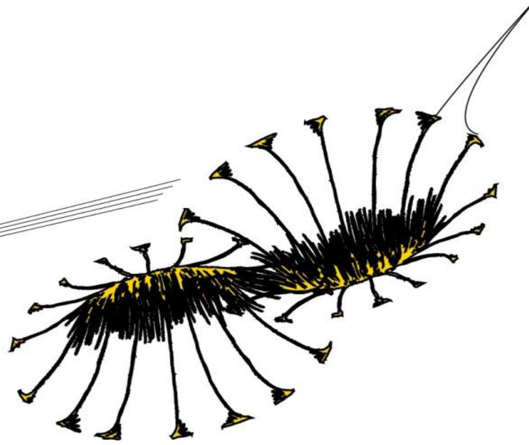
Saturation for LTSmin

Master Thesis

Final project (course code 192199978)
Course year 2011 – 2012

Tien Loong Siaw (s0045217)

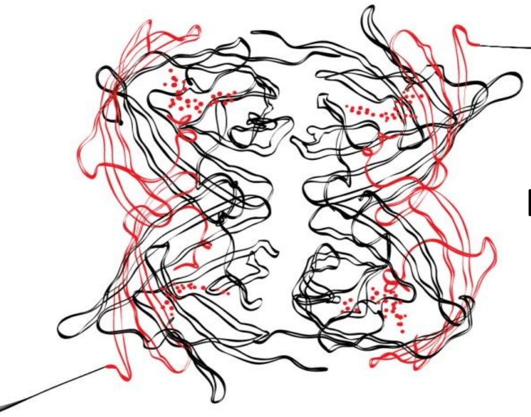


Master Computer Science
Track Software Engineering

Graduation committee:

- prof. dr. J.C. van de Pol
- dr. J. Ketema
- prof. dr. ir. A. Rensink

Department of Formal Methods and Tools (FMT)
University of Twente



Enschede, Wednesday 15th of February 2012

Abstract

State space generation or reachability analysis plays an important role in model checking, but a disadvantage of current techniques lies in the fact that they require quite a lot of time and memory to come up with a result when using real-life system models. Symbolic state space generation using known traversal techniques as breadth-first search and chaining are quite common practice, but their performance on real-life system models with a large transition relation remains an issue to be tackled.
In the past decade a relatively new traversal technique has emerged, named Saturation. This traversal technique has proven itself to be a good competitor to traditional symbolic state space generation techniques for handling models with extremely large state spaces. It originates from the research group of professor Gianfranco Ciardo (University of California at Riverside, USA).

The main goal of this Master project is to design and implement the aforementioned Saturation-based approach in the LTSmin toolset, which is a set of verification tools developed by the Formal Methods and Tools group (FMT) at the University of Twente. The main features of this toolset is its setup in architectural layers to separate language-specific details from verification algorithms and the use of an interface for presenting the (partitioned) transition relation of the model, called PINS.
The goal includes making adjustments to the LTSmin architecture for Saturation to work properly and comparing the implemented Saturation algorithm with other available symbolic reachability techniques in the LTSmin toolset.

An analysis of the Saturation-based approaches by G. Ciardo and the architecture of the LTSmin toolset is performed to choose the Saturation approach which fits best into the LTSmin toolset. The chosen (stand-alone) Saturation algorithm is then adjusted for implementation. The first version of this adjusted algorithm takes a different approach when updating the partitioned transition relation of the model compared to the algorithm proposed by G. Ciardo and reuses available functionality from the LTSmin toolset. An evaluation of this first version of the Saturation algorithm is performed comparing it to the other available reachability methods breadth-first search, chaining and an LTSmin version of a Saturation-like approach.

The evaluation reveals that the update process of the partitioned transition relation does not perform well for complex models resulting in bad time performance. Therefore the update process of the partitioned transition relation is revised. Hereby inspiration is taken from the update process used by G. Ciardo. This has resulted in a Saturation algorithm that outperforms the previous version of this Saturation algorithm and comes out on top of the traditional traversal techniques breadth-first search and chaining with regard to time performance. Compared to the LTSmin version of a Saturation-like algorithm it performs competitively well too.

Furthermore an existing reachability option of the LTSmin toolset which basically computes fix-points has been readjusted to make use of Saturation, where updates to the partitioned transition relation is done outside Saturation. This algorithm turns out to perform as good as well compared to breadth-first search and chaining, but not as great as the revised version of the stand-alone Saturation algorithm. A similar result is obtained as with the stand-alone Saturation algorithm when compared to the LTSmin version of a Saturation-like algorithm.

Preface

Initially I began my life as a student at the University of Twente in the autumn of 2002 by studying Industrial Design, but after fours years I realised this is not what I wanted to continue later in my career. So a year after I obtained my Bachelor degree in Industrial Design (in the autumn of 2005) I took the decision to switch over to Computer Science (in the autumn of 2006) where I hoped to find the career path that I was looking for. My interests in the field of Computer Science were initially in software design and especially in providing the graphical user interface of a software program. In numerous study projects and assignments throughout the years I was mainly responsible for the creative and user-oriented side of the project. Being creative and creating things was and still is one of the things I cherish in my life.
In the last few years of studying Computer Science I discovered other interests in this field, namely software verification and algorithmic design challenges. The latter is due to a number of individual projects that I performed, such as designing and implementing an algorithm to detect rip currents in images of a shoreline (as part of the minor at the ITC in the winter of 2009), displaying UML diagrams more user-friendly on a screen (in my traineeship at Novulo in the second half of 2010) and adapting the Saturation algorithm as new state space generation technique for the LTSmin toolset, which is the subject of my graduation project.
In all these individual projects concerning an algorithmic design and implementation, I was initially quite hesitant about the assignments themselves, because the descriptions of the assignments were very technical and it is hard to imagine what challenges you have to face. But in all these cases it turned out that my initial hesitation was unnecessary. It was interesting to work on such challenging projects, together with the good support that was provided by my supervisors.

For my graduation project I came into contact with Jaco van de Pol at the end of 2010 and he suggested me to work on the Saturation algorithm as state space generation technique. Since then I worked on this subject for a year now which started with a better understanding in the new technique in Research Topics, followed by the graduation project in July 2011. Next to my main supervisor Jaco van de Pol, my weekly supervisor used to be Michael Weber. He was of great assistance when I faced difficulties, but nearing the end of Research Topics I noticed that Michael was busy preparing himself moving to the USA. After his departure and the completion of Research Topics, I was assigned a new weekly supervisor, namely Jeroen Ketema. With him as supervisor, I had regular weekly meetings to discuss my progress and to provide me with useful feedback. Although there were times he could get agitated slightly for my lack of clarity in some of my e-mail correspondences with him, he also helped me out at times when I was stuck when working with the LTSmin toolset. Next to the weekly meetings with Jeroen, I also had monthly meetings with both Jeroen and Jaco, which mainly focused on my progress. With their guidance and feedback I can finally present to you this report.

After studying for almost ten years at the University of Twente, my life as student comes to an end. First of all I want to thank my supervisors Jaco and Jeroen for their guidance, structural feedback and good response on my work. Secondly I would also like to thank Michael for being my initial supervisor on this subject and for his help on my initial work. Furthermore I would also like to thank my study advisor for supporting me throughout the years and give a helping hand from time to time. My final thanks go to my friends and family, especially my parents for having the patience to let me finish my studies. Although they did not show their support explicitly, they gave me the strength to persist in my studies.

Tien Loong Siaw
Enschede, 15th of February 2012

Table of contents

List of figures

List of listings

List of tables

1 Introduction

Real-life critical software systems are developed with reliability and correctness in mind. During the development of such systems formal verification can play an important role. In formal verification the system under development is abstracted into a model which is checked against the given specifications to assess any errors and design flaws that may occur. One of such formal verification techniques is model checking and tools exist which are capable of automatically performing such verification on formal properties (which are the formalizations of the given specifications). Such tools basically assess these properties by considering all states of the model reachable from the initial state(s), which is referred to as state space generation or reachability analysis. The tool exhaustively searches all possible reachable states of the model using a transition relation or next-state function of the model. The latter indicates how a model can undergo a transition from one state to another. A major hurdle here is the inability to deal with large complex models containing billions of states.
Traditionally two approaches have been used for state space generation [7]:

- Explicit state space generation: states are found individually and stored explicitly one-by-one.
- Implicit or symbolic state space generation: multiple states are found and stored together in sets.

Unfortunately, both approaches suffer from the state space explosion problem in which case it becomes infeasible in both time and memory to compute the reachable state space when the system model becomes larger and more complex. The number of states that can be handled with the explicit approach lies roughly around 10^7 states [7] and this is mainly due to the problem that time and space complexity increase linearly with increasing reachable state space (when considering the fact that the exploration happens in a sequential manner). The symbolic approach can handle reachable state spaces up to approximately 10^{20} states [7] when the sets of states are stored using Binary Decision Diagrams (BDDs). This encoding is also used for storing the transition relations, so operations on BDDs are mainly affected by the number of nodes in the BDDs. The main challenges of the symbolic approach [7] are as follows:

- The maximum number of BDD nodes handled and stored, is called the peak BDD size and may require an excessive amount of memory that is not available, although the final BDD size can be quite compact.
- Handling BDDs as one rigid structure can be infeasible when they are very large and complex. Partitioning techniques can be used that partition the transition relation in more manageable subrelations or events. Conjunctive partitioning is used for synchronous system models, since the transition relation TR can be viewed as a conjunction of N individual events E_i, denoted as $TR = E_1 \wedge E_2 \wedge \ldots \wedge E_N$. Similarly disjunctive partitioning is used for asynchronous system models, where the transition relation TR is a disjunction of N individual events E_i, denoted as $TR = E_1 \vee E_2 \vee \ldots \vee E_N$. Also combinations of these partitioning techniques occur. The main challenge is to find a suitable and manageable partitioning for the transition relation.

For finding the reachable state space using the symbolic approach, breadth-first search traversal is the basic technique [10], which can come in the form of applying the transition relation on the set of newly discovered unexplored states or on the set of all discovered states so far as shown in Listing 1.1. A variant to this approach is chaining, in which each kind of event is fired separately on the set of newly discovered unexplored states. This set of states is updated immediately after firing each individual event and grows like a ‘chain’ during these firings. Instead of working with newly discovered unexplored states, it is also possible to use the set of all discovered states so far, which is actually the standard version of this approach. Both variations on chaining are shown in Listing 1.2. Some common variables and functions in these pseudo code algorithms are shortly mentioned in Listing 1.3.

When comparing breadth-first search and chaining, the latter appears to be a better traversal approach [10]. But for real-life complex system models, both approaches are still unable to handle extremely large state spaces. Next to these two approaches, a relatively new technique for traversing a state space of a system model has emerged to tackle this problem. This approach is called Saturation, which is developed by G. Ciardo [8]. It has proven itself as a good competitor to traditional symbolic state space generation techniques for handling models with extremely large state spaces and different versions of the approach have been devised.

```
set<state> visited := { init }
set<state> current := visited
set<state> next_set := { }
set<state> tmp := { }

while (current != { }) :
  next_set := { }
  foreach (int i in 1..evts.length() :
    tmp := next(current, evts[i])
    tmp := tmp - visited
    next_set := next_set + tmp
  visited := visited + next_set
  current := next_set
```

```
set<state> visited := { init }
set<state> old_vis := { }
set<state> tmp := { }

while (visited != old_vis) :
  old_vis := visited
  foreach (int i in 1..evts.length) :
    tmp := next(old_vis, evts[i])
    visited := visited + tmp

          'set + set' = set union
          'set - set' = set difference
```

Listing 1.1: Pseudo code of breadth-first search algorithm using newly unexplored states only (left) or all states (right).

```
set<state> visited := { init }
set<state> new_set := visited
set<state> tmp := { }

while (new_set != { }) :
  foreach (int i in 1..evts.length) :
    tmp := next(new_set, evts[i])
    tmp := tmp - visited
    new_set := new_set + tmp
  visited := visited + new_set
  new_set := new_set - visited
```

```
set<state> visited := { init }
set<state> old_vis := { }
set<state> tmp := { }

while (visited != old_vis) :
  old_vis := visited
  foreach (int i in 1..evts.length) :
    tmp := next(visited, evts[i])
    visited := visited + tmp

          'set + set' = set union
          'set - set' = set difference
```

Listing 1.2: Pseudo code of chaining algorithm using newly unexplored states only (left) or all states (right).

```
// Variables:                                The indices of arrays are numbered from
// Initial state of model.                   1 to length.
state init

// List containing all events (partitioned transition relations).
array<event> evts

// Function:
// Compute next (successor) states starting from from_states using event evt.
set<state> next(set<state> from_states, event evt)
```

Listing 1.3: Variables and function definitions for pseudo code in previous two listings.

