \\ T0 \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{array} \begin{array}{c} \begin{array}{ccccc} P0 & P1 & P2 & P3 & P4 \end{array} \\ \left[\begin{array}{ccccc} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{array}\right] \end{array}$	$\begin{array}{c} \\ c \\ p \end{array} \begin{array}{c} \begin{array}{ccc} i & b_0 & b_1 \end{array} \\ \left[\begin{array}{ccc} 1 & 1 & 1 \\ 1 & 1 & 1 \end{array}\right] \end{array}$

Example 3.9 (dependency matrices and projections)**.** For the projections we have

- $\rho_{T1}(\{\langle 1,0,0,0,0\rangle, \langle 0,1,0,1,0\rangle, \langle 0,0,1,1,0\rangle, \langle 0,1,0,0,1\rangle, \langle 0,0,1,0,1\rangle\}) = \{\langle 0\rangle, \langle 1\rangle\}$.
- $\rho_c(\{\langle 0,0,0\rangle, \langle 1,1,0\rangle, \langle 2,1,1\rangle\}) = \{\langle 0,0,0\rangle, \langle 1,1,0\rangle, \langle 2,1,1\rangle\}$.
- $\omega_{T1}(\{\langle 1,0,0,0,0\rangle, \langle 0,1,0,1,0\rangle, \langle 0,0,1,1,0\rangle, \langle 0,1,0,0,1\rangle, \langle 0,0,1,0,1\rangle\}) = \{\langle 0,0\rangle, \langle 1,0\rangle, \langle 0,1\rangle\}$.
- $\omega_c(\{\langle 0,0,0\rangle, \langle 1,1,0\rangle, \langle 2,1,1\rangle\}) = \{\langle 0,0,0\rangle, \langle 1,1,0\rangle, \langle 2,1,1\rangle\}$.

▲

If one compares this example to Example 1.10 one can see that ρ may give smaller sets of states than π. This greatly speeds up symbolic reachability analysis because we only have to compute successor states for the set of states given by ρ. Using the projection functions ω_i and ρ_i we can redefine the *next-state* function for a PTS. That is we extend the definition of the function to get the next states for only a particular transition group i.

Definition 3.10 (Partitioned Next-State function)**.**

$$\text{NEXT-STATE}_i(s \in \rho_i(S_\mathcal{P})) = \{s' \in \omega_i(S_\mathcal{P}) \mid s \to_i s'\}$$

■

As a last extension to the PINS interface we — for writing convenience extend the definition of the DM as follows.

Definition 3.11 (dependency matrix)**.** A *dependency matrix* $DM_{K\times N}$ for PTS $\mathcal{P}$ is a matrix with K rows and N columns defined in terms of the contents of RM, WM and MM with columns containing $\{\texttt{-}, \texttt{r}, \texttt{W}, \texttt{w}, \texttt{+}\}$, such that:

$$DM_{i,j} = \begin{cases} \texttt{-} \text{ (totally independent)} & \text{if } WM_{i,j} = 0 \text{ and } RM_{i,j} = 0 \text{ and } MM_{i,j} = 0, \text{ else} \\ \texttt{r} \text{ (read dependent)} & \text{if } WM_{i,j} = 0 \text{ and } MM_{i,j} = 0, \text{ else} \\ \texttt{w} \text{ (must-write dependent)} & \text{if } RM_{i,j} = 0 \text{ and } MM_{i,j} = 0, \text{ else} \\ \texttt{W} \text{ (may-write dependent)} & \text{if } RM_{i,j} = 0 \text{ and } WM_{i,j} = 0, \text{ else} \\ \texttt{+} \text{ (totally dependent)} & . \end{cases}$$

■

The writing convenience follows for example from the fact that $DM_{i,j} = \texttt{r} \implies WM_{i,j} = 0$ and $MM_{i,j} = 0$.

3.2 The PINS back-end

To be able to let multiple symbolic back-ends exploit our new notions for independence we first provide new notions for both the transition relation and the one step operation. Then we show how we can apply the transition relation on a set of states with LDDs. Other implementations such as BUDDY do not yet use our notion for independence, this is left as future work.

3.2.1 Symbolic reachability for Partitioned Transition Systems

The transition relation is changed such that a decision diagram only contains *read* nodes for a variable if the state slot represented by that variable has a read dependency. The same holds for *write* nodes.

$$\hookrightarrow(\boldsymbol{x}, \boldsymbol{x'}) = \bigvee_{i=1}^{K} \mathcal{D}_i(\boldsymbol{x}, \boldsymbol{x'}) = \bigvee_{i=1}^{K} (\hookrightarrow_i(\varpi_i(\boldsymbol{x}), \varpi'_i(\boldsymbol{x'})) \wedge \bigwedge_{\{1\le j\le N | DM_{i,j}=-\}} [x_j = x'_j]),$$

where $\varpi_i(\boldsymbol{x}) = (x_j)_{j\in\{j' | DM_{i,j'}\in\{\texttt{r},\texttt{W}\}\}}$ and $\varpi'_i(\boldsymbol{x}) = (x_j)_{j\in\{j' | DM_{i,j'}\in\{\texttt{w},\texttt{W}\}\}}$

The one step operation on a subvector is changed such that it knows not every variable in the transition relation has both a read and a write node.

$$\begin{aligned} &\text{STEP}(\mathcal{S}(\boldsymbol{x}), \hookrightarrow((x_i)_{X_i=1}, (x'_i)_{X'_i=1}), X, X') \\ &\quad = (\exists (x_i)_{X_i=1\wedge X'_i=1}(\mathcal{S}(\boldsymbol{x}) \wedge \hookrightarrow((x_i)_{X_i=1}, (x'_i)_{X'_i=1})))[x'_i := x_i \mid X'_i = 1] \end{aligned}$$

We extend our algorithm REACH-BFS-PREV and REACH-CHAIN-PREV to incorporate the separation of read and write dependencies as follows. We project the states in the current level only to the read variables. Then to the transition relation $\hookrightarrow_i^p$ a tuple is added in which the update is a state projected down to the write variables. Now, to make sure our LDD implementation can exploit the fact that elements in a tuple in the transition relation uses different projections we supply rows from the read and the write

dependency matrix.

Algorithm 9: REACH-BFS-PREV-RW

Data: RM, WM, MM, K, s^0
Result: $\mathcal{R}$

$WM \leftarrow WM \vee MM$;
$\mathcal{R} \leftarrow \{s^0\}$;
$\mathcal{L} \leftarrow \mathcal{R}$;
for $1 \leq i \leq K$ **do**
  $\mathcal{R}_i^p \leftarrow \varnothing$;
  $\hookrightarrow_i^p \leftarrow \varnothing$;
end
while $\mathcal{L} \neq \varnothing$ **do**
  for $1 \leq i \leq K$ **do**
    $\mathcal{L}^p \leftarrow \rho_i(\mathcal{L})$;
    for $s^p \in \mathcal{L}^p \setminus \mathcal{R}_i^p$ **do**
      $\hookrightarrow_i^p \leftarrow \hookrightarrow_i^p \cup \{(s^p, d^p) \mid d^p \in \text{NEXT-STATE}_i(s^p)\}$;
    end
    $\mathcal{R}_i^p \leftarrow \mathcal{R}_i^p \cup \mathcal{L}^p$;
  end
  $\mathcal{N} \leftarrow \varnothing$;
  for $1 \leq i \leq K$ **do**
    $\mathcal{N} \leftarrow \mathcal{N} \cup \text{STEP}(\mathcal{L}, \hookrightarrow_i^p, RM_i, WM_i)$;
  end
  $\mathcal{L} \leftarrow \mathcal{N} \setminus \mathcal{R}$;
  $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{N}$;
end

Our algorithm that uses chaining is changed similarly.

Algorithm 10: REACH-CHAIN-PREV-RW

Data: RM, WM, MM, K, s^0
Result: $\mathcal{R}$

$WM \leftarrow WM \vee MM$;
$\mathcal{R} \leftarrow \{s^0\}$;
$\mathcal{L} \leftarrow \mathcal{R}$;
for $1 \leq i \leq K$ **do**
  $\mathcal{R}_i^p \leftarrow \varnothing$;
  $\hookrightarrow_i^p \leftarrow \varnothing$;
end
while $\mathcal{L} \neq \varnothing$ **do**
  for $1 \leq i \leq K$ **do**
    $\mathcal{L}^p \leftarrow \rho_i(\mathcal{L})$;
    for $s^p \in \mathcal{L}^p \setminus \mathcal{R}_i^p$ **do**
      $\hookrightarrow_i^p \leftarrow \hookrightarrow_i^p \cup \{(s^p, d^p) \mid d^p \in \text{NEXT-STATE}_i(s^p)\}$;
    end
    $\mathcal{R}_i^p \leftarrow \mathcal{R}_i^p \cup \mathcal{L}^p$;
    $\mathcal{L} \leftarrow \mathcal{L} \cup \text{STEP}(\mathcal{L}, \hookrightarrow_i^p, RM_i, WM_i)$;
  end
  $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{L}$;
  $\mathcal{L} \leftarrow \mathcal{L} \setminus \mathcal{R}$;
end

3.2.2 Symbolic Reachability Implementation in LTSmin

We changed the LDD implementation to support the separation of the read and the write dependencies as follows. We changed the initialization of the cache (INIT-CACHE) to — starting from the bottom, first make a 'write' node for a variable and then make a 'read' node for a variable. The algorithm STEP-LDD

is changed as follows. First matching a node value is done only if a variable has a read dependency. Then again the cache is consulted. If there is a cache miss then it either writes all values in the relation to all nodes in the set if the variable should be written or it continues with the next value if the variable is only read. Then if a variable is both read and write dependent then we move down in the relation. Then if a variable has a write dependency then the actual value is written, else it may continue down the list. Again if the relation has no update or dependency on a variable it simply copies values from below and to the right. To indicate which indices in the state vector have read and write dependencies we provide two vectors; $P_i^R = \langle j \mid DM_{i,j} \in \{\mathtt{r}, \mathtt{W}\}\rangle$ and $P_i^W = \langle j \mid DM_{i,j} \in \{\mathtt{w}, \mathtt{W}\}\rangle$.

Algorithm 11: INIT-CACHE

Result: C

$C \leftarrow \{\epsilon\}$;
$r \leftarrow |P^R| - 1$;
$w \leftarrow |P^W| - 1$;
while $r \geq 0 \vee w \geq 0$ **do**
  if $w \geq 0 \wedge (r = -1 \vee P_w^W \geq P_r^R)$ **then**
    $C \leftarrow \text{NODE}(P_w^W, C, \{\epsilon\})$;
    $w = w - 1$;
  end
  if $r \geq 0 \wedge (w = -1 \vee P_r^R \geq P_w^W)$ **then**
    $C \leftarrow \text{NODE}(P_r^R, C, \varnothing)$;
    $r = r - 1$;
  end
end

Algorithm 12: STEP-LDD

```
Data: C, S, R, s, r, w
Result: L
1  if R = ∅ ∨ S = ∅ then return ∅;
2  if r = |P^R| ∧ w = |P^W| then return S;
3  if P^R_r = s then
4  |  S, R ← MATCH(S, R) ;
5  |  if R = ∅ ∨ S = ∅ then return ∅;
6  end
7  if IN-OP-CACHE(C, S, R) then return
   CACHE-LOOKUP(C, S, R);
8  L ← ∅ ;
9  O ← R ;
10 if P^R_r = s ∨ P^W_w = s then
11 |  if P^R_r ≠ s then
12 |  |  L ← STEP-LDD(C, RIGHT(S), R, s, r, w) ;
13 |  else
14 |  |  L ← STEP-LDD(C, RIGHT(S), RIGHT(R), s,
   |  |  r, w) ;
15 |  end
16 |  if P^R_r = s ∧ P^W_w = s then
17 |  |  R ← DOWN(R) ;
18 |  |  C ← DOWN(C) ;
19 |  end
20 |  if P^W_w = s then
21 |  |  L ← WRITE(C, S, R, L, s, r, w)
22 |  else
23 |  |  L ← READ(C, S, R, L, s, r, w)
24 |  end
25 else
26 |  L ← COPY(C, S, R, s, r, w)
27 end
28 ADD-TO-CACHE(C, S, O, L) ;
```

Algorithm 13: COPY

```
Data: C, S, R, s, r, w
Result: L
1 T_r ← STEP-LDD(C, RIGHT(S), R, s, r, w) ;
2 T_d ← STEP-LDD(C, DOWN(S), R, s, r, w) ;
3 L ← NODE(VAL(S), T_d, T_r)
```

Algorithm 14: WRITE

```
Data: C, S, R, L, s, r, w
Result: L
1  if P^R_r = s then
2  |  r ← r + 1 ;
3  end
4  if P^W_w = s then
5  |  w ← w + 1 ;
6  end
7  while R ∉ {{ϵ}} do
8  |  T ← STEP-LDD(DOWN(C), DOWN(S),
   |  DOWN(R), s + 1, r, w) ;
9  |  T ← NODE(VAL(R), T, ∅) ;
10 |  L ← UNION(L, T) ;
11 |  R ← RIGHT(R) ;
12 end
```

Algorithm 15: READ

```
Data: C, S, R, L, s, r, w
Result: L
1 if P^R_r = s then
2 |  r ← r + 1 ;
3 end
4 if P^W_w = s then
5 |  w ← w + 1 ;
6 end
7 T ← STEP-LDD(DOWN(C), DOWN(S), DOWN(R),
  s + 1, r, w) ;
8 T ← NODE(VAL(R), T, ∅) ;
9 L ← UNION(L, T) ;
```

3.3 PINS2PINS wrappers

With the three definitions introduced in Section 3.1 we introduce three new operations on the dependency matrix. Also we show how the existing row subsumption operation on the dependency matrix is changed. Changing the row subsumption operation is not trivial, because `w` may not subsume `-`.

3.3.1 New Dependency Matrix Operations

We introduce three new operations on the dependency matrix, namely `row-elm`, `col-elm` and $w < r$. The first two algorithms can remove rows and columns from the dependency matrix respectively. The algorithm $w < r$ can sort columns and rows in the dependency matrix more efficiently. We only illustrate how these algorithms should work and in which cases they benefit reachability analysis. We leave the implementation of these algorithms to future work.

Definition 3.12 (Row Elimination)**.** We can remove rows from the DM if there are only reads in those rows and one is only interested in the state space, not the transitions. The `row-elm` function returns al

row indices that can be removed from the DM:

$$\texttt{row-elm} = \{0 \leq i \leq K \mid \forall 0 \leq j \leq N \colon DM_{i,j} \in \{\texttt{-}, \texttt{r}\}\}.$$

■

One example in which this operation may prove useful is the Sokoban game. In a Sokoban game there are typically transition groups which check if the state of the game is 'finished'. This requires checking if all the blocks are on goal positions. Computing this often takes a long time. Performing the `row-elm` operation on the dependency matrix removes these checks. Note however that in the next chapter we will show a guard-splitting algorithm which can efficiently check whether all the blocks are on goal positions. With this algorithm we do not necessarily have to remove rows from the dependency matrix for a Sokoban model.

Definition 3.13 (Column Elimination)**.** We can remove columns from the DM if there are only writes in those columns and one is interested in the bisimulation of the original PTS. The `col-elm` function returns al column indices that can be removed from the DM:

$$\texttt{col-elm} = \{0 \leq j \leq N \mid \forall 0 \leq i \leq K \colon DM_{i,j} \in \{\texttt{-}, \texttt{w}, \texttt{W}\}\}.$$

■

There does not come one specific model to mind in which the `row-elm` operation can be useful. However one can imagine that performing reachability analysis on a bisimilar model is greatly improved w.r.t. to time and space.

The third algorithm that can be implemented is an algorithm named '$w < r$' that sorts columns in such a way that $\{\texttt{w}, \texttt{W}\}$ dependencies in the DM are put before `+` dependencies, `+` dependencies are put before `r` dependencies and `r` dependencies are put before `-` dependencies. Future work may show if the relational product operation in BDDs profit from this sorting algorithm.

3.3.2 Row Subsumption

Since `-` dependencies are not read and `w` does not allow copying values, `w` dependencies may not subsume `-` dependencies. In this section we describe how we can make row subsumption work with our new definitions for (in)dependence. The issue for row subsumption is illustrated in the following example.

Example 3.14 (row subsumption for `w`)**.** Consider the process P:

$$\begin{aligned} P(x, y, z) = & \\ & x \implies a.P(x, 1, 1) + \\ & x \implies b.P(x, y, 2), \end{aligned}$$

with initial state $P(1, 0, 0)$.

The DM DM of process P is:

$$\begin{array}{c} \\ a \\ b \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{bmatrix} \texttt{r} & \texttt{w} & \texttt{w} \\ \texttt{r} & - & \texttt{w} \end{bmatrix} \end{array}.$$

The state space of this process is as shown in Figure 3.3.

We can now show that if we let a `w` state slot subsume a `-` state slot we do not produce the same state space. The DM with row subsumption applied DM' is:

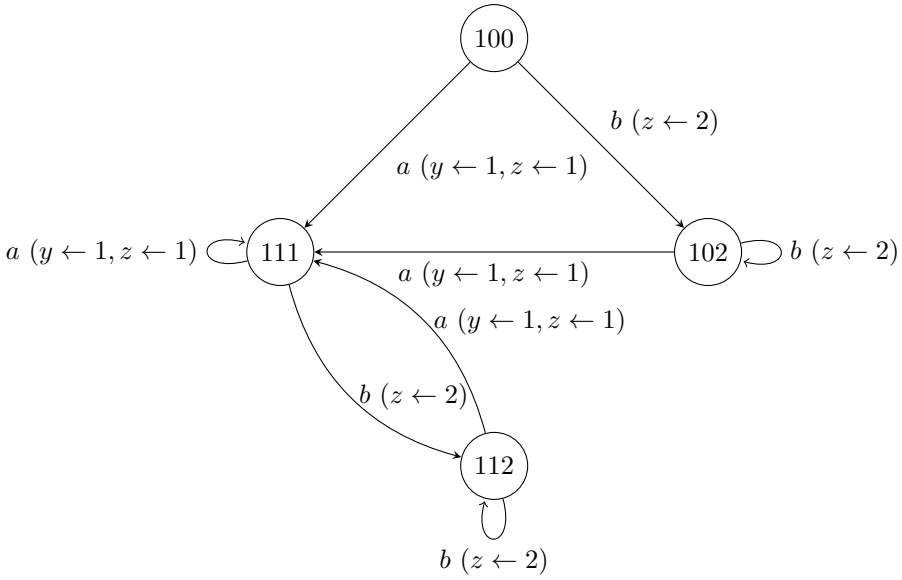


Figure 3.2: State space of process P using DM

$$\begin{array}{c} \\ a \\ b \end{array} \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{bmatrix} \mathtt{r} & \mathtt{w} & \mathtt{w} \\ \mathtt{r} & \mathtt{w} & \mathtt{w} \end{bmatrix} \end{array}.$$

With this DM the dependency of state slot y in transition group b changes into a `w`. This means that transition group b will always write a value from the initial state. Therefore the state space of P using DM' is as follows.

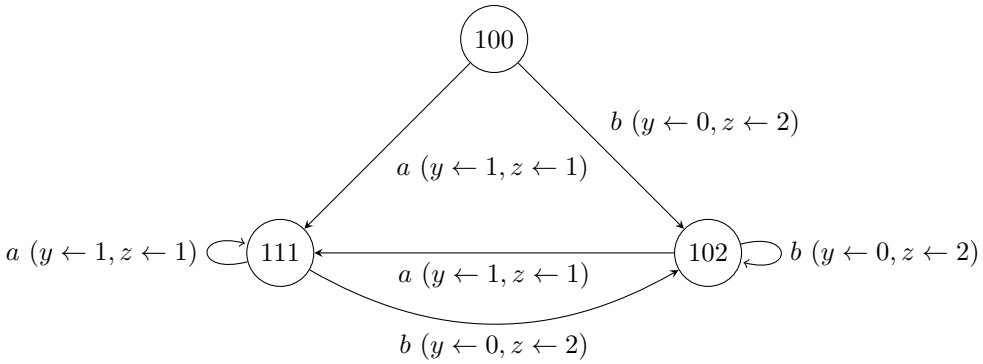
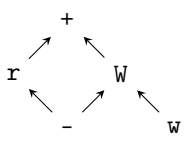


Figure 3.3: State space of process P using DM'

Because the two state spaces are obviously not equal; `w` may not subsume `-`.

▲

So we can not put a total ordering on the dependencies as we did in Chapter 1. We can however put a partial ordering on the dependencies as follows, such that `w` can not subsume `-`:



3.4 Implementation in LTSmin

To support separating dependencies in LTSmin we have changed all three major components in LTSmin. We have changed the Pins front-end, wrappers and back-end. In the front-end we have changed the `GBgetTransitionsShort` and `GBgetTransitionsLong`. In the wrappers we have changed how transformations can be done on three DMs. In the back-end we have changed how the transition relation is applied on a set of states using both read and write dependencies.

3.4.1 The Pins front-end

In Chapter 1 we stated that the `GBgetTransitionsShort` uses the projection π. We changed this to support both the ρ and ω projection. So now a symbolic back-end has to ask successor states for sets of states projected with ρ. The Pins front-ends mostly only implement the `GBgetTransitionsLong` function so we have to expand the short vector to a long vector with more values from the initial state. This is because in most cases short vectors projected with ρ are shorter than short vectors projected with π. Also a successor state given by a result of `GBgetTransitionsLong` is projected in the current situation with π. We now project successor states with ω. In our work we did not implement any RDM or WDM for any language front-end, because this was already done. However, with the definitions presented in this chapter it should be stated that one can get the RDM with `GBgetDMInfoRead` and the WDM with `GBgetDMInfoWrite`.

3.4.2 The Pins2Pins wrappers

Much work is done on DM operations. Two key problems are solved in this area. The first problem is that, since we have two new matrices we have to apply any transformation on three matrices. Namely the classic DM from Chapter 1, the RDM and the WDM. Determining which transformations to do and actually do them is a big architectural issue in the Pins2Pins wrapper. This issue can be solved in one of two ways.

1. The first solution to see which transformations we can do is as follows. For every transformation we create a new matrix. More specifically if we do an operation on rows we create a new matrix of length $N * 3$. If we do an operation on columns we create a matrix of height $K * 3$. The values of the RDM and WDM can be encoded in binary form in the new matrix. This is convenient because with this encoding we do not have to change the cost function or the subsumption functions as described in Chapter 1. The encoding for a state slot j and dependencies $\{\texttt{-}, \texttt{w}, \texttt{W}, \texttt{w}, \texttt{+}\}$ is as follows. The column r stands for read, w stands for write and c stands for copy.

$$\begin{array}{c} \\ \texttt{-} \\ \texttt{r} \\ \texttt{w} \\ \texttt{W} \\ \texttt{+} \end{array} \begin{array}{c} \begin{array}{ccc} r & w & c \end{array} \\ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \end{array}$$

   Note that this encoding precisely implements the partial ordering on dependencies described in Section 3.3.2. The disadvantage of this approach is that after every transformation we have to recreate an $N * 3$ or a $K * 3$ matrix.

2. The second solution is to compare values of all three matrices (RDM, WDM and MDM) and transforming each matrix individually. This approach makes comparing values in the matrices a little complex but we do not have to recreate an $N * 3$ or a $K * 3$ after each transformation. So we implemented this latter approach.

3.4.3 The PINS back-end

Implementing the separation of dependencies does not require many changes in the symbolic back-end. When initializing the sets (calling `init_domain`) we changed the projection from π to ρ and when initializing the transition relation we added both the ρ and ω projection to the relation. The second change we made was letting the LDD implementation of `vset_next` use the ρ and ω projection.

3.4.4 Compatibility

Our implementation of separating dependencies does not pose severe backward or forward compatibility issues between language front-ends such as MCRL2 or PROMELA and the PINS interface. The reason herefore is that if a language front-end does not support either the RDM or the WDM the symbolic back-end uses the DM as both the RDM and the WDM. This can be done because every dependency can be over-approximated to a `+`. The only issue w.r.t. compatibility is that the `vset_prev` for LDD is not implemented. This means that currently one can for example not find traces to certain states. Implementing `vset_prev` should not take more than a few hours.

3.4.5 Reproducablity

The experiments we did with our changes can be easily reproduced. To benchmark our changes we will first explain how to reproduce an environment for the current situation. Next we will explain how to set up an environment with our changes.

Environment for current situation

First compile and install LTSMIN as described here: `http://fmt.cs.utwente.nl/tools/ltsmin/`. When cloning the git repository use however the git repository at `git@github.com:Meijuh/ltsmin.git` and checkout the *next* branch. This version namely contains a fix w.r.t. caching in the `mdd_next` function. Install LTSMIN with support for DIVINE, PROMELA and MCRL2. Or install a subset of these languages if you do not want to reproduce experiments with all these languages. If you want to do experiments with MCRL2 make sure to install version 2012-10. Running an MCRL2 experiment can be done with the command `lps2lts-sym --mcrl2="--rewriter=jitty" /some-path/1394-fin.lps`.

Environment with separated dependencies

First compile and install LTSMIN as described in the previous section. Use however the *rw* branch in `git@github.com:Meijuh/ltsmin.git`. Also install a patched version for DIVINE from the *rw* branch in `git@github.com:Meijuh/Divine2.git`. To run a DIVINE experiment with our changes execute for example first the following command: `divine compile -lW iprotocol.5.dve`. This will compile a DVE model for LTSMIN with `W` dependencies over-approximated to `+`. Next you can execute the following command to perform symbolic reachability analysis on this model specification: `dve2lts-sym /some-path/iprotocol.5.dve2C`. When running PROMELA experiments you must also over-approximate `W` dependencies, thus run the command `spins -W model.prom` instead of `spins model.prom`.

3.5 Benchmarks

For a quick overview of our results please refer to the scatter plots in Figures 3.4 (run time) and 3.5 (memory usage) on Pages 43 and 44. Detailed information about our experiments are clearly given in tables, these can be found in appendix C.1 on Page 77. Note that appendix C.1 contains a small subset of our results. In this appendix only results for MCRL2 with chain-prev;no-sat are given. All other results can be found online, see 'Open online access to our benchmarking results' on Page 7.

One mCRL2 model that clearly benefits of separated dependencies is a model of the firewire protocol, i.e. 1394-fin.lps. This model has an interesting dependency matrix, because many state slots have only a 'w' dependency. Listing 3.2 shows a snapshot of this dependency matrix. Because the ρ projection does not project to 'w' state slots the result is fewer NEXT-STATE calls and shorter transition relations. Another mCRL2 model that benefit of our changes is a large variation of our discussed 1-safe Petri net in Figure 1.2. This model is vasy.lps. Note that vasy-init.lps is mostly the same model. In vasy-init.lps the complex first step is omitted. With the results of these two models we see that the Petri net can be computed with fewer NEXT-STATE calls. The Petri net also has smaller transition relations, because the plot of vasy-init.lps is put a little lower than the $y = x$ line. This is confirmed when looking at the detailed information in Table C.1 and Table C.2. With our changes the transition relation has 1374 nodes less.

Listing 3.2: Partial dependency matrix for 1394-fin.lps

```
...
23: ++ww+-rww------------+wwwwwwwww-w
24: ++ww+-rww------------+wwwwwwwww-w
25: ++ww+-rww------------+wwwwwwwww-w
26: ++ww+-rww------------+wwwwwwwww-w
27: ++ww+-rww------------+wwwwwwwww-w
28: ++ww+-rww------------+wwwwwwwwww-
29: ++ww+-rww------------+wwwwwwwwww-
30: ++ww+-rww------------+www+-wwwww-
31: ++ww+-rww------------+w+ww-wwwww-
32: ++ww+-rww------------+wwww-wwwww-
33: ++ww+-rww------------+wwwwwwwwww-
34: ++ww+-rww------------+wwwwwwwww--
35: ++ww+-rww------------+wwwwwwwww--
...
```

The scatter plots on Page 43 show a small speedup (-29 seconds) for cambridge.7.dve2C. In the results that can be found online one can see that the transition relation and the set of projected states are a little smaller (-1%), this is interesting because the speedup is more than 1%. Another interesting aspect of this model is the dependency matrix. Listing 3.3 shows a well represented snapshot of the original dependency matrix. One can indeed see that the few `w` dependencies indeed result in the decrease of the size of the transition relation and the set of projected states. Even more interesting is the series of `+` dependencies. These dependencies are actually `W` dependencies over-approximated to a `+`. In future work where we support copying values, e.g. by reserving special node values in the symbolic back-end we might get a much better result. Many DVE and also PROMELA models contain buffers and channels modeled as an array. Therefore cambridge.7.dve2C is not at all the only model which may profit from support for copying values.

Listing 3.3: Partial dependency matrix for cambridge.7.dve2C

```
...
15: ++r-----------------------------------------------
16: +rr-----------------------------------------------
17: ---+--++++++++++++++++++++++-----------------------
18: ---+-w++++++++++++++++++++++-----------------------
19: ---+-w++++++++++++++++++++++-----------------------
20: ---+-w++++++++++++++++++++++-----------------------
21: ---+-w++++++++++++++++++++++-----------------------
22: ---+w-++++++++++++++++++++++-----------------------
23: ---+--++++++++++++++++++++++-----------------------
24: ---+-w++++++++++++++++++++++-----------------------
25: ---+--++++++++++++++++++++++-----------------------
```

```
26: ---+-w++++++++++++++++++++-----------------------
27: ---+-w++++++++++++++++++++-----------------------
28: ---+rr-----------------------------------------------
29: ---+rr-----------------------------------------------
30: ---++r-----------------------------------------------
31: ---+rr-----------------------------------------------
32: ---++r-----------------------------------------------
...
```

In our scatter plots we only show one PROMELA model. There are many PROMELA models which do not significantly benefit of our changes. There are however exceptions, for example the elevator2.3.prom.spins model. The results for this model can be found online (see 'Open online access to our benchmarking results', Page 7).

The PROMELA compiler for LTSMIN appears to have a bug in the `-W` program option. This option should add a read dependency for may-write (`W`) dependencies. In the case were the PROMELA compiler performs internal reachability analysis with `spins_simple_reach` for atomic statements a problem occurs. Some `W` dependencies are namely not correctly over-approximated. So, all the models on which internal reachability analysis is performed we do not get correct state spaces. These results are thus omitted from our figures. We might actually get interesting results if this bug is fixed in the PROMELA compiler.