1.1 Goal statement

The Formal Methods and Tools (FMT) research group at the University of Twente has developed its own verification toolset called LTSmin which is the result from years of research and includes a tool for symbolic model checking [5]. It contains a clean medium for representing the states of a model in a uniform way where an interface, called PINS is used [4]. Using this interface the LTSmin toolset is capable of separating the language specific details of modelling languages, from the model checking algorithms used for verification. The tool implements several symbolic reachability algorithms, such as breadth-first search and chaining. Also LTSmin versions of Saturation-like algorithms are implemented, but these have not yet been compared with those by G. Ciardo.

The aim is to extend the symbolic reachability algorithms in the LTSmin toolset with the Saturation-based approach by G. Ciardo. The approach should be incorporated in the tool with its architecture and PINS interface in mind, and should maintain the benefits of keeping the algorithm independent from any language module. Basically the tool should eventually be capable of using Saturation as reachability technique for state space generation of any model that the tool can handle.

The main goal of this project is summarized as follows:
“Design and assess a Saturation-based approach as devised by G. Ciardo for the LTSmin toolset.”

To achieve the given goal, the following research questions provide guidance in this project:

- *“Different versions of the Saturation approach exist and which one of these is a good candidate for extending the symbolic reachability techniques of the LTSmin toolset?”*
- *”How can the chosen version of the Saturation approach be implemented in the LTSmin toolset and what are the consequences for the tool’s architecture?”*
- *“How does the chosen Saturation approach perform in the LTSmin toolset compared to other symbolic reachability techniques in the tool?”*

1.2 Document structure

The report follows the research questions to reach the main goal of this project. Chapter 2 gives an overview of the different Saturation approaches that exist and how they evolved over time. It will provide a basis for the first research question. Chapter 3 explains the architecture of the LTSmin toolset and the main focus lies with parts of the toolset that deal with reachability analysis. This will provide a basic understanding of how the LTSmin toolset is set up for answering the second research question. Chapter 4 gives an answer to the first two research questions by discussing the underlying choice of the Saturation approach to implement and how the algorithm can be incorporated in the LTSmin toolset. It also continues with discussing the issues that arise during actual implementation in the LTSmin toolset and how these are solved (relating to the second research question as well). Chapter 5 finalizes the implementation of the chosen Saturation approach by evaluating its performance, as indicated in the third research question. But the report does not end here and continues with an improvement on the implemented Saturation approach in chapter 6. This chapter discusses the design, implementation and evaluation of an improved version of the implemented Saturation approach and will therefore also contribute to the third research question. The report ends in chapter 7 with how the main goal of implementing a Saturation approach into the LTSmin toolset is reached and what future directions can be taken to make the approach a viable alternative next to the available reachability options of the LTSmin toolset.

2 Overview of Saturation approaches

This chapter starts with basic definitions of state spaces and next-state functions (or transition relations) of system models. Section 2.2 and 2.3 each give an overview of one of the two main categories of the Saturation approaches as devised by G. Ciardo.

2.1 Definitions and encodings for state space and next-state function

A discrete-state model is used for encoding a model of the system under consideration [10][7].

Definition 2.1. A discrete-state model is a tuple $(\hat{\mathcal{S}}, \mathcal{N}, \mathcal{S}^{init})$ [7], where

- $\hat{\mathcal{S}}$ is the set of all possible states $\boldsymbol{i}$ of the system (reachable or not);
- $\mathcal{N}: \hat{\mathcal{S}} \rightarrow 2^{\hat{\mathcal{S}}}$ is the next-state function, $\mathcal{N}(\boldsymbol{i})$ consists of all states reachable in one step from $\boldsymbol{i}$;
- $\mathcal{S}^{init} \subseteq \hat{\mathcal{S}}$ is the set of initial states.

For discussing the Saturation approaches, the set $\mathcal{S}^{init}$ will only contain a single initial state and the approaches can be extended to multiple initial states. The set of reachable states of the system is denoted by $\mathcal{S} \subseteq \hat{\mathcal{S}}$.

The model is usually decomposed into a number of $\mathcal{K}$ components and a model's state $\boldsymbol{i}$ is a vector written as a $\mathcal{K}$-tuple $(i_{\mathcal{K}}, \ldots, i_1)$ with $\hat{\mathcal{S}} = \mathcal{S}_{\mathcal{K}} \times \ldots \times \mathcal{S}_1$ and $i_\ell \in \mathcal{S}_\ell$ for all $1 \leq \ell \leq \mathcal{K}$. Each state slot i_ℓ in the state vector corresponds to a local state of the corresponding component ℓ. For convenience, the local states of a component ℓ are assumed to be integer values ranging from 0 to $n_\ell - 1$, with n_ℓ the size of the local state space of component ℓ [10].
When the discrete-state model uses $\mathcal{K}$ variables in its specification and a model's state consists of enumerating the values of these $\mathcal{K}$ variables (corresponding to the Cartesian product $\mathcal{S}_{\mathcal{K}} \times \ldots \times \mathcal{S}_1$), then a state slot in a state vector corresponds to one of these $\mathcal{K}$ variables (one of the elements of the Cartesian product). The variables are ordered in a predefined way (mostly as part of the model's specification) and the local state space sizes correspond to the domains of these variables.

For every Saturation approach the state space is stored as a quasi-reduced $\mathcal{K}$-level Multi-valued Decision Diagram [10]. They are almost similar to BDDs, but instead of 2 outgoing arcs per node, each node contains n_ℓ outgoing arcs. Each level ℓ corresponds to the accompanying component ℓ of the model.
The next-state function is stored in different ways for different Saturation approaches and is discussed later. Mostly it is not stored in a single rigid data structure, but split into smaller manageable parts.

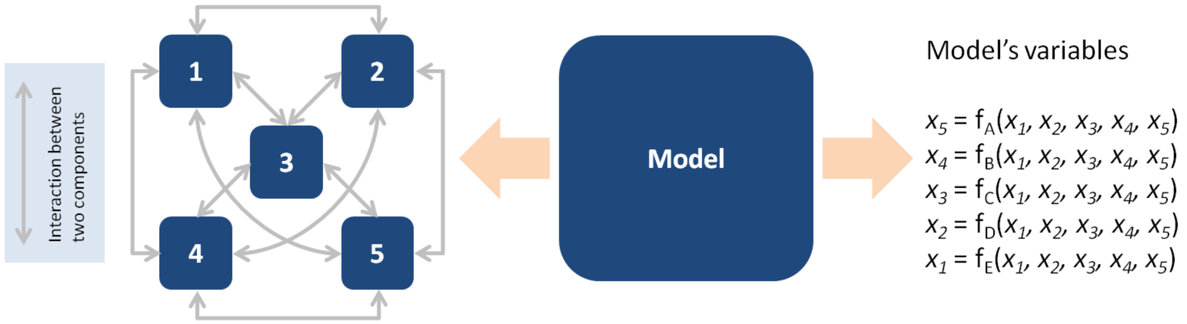
Definition 2.2. A quasi-reduced $\mathcal{K}$-level Multi-valued Decision Diagram (abbreviated to MDD) is a directed acyclic edge-labelled multi-graph [7], where:

- Nodes are put into levels ranging from $\mathcal{K}$ to 0 (i.e. $\mathcal{K} + 1$ levels).
- Level $\mathcal{K}$ contains a single non-terminal root node r and one or more non-terminal nodes are situated at levels $\mathcal{K} - 1$ to 1.
- Level 0 only has two terminal nodes, namely $\mathcal{F}$ and $\mathcal{T}$ (False and True MDD nodes, respectively).
- A non-terminal node p at level ℓ contains n_ℓ arcs pointing to nodes at the lower level $\ell - 1$.
- Duplicate nodes (with same node values and same set of incoming and outgoing arcs) are not allowed, but redundant (non-terminal) nodes at level ℓ where all arcs of the node point to the same node at level $\ell - 1$ are (hence quasi-reduced).

A path starting from the root node of an MDD and ending in the True node indicates a reachable state of the model. The MDD as defined above is not used as such, since this would also store the MDD nodes of unreachable states and the False MDD node. In practice, a reduced version of this MDD is stored, containing only the reachable states. That redundant MDD nodes are stored, is an artefact of building the MDD level by level from the bottom-up by the Saturation algorithm (more about this in section 2.2.1).

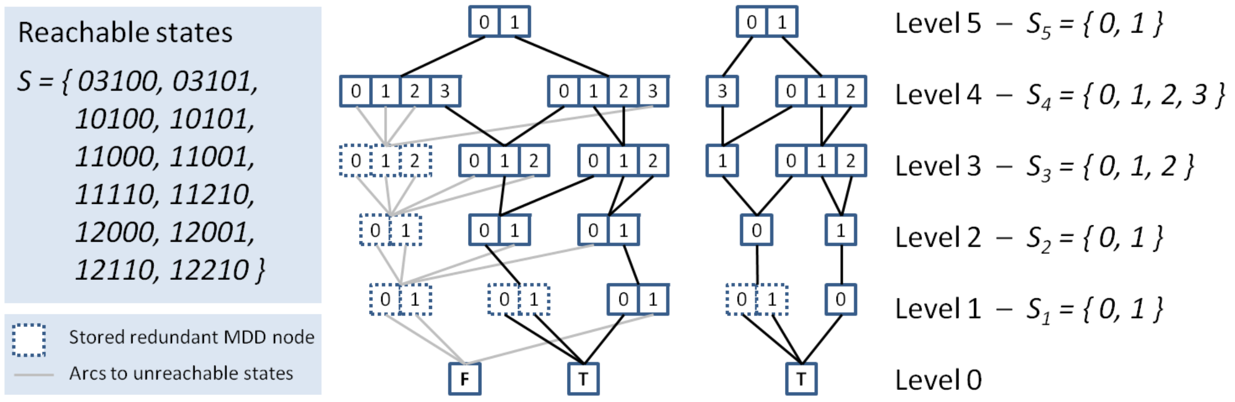
■ From model to MDD example

To give a sense what an MDD of a reachable state space looks like, consider one has a model specification and that the entire model itself can be decomposed into 5 smaller components which interact with each other in a certain way (so $\mathcal{K} = 5$), as shown on the left in Figure 2.1. Another representation is that the model is specified using 5 variables and each is defined by a (mathematical) function, as shown on the right in Figure 2.1. A state vector of this model is written as a 5-tuple $(i_5, i_4, i_3, i_2, i_1)$ with $\hat{\mathcal{S}} = \mathcal{S}_5 \times \mathcal{S}_4 \times \mathcal{S}_3 \times \mathcal{S}_2 \times \mathcal{S}_1$. Each component has its own local state space size n_ℓ and the values of the local state spaces are given on the right in Figure 2.2 (and likewise for the domain of the 5 variables of the model).
Furthermore imagine the model can have a set of reachable states $\mathcal{S}$ as given on the left in Figure 2.2 and this set is stored in an MDD, as depicted in the two MDDs. The left MDD stores every single state of the model and each MDD node has n_ℓ outgoing arcs (according to **Definition 2.2**), while the right MDD only stores the reachable states of the model.
A path starting from the root node at level 5 to the True MDD node at level 0 represents a reachable state of the model. So for state (0, 3, 1, 0, 1) one would start at the root node (at level 5) and take the arc pointing out of entry value 0. Arriving at the MDD node at level 4, one needs to take the arc heading out of entry value 3 to the next MDD node at level 3, and so on until the True MDD node at level 0.
The MDDs do not contain duplicate nodes, since there are no two MDD nodes with the same entry values and that have the same set of incoming and outgoing arcs (pointing to the same set of MDD nodes).