Separating dependencies should in theory always produce faster computations. This appears to be indeed the case. However, most of the results are not significant. There are probably two reasons herefore. First, we find that the dependency matrices of most models are already sparse. Second, we believe that support for copying values by reserving a special value in the transition relation may produce great speed ups. Because dependency matrices are already sparse we might see better results if we re-run our experiments with optimizations to the dependency matrix such as row subsumption. Row subsumption combines transition relations into a larger ones. However, since we remove nodes from the transition relation with our changes, these relations may actually become notably smaller.

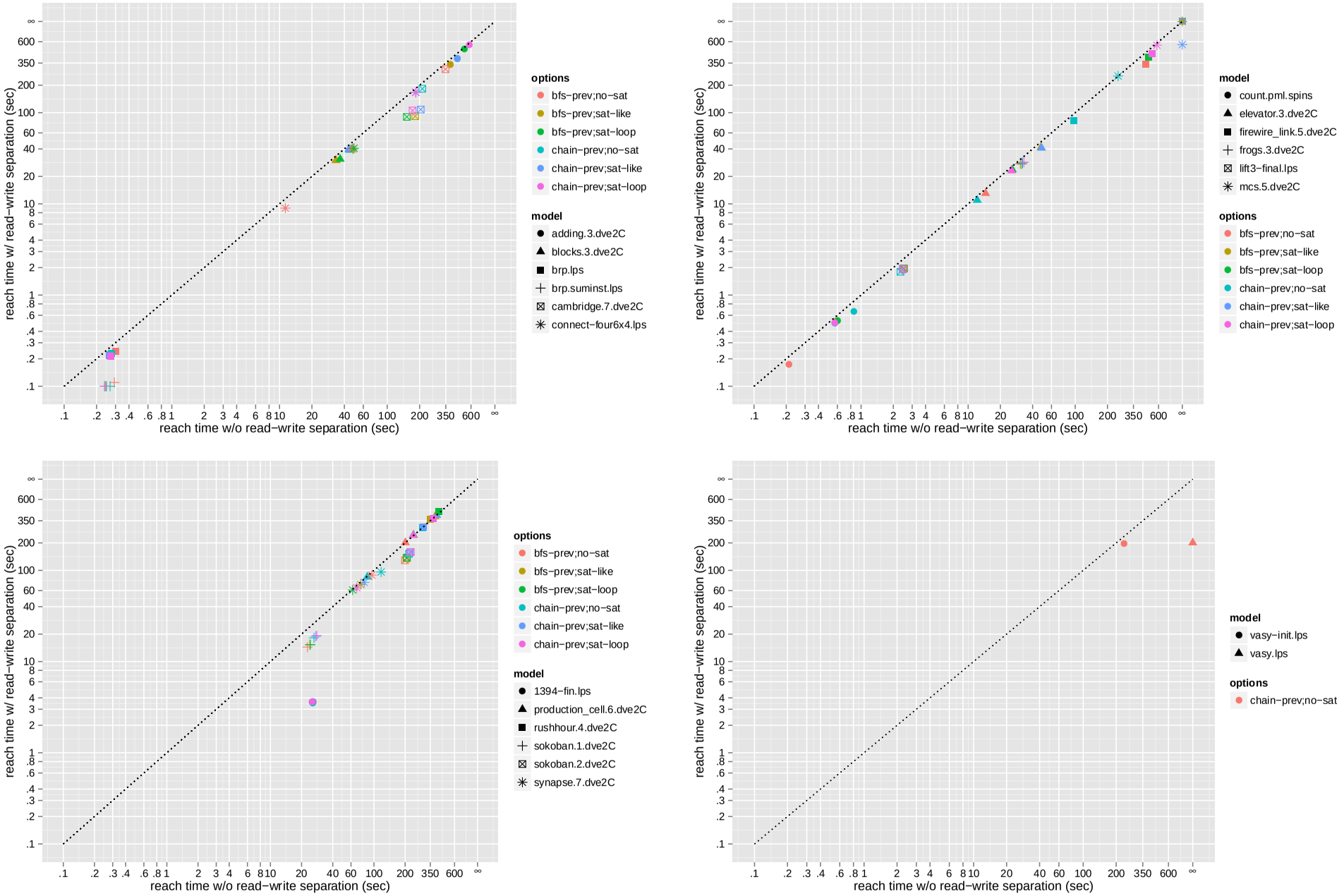


Figure 3.4: Time scatter plots 1 – 20 for with read/write separation

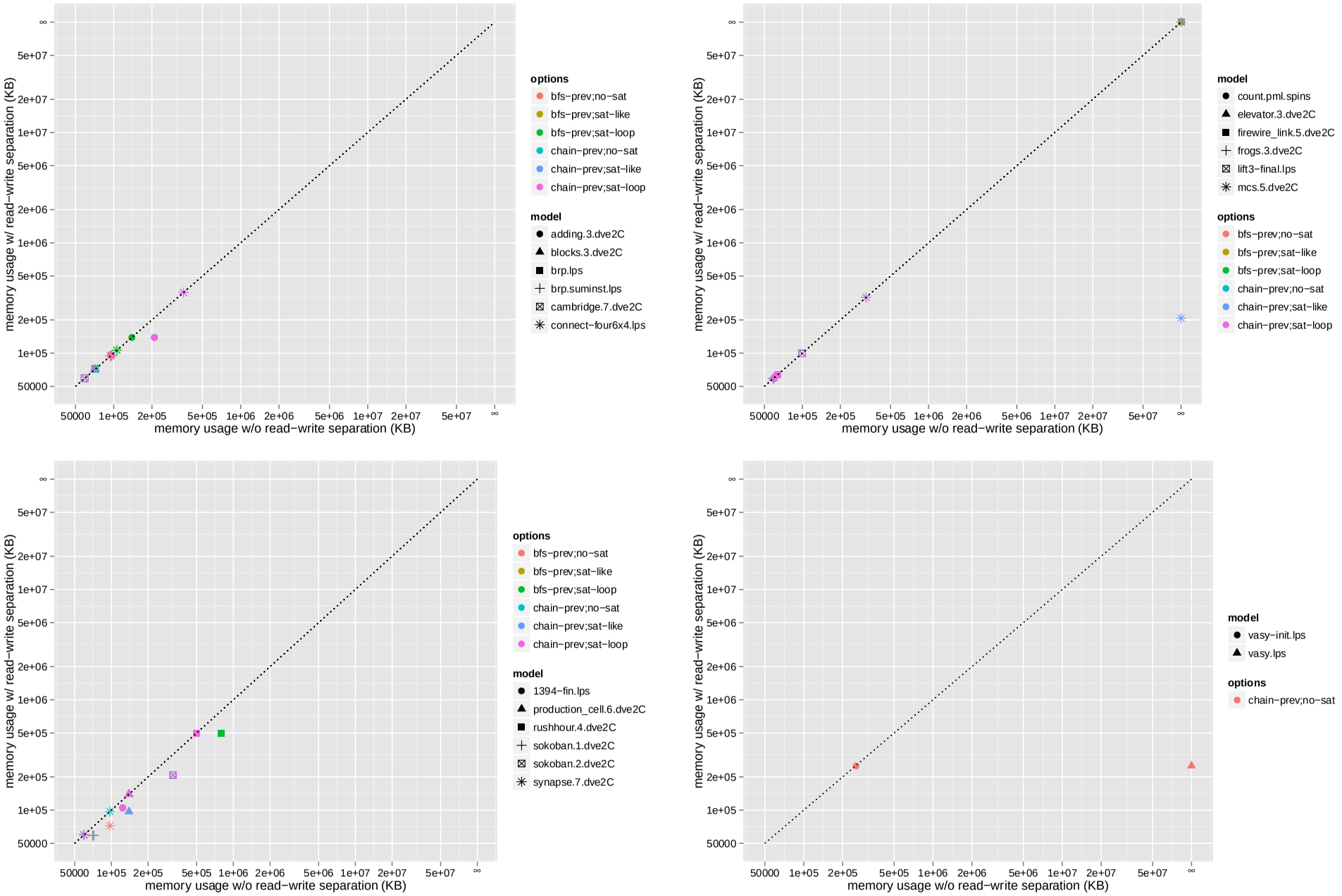


Figure 3.5: Memory scatter plots 1 – 20 for with read/write separation

3.6 Future Work

On the subject of separating dependencies there is still some work to be done on both the back-ends and front-ends for LTSmin.

- Support for may-write dependencies in the symbolic back-ends, such that we do not have to over-approximate these dependencies to a `+`. With this improvement we may see better results because we can save many NEXT-STATE calls. Consider for example Listing 3.3. This dependency matrix contains lots of `+` dependencies, which are actually `W` dependencies. If we reserve a special node value for copying in the transition relation then we do not have to project to these state slots and thus get less states to compute successors for. Furthermore if we reserve a value for copying then we get even smaller transition relations.
- Fix a bug in Promela where may-write dependencies are not correctly over-approximated to a `+` when internal reachability is performed.
- Implement the `vset_next` function in all symbolic back-ends other than LDD.
- Implement the `vset_prev` function in all symbolic back-ends.
- Fix `vset_least_fixpoint` to support separated read and write dependencies.
- Investigate how saturation can profit from separated read and write dependencies.
- Re-do the experiments with optimizations to the dependency matrix. We expect improvements when for example row subsumption is used.

3.7 Conclusion

The problem we approached in this chapter involves separating dependencies. Instead of simply looking at whether a transition group is dependent on a state slot. We separate this into either a read or write dependency. By handling these two dependencies distinctly we have shown that we can improve the time and memory usage for the computation of the state space. Two key changes to symbolic reachability algorithms entail these improvements. The first improvement is that using a projection (ρ) which only looks at read dependencies we can greatly reduce the amount of NEXT-STATE calls. Second, it is indeed possible to remove read nodes from the transition relation if a state slot that is represented by that read node is not read. Similarly we can remove write nodes if the state slot is not written. This latter change results in smaller transition relations. If we would re-do our experiments with optimizations to the dependency matrix, such as row subsumption we expect even better results. Removing nodes from the transition relation namely reduces the chance of a blow up in the number of nodes when a bad variable ordering is used. The effects of this blow up is probably reduced even more when row subsumption is used.

Our benchmarks show that every computation of state spaces benefit of our improved algorithms. If a model has `w` (must-write) dependencies in the dependency matrix then the transition relation is smaller. This reduces the memory footprint as well as the time needed to compute the state space. Even better; a model of a 1-safe Petri net shows a massive speed up. This is due to the fact that the amount of NEXT-STATE calls is significantly reduced to the extent that the model suddenly becomes tractable for LTSmin.

Because Promela and Dve models often use buffers and channels modeled as an array, it is essential that we provide better support for `W` (may-write) dependencies. We can do this by reserving a special node value in the symbolic back-ends to support copying values. We believe that this improvement will show great speed ups for many Promela and Dve models.

Working on separating dependencies was harder than expected in our research proposal. This is due to the fact that Partial Order Reduction uses a different notion for write dependent than which is usable in symbolic algorithms. The amount of time we spent on this issue prevented us from examining how saturation can benefit from our work described in this chapter. We expect however that also saturation

can greatly benefit of our work here. An implementation of saturation such as `reach_sat_like` tries to compute successors for subsets of transition groups. The saturation implementation tries to only compute successors for transition groups that could possibly have successors because of interleaved dependencies between transition groups [9]. This is opposed to our algorithm REACH-CHAIN-PREV where only every transition group is used to compute successors. When looking at interleaved dependencies saturation looks at the dependency matrix described in Chapter 1. This means that saturation does not look at separate read and write dependencies. It thus assumes that every dependent state slot writes. If saturation would look at our more fine-grained notion of dependencies it may conclude that some transition groups may not have successors because the interleaved dependencies are actually not write dependencies.

Chapter 4

Guard-based Symbolic Reachability

In the previous chapter we saw how improved notions of dependencies can speed up symbolic reachability analysis. This chapter focuses less on dependencies. In this chapter we show how we can extend reachability algorithms REACH-BFS-PREV and REACH-CHAIN-PREV with guard-splitting. Guard-splitting involves evaluating individual conjuncts from a condition separately from the NEXT-STATE computation.

4.1 Background

For this chapter we need to define a new operation on indexed sets. For this an *indexed set* for some domain D is defined as follows. Typically in PINS the domain is the set of natural numbers ($\mathbb{N}$).

Definition 4.1 (indexed set)**.**

$$D^I \stackrel{\text{def}}{=} \prod_{i \in I} D_i,$$

where I is the index set. ■

The join relation between to sets is defined as follows.

Definition 4.2 (join on indexed sets)**.** Let I_1, I_2 be index sets and $S_1 \subseteq D^{I_1}$, $S_2 \subseteq D^{I_2}$ be indexed sets then

$$S_1^{I_1} \bowtie S_2^{I_2} = \{s \in D^{I_1 \cup I_2} \mid \Pi^{I_1}(s) \in S_1 \wedge \Pi^{I_2}(s) \in S_2\}.$$

■

Example 4.3. Let $D = \{a, b, c\}$, $I = \{1, 2, 3, 4, 5\}$, $S_1 = \{(a, a, a), (a, b, c), (b, b, b)\}$, $S_2 = \{(a, a, a), (a, b, c), (b, b, b)\}$, $I_1 = \{1, 2, 3\}$, $I_2 = \{3, 4, 5\}$ then $S_1 \bowtie S_2 = \{(a, a, a, a, a), (a, a, a, b, c), (b, b, b, b, b)\}$. ▲

In Section 4.5.1 we will show how the the join operation can be used on symbolic sets rather than on indexed sets.

4.2 The Monolithic Next-State Interface

We first extend our definition of TSs with guards. We do this so that we can also extend the definition of PTSs with guards.

Definition 4.4 (Transition System)**.** A TS is a structure $\langle S, \rightarrow, s^0, G \rangle$, G are the set guards. A guard is a total function that maps a state to *false*, *maybe* or *true*, i.e. $g : S \rightarrow \{-, ?, +\}$. We write $g(s)$ to denote the evaluation of the guard g in state $s \in S$. The logical $\wedge$ of two guards is defined as

$$\begin{array}{c|cccc} \wedge & - & ? & + \\ - & - & - & - \\ ? & - & ? & ? \\ + & - & ? & + \end{array}.$$

■

So, a guard can evaluate to − (false), + (true) or ? (maybe). Why we need to support that guards can evaluate to maybe is illustrated in Example 4.7.

4.3 The Partitioned Next-State Interface

Here we extend the definition of PTSs from Chapter 1 with guards. With the notion of guards we provide two new dependency matrices for PTSs. Furthermore we show how a variation on the NEXT-STATE function can benefit of these matrices.

Definition 4.5 (Partitioned Transition System)**.** A PTS is a structure $\mathcal{P} = \langle S_\mathcal{P}, \hookrightarrow_\mathcal{P}, s^0_\mathcal{P}, \langle\langle g_1, \ldots, g_M\rangle, \langle G_1, \ldots, G_K\rangle\rangle\rangle$. $G_\mathcal{P} = \langle \Gamma, G\rangle = \langle\langle g_1, \ldots, g_M\rangle, \langle G_1, \ldots, G_K\rangle\rangle$. The vector $\langle g_1, \ldots, g_M\rangle$ contains the guards and the vector $\langle G_1, \ldots, G_K\rangle$ contains sets of indices $(\forall 1 \leq k \leq K \colon G_k \subseteq \mathbb{N}^+_{\leq M})$ for the vector of guards. A transition group $\rightarrow_\mathrm{i}$ is *guarded* by some vector $G' = (\langle g_1, \ldots, g_M\rangle_k)_{k \in G_i}$ if

1. $\forall s \in R : (\exists k \in G_i : g_k(s) = -) \implies \nexists s' \in R : (s, s') \in \rightarrow_\mathrm{i}$, i.e. the existance of a *false* guard implies there is no successor state.
2. $\forall s \in R : (\forall k \in G_i : g_k(s) = +) \implies \exists s' \in R : (s, s') \in \rightarrow_\mathrm{i}$, i.e. all guards *true* implies at least once successor state.
3. $\forall s \in R : (\exists k \in G_i : g_k(s) = ?) \implies \exists k' \in G_i \colon k \neq k' \wedge g_{k'}(s) = -$, i.e. if a guard evaluates to *maybe* then there must be another guard that evaluates to *false*, i.e. the guarded transition does not depend on the evaluation of g_k.

The defined TS of $\mathcal{P}$ is $\langle S_\mathcal{P}, \rightarrow_\mathcal{P}, s^0, G_\mathcal{P}\rangle$.

■

We have extended the definition of PTSs with two new vectors. One vector contains the guards which are functions on states. The other vector contains sets of indices to denote which guard belongs to which transition group. Typically in modeling languages a guard is a single conjunct in the condition of a transition group. A guard can be evaluated separately from the NEXT-STATE function. The vector $\langle G_1, \ldots, G_K\rangle$ indicates for each transition group which guard is conjunctively bound to which other guards. The following example shows how we can identify guards in the 1-safe Petri net of Figure 1.2.

Example 4.6 (Guards in a PTS)**.** We denote the PTS $\mathcal{P}$ of the Petri net as follows.

$$\begin{aligned} \mathcal{P} = \langle &\{\langle 1,0,0,0,0\rangle, \langle 0,1,0,1,0\rangle, \langle 0,0,1,1,0\rangle, \langle 0,1,0,0,1\rangle, \langle 0,0,1,0,1\rangle\}, \\ &\{(\langle 1,0,0,0,0\rangle, \langle 0,1,0,1,0\rangle), \ldots\}, \langle 1,0,0,0,0\rangle, \\ &\langle\langle [P0], [P1], [P2], [P3], [P4]\rangle, \langle \{0\}, \{1\}, \{2\}, \{3\}, \{2,4\}\rangle\rangle\rangle, \\ &\text{with } N = 5,\ K = 6 \text{ and } M = 5. \end{aligned}$$

▲

In this example it may not be clear why a guard can evaluate to maybe. Consider Example 4.7.

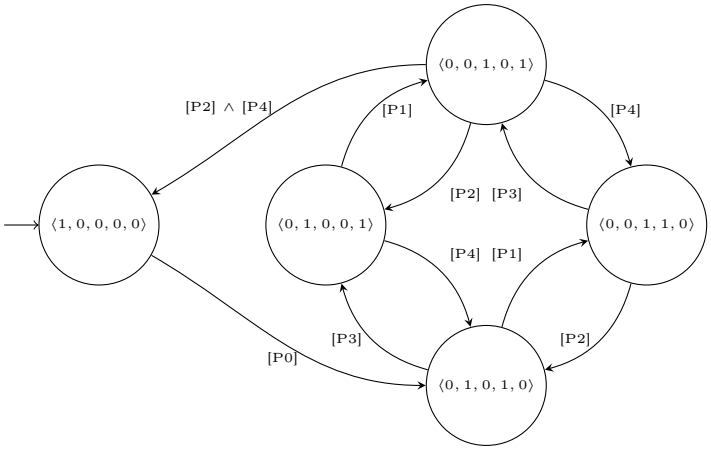


Figure 4.1: Example partinioning of the Petri net in Figure 1.2 with guards, transition groups are omitted.

Example 4.7 (Guard evaluation to maybe)**.**

$$\text{someterm} \wedge \text{false} \quad (1)$$

$$\text{if (p != null \&\& p.method()) \{ // next state \}} \quad (2)$$

▲

Example 4.7.1 is a term with two conjuncts: 'someterm' and 'false'. If a term rewriting system such as mCRL2 tries to evaluate 'someterm' as a guard than the result will be 'someterm'. Which we denote as ? since the truth value of the whole term depends one the evaluation of 'false'. The second example is syntax we can find in languages such as Java. Since every element in the vector $\langle G_1, \ldots, G_K \rangle$ is a set we can evaluate the guard 'p.method()' before we evaluate 'p != null'. If 'p' is a null pointer then 'p.method()' can not evaluate to true or false. A language front-end should therefore evaluate it to maybe.

4.3.1 State slot dependencies

With our notion of guards we provide two new definitions of independence which closely relate to the read-independent (Definition 3.1). Along with these definitions we will also give their respective dependency matrices and projections. With this information we can efficiently evaluate guards and compute successor states.

Definition 4.8 (guard-independent [7])**.** Given a PTS $\mathcal{P} = \langle S_\mathcal{P}, \rightarrow_\mathcal{P}, s^0, G_\mathcal{P} \rangle$ state label i is independent on state slot j if $\langle g_1, \ldots, g_i, \ldots, g_M \rangle$ for all $\langle s_1, \ldots, s_N \rangle \in R$, whenever $\exists v \in \{-, ?, +\}. g_i(\langle s_1, \ldots, s_j, \ldots, s_N \rangle) = v$ then

1. for all $r_j \in S_j$, we also have $g_i(\langle s_1, \ldots, r_j, \ldots, s_N \rangle) = v$, i.e. the value of state slot j is not relevant for the evaluation of state label i.

■

This definition says that if we have a state s and a state slot j we should be able to assign any value to state slot j and not change the outcome of the evaluation of the guard. In practice we make guard i dependent on state slot j if the variable that is represented by state slot j occurs in guard i.

The Guard Dependency Matrix (GDM) is defined as follows.

Definition 4.9 (guard dependency matrix)**.** A GDM $G_{M \times N} = GDM(\mathcal{P})$ for PTS $\mathcal{P}$ is a matrix with M rows and N columns containing $\{0, 1\}$ such that if $G_{i,j} = 0$ then guard i is *independent* on state slot j. For any guard $1 \leq i \leq M$, we define γ_i as the projection $\gamma_i : S \rightarrow \Pi_{\{1 \leq j \leq N | G_{i,j} = 1\}} S_j$. ■

Recall Definition 1.5; the definition of an LPS in mCRL2. The GDM for LPS X $GDM(X) = GM^X_{M\times N}$, is defined as follows.

$$GM^X_{k,j} = \begin{cases} 1 & \text{if } x_j \text{ occurs in } g_k, \\ 0 & \text{otherwise.} \end{cases}$$

Example 4.10 (Guard Dependency Matrix)**.** The GDM for the the Petri net in Figure 1.2 is:

$$\begin{array}{c} \\ [P0] \\ [P1] \\ [P2] \\ [P3] \\ [P4] \end{array} \begin{array}{c} \begin{array}{ccccc} P0 & P1 & P2 & P3 & P4 \end{array} \\ \left[\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

and $\gamma_{[P0]}(\{\langle 1,0,0,0,0\rangle, \langle 0,1,0,1,0\rangle, \langle 0,0,1,1,0\rangle, \langle 0,1,0,0,1\rangle, \langle 0,0,1,0,1\rangle\}) = \{\langle 0\rangle, \langle 1\rangle\}$.

▲

In the above example one can see that if we evaluate a guard for two states $\{\langle 0\rangle, \langle 1\rangle\}$ we know for which states in the original PTS the guard holds.

Definition 4.11 (update-independent)**.** Given a PTS $\mathcal{P} = \langle S_\mathcal{P}, \rightarrow_\mathcal{P}, s^0\rangle$ transition group i is *update-independent* on state slot j if for all $\langle s_1, \ldots, s_N\rangle, \langle t_1, \ldots, t_N\rangle \in S_\mathcal{P}$, whenever $\forall k \in G_i \colon g_k(\langle s_1, \ldots, s_N\rangle) \wedge \langle s_1, \ldots, s_j, \ldots, s_N\rangle \rightarrow_\mathrm{i} \langle t_1, \ldots, t_j, \ldots, t_N\rangle \in \rightarrow_\mathcal{P}$ it holds that $(s_j = t_j \wedge \forall r_j \in S_j \colon \langle s_1, \ldots, r_j, \ldots, s_N\rangle \rightarrow_\mathrm{i} \langle t_1, \ldots, r_j, \ldots, t_N\rangle) \vee (\forall r_j \in S_j \colon \langle s_1, \ldots, r_j, \ldots, s_N\rangle \rightarrow_\mathrm{i} \langle t_1, \ldots, t_j, \ldots, t_N\rangle)$. I.e. the value of state slot j is not relevant in transition group i. Whether some transition group i and state slot j is not update-independent may be over-approximated. ■

This definition is thus very similar to Definition 3.1. However, we do not look at dependencies for guards. In practice the corresponding dependency matrix is very similar to the RDM but with the dependencies for the guards omitted. A definition and example for the Update Dependency Matrix (UDM) for mCRL2 will be given in Subsection 4.4.2.

Definition 4.12 (update dependency matrix)**.** A UDM $UM_{K\times N} = UDM(\mathcal{P})$ for PTS $\mathcal{P}$ is a matrix with K rows and N columns containing $\{0,1\}$ such that if $UM_{i,j} = 0$ then transition group i is *update-independent* on state slot j. For any transition group $1 \leq i \leq K$, we define υ_i as the projection $\upsilon_i : S \rightarrow \Pi_{\{1\leq j\leq N \mid UM_{i,j}=1\}} S_j$. ■

Using the projection functions υ_i and ω_i we can define the NEXT-UPDATE function for a PTS. That is, we base the definition on the NEXT-STATE function and accept only projected states which are not *update-independent* on a transition group.

Definition 4.13 (Partitioned Next-Update function)**.**

$$\text{NEXT-UPDATE}_i(s \in \upsilon_i(S_\mathcal{P})) = \{s' \in \omega_i(S_\mathcal{P}) \mid s \rightarrow_\mathrm{i} s'\}$$

Note: calling the NEXT-UPDATE function requires that all the relevant guards for s evaluate to $+$. ■

For writing convenience we change the definition of the DM as follows.

Definition 4.14 (dependency matrix)**.**

$$DM_{i,j} = \begin{cases} \texttt{-} & \text{if } WM_{i,j} = 0 \text{ and } UM_{i,j} = 0 \text{ and } MM_{i,j} = 0, \text{ else} \\ \texttt{r} & \text{if } WM_{i,j} = 0 \text{ and } MM_{i,j} = 0, \text{ else} \\ \texttt{W} & \text{if } UM_{i,j} = 0 \text{ and } WM_{i,j} = 0, \text{ else} \\ \texttt{w} & \text{if } UM_{i,j} = 0 \text{ and } MM_{i,j} = 0, \text{ else} \\ \texttt{+} & . \end{cases}$$

■

The writing convenience follows from the fact that dependencies from the guards are not included.

4.4 The Pins front-end

To support guard-splitting for an LPS we refine Definition 1.5; the definition of an LPS. The definition shows how the vector Γ and the vector G for an LPS is built.

Definition 4.15 (Linear Process System)**.**

$$X(x_1,\ldots,x_N) = \sum_{i=1}^{K} \sum_{e_i \in E_i} C_i \implies a(t_{i,0}).X(t_{i,1},\ldots,t_{i,N}) =$$
$$\sum_{i=1}^{K} \bigwedge_{g \in \mathcal{G}_i} g \implies \sum_{e_i \in E_i} C'_i \implies a(t_{i,0}).X(t_{i,1},\ldots,t_{i,N}),$$

where g is a boolean expression over $x_1,\ldots,x_N$. Note that indeed $C_i = \bigwedge_{g\in\mathcal{G}_i} g \wedge C'_i$, for a summand i. But not always $C_i = \bigwedge_{g\in\mathcal{G}_i} g$, because an expression e_i in the summation $\sum_{e_i\in E_i}$ may introduce local variables which occur in C_i. More precisely

$$\forall e_i \in E_i.\, \forall v \in \text{vars}(e_i)\colon v \text{ occurs in } C_i \implies v \text{ occurs in } C'_i \wedge (\forall g \in \mathcal{G}_i\colon v \text{ does not occur in } g), \text{ also}$$
$$C_i = \bigwedge_{g\in\mathcal{G}_i} g \implies C'_i = \text{true}.$$

Let $S^X_{\mathcal{P}}$ be all the states in process X. To make sure all guards $\langle g_1,\ldots,g_M\rangle$ in Γ are unique we guarantee the following.

$$\text{Let } U = \bigcup_{1\le i\le K} \mathcal{G}_i, \text{ then}$$

$$\forall g, g' \in U.\, \forall s \in S^X_{\mathcal{P}}\colon g(s) = g'(s) \implies g \in \Gamma \wedge g' \notin \Gamma.$$

■

Note that in term rewriting systems such as mCRL2 it is easy to show that two guards are equal, because when testing the equality of two terms (guards) the equation rewrites to true (and terminates) when guards are equal (or false when they are not). Even when – for example – the operands in a binary operation are in different order.

4.4.1 Dependency Matrices for mCRL2

We define the contents of the Pins UDM for LPS X $UDM(X) = UM^X_{K\times N}$ as follows.

$$UM^X_{i,j} = \begin{cases} 1 & \text{if } \exists 0 \le k \le N : x_j \text{ occurs in } t_{i,k} \wedge (j \neq k \vee t_{i,j} \neq x_j) \vee x_j \text{ occurs in } C'_i, \\ 0 & \text{otherwise.} \end{cases}$$

Example 4.16 (Update Dependency Matrix)**.** The UDM for the Petri net in Figure 1.2 is:

$$\begin{array}{c} \\ T0 \\ T1 \\ T2 \\ T3 \\ T4 \\ T5 \end{array} \begin{array}{c} \begin{array}{ccccc} P0 & P1 & P2 & P3 & P4 \end{array} \\ \left[\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}.$$

and $\upsilon_{TI}(\{\langle 1,0,0,0,0\rangle, \langle 0,1,0,1,0\rangle, \langle 0,0,1,1,0\rangle, \langle 0,1,0,0,1\rangle, \langle 0,0,1,0,1\rangle\}) = \{\}$. ▲

In this example the UDM does not contain any dependency. The reason herefore is that the value of state slots of successor states in the Petri net do not require reading $P0 \ldots P4$. Variables $P0 \ldots P4$ are simply assigned true or false.

4.4.2 Guard-splitting algorithms for mCRL2

Finding an optimal guard splitting algorithm such that calculating the state space using this algorithm is as fast as possible is hard. The problem is inherently hard because splitting guards is like transforming a propositional formula to Conjunctive Normal Form (CNF). The following example shows that transforming a certain formula to CNF blows up the number of conjuncts.

Example 4.17. Transforming the non-CNF formula $(X_1 \wedge Y_1) \vee (X_2 \vee Y_2) \vee \ldots \vee (X_n \vee Y_n)$ into CNF formula $(X_1 \vee \ldots \vee X_{n-1} \vee X_n) \wedge (X_1 \vee \ldots \vee X_{n-1} \vee Y_n) \wedge \ldots \wedge (Y_1 \vee \ldots \vee Y_{n-1} \vee Y_n)$ results in 2^n conjuncts; each conjunct contains either X_i or Y_i for each i. ▲

There exists however a method to linearly increase the size of the formula, but it introduces a new variable. We can transform the formula above in CNF by adding variables $Z_1, \ldots, Z_n$ as follows $(Z_1 \vee \ldots \vee Z_n) \wedge (\neg Z_1 \vee X_1) \wedge (\neg Z_1 \vee Y_1) \wedge \ldots \wedge (\neg Z_n \vee X_n) \wedge (\neg Z_n \vee Y_n)$. We suspect however that finding $Z_1, \ldots Z_n$ is just as hard as calculating the entire state space.