Interaction between two components
1
2
3
4
5
Model
Model's variables
$x_5 = f_A(x_1, x_2, x_3, x_4, x_5)$
$x_4 = f_B(x_1, x_2, x_3, x_4, x_5)$
$x_3 = f_C(x_1, x_2, x_3, x_4, x_5)$
$x_2 = f_D(x_1, x_2, x_3, x_4, x_5)$
$x_1 = f_E(x_1, x_2, x_3, x_4, x_5)$

Figure 2.1: Example of a model represented as a decomposition (left) and as a function (right).



Reachable states
S = { 03100, 03101, 10100, 10101, 11000, 11001, 11110, 11210, 12000, 12001, 12110, 12210 }
Stored redundant MDD node
Arcs to unreachable states
F
T
Level 5 – S_5 = { 0, 1 }
Level 4 – S_4 = { 0, 1, 2, 3 }
Level 3 – S_3 = { 0, 1, 2 }
Level 2 – S_2 = { 0, 1 }
Level 1 – S_1 = { 0, 1 }
Level 0

Figure 2.2: Example MDDs which store every state (left) and only reachable states (right).

2.2 Saturation approaches for Kronecker-consistent models

The first group of Saturation approaches require that the decomposition of the next-state function is Kronecker consistent, meaning that the effect of an event e on component ℓ only depends on its corresponding local state space. Thus, it is possible to find $\mathcal{K} \cdot |\mathcal{E}|$ subfunctions $\mathcal{N}_{\ell,e}$, where $\mathcal{E}$ denotes the set of events. Some modelling formalisms such as basic Petri Nets satisfy this property automatically, but this is not always the case. In these other cases, one can still obtain a Kronecker-consistent decomposition of the next-state function by splitting and/or combining events, or performing these operations on the model components. This needs to be done manually before providing the model to the Saturation algorithm, because currently there is no technique available that can handle this automatically for every general type of model (it can for certain types of Petri Nets as implemented in the research tool SMART by G. Ciardo [7][6]).

The next-state function is represented as a Kronecker product on Boolean adjacency matrices. A Kronecker product on two matrices is defined below.

Definition 2.3. Kronecker product (or matrix direct product) of two matrices [25].

- Given: $m \times n$ matrix A and $p \times q$ matrix B
- Result: $(mp) \times (nq)$ matrix C with elements $c_{xy} = a_{ij} b_{k\ell}$, where $x \equiv p(i-1)+k$ and $y \equiv q(j-1)+\ell$.
- Notation: C = A $\otimes$ B

Basically, the Kronecker product on two matrices is a form of matrix multiplication, where intuitively each value of matrix A is multiplied with the entire matrix B. Take for example the Kronecker product of a 1×3 matrix A and a 2×3 matrix B, which results in a 2×9 matrix C, as shown in Figure 2.3.

$$\mathrm{A} \otimes \mathrm{B} = \begin{bmatrix} a_{11}\mathrm{B} & a_{12}\mathrm{B} & a_{13}\mathrm{B} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{11}b_{13} & a_{12}b_{11} & a_{12}b_{12} & a_{12}b_{13} & a_{13}b_{11} & a_{13}b_{12} & a_{13}b_{13} \\ a_{11}b_{21} & a_{11}b_{22} & a_{11}b_{23} & a_{12}b_{21} & a_{12}b_{22} & a_{12}b_{23} & a_{13}b_{21} & a_{13}b_{22} & a_{13}b_{23} \end{bmatrix}$$

Figure 2.3: Example of Kronecker product for a 1 × 3 matrix A and a 2 × 3 matrix B.

Each row and column of the Boolean adjacency matrix $\mathcal{N}_{\ell,e}$ specifies the local state values for a particular component ℓ and a matrix cell simply indicates if a local state value (row) can go to another local state value (column). These Boolean adjacency matrices $\mathcal{N}_{\ell,e}$ are sorted per event e and component ℓ. The Kronecker product of all these submatrices $\mathcal{N}_{\ell,e}$ then forms the next-state function. So a decomposition of the next-state function is: $\mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e$ and each $\mathcal{N}_e$ represents the following Kronecker product: $\mathcal{N}_e = \mathcal{N}_{\mathcal{K},e} \otimes \ldots \otimes \mathcal{N}_{1,e}$, where each submatrix $\mathcal{N}_{\ell,e}$ represents a part of the next-state function for component ℓ and event e. Table 2.1 visualizes this product in matrix form.

		Events					
		1	2	$\dots$	e	$\dots$	n
	$\mathcal{K}$	$\mathcal{N}_{\mathcal{K},1}$	$\mathcal{N}_{\mathcal{K},2}$	$\dots$	$\mathcal{N}_{\mathcal{K},e}$	$\dots$	$\mathcal{N}_{\mathcal{K},n}$
	$\mathcal{K}-1$	$\mathcal{N}_{\mathcal{K}-1,1}$	$\mathcal{N}_{\mathcal{K}-1,2}$		$\mathcal{N}_{\mathcal{K}-1,e}$		$\mathcal{N}_{\mathcal{K}-1,n}$
	$\vdots$	$\vdots$					$\vdots$
Levels	ℓ	$\mathcal{N}_{\ell,1}$	$\mathcal{N}_{\ell,2}$		$\mathcal{N}_{\ell,e}$		$\mathcal{N}_{\ell,n}$
	$\vdots$	$\vdots$					$\vdots$
	2	$\mathcal{N}_{2,1}$	$\mathcal{N}_{2,2}$		$\mathcal{N}_{2,e}$		$\mathcal{N}_{2,n}$
	1	$\mathcal{N}_{1,1}$	$\mathcal{N}_{1,2}$	$\dots$	$\mathcal{N}_{1,e}$	$\dots$	$\mathcal{N}_{1,n}$

Table 2.1: Kronecker product of next-state function in matrix form.

An advantage of storing the next-state function in this manner, is that the submatrices $\mathcal{N}_{\ell,e}$ are quite sparse. Moreover, if the occurrence of certain events do not affect the local state space of a component ℓ, then the submatrix $\mathcal{N}_{\ell,e}$ is simply the identity matrix and can be stored more efficiently [10].

In the following subsections two variants of this group of Saturation approaches will be discussed in detail, as summarized in Table 2.2. For convenience, they will be denoted as Kronecker Saturation approaches from now on.

	Kronecker Saturation approaches	
Name	Kronecker Prebuilt Saturation	Kronecker On-the-fly Saturation
Explanation	Next-state function is Kronecker consistent and is built in advance (prebuilt) before starting the Saturation process. (Local state space sizes n_ℓ are known.)	Next-state function is Kronecker consistent and built on-the-fly during the Saturation process. (Local state space sizes n_ℓ are unknown.)

Table 2.2: Subcategories of Kronecker Saturation approaches.

■ Running example 1 – Model description

As a running example for Kronecker Saturation throughout this report, a simple program model is used which consists of three (interleaving) parallel processes and three global variables x, y and z. To obtain a Kronecker-consistent next-state function, the individual updating statements (not conditions) of the processes can only rely on at most a single variable, namely their own variable. Each process is considered to consist of a single event and the three variables x, y and z correspond to level 3, 2 and 1 of the MDD, respectively. An overview of the program model is given below in Table 2.3.

Level	Variable	Initial value	Local state space
3	x	0	$S_3 = \{0, 1, 2\}$
2	y	0	$S_2 = \{0, 1, 2, 3, 4\}$
1	z	1	$S_1 = \{0, 1\}$

Event	Process of program
0	$if\ x < 2\ and\ y < 3\ then$ $\{x = x + 1 ; y = y + 2 ; \}$
1	$if\ y > 0\ and\ z > 0\ then$ $\{y = y - 1 ; z = z - 1 ; \}$
2	$if\ x < 2\ and\ z > 0\ then$ $\{x = x + 1 ; z = z - 1 ; \}$

Table 2.3: Example model 1 with a Kronecker-consistent next-state function.

■ Running example 1 – Kronecker matrix of next-state function

Following example model 1, the submatrices $\mathcal{N}_{l,e}$ per level and event of the entire next-state function can be determined, which is shown in (Kronecker) matrix form in Figure 2.4. So the entire matrix depicted forms the Kronecker product of the complete next-state function $\mathcal{N}$ and each (large) matrix cell contains a Boolean adjacency matrix $\mathcal{N}_{l,e}$ (for readability reasons crosses are used to indicate the value True and empty cells indicate the value False).

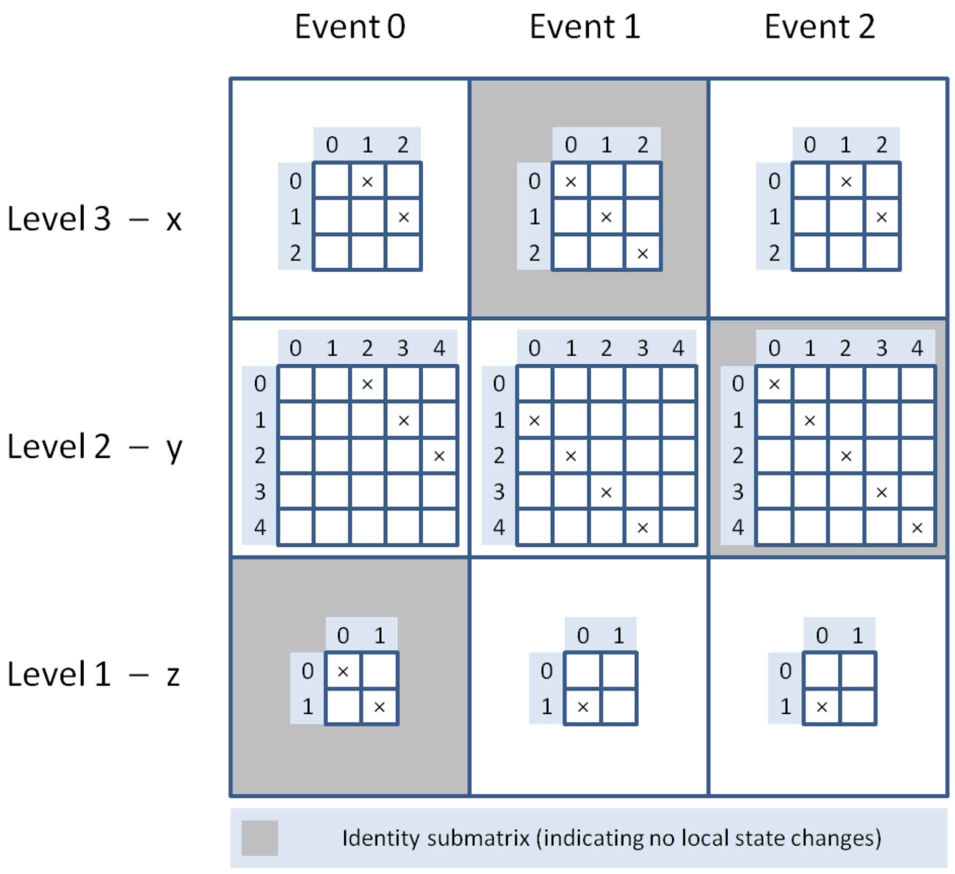


Figure 2.4: Kronecker product of next-state function in matrix form for example model 1.

2.2.1 Kronecker Prebuilt Saturation approach

The most basic version of the Kronecker Saturation approach is first presented in [8] and later more extensively discussed in [10] and [7]. Next to the requirement of Kronecker consistency of the next-state function, this version of the Saturation algorithm also requires that n_l, the local state space size of component l, is known in advance. With this knowledge, it is easier to map local states to their corresponding local state values (ranging from 0 to $n_l - 1$). And with the known local state space sizes, the next-state function can be prebuilt as a Kronecker product of Boolean adjacency matrices $\mathcal{N}_{l,e}$ before generating the reachable state space using Saturation. Therefore this variant of the Kronecker Saturation approach will be denoted as Kronecker Prebuilt Saturation.

The Kronecker Prebuilt Saturation approach makes use of the notion of node saturation, which basically means that an MDD node at a certain level is only saturated if this node and all nodes at lower levels do not change anymore with regard to the firing of events that only affect these levels. Hereby the concept of event locality comes into play, where certain events only affect certain MDD levels. For each event, the

range of levels can be determined in which the local states of the corresponding levels undergo a transition to another local state and these are indicated with the top- and bottommost levels per event e, respectively $\mathcal{Top}(e)$ and $\mathcal{Bot}(e)$. Levels above and below this range are not affected by event e, since these only contain identity transitions. For the Saturation algorithm, this means that in a level ℓ an event e is only fired when it falls within the range between $\mathcal{Top}(e)$ and $\mathcal{Bot}(e)$, avoiding unnecessary firings where the local state space remains the same. Hereby the entire set of events $\mathcal{E}$ is partitioned into event sets per level ℓ, namely $\mathcal{E}_\ell = \{ e \in \mathcal{E}\colon \mathcal{Top}(e) = \ell \}$.
In the traditional symbolic approach the unsaturated BDD nodes found, are mostly immediately replaced by (a number of) new BDD nodes that encode a larger set of states [7]. The number of unsaturated nodes is very unpredictable and changes all the time, which contributes largely to the peak BDD size. This is avoided in the Saturation algorithm by getting rid of unsaturated nodes as early as possible.

■ Running example 1 – $\mathcal{Bot}(e)$ and $\mathcal{Top}(e)$

In Table 2.4 $\mathcal{Bot}(e)$ and $\mathcal{Top}(e)$ of the three events are shown for example model 1. So for example the bottommost level for event 0 is level 2 and this means that every submatrix $\mathcal{N}_{\ell,e}$ below level 2 of event 0 are just identity matrices (containing only identity transitions). And this is similar for the topmost level, but then for submatrices $\mathcal{N}_{\ell,e}$ above the topmost level.
The submatrices within the range of $\mathcal{Bot}(e)$ and $\mathcal{Top}(e)$ for a certain event e can still contain identity submatrices, as is the case with event 3, as can be seen in Figure 2.4.

Event e	$\mathcal{Bot}(e)$	$\mathcal{Top}(e)$
0	Level 2	Level 3
1	Level 1	Level 2
2	Level 1	Level 3

Table 2.4: Bottom- and topmost levels for each event e of example model 1.

The resulting Kronecker Prebuilt Saturation algorithm is described in [8], [10] and [7] (although these papers show some minor differences in the pseudo code, the basic structure of the algorithm is the same). The Saturation algorithm consists of three main functions, namely `generate`, `saturate` and `fire`.
The function `generate` has as main purpose to create an empty MDD node for a state slot at level ℓ of the initial state vector and saturate it by calling the function `saturate`. After this MDD node has been saturated, the next state slot of the initial state vector is handled until no more state slots are left.
The function `saturate` is used to saturate an MDD node on a certain level ℓ. It tries to fire every possible event from the set $\mathcal{E}_\ell$ (iterating over this event set), where MDD nodes on lower levels are turned into saturated ones per event. Hereby the current MDD node is only updated in-place if new (local) states are discovered. In this case every event from the set $\mathcal{E}_\ell$ needs to be iterated over and fired again to check if it is possible to discover more reachable states. Finally the updated MDD node is placed in the unique (lookup) table.
The function `fire` is almost similar to the previous function, but only saturates lower-level MDD nodes for a particular event e and this function is called only with an MDD node on levels lower than $\mathcal{Top}(e)$. It traverses down the MDD until $\mathcal{Bot}(e)$ is reached, if a certain MDD node and event combination has been handled before, or if no more successors are found. Before unwinding out of the recursive calls of `fire`, any newly found states are saturated as well with the function `saturate`. Basically function `fire` calculates the next (local) states or relational product for a certain MDD.

So for a given model, after creating an empty MDD node for a state slot (at level ℓ) of the initial state vector, this MDD node is saturated by considering all events from the set $\mathcal{E}_\ell$. It then fires each event separately by traversing the MDD downwards (by calculating the relational product) and saturating it on the way from the bottom-up when returning back to the MDD node it started with. Saturation is then repeated for the same events if new (local) states are found. Otherwise the whole process is repeated for

the next state slot of the initial state vector until the last state slot of the initial state vector is saturated and the result is the reachable state space of the given model.
The pseudo code of the described functions can be found in Listing 2.1 (for now the highlighted parts can be safely ignored). Furthermore Listing 2.2 provides some explanations for a couple of operations on MDD nodes. Figure 2.5 gives a pictorial overview of the described Saturation process, and some definitions regarding the working MDD level are explained in Figure 2.6, which indicates the current working level for which a Saturation process is started for events that start at a topmost level.

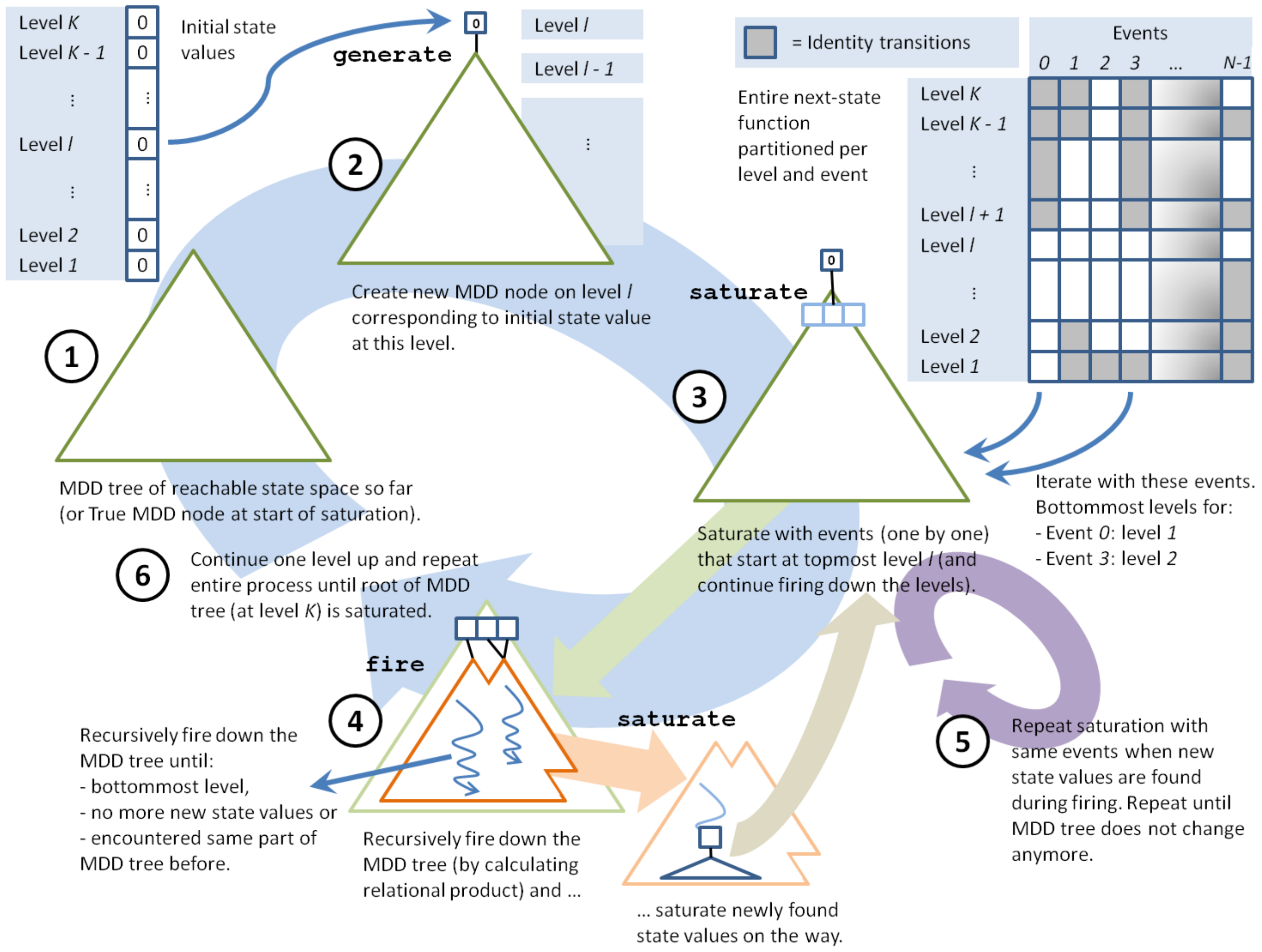

Level *K*
Level *K* - *1*
Level *l*
Level *2*
Level *1*
0
Initial state values
generate
Level *l*
Level *l* - *1*
2
Create new MDD node on level *l* corresponding to initial state value at this level.
= Identity transitions
Events
0 1 2 3 ... N-1
Entire next-state function partitioned per level and event
Level *K*
Level *K* - *1*
Level *l* + *1*
Level *l*
Level *2*
Level *1*
saturate
1
MDD tree of reachable state space so far (or True MDD node at start of saturation).
3
Iterate with these events. Bottommost levels for:
- Event *0*: level *1*
- Event *3*: level *2*
Saturate with events (one by one) that start at topmost level *l* (and continue firing down the levels).
6
Continue one level up and repeat entire process until root of MDD tree (at level *K*) is saturated.
fire
4
saturate
5
Recursively fire down the MDD tree until:
- bottommost level,
- no more new state values or
- encountered same part of MDD tree before.
Recursively fire down the MDD tree (by calculating relational product) and ...
... saturate newly found state values on the way.
Repeat saturation with same events when new state values are found during firing. Repeat until MDD tree does not change anymore.

Figure 2.5: Pictorial overview of Saturation process in a nutshell.