Transforming any propositional formula to CNF is done by using certain *laws of logic*, namely the following:

double negative law $\neg\neg P \equiv P$.

de Morgan's laws $\neg(P \wedge Q) \equiv (\neg P) \vee (\neg Q)$ and $\neg(P \vee Q) \equiv (\neg P) \wedge (\neg Q)$.

Distributive laws $(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$ and $(P \vee (Q \wedge R)) \equiv ((P \vee Q) \wedge (P \vee R))$.

We provide descriptions of algorithms which use none, some or all of these laws to transform all conditions (C_i) into a conjunctively joined set of guards $\mathcal{G}_i$ and adds them (unique guards only) to Γ.

$\mathcal{A}_{|\mathcal{G}_i|=1}$ The algorithm $\mathcal{A}_{|\mathcal{G}_i|=1}$ views the entire condition C_i of a summand as one guard. This algorithm does not require the use of any of the above laws.

$\mathcal{A}_{|\mathcal{G}_i|\geq 1}$ The algorithm $\mathcal{A}_{|\mathcal{G}_i|\geq 1}$ splits the condition into guards on every $\wedge$ operator without transforming C_i first.

$\mathcal{A}_{\text{CNF}}$ The algorithm $\mathcal{A}_{\text{CNF}}$ transforms C_i into CNF such that $\mathcal{G}_i$ is as large as possible.

$\mathcal{A}_{+1}$ The algorithm $\mathcal{A}_{+1}$ introduces an extra variable to C_i to make sure the size of the formula increases linearly.

$\mathcal{A}_h$ The algorithm $\mathcal{A}_h$ uses heuristics to find a good set $\mathcal{G}_i$.

$\mathcal{A}_\neg$ The algorithm $\mathcal{A}_\neg$ pushes the negation in C_i as much inwards as possible by using $\neg\neg P \equiv P$ and $\neg(P \wedge Q) \equiv (\neg P) \vee (\neg Q)$. This algorithm assumes that C_i already an efficient condition and only uses the laws to obtain the conjuncts.

Currently the SpinS [11] compiler uses algorithm $\mathcal{A}_{|\mathcal{G}_i|=1}$, so optimizations can be made here. We have implemented algorithm $\mathcal{A}_{|\mathcal{G}_i|\geq 1}$ for mCRL2 as shown in Algorithm 16 and leave the other algorithms for future work.

- SPLIT$(T) = \{t_0, \ldots, t_n \mid \bigwedge_{t \in \{t_0,\ldots,t_n\}} t = T\}$ we use to split a term into multiple terms connected by a $\wedge$.
- CONDITIONS(L) returns a vector of all conditions in LPS L.
- CONDITION(L, i, T) sets term T as the condition in summand i in LPS L.
- CONDITION(L, i) gets the condition in summand i in LPS L.

- PROC-VARS(L) returns a set of process variables in LPS L.
- VARS(T) returns a set of variables in term T.

Algorithm 16: $\mathcal{A}_{|G_i|\geq 1}$

```
   Data: LPS L
   Result: Γ, G, L
 1 C ← CONDITIONS(L) ;
 2 Γ ← ⟨⟩;
 3 G ← ⟨⟩;
 4 for 0 ≤ i ≤ |C| do
 5 |   ⋀ ← SPLIT(C_i);
 6 |   J ← {n ∈ ℕ | 1 ≤ n ≤ |Γ|};
 7 |   CONDITION(L, i, true);
 8 |   G_i ← ∅;
 9 |   for g ∈ ⋀ do
10 |   |   if VARS(g) \ PROC-VARS(L) ≠ ∅ then
11 |   |   |   c ← CONDITION(L, i);
12 |   |   |   c ← c ∧ g;
13 |   |   |   CONDITION(L, i, c);
14 |   |   else
15 |   |   |   n ← |Γ| + 1;
16 |   |   |   for j ∈ J do
17 |   |   |   |   if Γ_j = g then
18 |   |   |   |   |   n ← j;
19 |   |   |   |   |   break;
20 |   |   |   |   end
21 |   |   |   end
22 |   |   |   G_i ← G_i ∪ {n};
23 |   |   |   if n = |Γ| + 1 then
24 |   |   |   |   Γ_n ← g;
25 |   |   |   end
26 |   |   end
27 |   end
28 end
```

For convenience we define the following function.

- REWRITE(t, σ) rewrites the term t using substitution σ.

Algorithm 17: EVAL-GUARD

```
   Data: i, s ∈ γ_i(S_P)
   Result: r
 1 for v ∈ VARS(Γ_i) do
 2 |   σ(v) ← s_v;
 3 end
 4 t ← REWRITE(Γ_i, σ);
 5 if t = ⊤ then
 6 |   r ← +;
 7 else if t = ⊥ then
 8 |   r ← −;
 9 else
10 |   r ← ?;
11 end
```

4.5 Symbolic reachability for Partitioned Transition Systems

LTSmin supports many symbolic back-ends. We first define the join relation as a symbolic set operation. Then we provide improvements to both REACH-BFS-PREV and REACH-CHAIN-PREV, which use the join operation. All symbolic back-ends can implement the join operation and use the algorithms we provide in this section. In this thesis we however only implement the join operation in LDD. We leave implementing the join operation for other symbolic back-ends such as BuDDy as future work.

$$\text{JOIN}(S_1(\varpi_1(\boldsymbol{x})), S_2(\varpi_2(\boldsymbol{x}))) = \varpi_{1+2}(\boldsymbol{x}) \mapsto S_1(\varpi_1(\boldsymbol{x})) \wedge S_2(\varpi_2(\boldsymbol{x})),$$
where ϖ_1, ϖ_2 and ϖ_{1+2} are projections and ϖ_{1+2} is well defined.

To support guard-splitting in REACH-BFS-PREV we change the algorithm as follows. For each level we evaluate guards we do not know yet. We keep track of the set $\mathcal{F}_i$ so we do not have to re-evaluate states in $\mathcal{F}_i$ for guard i. Then we reduce the states in the current level by checking which states have the necessary guards evaluated to true. This reduction of set $\mathcal{L}_i$ is exactly the key improvement guard-splitting brings. The set $\mathcal{L}_i$ will in many cases be smaller then the set $\mathcal{L}$ which we used in previous versions of REACH-BFS-PREV. Note that checking whether there are states which have guards evaluated to maybe is not done here. In the Section 4.9 it is shown how this can be done in future work.

Algorithm 18: REACH-BFS-PREV

Data: UM,WM,MM,K,M,s^0,Γ,G
Result: $\mathcal{R}$

```
1  WM ← WM ∨ MM;
2  R ← {s^0};
3  L ← R;
4  for 1 ≤ i ≤ K do
5  |   R_i^p ← ∅;
6  |   ↪_i^p ← ∅;
7  |   L_i ← ∅;
8  end
9  for 1 ≤ i ≤ M do
10 |   T_i^p ← ∅;
11 |   F_i^p ← ∅;
12 end
13 while L ≠ ∅ do
14 |   for 1 ≤ i ≤ M do                                          } Evaluate guards
15 |   |   L^p ← γ_i(L);
16 |   |   for s^p ∈ L^p \ T_i^p \ F_i^p do
17 |   |   |   if g_i(s_p) = + then
18 |   |   |   |   T_i^p ← T_i^p ∪ {s_p};
19 |   |   |   else
20 |   |   |   |   F_i^p ← F_i^p ∪ {s_p};
21 |   |   |   end
22 |   |   end
23 |   end
24 |   for 1 ≤ i ≤ K do                                          } Reduce current level
25 |   |   L_i ← L;
26 |   |   for l ∈ G_i do
27 |   |   |   if L_i = ∅ then break;
28 |   |   |   L_i ← T_l^p ⋉ L_i;
29 |   |   end
30 |   |   if L_i ≠ ∅ then                                      } Build the transition relation
31 |   |   |   L^p ← υ_i(L_i);
32 |   |   |   for s^p ∈ L^p \ R_i^p do
33 |   |   |   |   ↪_i^p ← ↪_i^p ∪ {(s^p, d^p) | d^p ∈ NEXT-UPDATE_i(s^p)};
34 |   |   |   end
35 |   |   |   R_i^p ← R_i^p ∪ L^p;
36 |   |   end
37 |   end
38 |   N ← ∅;
39 |   for 1 ≤ i ≤ K do                                         } Build the state space
40 |   |   if L_i ≠ ∅ then
41 |   |   |   N ← N ∪ STEP(L_i, ↪_i^p, UM_i, WM_i);
42 |   |   end
43 |   end
44 |   L ← N \ R;
45 |   R ← R ∪ N;
46 end
47
```

Our chaining algorithm can not learn all the guards at once per level. We need to learn relevant guards

when we expand the transition relation for each transition group.

Algorithm 19: REACH-CHAIN-PREV

Data: $UM, WM, MM, K, M, s^0, \Gamma, G$
Result: $\mathcal{R}$

$WM \leftarrow WM \vee MM$;
$\mathcal{R} \leftarrow \{s^0\}$;
$\mathcal{L} \leftarrow \mathcal{R}$;
for $1 \leq i \leq K$ **do**
  $\mathcal{R}_i^p \leftarrow \varnothing$;
  $\hookrightarrow_i^p \leftarrow \varnothing$;
  $\mathcal{L}_i \leftarrow \varnothing$;
end
for $1 \leq i \leq M$ **do**
  $\mathcal{T}_i^p \leftarrow \varnothing$;
  $\mathcal{F}_i^p \leftarrow \varnothing$;
end
while $\mathcal{L} \neq \varnothing$ **do**
  for $1 \leq i \leq K$ **do**
    $\mathcal{L}_i \leftarrow \mathcal{L}$;
    for $l \in G_i$ **do**
      if $\mathcal{L}_i = \varnothing$ **then** break;
      $\mathcal{L}^p \leftarrow \gamma_i(\mathcal{L}_i)$;
      for $s^p \in \mathcal{L}^p \setminus \mathcal{T}_l^p \setminus \mathcal{F}_l^p$ **do**
        if $g_l(s_p) = +$ **then**
          $\mathcal{T}_l^p \leftarrow \mathcal{T}_l^p \cup \{s_p\}$;
        else
          $\mathcal{F}_l^p \leftarrow \mathcal{F}_l^p \cup \{s_p\}$;
        end
      end
      $\mathcal{L}_i \leftarrow \mathcal{T}_l^p \bowtie \mathcal{L}_i$;
    end
    if $\mathcal{L}_i \neq \varnothing$ **then**
      $\mathcal{L}^p \leftarrow \upsilon_i(\mathcal{L}_i)$;
      for $s^p \in \mathcal{L}^p \setminus \mathcal{R}_i^p$ **do**
        $\hookrightarrow_i^p \leftarrow \hookrightarrow_i^p \cup \{(s^p, d^p) \mid d^p \in \text{NEXT-UPDATE}_i(s^p)\}$;
      end
      $\mathcal{R}_i^p \leftarrow \mathcal{R}_i^p \cup \mathcal{L}^p$;
      $\mathcal{L} \leftarrow \mathcal{L} \cup \text{STEP}(\mathcal{L}_i, \hookrightarrow_i^p, UM_i, WM_i)$;
    end
  end
  $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{L}$;
  $\mathcal{L} \leftarrow \mathcal{L} \setminus \mathcal{R}$;
end

4.5.1 Symbolic Reachability Implementation in LTSmin

For guard-splitting we have added one new operation on two sets. The join operation takes two sets as input and returns a new set with new projection information. On common state slots between the input sets we find matching values. If set B is 'behind' set A on the state slots we walk over then we take all

values from the state slot of set B. If set A is 'behind' on set A we do the same for set A.

Algorithm 20: JOIN-LDD

```
   Data: X, Y, A, B, a, b
   Result: C
 1 if A = ∅ ∨ B = ∅ then return ∅;
 2 if A = B then return A;
 3 if a = |P^A| then return B;
 4 if b = |P^B| then return A;
 5 if P_a^A = P_b^B then
 6 |   if IN-OP-CACHE(∅, A, B) then return CACHE-LOOKUP(∅, A, B);
 7 |   if VAL(A) = VAL(B) then
 8 |   |   R ← JOIN-LDD(X, Y, RIGHT(A), RIGHT(B), a, b);
 9 |   |   D ← JOIN-LDD(DOWN(X), DOWN(Y), DOWN(A), DOWN(B), a + 1, b + 1);
10 |   |   C ← NODE(VAL(A), D, R);
11 |   else if VAL(A) < VAL(B) then
12 |   |   C ← JOIN-LDD(X, Y, RIGHT(A), B, a, b);
13 |   else
14 |   |   C ← JOIN-LDD(X, Y, A, RIGHT(B), a, b);
15 |   end
16 |   ADD-TO-CACHE(∅, A, B, C);
17 else if P_a^A > P_b^B then
18 |   if IN-OP-CACHE(X, A, B) then return CACHE-LOOKUP(X, A, B);
19 |   R ← JOIN-LDD(X, Y, A, RIGHT(B), a, b);
20 |   D ← JOIN-LDD(X, DOWN(Y), A, DOWN(B), a, b + 1);
21 |   C ← NODE(VAL(B), D, R);
22 |   ADD-TO-CACHE(X,A,B,C);
23 else
24 |   if IN-OP-CACHE(Y, A, B) then return CACHE-LOOKUP(Y, A, B);
25 |   R ← JOIN-LDD(X, Y, RIGHT(A), B, a, b);
26 |   D ← JOIN-LDD(DOWN(X), Y, DOWN(A), B, a + 1, b);
27 |   C ← NODE(VAL(A), D, R);
28 |   ADD-TO-CACHE(Y, A, B, C);
29 end
```

4.6 Pins2Pins wrappers

When using guard-splitting implementing regrouping functions on the dependency matrix becomes challenging. This is due to the fact that determining whether a state has successors moves from a language front-end to the symbolic back-end. Thus the behaviour of regrouping functions can not be hidden well in a Pins2Pins wrapper. We leave it for future work to implement regrouping functions on the dependency matrix in combination with guard-splitting.

4.7 Implementation in LTSmin

4.7.1 The Pins front-end

Since guard-splitting for Promela and Dve is already done in previous work [7, 11]. The following Pins gb functions already exist, but have been implemented for mcrl2.

PINS method	Input arguments	Return value	description
`GBgetGuard`	*model*, *group index*	set of indices	Returns all guard numbers for a transition group. More precisely, it returns an element in the vector $G = \langle G_1, \ldots, G_K \rangle$.
`GBgetStateLabelShort`	*model*, *guard index*, *state*	$\{-, ?, +\}$	Returns the evaluation of a guard for a state.
`GBgetStateLabelGroupInfo`	*model*	Returns the Guard Dependency Matrix	Returns the GDM for a model.

Table 4.1: PINS GreyBox interface functions for guard-splitting in `pins.h`

Function	Input arguments	Return value	Description
`vset_join`	vset_t, vset_t	vset_t	joins two sets with the relational $\bowtie$ operation and returns a new set with new projection information.

Table 4.2: PINS state vector operations for guard-splitting

4.7.2 The PINS2PINS wrappers

Currently we have inhibited any operation on dependency matrices, because they do not work yet with guard-splitting.