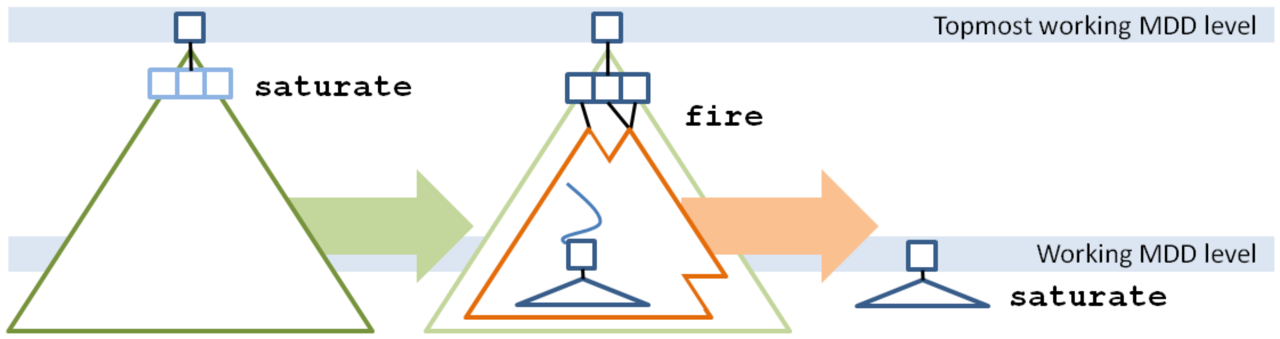


Figure 2.6: Visualizing the term (topmost) working MDD level, when entering the MDD from above.

```
// Generate MDD tree of reachable state space, starting from initial state.
MDDnode generate(array<int> init_vector) :
    MDDnode p := getMDDTrue()
    for (int level in 1 .. init_vector.length()) :
        i := getAllStateValsOfLevel(level).add(init_vector[level])
        confirm(level, i)
        MDDnode r := createEmptyMDD(level)
        r.setArc(init_vector[level], p)    // On-the-fly: r.setArc(i, p)
        p := saturate(level, r)
    return p

// Saturate MDD subtree in a certain level.
MDDnode saturate(int l, MDDnode p) :
    set<event> topLvlEvts := getEventsOnTopLevel(l)
    while (! topLvlEvts.empty() ) :
        event evt := topLvlEvts.pickEvent()
        foreach (int i in getLocalStateVals(evt, l, p)) :
            MDDnode f := fire(evt, l - 1, p.getArc(i))
            foreach (int j in getNextStates(evt, l, i)) :
                MDDnode u := unionMDDs(l - 1, f, p.getArc(j))
                if (u != p.getArc(j)) :
                    if (! getAllStateValsOfLevel(l).contains(j)) :
                        confirm(l, j)
                    p.setArc(j, u)
                    topLvlEvts := getEventsOnTopLevel(l)
    return checkMDDnode(UT[l], p)

// Fire event on MDD subtree in a certain level
// for finding new MDD subtrees to saturate.
MDDnode fire(event evt, int l, MDDnode q) :
    if (l < getBottomLevelForEvent(evt)) :
        return q
    MDDnode s := findInLookupTable(FC[l], (q, evt))
    if (s != null) :
        return s
    s := createEmptyMDD(l)
    foreach (int i in getLocalStateVals(evt, l, q)) :
        MDDnode f := fire(evt, l - 1, q.getArc(i))
        foreach (int j in getNextStates(evt, l, i)) :
            if (! getAllStateValsOfLevel(l).contains(j)) :
                confirm(l, j)
            s.setArc(j, unionMDDs(l - 1, f, s.getArc(j)))
    s := saturate(l, s)
    insertInLookupTable(FC[l], (q, evt), s)
    return s

// Return set of (local) state slot values for which a certain event can fire
// in a certain level for MDD subtree.
set<int> getLocalStateVals(event evt, int l, MDDnode p) :
    set<int> localStateVals
    foreach (int i in getAllStateValsOfLevel(l)) :
        if (p.getArc(i) != getMDDFalse() && ! getNextStates(evt, l, i).empty()) :
            localStateVals.add(i)
    return localStateVals
```

The indices of arrays are numbered from 1 to length().

Global lookup tables (per level)

- FC : Fire Cache
- UT : Unique Table

Function in pseudo code	Notation
getEventsOnTopLevel(l)	$\mathcal{E}_\ell$
getNextStates(alpha, l, i)	$\mathcal{N}_{\ell,\alpha}(i)$
getBottomLevelForEvent(alpha)	$\mathcal{B}ot(\alpha)$
getAllStateValsOfLevel(l)	$\mathcal{S}_\ell$

Listing 2.1: Pseudo code of Kronecker Prebuilt Saturation algorithm with additional code for On-the-fly version highlighted in light-blue [10].

```
// Create a new empty MDD node at the given level.
MDDnode createEmptyMDD(int level)

// Check if the given MDD tree is already stored in the given lookup table. If so,
// the value from the lookup table is returned, otherwise the given MDD tree is
// inserted into the lookup table and returned.
MDDnode checkMDDnode(array<MDDnode> lookup_table, MDDnode set)

// Return MDD node pointed to by arc belonging to the given local state value.
// If current MDD node does not contain the given local state value, then the False
// MDD node is returned.
MDDnode getArc(int value)

// Set arc (belonging to the given local state value) to point to the given MDD
// node. If current MDD node does not contain the given local state value, then
// insert a new entry with this local state value and set the arc to the given MDD
// node.
void setArc(int value, MDDnode set)

// Return the True MDD node.
MDDnode getMDDTrue()

// Return MDD tree of given state vector (consisting of integer state values).
MDDnode getInitMDD(array<int> state_vector)
```
Listing 2.2: Function definitions used in pseudo code of Kronecker Saturation algorithms.

2.2.2 Kronecker On-the-fly Saturation approach

In [9] an extension to the Kronecker Prebuilt Saturation approach is described which lifts the requirement that the local state space sizes n_ℓ are known in advance. It turns out that when n_ℓ is determined for each component ℓ beforehand, one can encounter either an infinite value for the local state space or spurious states (in which case n_ℓ may be larger than necessary). Manually manipulating the model in advance to cope with this issue is not a good solution, due to newly introduced (human) errors.

The next version of the Saturation algorithm deals with two issues in parallel, namely determining the smallest size of each local state space and generating the reachable state space of the model. In this algorithm new locally reachable states are discovered and only those that are also globally reachable are stored by indicating that they are confirmed to occur in the reachable state space. So it is only viable to consider transitions originating from confirmed local states, and ignoring the ones starting from unconfirmed local states. For this to work, the algorithm needs to keep track of the possible firings of events starting from these confirmed states (and this also explains the name On-the-fly Saturation, since it discovers (local) transitions on-the-fly). This is done by incrementally building the next-state function (as before as Boolean adjacency matrices $\mathcal{N}_{\ell,e}$) to keep track of which firings of events are possible. In the Prebuilt version of the Saturation algorithm the submatrices $\mathcal{N}_{\ell,e}$ are prebuilt and are always square-shaped. But now they are generated on-the-fly, these submatrices mostly have a rectangular size during its evolution with the rows corresponding to confirmed local state values and the columns to both confirmed and unconfirmed local state values (these local state values are discovered with the `confirm` function, as will be explained below). At the end of its evolution some submatrices $\mathcal{N}_{\ell,e}$ can end up in a square shape, meaning that all available local state values are globally reachable.
So when there is a transition from local state value i to local state value j in a submatrix and j is confirmed, than there also exists a row j in this submatrix. When j is unconfirmed, this row does not exist (meaning that j is locally reachable, but not (yet) globally reachable). And j can become confirmed if it is discovered to be globally reachable later during saturation.

■ Running example 1 – Fictitious on-the-fly confirmation of next-state function

The initial state values are confirmed one by one, but if all initial state values would be confirmed at the beginning, then the Kronecker product of the next-state function would look in (Kronecker) matrix form like the one as depicted in Figure 2.7. Only one row of each adjacency submatrix has been discovered here and the cells in the row until the local state transitions (indicated by crosses) are stored (so the adjacency submatrices have a rectangular shape).

When one of the local state transitions to an unconfirmed local state is taken (cross located in a yellow cell), `confirm` is called first to confirm the row in all submatrices at the same level corresponding to the local state to which it is going to by the local transition. So take for example the submatrix $\mathcal{N}_{3,0}$ for level 3 and event 0. Before a transition is performed from local state value 0 to local state value 1 (indicated by cross), the row for local state value 1 needs to be confirmed first in all submatrices at level 3. During the confirmation, the rectangular shape of the submatrices is maintained. In this case the number of columns of the submatrix is increased from 2 to 3 as is shown in Figure 2.8.

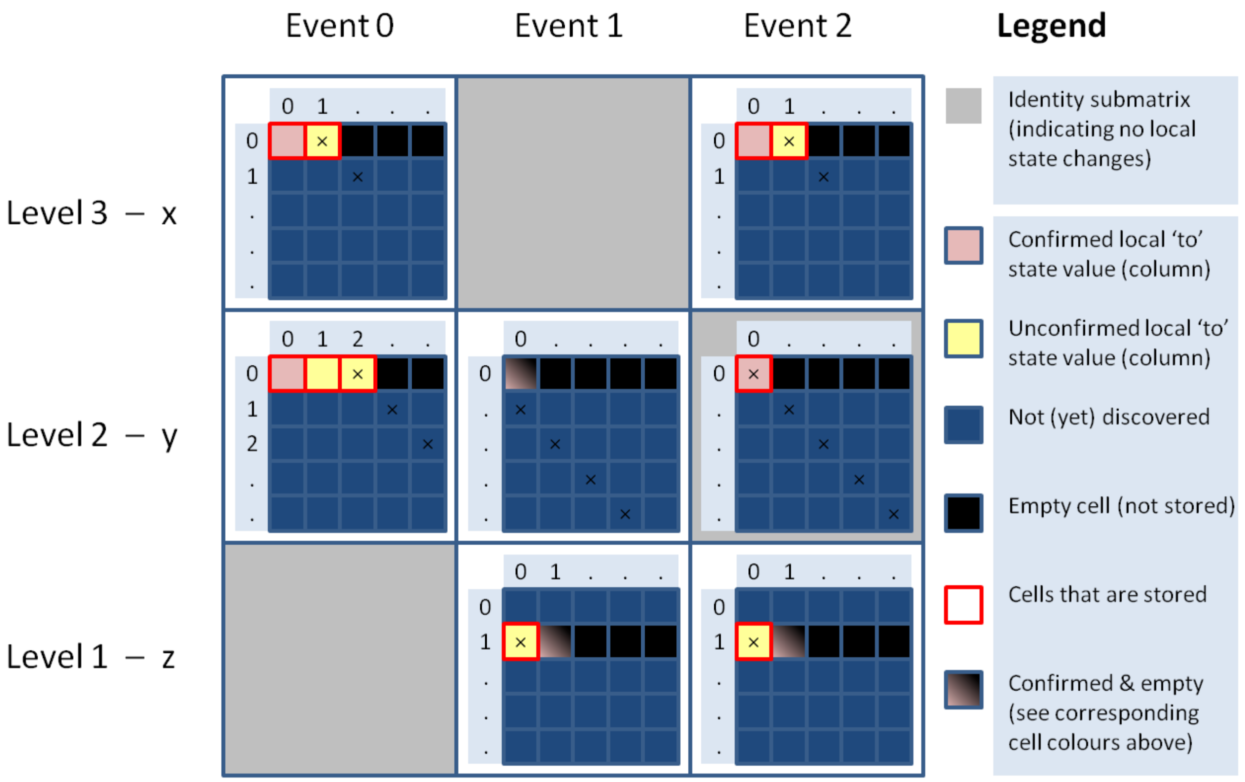

Event 0
Event 1
Event 2
Legend
Level 3 – x
Level 2 – y
Level 1 – z
Identity submatrix (indicating no local state changes)
Confirmed local 'to' state value (column)
Unconfirmed local 'to' state value (column)
Not (yet) discovered
Empty cell (not stored)
Cells that are stored
Confirmed & empty (see corresponding cell colours above)

Figure 2.7: Fictitious Kronecker matrix of next-state function for example model 1 after confirming all initial state values.

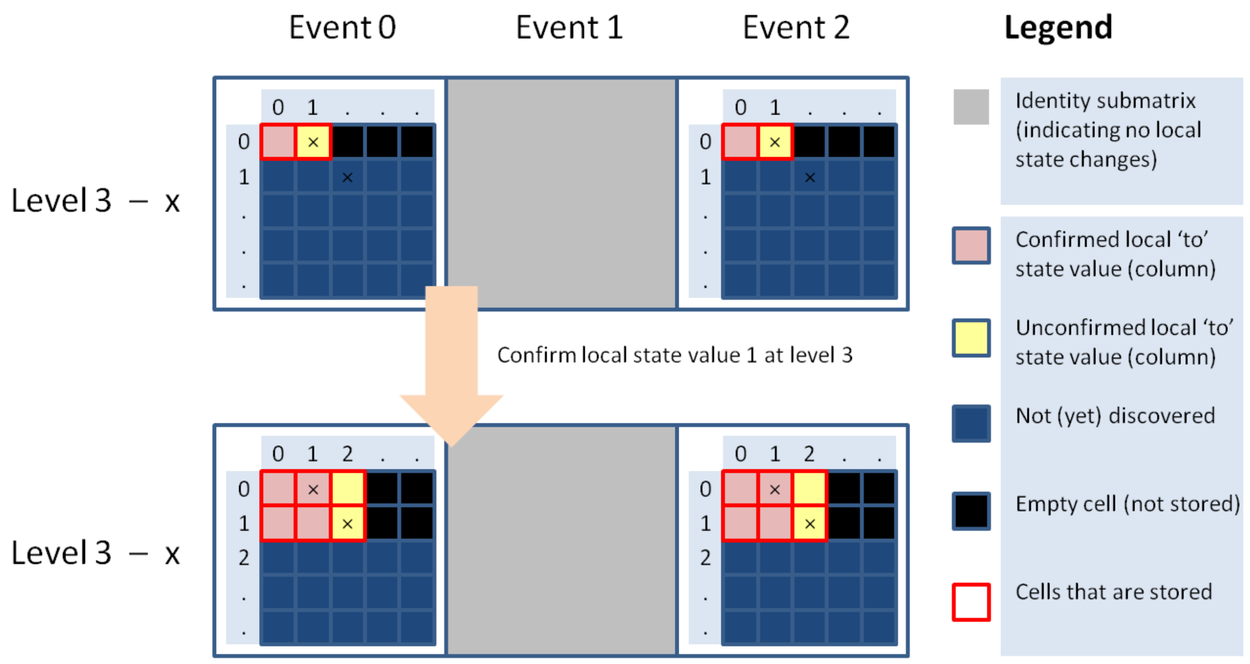

Event 0
Event 1
Event 2
Legend
Level 3 – x
Confirm local state value 1 at level 3
Level 3 – x
Identity submatrix (indicating no local state changes)
Confirmed local 'to' state value (column)
Unconfirmed local 'to' state value (column)
Not (yet) discovered
Empty cell (not stored)
Cells that are stored

Figure 2.8: Visualization of confirmation process for local state value 1 at level 3 (which only affect matrix cells at level 3) for example model 1.

The resulting Kronecker On-the-fly Saturation algorithm has basically the same structure as the Kronecker Prebuilt Saturation algorithm, but a new function `confirm` is added which is confirming newly found local state values [9][10]. This function basically looks for every Boolean adjacency matrix $\mathcal{N}_{\ell,e}$ where ℓ corresponds to the current level. While iterating over each event e, for each submatrix $\mathcal{N}_{\ell,e}$ that is not equal to an identity transition (except when it lies in the range of $\mathcal{B}ot(e)$ and $\mathcal{T}op(e)$), it adds a new row to this submatrix for the newly discovered local 'from' state value and extracts the local 'to' state values it heads to in this row. The local 'to' state values can contain both confirmed as unconfirmed local state values.
In practice the local state values are not discovered in numerical order from 0 to $n_\ell - 1$. A more efficient way to add new rows to the submatrix $\mathcal{N}_{\ell,e}$ is to map the real local state value to the row index of the first occurring empty row in the submatrix. In this way new rows are added adjacently to the existing rows in submatrix $\mathcal{N}_{\ell,e}$ and the algorithm uses the mapped local state values in the algorithm.
The corresponding pseudo code of this function clarifies this in Listing 2.3, where the highlighted code segments indicate the use of mappings for the real local state values.

```
// Constant integer indicating that a certain mapping (for a key) does not exist.
constant int NULL_MAPPING := MIN_INTEGER

// Confirm the newly found ('from' local state) value at the given level and update
// affected events by inserting new transitions using model specifications.
void confirm (int level, int from_value) :
  int real_from_value := getInvMapping(from_value)
  foreach (event evt with getNextStates(evt, level) != getIDMatrix(level)) :
    foreach (real_to_value in getNextStatesFromModel(evt, level, real_from_value)) :
      int to_value := getMapping(real_to_value)
      if (to_value != NULL_MAPPING) :
        to_value := createMapping()
      setNextStates(evt, level, from_value, to_value)
  getAllStateValsOfLevel(level).add(from_value)
```
Listing 2.3: Pseudo code of confirmation process for updating events in Kronecker On-the-fly Saturation algorithm [10].

In the algorithm some additional function calls to `confirm` are added (see highlighted code in Listing 2.1). The confirmation in the function generate is obvious, because it needs to confirm the initial local state value at each level. The other two `confirm` functions are located at points when it is certain that a new local state value is globally reachable.

The difference between Kronecker Prebuilt and Kronecker On-the-fly Saturation algorithms in terms of implementation details of the next-state function is given in Table 2.5.

	Kronecker Saturation algorithm	
	Prebuilt	**On-the-fly**
Next-state function	Static and partitioned into Boolean matrices in advance. They are only read.	Dynamic and partitioned into Boolean matrices. They are created incrementally (during saturation) from a high-level transition spec and read.

Table 2.5: Differences in implementation details between Kronecker Prebuilt and Kronecker On-the-fly Saturation.

2.3 Saturation approaches for general models

In [11] and [23] the Saturation approach is further extended, by handling decompositions of the next-state function that are not Kronecker consistent and this gives us the second group of Saturation approaches. To handle these, the next-state function is partitioned using disjuncts over each occurring event and conjuncts representing the transitions that synchronously update several (local) states at once [11]. The conjuncts are divided to handle the following two cases:

- A set of enabling conjuncts indicating when an event e can occur.
- A set of updating conjuncts indicating how each local state can transition to another local state when event e occurs.

In [11] this distinction within conjuncts is made explicitly, but is not necessary when using this group of Saturation algorithms as explained in [23].
Now for the encoding of the next-state function, variants of MDDs are used which will be discussed per approach in the following subsections.

Just as the Kronecker Saturation approaches, two variants exist which will be discussed in the following subsections, as summarized in Table 2.6. For convenience, they will be denoted as General Saturation approaches from now on.

	General Saturation approaches	
Name	General Prebuilt Saturation	General On-the-fly Saturation
Explanation	Next-state function from general types of models is built in advance (prebuilt) before starting the Saturation process. (Local state space sizes n_ℓ are known.)	Next-state function from general types of models is built on-the-fly during the Saturation process. (Local state space sizes n_ℓ are unknown.)

Table 2.6: Subcategories of General Saturation approaches.

■ Running example 2 – Model description

As a running example for general Saturation throughout this report, a simple program model is used which consists of three (interleaving) parallel processes and three global variables x, y and z. The next-state function does not need to be Kronecker-consistently decomposable and so the individual statements of the processes can rely on multiple variables (in this example at most two variables). Each process is considered to be an event and the three variables x, y and z correspond to level 3, 2 and 1 of the MDD, respectively. The program model is summarized below in Table 2.7.

Level	Variable	Initial values	Local state space
3	x	0	$S_3 = \{0, 1, 2\}$
2	y	0	$S_2 = \{0, 1, 2\}$
1	z	0	$S_1 = \{0, 1, 2\}$

Event	Process of program
0	$\mathit{if}\ x < y\ \mathit{then}\ \{\ x = x + 1\ ;\ \}$
1	$\mathit{if}\ y < 2\ \mathit{then}\ \{\ y = y + 1\ ;\ \}$
2	$z = x;$

Table 2.7: Example model 2 with a general next-state function.

■ Running example 2 – Disjunctive & conjunctive partitioning

The disjunctive partitioning of the transition relation of example model 2 is already given above: each event represents a disjunct. The conjuncts for each of these disjuncts would be as given in Table 2.8.

Event	Enabling conjunct	Updating conjunct
0	$x < y$	$x = x + 1$
1	$y < 2$	$y = y + 1$
2	True	$z = x$

Table 2.8: Partitioning of events in enabling and updating conjuncts for example model 2.

2.3.1 General Prebuilt Saturation approach

The General Prebuilt Saturation algorithm needs to know the local state space sizes (or variable domains) and the partitioned next-state function prior to running the Saturation algorithm, just as is the case with the Kronecker Prebuilt Saturation algorithm. Also the notions of node saturation, event locality and in-place updates are present in the General version of the algorithm. Compared to the Kronecker Saturation algorithms, the pseudo code of the General version of the algorithm is set up slightly different, but the main structure is still visible [23] (in [11] the pseudo code of the algorithm is given as well, but is written more abstractly). The General Saturation algorithm now consists of four main functions, namely `generate`, `saturate`, `doFixPoint` and `relProd`.

The function `generate` is different from the one in the Kronecker Prebuilt Saturation algorithm in the sense that it creates the entire MDD of the initial state at once (for all initial state values), before starting the saturation process by calling `saturate`.

The function `saturate` looks for successor MDD nodes of a given MDD node to saturate and to perform a fix-point computation on them, which is then stored in a unique (lookup) table. Prior to looking for successor MDD nodes, it first checks if the given MDD node has been handled before.

The main work of saturating MDD nodes now lies with the function `doFixPoint`. Just as is the case in the function `saturate` of the Kronecker Prebuilt Saturation algorithm, it tries to saturate a given MDD node on a certain level ℓ.

And the function `relProd` is now handling the work of performing saturation on lower-level MDD nodes for a particular event e, similar to `fire` in the Kronecker Prebuilt Saturation algorithm (calculating the next (local) states or relational product for an MDD). And before returning the updated MDD on the given level (with newly found next (local) states), it performs a saturation process on the current and lower-level MDD nodes to saturate any newly found (local) states.

For the pseudo code of these functions, one can take a look in Listing 2.4 (using the same MDD operations as explained in Listing 2.2) and for now ignore the highlighted parts.

The given algorithm in [23] partitions the next-state function by event and stores each of these as quasi-reduced $2K$-level MDDs. The levels of such partitioned MDDs indicate if it concerns an unprimed level from which a transition is performed (local 'from' state value) or a primed level to which a transition is going (local 'to' state value). The partitioning using conjuncts and disjuncts is as follows: the disjuncts are actually the MDDs for each event separately and each of these MDDs are built up from the intersection of individual MDDs representing the conjuncts. The algorithm does not require that the individual conjuncts are stored explicitly and works just as well with storing only the disjuncts (as MDDs for each event). But in Listing 2.4 it is assumed that the conjuncts of the entire next-state function are stored (indicated by the function `buildTransitionMDDFromConjuncts` in line 25 of pseudo code).
The next-state function can also be partitioned by level (or by variable), but this is basically a variation on the disjunctive partitioning scheme. In this case, for each level l a transition relation is constructed by taking the union over the partitioned transition relations per event e with $Top(e) = l$.

The original pseudo code of the algorithm in [23] does not use the range of levels bounded by $Top(e)$ and $Bot(e)$. This is only done for clarity reasons to avoid clutter in the pseudo code and is as such very time-consuming. This is due to the fact that every level of the partitioned MDD of the next-state function needs to be traversed, including identity transitions at upper and lower levels of the MDD. But the given algorithm can be easily adjusted to take into account the range of levels between $Top(e)$ and $Bot(e)$, so that only the levels of the MDD for the next-state function that matter are traversed (as indicated in line 37 in Listing 2.4, where $Bot(e)$ is taken into account). By making use of this, the partitioned MDDs of the next-state function can be reduced further by storing only MDD nodes on levels within this range.
With this setup the partitioned MDDs of the next-state function contain nodes at every level within the mentioned range, even if identity transitions exist at certain intermediate levels (which will be denoted as intermediate identity transitions from now on). The given pseudo code in Listing 2.4 assumes that such intermediate identity transitions are stored in the MDDs of the next-state function.

■ Running example 2 – Partitioning of next-state function per event

For each of the three events of example model 2 an MDD is constructed in advance and these are depicted in Figure 2.9, which are equivalent to the disjunctive partitioning of the next-state function (the individual conjuncts are not visible). Storing these disjuncts is enough for the algorithm to work. As described above, only MDD nodes within the range of levels bounded by $Top(e)$ and $Bot(e)$ are stored, also for intermediate identity transitions (which are highlighted in grey in the figure).