4.7.3 The PINS back-end

To implement guard-splitting in the symbolic back-end we thus changed the REACH-BFS-PREV and the REACH-CHAIN-PREV algorithm. To this end we added some functions to the symbolic back-end and we will also show how the symbols in the algorithms map to variables used in the symbolic back-end. This information can be found in the Tables 4.2 and 4.3.

4.7.4 Compatibility

The symbolic back-end by default uses guards and thus guard-splitting in the reachability algorithms if a language front-end provides these guards. However there are some compatibility issues. The first issue is that all dependency matrix operations are not supported if we use guard-splitting. The main reason herefore is that the responsibility of what knows whether a condition holds for a state is moved from the front-end to the back-end. This change is not at all supported by the PINS2PINS wrapper. And we thus leave it for feature work to come up with a good design approach to enable dependency matrix operations in combination with guard-splitting. The other — not so large of an issue is that there are many symbolic back-ends which do not yet implement the relational join operation. Currently, if a user

Variable	Symbol	Type	Description
`level_reduced`	$\mathcal{L}_i$	vset_t	Symbolically stored set of states in a particular level, reduced with a join operation

Table 4.3: PINS variables for symbolic reachability w/ guard-splitting

performs symbolic reachability analysis with any symbolic back-end other than LDD the user will get an error.

4.7.5 Reproducablity

The experiments we did with our changes can be easily reproduced. Please refer to Section 3.4.5 to see how to set up an environment with separation of dependencies.

Environment with guard-splitting

First compile and install LTSMIN as described in Section 3.4.5. Use however the *guard* branch in `git@github.com:Meijuh/ltsmin.git`. Note that the UDM for DIVINE is not completely implemented thus one can not run all DVE models from the BEAM database with guard-splitting. When running experiments with PROMELA do not forget to add the `-W` option as described in the previous chapter. To run experiments with MCRL2 install version 2012-10 with a patch (for guard-splitting) applied. This patch can be found at `https://gist.github.com/Meijuh/9617728`. Now without any specific options for guard-splitting one can perform reachability analysis with the following command: `lps2lts-sym --mcrl2="--rewriter=jitty" /some-path/1394-fin.lps`.

4.8 Benchmarks

We expected that guard-splitting would improve the computation of state spaces of models such as Sokoban. For larger models of Sokoban, such as screen.1 this appears to be indeed the case. However, not all experiments seem to benefit our guard-splitting in terms of run time. Memory usage is however improved.

If one looks at Table C.3 and Table C.4 one can see that the values in the column $[\![\pi(\mathcal{R})]\!]$ are significantly lower with guard-splitting. Lower values in this column mean that there are fewer states to compute successors for (because we know some guards do not hold). Furthermore the amount of nodes in the transition relation is less, but now the sets $[\![\mathcal{F}]\!]$ and $[\![\mathcal{T}]\!]$ also use some nodes. It is yet inconclusive if we can reduce the total amount of nodes in the transition relation, the set of states which evaluate to false for guards and the set of states which evaluate to true for guards when row subsumption is used. Currently the amount is not *significantly* lower because in many models the transition relation is already relatively small.

The most interesting results for guard-splitting are larger Sokoban models in MCRL2, such as screen.1-deadlock.lps and the model of the firewire protocol; 1394-fin.lps. For PROMELA the most interesting results are the models of the dining philosophers (phil-10.pr and phil-15.pr). For DVE there are more interesting results such as: iprotocol.7.dve2C, firewire_link.5.dve2C and peg_solitaire.5.dve2C. Both 1394-fin.lps and firewire_link.5.dve2C are models of the firewire protocol. If we look at the detailed results that can be found online (see 'Open online access to our benchmarking results', Page 7) we see that there is a good ratio between the number of guards and the number of transition groups. In both models the transition relation contain about 1000 nodes less. The amount of nodes to store the set of projected states is in both cases **99%** less. The MCRL2 model for the firewire protocol (1394-fin.lps) has a strange result. The experiment with guard-splitting shows an increase in the amount of nodes to store the set of reachable states. We believe this may be due to a bad variable ordering which becomes notable when using guard-splitting.

The DVE model 'peg_solitaire.5.dve2C' is considerably slower with guard-splitting. If one looks at the results which can be found online one can see that the amount of guards is almost equal to the number of transition groups. This is very inefficient, because for every transition group we have to evaluate a guard. And if this guard in most cases evaluate to true then we have to do both a guard evaluation and a NEXT-STATE call. The amount of nodes to store all the states that evaluate to true for a guard is very large. The amount of nodes is half that of the amount of nodes to store the projected set of states

without guard-splitting. This is a bad result and explains why we do not see a run time improvement for peg_solitaire.5.dve2C.

A good explanation for why guard-splitting is not beneficial to the computation of models such as WMS.lps and WMS.suminst.lps is that almost all states in the current level satisfy all guards. This means that in these cases there is an overhead of calls to the language front-end.

In our implementation of guard-splitting in LTSmin there is a memory leak. This is the reason why we did not include scatter plots for memory usage. Because of the memory leak we must make the note that the measurements for run time may be a little biased. This is because our implementation spends less time on freeing memory. We were however able to come up with a quick and dirty solution to this problem. With this solution we did not re-do all our experiments, because this would take to long. We did however run a large Sokoban model which previously would run out of memory due to the memory leak. The model is screen 38 without deadlock situations of the Sokoban game. With guard-splitting reachability analysis took about 1800 seconds, without guard-splitting it took 9000 seconds. Our experiment with guard-splitting used about 250 MB of memory while without guard-splitting 140 MB of memory was used. The Sokoban screen without deadlock has 2.604.179.766 states.

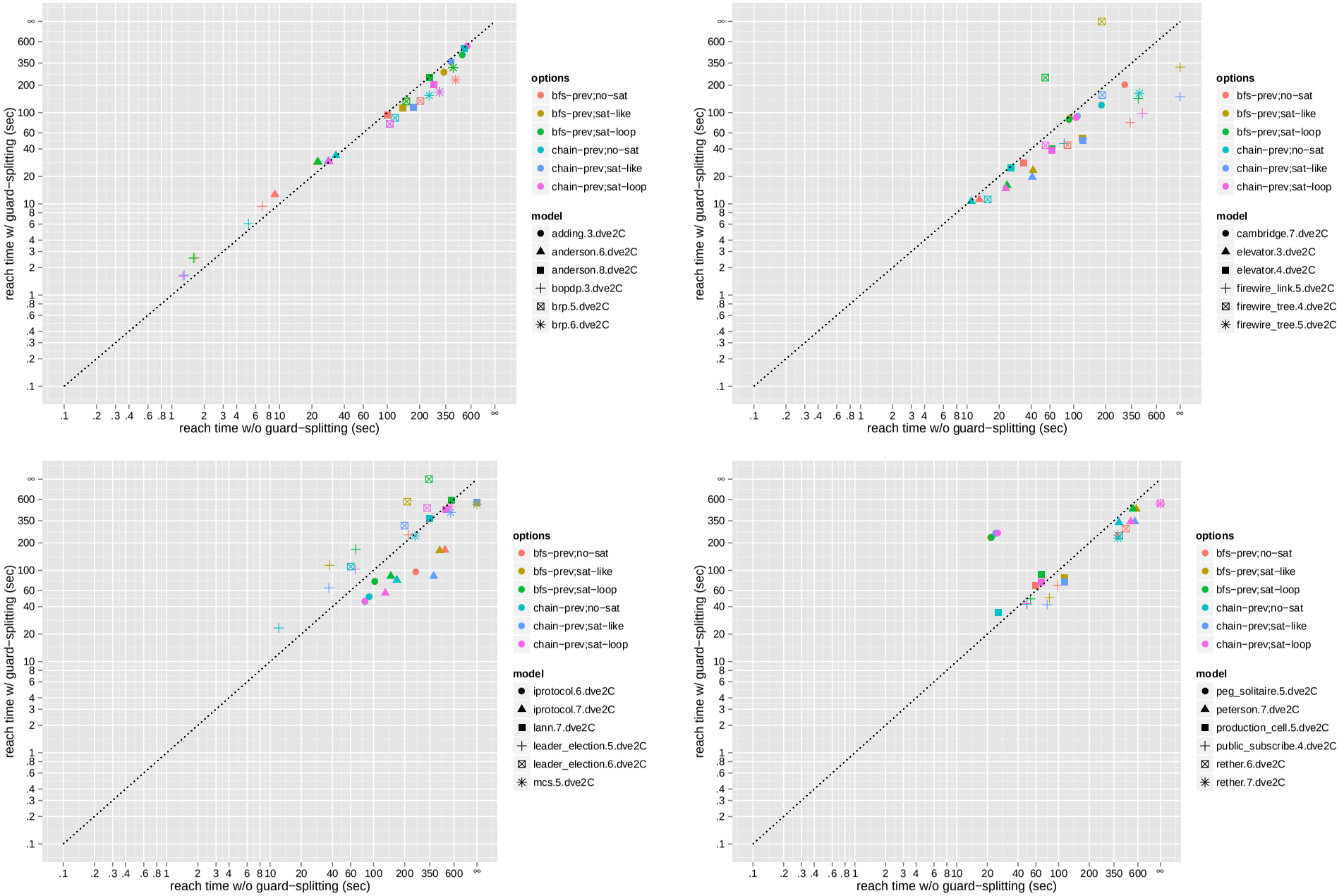


Figure 4.2: Time scatter plots 1 – 24 for with guard-splitting

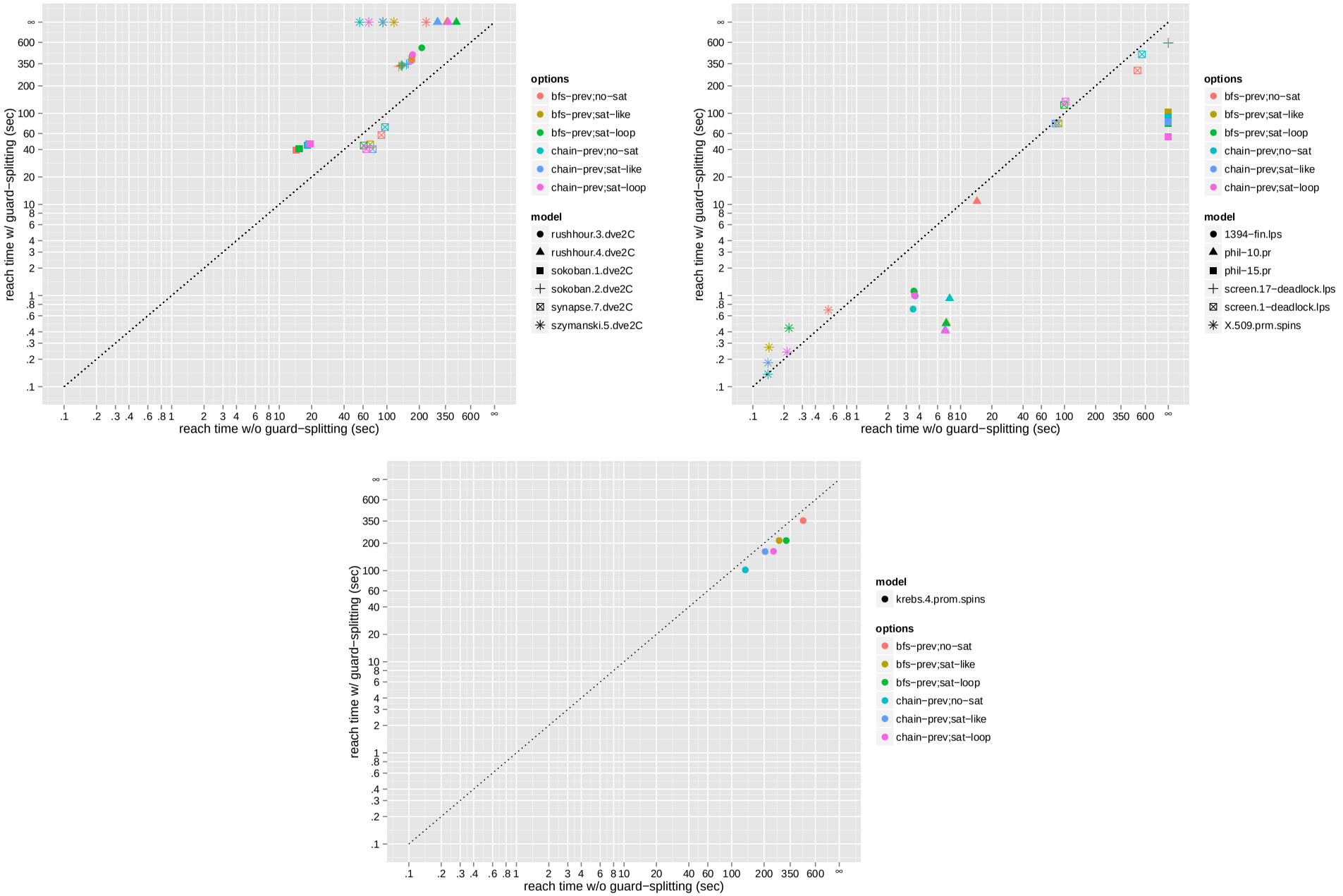


Figure 4.3: Time scatter plots 25 – 37 for with guard-splitting

4.9 Future Work

We can also bring the following improvements to LTSmin.

- Implement the update matrix correctly in DiVinE.
- Implement the `vset_join` function in all symbolic back-ends other than LDD.
- Implement guard-splitting in `reach-chain` and `reach-bfs`.
- Implement other guard-splitting algorithms for mCRL2.
- Implement the maybe check as described in Algorithms 21 and 22. Note that these algorithms do not check if the models are correct under lazy evaluation for languages such as Java. E.g. the algorithms assume no transition of the form '`if (p.method() && p != null ) { // next state }`' exist in the model and in some point in time '`p.method()`' is called with '`p == null`'. In the future this may be solved by checking each conjunction of guards is never equal to 'maybe'. More precisely if a transition contains the condition $g_1 \wedge \ldots \wedge g_n$ and we have a state s, then we must check: $g_1(s) \neq ?$ and $g_1(s) \wedge (g_2(s) \neq ?)$ and ... and $g_1(s) \wedge \ldots \wedge (g_n(s) \neq ?)$. This check can be built into front-ends for languages that use lazy evaluation.
- In our benchmarks we have found that the default options to the symbolic back-end may not always produce the best results. For example, if we increase the size of the node table and cache then we see much better run times. As future work we can investigate what more optimal options are to the symbolic back-end.
- Like in Chapter 3 it is interesting to see what the results are like if we re-do our experiments with optimizations to the dependency matrix such as row subsumption.
- Fix the apparent issue with the garbage collector for the `vset_join` function in LDD.
- Implement the Do-Not-Accord (DNA) matrix in mCRL2 to support Partial Order Reduction.

4.10 Conclusion

In our problem statement in Chapter 2 we showed that the condition could be removed from the transition relation and stored separately. In our approach we have exactly done that. Results show that there are some models which benefit in terms of time and memory usage of guard-splitting. There are however also models which do not benefit of guard-splitting. Models which do not benefit of guard-splitting often have conditions which are satisfied by many or all states in the current level when computing the state space.

In this chapter we have given a notion for PTSs (Definition 4.5) with support for guard-splitting. The definition extends the original notion of PTSs with guards and sets which indicate which guard is relevant to a transition group. We have also given two notions of independence for guards and the update part of a transition group. In this chapter we described in detail how to perform guard-splitting for mCRL2.