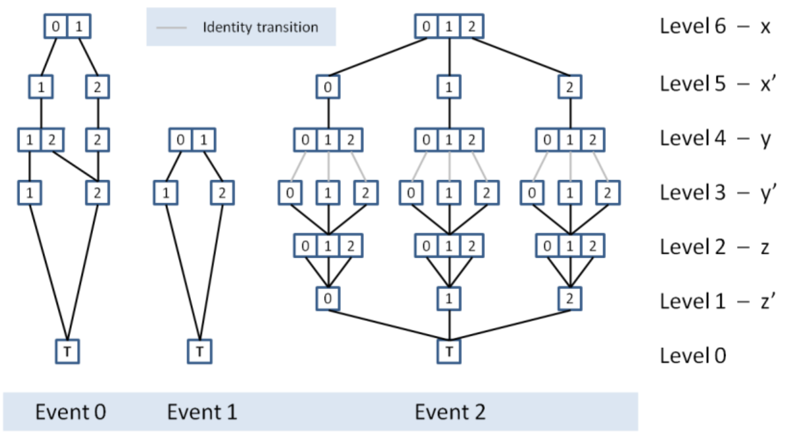


Figure 2.9: MDDs of partitioned next-state function per event for example model 2.

```
// Generate MDD tree of reachable state space, starting from initial state.
// On-the-fly: confirmation of initial state values in getInitMDD(init).
MDDnode generate(array<int> init_vector) :
    return saturate(init_vector.length(), getInitMDD(init_vector))

// Saturate MDD subtree in a certain level.
MDDnode saturate(int l, MDDnode p) :
    MDDnode t := findInLookupTable(SC[l], p)
    if (t != null) :
        return t
    t := createEmptyMDD(l)
    foreach (int i in getLocalStateVals(l, p)) :
        t.setArc(i, saturate(l - 1, p.getArc(i)))
    t := checkMDDnode(UT[l], doFixPoint(l, t))
    insertInLookupTable(SC[l], p, t)
    return t

// Do fix-point computation on MDD subtree for every event in given topmost level.
MDDnode doFixPoint(int l, MDDnode t) :
   MDDnode old_t := createEmptyMDD(l)
   while (old_t != t) :
      old_t := t
      foreach (event evt in getEventsOnTopLevel(l)) :
         confirm(foundNewStateVals)
         MDDnode r := buildTransitionMDDFromConjuncts(evt)
         foreach (int i in getLocalStateVals(l, t)) :
            foreach (int j in getLocalStateVals(l, r.getArc(i))) :
               if (! getAllStateValsOfLevel(l).contains(j)) :
                  foundNewStateVals.add(l, j)
               MDDnode u := relProd(l - 1, t.getArc(i), r.getArc(i).getArc(j), evt)
               t.setArc(j, unionMDDs(l - 1, t.getArc(j), u))
   return t

// Perform relational product on MDD subtree in a certain level under a given event
// for finding new MDD subtrees to saturate.
MDDnode relProd(int l, MDDnode s, MDDnode r, event evt) :
    if (l < getBottomLevelForEvent(evt)) :
        return s
    MDDnode t := findInLookupTable(RC[l], (s, r))
    if (t != null) :
        return t
    t := createEmptyMDD(l)
    foreach (int i in getLocalStateVals(l, s, r)) :
        foreach (int j in getLocalStateVals(l, r.getArc(i))) :
            if (! getAllStateValsOfLevel(l).contains(j)) :
                  foundNewStateVals.add(l, j)
            MDDnode u := rel_prod(l - 1, s.getArc(i), r.getArc(i).getArc(j), evt)
            t.setArc(j, unionMDDs(l - 1, t.getArc(j), u))
    t := saturate(l, checkMDDnode(UT[l], t))
    insertInLookupTable(RC[l], (s, r), t)
    return t

// Get set of (local) state slot values in a certain level for MDD subtree
// (for which transition can occur).
set<int> getLocalStateVals(int l, MDDnode p) : // Version with parameter MDDnode r
    set<int> localStateVals
    foreach (int i in getAllStateValsOfLevel(l)) :
        // Extra condition for version: && r.getArc(i) != getMDDFalse()
        if (p.getArc(i) != getMDDFalse()) :
            localStateVals.add(i)
    return localStateVals
```

The indices of arrays are numbered from 1 to length().

Function in pseudo code	**Notation**
getEventsOnTopLevel(l)	$\mathcal{E}_l$
getBottomLevelForEvent(alpha)	$Bot(\alpha)$
getAllStateValsOfLevel(l)	$\mathcal{S}_l$

Global lookup tables (per level)

- SC : Saturation Cache
- RC : Rel-prod Cache
- UT : Unique Table

Listing 2.4: Pseudo code of General Prebuilt Saturation algorithm with additional code for On-the-fly version highlighted in light-blue [23].

2.3.2 General On-the-fly Saturation approach

Just as is the case with the Kronecker On-the-fly Saturation algorithm, the restriction of having to know the local state space sizes in advance can be lifted. The resulting algorithm also has to deal with updating the next-state function in parallel to generating the reachable state space, where newly discovered local state values need to be confirmed as globally reachable. The algorithm as described for the General Prebuilt Saturation algorithm can be used with the addition of confirmation for newly found local state values.
The confirmation function works differently from the one in Kronecker On-the-fly Saturation, which is due to the data structure used for storing the entire next-state function. When a new local state value is encountered inside the function `doFixpoint`, this new value is not immediately confirmed. Newly found values are put on hold (highlighted in lines 28 – 29 and 45 – 46 in Listing 2.4) and are confirmed in the next iteration in the while-loop of the function `doFixpoint` by calling the function `confirm` (highlighted in line 24 in Listing 2.4). During the confirmation process, only the MDDs representing the conjuncts with the affected level are updated (confirmed) with the new local state value. And when function `buildTransitionMDDFromConjuncts` is called, it rebuilds the MDD for the given event by performing the intersection over the MDDs of the conjuncts. Listing 2.5 gives the pseudo code of function `confirm` for updating all affected MDD conjuncts.

```
// Confirm new local state values corresponding to a certain level by updating all
// affected conjuncts which involve the given level.
void confirm(array<(int, int)> foundNewStateVals) :
  foreach ((int level, int new_value) in foundNewStateVals) :
    getAllStateValsOfLevel(level).add(new_value)
    // Assume that MDD conjuncts can be identified with a unique ID (conjID) and
    // the top level of an MDD conjunct can be retrieved with getTopLevel().
    foreach ((event evt, int conjID) in getMDDConjunctsWithLevel(level)) :
      MDDnode newConj := buildMDDConjunctFromModel(evt, conjID, level, new_value)
      MDDnode oldConj := getMDDConjunct(evt, conjID)
      MDDnode unitedConjs := unionMDDs(oldConj.getTopLevel(), oldConj, newConj)
      storeMDDConjunct(evt, conjID, unitedConjs)
```
Listing 2.5: Pseudo code of confirmation process for updating events in General On-the-fly Saturation algorithm [23].

In [11] and [23] a proposal has been given for the storage of the next-state function, which still partitions the MDDs per event (or level). Instead of using quasi-reduced $2\mathcal{K}$-level MDDs, the next-state function is partitioned into fully-identity reduced $2\mathcal{K}$-level MDDs where an arc can skip a level (the usual MDD operations used for quasi-reduced MDDs cannot be applied here, because they assume that intermediate identity transitions are stored as well). The point of using such an MDD structure is the fact that intermediate identity transitions are not stored explicitly while still maintaining an efficient encoding for the next-state function. But the disadvantage lies in updating the MDDs when a new local state value is found and building larger MDDs from the partitioned MDDs. [23] has come up with a solution by introducing extensible MDDs, where the reduction technique for MDD nodes differs per level in the MDD. Performing operations on such MDDs as union and intersection now becomes more complex. Since the proposed solutions are mostly an implementation solution to the MDD data structure, they will not be discussed further in this report.

An overview of the differences between the General Prebuilt and the General On-the-fly Saturation algorithms in terms of implementation details of the next-state function is given in Table 2.9. Although General Saturation does not pose a restriction on the next-state function of a model, it is in principle capable of using models with a Kronecker-consistent decomposable next-state function. However, storing such next-state functions using MDDs is not as efficient as storing those using Boolean matrices.

	General Saturation algorithm	
	Prebuilt	**On-the-fly**
Next-state function	Static and partitioned into transition MDDs per event in advance. They are only read.	Dynamic and partitioned into transition MDDs per event. They are created incrementally (during saturation) from a high-level transition spec and read.

Table 2.9: Differences in implementation details for General Prebuilt and General On-the-fly Saturation.

2.3.3 General Saturation approach using Matrix Diagrams

In this report the main focus will be on using MDDs as data structure for storing sets of states and the next-state function. For completeness a reference to another Saturation approach is given to show how versatile it is regarding data structure.

In [18] another approach is presented by A. Miner for dealing with general models with transition relations which are not Kronecker consistent. Here a variation on MDDs, called Matrix Diagrams (abbreviated to MxDs) is used for storing the next-state function. They are similar to MDDs, but each non-terminal node is a matrix of size $n_\ell \times n_\ell$. These contain pointers to other (matrix) nodes. Furthermore identity transitions are not stored, so it is in fully-identity reduced form.
Transition relations are partitioned per event as well, but the underlying decomposition into smaller manageable parts is different. When in an event of the model certain levels are dependent on each other (e.g. certain model components interact with each other), they are grouped together into one MxD, called a Group. If a Group only depends on one level, it is either an identity transition or a transition that only relies on itself and/or some constant value. A transition relation MxD belonging to a certain event is basically the cross product of these Groups.
So when dealing with conjuncts that have overlapping variables, one has to group these conjuncts together in one MxD. This is not necessary when working with the approach as presented by G. Ciardo and each conjunct is stored individually.
Here also Prebuilt and On-the-fly versions of the Saturation algorithm exist and the difference with that of G. Ciardo is mainly due to implementation details of handling MxDs instead of MDDs.

■ Running example 2 – Matrix diagrams of next-state function per event

The partitioning of the next-state function into MDDs for example model 2, as shown in Figure 2.9, can also be depicted using MxDs as shown in Figure 2.10. Each matrix node indicates how a local transition occurs with the row indicating the ‘from’ local state value and the column indicating the ‘to’ local state value. The matrix cell corresponding to this local transition is marked with a pointer to the next matrix node on a lower level. The difference with MDDs is that the levels are not duplicated and that each non-terminal node is a matrix node (in this case of size 3×3, because the local state space sizes n_ℓ are the same for all three variables in this example model). Identity transitions (also intermediate ones) are not stored.

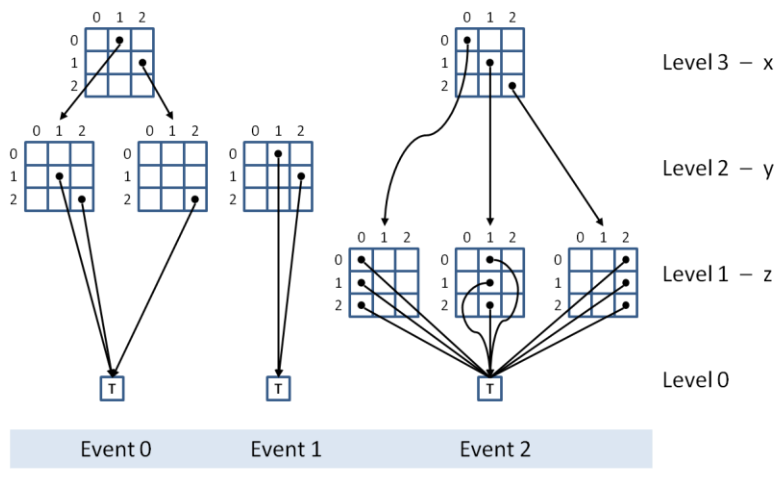


Figure 2.10: Matrix Diagrams of partitioned next-state function per event for example model 2.