While working on guard-splitting for mCRL2 we noticed that we had to support a ternary logic for the evaluation of guards (Example 4.7). Together with the new join algorithm (Algorithm 20) for LDD is what makes guard-splitting an interesting extension to symbolic reachability analysis.

Our benchmarks indicate there are clearly models which benefit of guard-splitting, such as models of Sokoban, a model of the firewire protocol and models of dining philosophers. For these models we see great run-time improvements because we can prevent many calls to language front-ends. These calls can be prevented, because with guard-splitting we know which states can not have successors. This is also shown in Table C.3 and Table C.4 in the latter table the amount of nodes to represent states for which successors have to be computed ($[\![\pi(\mathcal{R})]\!]$) is significantly smaller for all models. Furthermore, since we do not have to store the evaluation of the condition in the transition relation one can also see in the same table that in all models the size of the transition relation is also smaller.

Algorithm 21: REACH-BFS-PREV W/ MAYBE CHECK

```
   Data: UM,WM,MM,K,M,s^0,Γ,G
   Result: R
 1 ...;
 2 while L ≠ ∅ do
 3 |   for 1 ≤ i ≤ M do
 4 |   |   M_i^p ← ∅;
 5 |   |   L^p ← γ_i(L);
 6 |   |   for s^p ∈ L^p \ T_i^p \ F_i^p do
 7 |   |   |   if g_i(s_p) = + then
 8 |   |   |   |   T_i^p ← T_i^p ∪ {s_p};
 9 |   |   |   else if g_i(s_p) = − then
10 |   |   |   |   F_i^p ← F_i^p ∪ {s_p};
11 |   |   |   else if g_i(s_p) = ? then
12 |   |   |   |   F_i^p ← F_i^p ∪ {s_p};
13 |   |   |   |   M_i^p ← M_i^p ∪ {s_p};
14 |   |   |   end
15 |   |   end
16 |   end
17 |   for 1 ≤ i ≤ K do
18 |   |   L_i ← L;
19 |   |   for l ∈ G_i do
20 |   |   |   if L_i = ∅ then break;
21 |   |   |   L_i ← T_l^p ⋈ L_i;
22 |   |   end
23 |   |   if L_i ≠ ∅ then
24 |   |   |   for l ∈ G_i() do
25 |   |   |   |   if L_i ⋈ M_l^p ≠ ∅ then ERROR;
26 |   |   |   end
27 |   |   end
28 |   |   ...;
29 |   end
30 |   ...;
31 end
```

Because work on Chapter 3 took more work than anticipated we were not able to examine in detail how advanced reachability strategies such as saturation can profit from guard-splitting. It is however the case that we store the evaluation of guards globally. This means that also in saturation — where we only look at subsets of transition groups, we can prevent computing successor states for these transition groups. Investigation is however necessary to see if saturation can profit on a 'higher level' from guard splitting, i.e. before performing `reach-chain-prev` on a subset of transition groups.

When writing our research proposal we incorrectly expected that simply implementing guard-splitting for mCRL2 we would also have Partial Order Reduction for mCRL2. This appeared to not be the case. For mCRL2 the DNA matrix has to be implemented [7].

To make guard-splitting in LTSMIN production ready quite some work has to be done as described in Section 4.9. Most notable future work to be done is checking whether model specifications are complete. This is shown with the maybe check in Algorithms 21 and 22. Furthermore the `vset_join` operation should be implemented in all symbolic back-ends.

Algorithm 22: REACH-CHAIN-PREV W/ MAYBE CHECK

Data: UM,WM,MM,K,M,s^0,Γ,G
Result: $\mathcal{R}$

```
1  ...;
2  while L ≠ ∅ do
3      for 1 ≤ i ≤ K do
4          L_i ← L;
5          for l ∈ G_i do
6              if L_i = ∅ then break;
7              M_l^p ← ∅;
8              L^p ← γ_i(L_i);
9              for s^p ∈ L^p \ T_l^p \ F_l^p do
10                 if g_l(s_p) = + then
11                     T_l^p ← T_l^p ∪ {s_p};
12                 else if g_l(s_p) = − then
13                     F_l^p ← F_l^p ∪ {s_p};
14                 else if g_l(s_p) = ? then
15                     F_l^p ← F_l^p ∪ {s_p};
16                     M_l^p ← M_l^p ∪ {s_p};
17                 end
18             end
19             L_i ← T_l^p ⋉ L_i;
20         end
21         if L_i ≠ ∅ then
22             for l ∈ G_i() do
23                 if L_i ⋉ M_l^p ≠ ∅ then ERROR;
24             end
25         end
26         ...;
27     end
28     ...;
29 end
```

Bibliography

[1] S. C. C. Blom, J. C. van de Pol, and M. Weber. Bridging the Gap between Enumerative and Symbolic Model Checkers. Technical Report TR-CTIT-09-30, Centre for Telematics and Information Technology University of Twente, Enschede, June 2009.

[2] Stefan Blom and Jaco Pol van de. Symbolic Reachability for Process Algebras with Recursive Data Types. In J.S. Fitzgerald, A.E. Haxthausen, and H. Yenigun, editors, *Theoretical Aspects of Computing*, volume 5160 of *Lecture Notes in Computer Science*, pages 81–95, Berlin, Germany, August 2008. Springer Verlag.

[3] Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and Symbolic Reachability. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359. Springer, 2010.

[4] Gianfranco Ciardo and Andy Jinqing Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Correct Hardware Design and Verification Methods*, pages 146–161. Springer, 2005.

[5] Alfons Laarman, Elwin Pater, Jaco van de Pol, and Michael Weber. Guard-Based Partial-Order Reduction. In Ezio Bartocci and C. R. Ramakrishnan, editors, *SPIN*, volume 7976 of *Lecture Notes in Computer Science*, pages 227–245. Springer, 2013.

[6] Alfons Laarman, Jaco van de Pol, and Michael Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 506–511. Springer, 2011.

[7] Elwin Pater. Partial Order Reduction for PINS. Master's thesis, University of Twente, March 2011.

[8] Radek Pelánek. Beem: Benchmarks for explicit model checkers. In Dragan Bosnacki and Stefan Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer, 2007.

[9] Tien Loong Siaw. Saturation for LTSmin. Master's thesis, University of Twente, 2012.

[10] Steven Skiena. *The Algorithm Design Manual: Text*, volume 1. Springer, 1998.

[11] Freark I. van der Berg and Alfons W. Laarman. SpinS: Extending LTSmin with Promela through SpinJa. In *Proceedings the Sixth International Workshop on the Practical Application of Stochastic Modelling (PASM) and the Eleventh International Workshop on Parallel and Distributed Methods in Verification (PDMC).*, volume 296, pages 95 – 105. 2013.

[12] T. van Dijk, A. W. Laarman, and J. C. van de Pol. Multi-Core BDD Operations for Symbolic Reachability. In K. Heljanko and W. J. Knottenbelt, editors, *11th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2012, London, UK*, Electronic Notes in Theoretical Computer Science, Amsterdam, September 2012. Elsevier.

[13] Kirsten Winter. Optimising Ordering Strategies for Symbolic Model Checking of Railway Interlockings. *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, pages 246–260, 2012. Section 3.

Appendices

Appendix A

Acronyms

Notation	Description
BDD	Binary Decision Diagram.
BFS	Breadth First Search.
BUDDY	BuDDy: A BDD package.
CNF	Conjunctive Normal Form.
DAG	Directed Acyclic Graph.
DM	Dependency Matrix.
DNA	Do-Not-Accord.
GB	GreyBox interface.
GDM	Guard Dependency Matrix.
LDD	List Decision Diagram.
LPS	Linear Process System.
MDD	Multi-way Decision Diagram.
MDM	May-write Dependency Matrix.
PINS	Partitioned Interface to the Next State.
POR	Partial Order Reduction.
PTS	Partitioned Transition System.
RDM	Read Dependency Matrix.
TS	Transition System.
UDM	Update Dependency Matrix.
WDM	Must-write Dependency Matrix.

Appendix B

Implementation in LTSmin

Pins method	**Input arguments**	**Return value**	**description**
`GBgetDMInfo`	*model*	matrix	Returns the DM.
`GBgetInitialState`	*model*	*state*	Returns the initial state.
`GBgetTransitionsShort`	*model*, *group number* ($1 \leq i \leq K$), *source state* (as projected state vector, according to π_i), call back function	number of successors	For a given *model*, enumerate the transitions of the given transition starting from the *source state*. The callback function together with the context information is used to return a list of successor states, as a list of projected indexed state vectors. This function is mostly an implementation of the NEXT-STATE function.
`GBgetTransitionsLong`	*model*, *group number* ($1 \leq i \leq K$), *source state* (as indexed state vector)	of successors	Idem, but now using a normal state vector.
`GBregroup`	*model*, *regroup specification*	–	applies variable reordering and transition regrouping to the DM of a *model*.

Table B.1: Pins GreyBox interface in `pins.h`

Name	Library	Description
list	ATermDD	Full-fledged implementation of LDDs using linked-lists with integer as node value and building on ATerm objects.
tree	ATermDD	Full-fledged implementation of MDDs using binary trees with integer as node value and building on ATerm objects.
fdd	BuDDy	Wrapper around BuDDy library.
ddd	LibDDD	Wrapper around libDDD library.
ldd	listDD	Full-fledged implementation of LDDs with integer as node value.
sylvan	Sylvan	multi-core BDD library using task-based work-stealing algorithms and scalable data structures [12]

Table B.2: Pins gb example

Variable	Symbol	Type	Description
`current_level`, `new_states`	$\mathcal{L}$	vset_t	Symbolically stored set of states in a particular level
`next_level`	$\mathcal{N}$	vset_t	Symbolically stored set of states up to and including a particular level
`visited`	$\mathcal{R}$	vset_t	Symbolically stored set of reachable states.
`nGrps`	K	int	Number of transition groups.
`group_next`	$\hookrightarrow_i^p$	vrel_t	Symbolically stored transition relation per transition group i
`group_explored`	$\mathcal{R}_i^p$	vset_t	Symbolically stores set of states per transition group i
`domain`		vdom_t	a type for a domain
`projs`	P, i.e. $\langle j \mid DM_{i,j} = 1 \rangle$	proj_info	information about projections of symbolically stored set of states

Table B.3: Pins variables for symbolic reachability

Function	Input arguments	Description
expand_group_next	*transition group number*, vset_t	Build the transition relation for *transition group number*. This corresponds to Line 11 in Algorithm 2.

Table B.4: Pins functions for symbolic reachability

Function	Input arguments	Return value	Description
`vset_create`	vdom_t, length of state vector (N), projections of state vectors	vset_t	new instance of state vector with a length according to N, also contains auxiliary projection data.
`vset_add`	*vset_t*, state (as indexed state vector)	vset_t	the *vset_t* with the given state added.
`vset_is_empty`	vset_t	boolean	returns whether the given state vector is empty.
`vset_equal`	vset_t src,dst	boolean	returns whether the two given state vectors are equal.
`vset_copy`	vset_t *src*,*dst*	void	copies state vector *src* into *dst*.
`vset_project`	vset_t *src*, *dst*	void	makes a projection of *src* into *dst*; *dst* already contains a list of state slots to project to.
`vset_union`	vset_t *src*,*dst*	void	$dst := dst \cup src$
`vset_intersect`	vset_t *src*, *dst*	void	$dst := dst \cap src$
`vset_minus`	vset_t *src*, *dst*	void	$dst := dst - src$
`vset_enum`	vset_t, callback function, context	void	Enumerate every state of the given `state vector` and use the callback function and context to perform operations on these states.
`vset_zip`	vset_t *src*, *dst*	void	$dst := (dst \cup src) \cup src - dst$
`vset_clear`	vset_t	void	Removes all states in the state vector.
`vset_reorder`	vdom_t	void	Reorder the variables in the state vector?.
`vset_next`	vset_t *src*, *dst*, vrel_t	void	$dst := \{y \mid \exists x \in src.x\ rel\ y\}$

Table B.5: Pins state vector operations

Function	Input arguments	Return value	Description
`vrel_create`	vdom_t, length of state vector, projections of state vector	vrel_t	Returns a new transition relation with two times the size of the state vector and projection information.
`vrel_add`	*vrel_t*, source state (as indexed state vector), successor state (as indexed state vector)	void	Adds a source and successor state to *vrel_t*

Table B.6: Pins transition group operations

Appendix C

Results

This section contains a subset of our results. The results here are separated into two sections. One for read and write separation (results for Chapter 3) and one for guard-splitting (results for Chapter 4). Each section contains a list of models which do not show significant differences. The detailed results of these models are however shown in Tables C.1 and C.2 for separated dependencies and C.3 and C.4 for guard-splitting. The results in these tables show slight differences which are not visible in the scatter plots. Along with the tables we give a legend with an explanation for the header of each table. All our benchmark results can be downloaded online, see Section 'Open online access to our benchmarking results' on Page 7.

C.1 Separated Dependencies

List of results for read/write separation with insignificant differences:

- anderson.6.dve2C
- anderson.8.dve2C
- at.5.dve2C
- bke.lps
- bopdp.3.dve2C
- brp.5.dve2C
- brp.6.dve2C
- brp.prm.spins
- cyclic_scheduler.3.dve2C
- cyclic_scheduler.4.dve2C
- dbm.prm.spins
- dining_10.lps
- dining_10.suminst.lps
- elevator2.3.prom.spins
- elevator.4.dve2C
- exit.4.dve2C
- extinction.3.dve2C
- extinction.4.dve2C
- fgs.promela.spins
- firewire_tree.4.dve2C
- firewire_tree.5.dve2C
- fischer.5.dve2C
- fischer.6.dve2C
- iprotocol.6.dve2C
- iprotocol.7.dve2C
- krebs.3.dve2C
- krebs.4.dve2C
- krebs.4.prom.spins
- lamport.6.prom.spins
- lamport_na.5.dve2C
- lamport_nonatomic.3.prom.spins
- lann.5.dve2C
- lann.7.dve2C
- leader_election.5.dve2C
- leader_election.6.dve2C
- leader_filters.6.dve2C
- loyd.2.dve2C
- magic_square.lps
- p319.pml.spins
- peg_solitaire.5.dve2C
- peterson.7.dve2C
- phil-10.pr
- phil-15.pr
- phils.5.prom.spins
- phils.7.dve2C
- phils.8.dve2C
- pouring.2.prom.spins
- production_cell.5.dve2C
- public_subscribe.4.dve2C
- reader_writer.3.prom.spins
- rether.6.dve2C

- rether.7.dve2C
- rushhour.3.dve2C
- schedule_world.3.dve2C
- screen.17-deadlock.lps
- screen.1-deadlock.lps
- SMS.lps
- SMS.suminst.lps
- snake.lps
- snake.suminst.lps
- sorter.4.dve2C
- szymanski.5.dve2C
- tree.lps
- tree.suminst.lps
- WMS.lps
- WMS.suminst.lps
- X.509.prm.spins

Legend for detailed results for read/write separation:

model	name of the model
options	options to the reachability algorithm
backend	the reachability algorithm
frontend	the language front-end
n	number of times the experiment is repeated
N	length of the state vector
K	number of transition groups
checks	number of times NEXT-STATE is called
ns	number of times the transition relation is applied
$\overline{\textbf{rt}}$	average time in seconds for the reachability algorithm excluding set-up time
rt-σ	standard deviation for the reachability algorithm excluding set-up time
$\lvert\mathcal{R}\rvert$	amount of reachable states
$[\![\mathcal{R}]\!]$	amount of nodes to store the reachable states
$[\![\hookrightarrow]\!]$	amount of nodes to store the transition relation
$[\![\pi(\mathcal{R})]\!]$	amount of nodes to store the projected reachable states
$\overline{\textbf{at}}$	average time in seconds for the reachability algorithm including set-up time
at-σ	standard deviation for the reachability algorithm including set-up time
$\overline{\textbf{mem}}$	average amount of memory in kilobytes needed for the experiment
mem-σ	standard deviation for the memory needed for the experiment
-	the experiment ran out of time.

model	backend	n	N	K	checks	ns	$\overline{\text{rt}}$	rt-σ	$\lvert\mathcal{R}\rvert$	$[\![\mathcal{R}]\!]$	$[\![\hookrightarrow]\!]$	$[\![\pi(\mathcal{R})]\!]$	$\overline{\text{at}}$	at-σ	$\overline{\text{mem}}$	mem-σ
1394-fin.lps	sym	3	34	925	49950	49950	25.667	0.113	1.89e+05	7384	11079	791364	25.92	0.107	123244	0
SMS.suminst.lps	sym	3	18	54	648	648	0.087	0.005	3.64e+04	1266	825	2227	0.12	0	94448	0
WMS.lps	sym	3	30	56	2464	2464	280.017	0.351	2.51e+10	15368	5696	6832	280.157	0.351	136784	0
WMS.suminst.lps	sym	3	30	101	4343	4343	308.427	0.63	2.51e+10	15368	8369	13955	308.557	0.63	136936	0
bke.lps	sym	3	19	17	119	119	13.293	0.054	1.11e+04	7816	5493	13301	13.377	0.056	105932	0
brp.lps	sym	3	20	18	378	378	0.277	0.005	1.40e+04	1823	1952	2427	0.313	0.005	95812	0
brp.suminst.lps	sym	3	20	78	1638	1638	0.267	0.005	1.40e+04	1823	2632	12126	0.303	0.005	95732	0
dining_10.lps	sym	3	30	210	4200	4200	0.34	0	9.37e+05	569	540	2708	0.383	0.005	94072	0
dining_10.suminst.lps	sym	3	30	50	1000	1000	0.23	0	9.37e+05	569	380	628	0.247	0.005	93372	0
lift3-final.lps	sym	3	30	36	540	540	2.333	0.012	7.57e+03	9916	6651	25712	2.427	0.017	99676	90.51
magic_square.lps	sym	3	1	1	2	2	46.933	0.387	2.00e+00	4	4	4	46.947	0.386	456618.667	422.378
screen.1-deadlock.lps	sym	3	56	222	29304	29304	557.033	0.493	4.37e+07	11377	1389	2525	557.07	0.49	94124	0
snake.lps	sym	3	6	4	52	52	81.117	0.334	7.60e+04	74731	124503	232750	81.273	0.333	309613.333	1.886
snake.suminst.lps	sym	3	6	10	130	130	79.013	0.047	7.60e+04	74621	134713	612534	79.243	0.047	310066.667	1.886
tree.lps	sym	3	1	2	22	22	0.287	0.005	1.02e+03	1027	2052	2054	0.303	0.005	94312	0
tree.suminst.lps	sym	3	1	2	22	22	0.283	0.005	1.02e+03	1027	2052	2054	0.303	0.005	94312	0
vasy-init.lps	sym	3	484	775	26350	26350	234.85	0.967	9.79e+21	8182	6754	7733	235.137	0.967	252456	0
vasy.lps	sym	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table C.1: Results w/o read-write separation for mCRL2 (chain-prev-no-sat)

model	backend	n	N	K	checks	ns	$\overline{\text{rt}}$	rt-σ	$\lvert\mathcal{R}\rvert$	$[\![\mathcal{R}]\!]$	$[\![\hookrightarrow]\!]$	$[\![\pi(\mathcal{R})]\!]$	$\overline{\text{at}}$	at-σ	$\overline{\text{mem}}$	mem-σ
1394-fin.lps	sym	3	34	925	49950	49950	3.513	0.009	1.89e+05	7384	7157	203103	3.747	0.012	105500	0
SMS.suminst.lps	sym	3	18	54	648	648	0.087	0.005	3.64e+04	1266	524	1285	0.127	0.005	94452	0
WMS.lps	sym	3	30	56	2464	2464	281.813	0.471	2.51e+10	15368	4212	6287	281.953	0.471	137752	0
WMS.suminst.lps	sym	3	30	101	4343	4343	315.347	0.258	2.51e+10	15368	6304	12137	315.48	0.262	138352	0
bke.lps	sym	3	19	17	119	119	12.37	0.028	1.11e+04	7816	4675	7888	12.45	0.028	106122.667	3.771
brp.lps	sym	3	20	18	378	378	0.23	0	1.40e+04	1823	1328	1001	0.267	0.005	95540	62.225
brp.suminst.lps	sym	3	20	78	1638	1638	0.09	0	1.40e+04	1823	1222	1675	0.127	0.005	94868	62.225
dining_10.lps	sym	3	30	210	4200	4200	0.403	0.005	9.37e+05	569	500	2311	0.453	0.005	94236	0
dining_10.suminst.lps	sym	3	30	50	1000	1000	0.25	0	9.37e+05	569	340	551	0.267	0.005	93376	0
lift3-final.lps	sym	3	30	36	540	540	1.803	0.012	7.57e+03	9916	5154	7852	1.893	0.012	99496	90.51
magic_square.lps	sym	3	1	1	2	2	46.933	0.34	2.00e+00	4	4	4	46.947	0.344	456388	276.512
screen.1-deadlock.lps	sym	3	56	222	29304	29304	557.57	0.318	4.37e+07	11377	1383	2525	557.603	0.323	94156	0
snake.lps	sym	3	6	4	52	52	80.87	0.653	7.60e+04	74731	115209	232750	81.037	0.653	309617.333	1.886
snake.suminst.lps	sym	3	6	10	130	130	79.123	0.327	7.60e+04	74621	125329	612534	79.353	0.327	310072	0
tree.lps	sym	3	1	2	22	22	0.29	0	1.02e+03	1027	2052	2054	0.307	0.005	94316	0
tree.suminst.lps	sym	3	1	2	22	22	0.283	0.005	1.02e+03	1027	2052	2054	0.303	0.005	94316	0
vasy-init.lps	sym	3	484	775	26350	26350	196.39	0.253	9.79e+21	8182	5380	4294	196.693	0.253	251504	0
vasy.lps	sym	3	485	776	26384	26384	200.893	0.42	9.79e+21	9387	5444	4340	201.203	0.428	252078.667	29.273

Table C.2: Results w/ read-write separation for mCRL2 (chain-prev-no-sat)

C.2 Guard-splitting

List of results for guard-splitting with insignificant differences:

- at.5.dve2C
- bke.lps
- blocks.3.dve2C
- brp.lps
- brp.prm.spins
- brp.suminst.lps
- connect-four6x4.lps
- count.pml.spins
- cyclic_scheduler.3.dve2C
- cyclic_sc_heduler.4.dve2C
- dbm.prm.spins
- dining_10.lps
- dining_10.suminst.lps
- elevator2.3.prom.spins
- exit.4.dve2C
- extinction.3.dve2C
- extinction.4.dve2C
- fgs.promela.spins
- fischer.5.dve2C
- fischer.6.dve2C
- frogs.3.dve2C
- krebs.3.dve2C
- krebs.4.dve2C
- lamport.6.prom.spins
- lamport_na.5.dve2C
- lamport_nonatomic.3.prom.spins
- lann.5.dve2C
- leader_filters.6.dve2C
- lift3-final.lps
- loyd.2.dve2C
- magic_square.lps
- p319.pml.spins
- phils.5.prom.spins
- phils.7.dve2C
- phils.8.dve2C
- pouring.2.prom.spins
- production_cell.6.dve2C
- reader_writer.3.prom.spins
- schedule_world.3.dve2C
- SMS.lps
- SMS.suminst.lps
- snake.lps
- snake.suminst.lps
- sorter.4.dve2C
- tree.lps
- tree.suminst.lps
- vasy-init.lps
- vasy.lps
- WMS.lps
- WMS.suminst.lps

Legend for detailed results for guard-splitting:

M	number of guards
$[\![\mathcal{F}]\!]$	amount of nodes needed to store all projected reachable states that do not satisfy guards
$[\![\mathcal{T}]\!]$	amount of nodes needed to store all projected reachable states that satisfy guards

model	backend	n	N	K	checks	ns	$\overline{\text{rt}}$	rt-σ	$\lvert\mathcal{R}\rvert$	$[\![\mathcal{R}]\!]$	$[\![\hookrightarrow]\!]$	$[\![\pi(\mathcal{R})]\!]$	$\overline{\text{at}}$	at-σ	$\overline{\text{mem}}$	mem-σ	M	$[\![\mathcal{F}]\!]$	$[\![\mathcal{T}]\!]$
1394-fin.lps	sym	3	34	925	49950	49950	3.513	0.009	1.89e+05	7384	7157	203103	3.747	0.012	105500	0	-1	-1	-1
SMS.suminst.lps	sym	3	18	54	648	648	0.087	0.005	3.64e+04	1266	524	1285	0.127	0.005	94452	0	-1	-1	-1
WMS.lps	sym	3	30	56	2464	2464	281.813	0.471	2.51e+10	15368	4212	6287	281.953	0.471	137752	0	-1	-1	-1
WMS.suminst.lps	sym	3	30	101	4343	4343	315.347	0.258	2.51e+10	15368	6304	12137	315.48	0.262	138352	0	-1	-1	-1
bke.lps	sym	3	19	17	119	119	12.37	0.028	1.11e+04	7816	4675	7888	12.45	0.028	106122.667	3.771	-1	-1	-1
brp.lps	sym	3	20	18	378	378	0.23	0	1.40e+04	1823	1328	1001	0.267	0.005	95540	62.225	-1	-1	-1
brp.suminst.lps	sym	3	20	78	1638	1638	0.09	0	1.40e+04	1823	1222	1675	0.127	0.005	94868	62.225	-1	-1	-1
dining_10.lps	sym	3	30	210	4200	4200	0.403	0.005	9.37e+05	569	500	2311	0.453	0.005	94236	0	-1	-1	-1
dining_10.suminst.lps	sym	3	30	50	1000	1000	0.25	0	9.37e+05	569	340	551	0.267	0.005	93376	0	-1	-1	-1
lift3-final.lps	sym	3	30	36	540	540	1.803	0.012	7.57e+03	9916	5154	7852	1.893	0.012	99496	90.51	-1	-1	-1
magic_square.lps	sym	3	1	1	2	2	46.933	0.34	2.00e+00	4	4	4	46.947	0.344	456388	276.512	-1	-1	-1
screen.1-deadlock.lps	sym	3	56	222	29304	29304	557.57	0.318	4.37e+07	11377	1383	2525	557.603	0.323	94156	0	-1	-1	-1
snake.lps	sym	3	6	4	52	52	80.87	0.653	7.60e+04	74731	115209	232750	81.037	0.653	309617.333	1.886	-1	-1	-1
snake.suminst.lps	sym	3	6	10	130	130	79.123	0.327	7.60e+04	74621	125329	612534	79.353	0.327	310072	0	-1	-1	-1
tree.lps	sym	3	1	2	22	22	0.29	0	1.02e+03	1027	2052	2054	0.307	0.005	94316	0	-1	-1	-1
tree.suminst.lps	sym	3	1	2	22	22	0.283	0.005	1.02e+03	1027	2052	2054	0.303	0.005	94316	0	-1	-1	-1
vasy-init.lps	sym	3	484	775	26350	26350	196.39	0.253	9.79e+21	8182	5380	4294	196.693	0.253	251504	0	-1	-1	-1
vasy.lps	sym	3	485	776	26384	26384	200.893	0.42	9.79e+21	9387	5444	4340	201.203	0.428	252078.667	29.273	-1	-1	-1

Table C.3: Results w/o guard-splitting for mCRL2 (chain-prev-no-sat)

model	backend	n	N	K	checks	ns	$\overline{\text{rt}}$	rt-σ	$\lvert\mathcal{R}\rvert$	$[\![\mathcal{R}]\!]$	$[\![\hookrightarrow]\!]$	$[\![\pi(\mathcal{R})]\!]$	$\overline{\text{at}}$	at-σ	$\overline{\text{mem}}$	mem-σ	M	$[\![\mathcal{F}]\!]$	$[\![\mathcal{T}]\!]$
1394-fin.lps	sym	3	34	925	1091	1091	0.713	0.005	1.89e+05	8013	6198	1640	1.003	0.005	120920	248.902	249	7005	966
SMS.suminst.lps	sym	3	18	54	394	394	0.1	0	3.64e+04	1266	459	245	0.14	0	94544	0	60	397	218
WMS.lps	sym	3	30	56	2114	2114	266.137	0.219	2.51e+10	15368	4153	4585	266.28	0.218	831800	62.225	46	237	136
WMS.suminst.lps	sym	3	30	101	3126	3126	278.683	0.732	2.51e+10	15368	6165	5642	278.817	0.736	1304064	0	83	497	241
bke.lps	sym	3	19	17	67	67	11.81	0.033	1.11e+04	7816	4624	3811	11.91	0.033	105840	0	19	107	56
brp.lps	sym	3	20	18	273	273	0.29	0	1.40e+04	1823	1284	630	0.327	0.005	95620	0	18	119	54
brp.suminst.lps	sym	3	20	78	796	796	0.08	0	1.40e+04	1823	912	247	0.12	0	94992	0	47	318	194
dining_10.lps	sym	3	30	210	2373	2373	0.477	0.005	9.37e+05	569	430	830	0.57	0	95100	0	130	500	239
dining_10.suminst.lps	sym	3	30	50	823	823	0.363	0.005	9.37e+05	569	230	50	0.383	0.005	93896	0	90	347	270
lift3-final.lps	sym	3	30	36	276	276	2.233	0.009	7.57e+03	9916	5109	5671	2.333	0.009	99212	0	32	408	150
magic_square.lps	sym	3	1	1	1	1	53.49	0.324	2.00e+00	4	3	1	53.51	0.324	413905.333	159.644	1	3	3
screen.1-deadlock.lps	sym	3	56	222	22940	22940	443.917	0.161	4.37e+07	11377	902	222	443.967	0.169	5327008	329.266	153	552	443
snake.lps	sym	3	6	4	42	42	88.78	0.589	7.60e+04	74731	103622	62877	88.907	0.588	312538.667	1.886	4	40514	11591
snake.suminst.lps	sym	3	6	10	109	109	117.113	0.52	7.60e+04	74621	113740	66043	117.277	0.515	319464	0	20	148458	66686
tree.lps	sym	3	1	2	20	20	0.3	0	1.02e+03	1027	2052	1028	0.313	0.005	94382.667	1.886	1	515	514
tree.suminst.lps	sym	3	1	2	20	20	0.3	0	1.02e+03	1027	2052	1028	0.31	0	94384	0	1	515	514
vasy-init.lps	sym	3	484	775	10149	10149	201.487	0.382	9.79e+21	8182	4002	775	201.977	0.386	3455068	549.89	484	1438	1452
vasy.lps	sym	3	485	776	10150	10150	204.803	0.534	9.79e+21	9387	4065	776	205.293	0.532	3455781.333	766.153	485	1453	1455

Table C.4: Results w/ guard-splitting for mCRL2 (chain-prev-no-sat)