2.4 Summary of Saturation approaches

To summarize this chapter, an overview is given of the similarities and differences among the Saturation approaches. This is followed by an overview of how the Saturation approach evolved over time.

2.4.1 Similarities and differences among Saturation approaches

The basic idea of the Saturation approach is the same for all its different versions, where the reachable state space is stored as a quasi-reduced $\mathcal{K}$-level MDD. The algorithm performs a fix-point computation when saturating MDD nodes and this starts from the top of the MDD heading down the MDD tree structure. Starting from a topmost working level for certain events, the fix-point computation is performed for each of these events separately after each other, which involves calculating the relational product of the MDD so far. When unwinding out of the recursive calls back to the top of the MDD, newly updated MDD nodes are saturated on the way. So basically MDD nodes are saturated from the bottom to the top of the MDD and one can only head one level up when all MDD nodes on the current and lower levels are saturated. This process continues until the root of the MDD is saturated and this gives the reachable state space of the model under investigation.

The differences between the Saturation approaches can be found in the prerequisites and the data structures used for the next-state function. The Saturation algorithms can be divided into Prebuilt and On-the-fly versions, which differ only in the requirement that the local state space sizes (or domain of the model variables) n_ℓ are known respectively in advance or not. For Prebuilt Saturation the next-state function can be calculated in advance of the Saturation process; for On-the-fly Saturation it is updated in parallel with generating the reachable state space. Another division is made in the type of models the algorithms can handle: the Kronecker version of the algorithms can only handle models with a Kronecker-consistent decomposable next-state function, while for the general version this restriction is not necessary.
These differences in requirements give rise to the different data structures used for encoding the next-state function. In the Kronecker versions of the algorithms it is more efficient to partition and store the next-state function in a matrix-like structure per level and event, storing each of these as Boolean adjacency matrices. But for the general versions of the algorithm this is no longer possible and one has to switch to MDDs. Here the next-state function is partitioned into $2\mathcal{K}$-level MDDs per event, where the unprimed levels represent the local 'from' states and the primed levels the local 'to' states.

2.4.2 Evolution of Saturation approach

The Saturation approach started out with the Kronecker Prebuilt Saturation algorithm in [8] and is covered more extensively in [10] and [7]. Here it is required that the local state space sizes (or domain of model variables) n_ℓ are known in advance and that the partitioning of the next-state function is Kronecker consistent. The knowledge of the local state space sizes n_ℓ in advance posed some issues, such as spurious or infinite values for n_ℓ. To overcome this, [9] (and later [10]) presented the Kronecker On-the-fly Saturation algorithm which lifts the requirement that n_ℓ is known in advance. But now the next-state function and local state space sizes n_ℓ need to be updated in parallel with generating the reachable state space. In both algorithms the next-state function is partitioned per level and event into Boolean adjacency matrices $\mathcal{N}_{\ell,e}$.

The prerequisite that the next-state function needs to be partitioned in a Kronecker-consistent manner posed a large problem for using Saturation for general types of models. Although one can still try to decompose the next-state function in a Kronecker-consistent way by hand, it is not always feasible and practical to do this. So another approach has been developed to deal with this issue: General Prebuilt and General On-the-fly Saturation [11] [23]. For the Prebuilt version the local state space sizes n_ℓ still need to be known in advance, but this is no longer the case for the On-the-fly version. However, handling general types of models also means that the matrix-like data structure for the next-state function cannot be used anymore. The next-state function is now conjunctively and disjunctively partitioned using MDDs. The disjuncts are formed by the MDDs per event and within these MDDs the conjuncts can be found. For the algorithms the disjunctive partitioning into MDDs per event is enough to be able to work with it.

3 Architecture of LTSmin toolset

The FMT research group has developed the LTSmin toolset with the idea of having a unifying framework that combines the power of a variety of existing modelling languages and different kinds of verification techniques. It turns out that the verification techniques do not rely on a specific modelling language and with this in mind one has come up with an intermediate interface, called PINS.
This chapter first gives a global architectural overview of the LTSmin toolset, after which the attention is focused on the relevant parts of the LTSmin toolset for implementing the Saturation approach. These are PINS and its dependency matrix, the available MDD data structures and the symbolic reachability algorithms currently implemented in the LTSmin toolset.

3.1 High-level architecture of LTSmin toolset

The LTSmin toolset [4] is a model checker built in a modular fashion in the C programming language, where language-specific details for specifying models are split from the model checking algorithms which are used in the verification analysis. This is achieved using a layered architecture where the interface between two layers is provided by PINS: an Interface based on a Partitioned Next-State function. The tool consists of three layers, which are the language module front-end, the PINS2PINS wrappers and the reachability algorithmic back-end.

The front-end side of the LTSmin toolset handles the next-state function of a model written in a certain model specification language, and through the usage of PINS one does not have to deal with these language-specific (implementation) details. An advantage of providing such a front-end is the ability to work with different model specification languages; multiple types of such languages are in use in the field of model verification. The following languages are currently supported by the LTSmin toolset:

- The in-house developed in- and output language for the LTSmin toolset, called ETF, which stands for Enumerated Table Format [5].
- The state-based specification languages DVE and subsets of PROMELA [16]. The former is a specification language for the verification tool DiVinE [1], the latter are supported by the SpinJa model checker (a Java version of SPIN) [12] and the NIPS-VM (a virtual machine for state space generation) [24].
- The process algebraic languages µCRL [2] and mCRL2 [14]. Both are used for the specification and behavioural analysis of communicating processes (such as in distributed systems and protocols).

The back-end side of the LTSmin toolset contains algorithms for state space exploration, which also includes capabilities to check certain properties. The two main categories of reachability analysis are:

- Explicit reachability: exploring a state space by discovering individual states.
- Symbolic reachability: exploring a state space by discovering sets of states.

The reachability analysis of the first category can be performed in three different ways, which are:

- Enumerative/sequential reachability: using one processor core of a computer.
- Multi-core reachability: using multiple processor cores of a computer.
- Distributed reachability: using multiple computers in a network.

Between the front- and back-end layers an intermediate layer containing PINS2PINS wrappers is situated. These wrappers are able to perform transformations for optimizing the reachability process (only the wrapper for regrouping will be touched upon in section 3.2.2 and the rest will not be discussed any further). In Figure 3.1 an overview of the three layers of the LTSmin toolset is given and it also explains how the different layers communicate with each other through PINS.

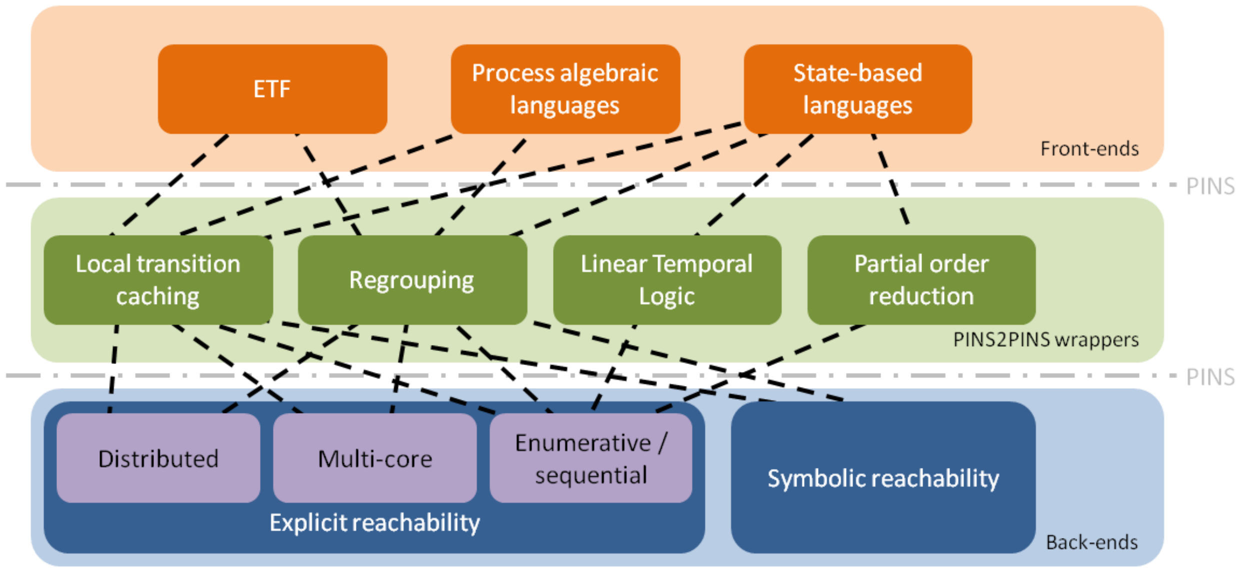


Figure 3.1: Layered architecture of LTSmin toolset with PINS.

When using the LTSmin toolset, the user can indicate which kind of tool and language to use, which is encoded in the command name. For example, calling the command etf-reach means using the in-house developed language ETF and the symbolic reachability tool. A selection of possible language and reachability tool combinations (only for state space exploration) is summarized in Table 3.1.

Modelling language	LTSmin command prefix
ETF	etf
DVE	dve2
µCRL	lpo
mCRL2	lps
NIPS-VM	nips

LTSmin command suffix	Reachability analysis
-reach	Symbolic reachability tool using decision diagrams for manipulating sets of states.
2lts-grey	Sequential state space enumeration tools that can either use decision diagrams or hash tables for sets of states.
2lts-mpi	Distributed state space enumeration tools that use distributed hash tables for sets of states.

Table 3.1: Selection of command pre- and suffixes of state space exploration tools of LTSmin toolset.

Before delving into the symbolic reachability tools of the LTSmin toolset, section 3.2 will give an overview of the most relevant aspects of PINS and section 3.3 will go into the underlying MDD data structure used for storing the reachable state space and the transition relation.

3.2 LTSmin toolset – PINS

This section will first provide a semantic model used to describe Labelled Transition Systems, which forms the basis for describing PINS and its dependency matrix to store information about the transition relation of the model.

3.2.1 Semantic model of Labelled Transition Systems

The LTSmin toolset works with Labelled Transition Systems (abbreviated to LTSs) as the underlying semantic model for describing a certain model of a system. The transition system is able to handle labels on its edges and states, but this is not needed for our purposes and will not be discussed further in this report. The definition of an LTS (without labels on edges and states) is similar to a discrete-state model, as given in **Definition 2.1**.
Working with such a monolithic transition system is not very useful and in most situations the transition system is based on a model consisting of multiple smaller components, which all contribute to a global state. There are also situations where only a subset of these components contributes to a certain transition, which is known as event locality. This results in a fine-grained version of a transition system where sets of states are represented using a Cartesian product and the transition relation is partitioned into a number of transition groups.

Definition 3.1. A Partitioned Transition System (abbreviated to PTS) is a structure $\mathcal{P} = \langle\langle \mathcal{S}_1, \dots, \mathcal{S}_\mathcal{K}\rangle, \langle \rightarrow_1, \dots, \rightarrow_\mathcal{M}\rangle, \langle s^0_1, \dots, s^0_\mathcal{K}\rangle\rangle$ [4], where:

- The set of local state sets $\mathcal{S}_1, \dots, \mathcal{S}_\mathcal{K}$ define the set of (all possible) states $\mathcal{S}_\mathcal{P} = \mathcal{S}_1 \times \dots \times \mathcal{S}_\mathcal{K}$.
- The transition groups $\rightarrow_i \subseteq \mathcal{S}_\mathcal{P} \times \mathcal{S}_\mathcal{P}$, $(1 \leq i \leq \mathcal{M})$ define the transition relation $\rightarrow = \bigcup_{i=1}^{\mathcal{M}} \rightarrow_i$.
- The initial state is $s^0 := \langle s^0_1, \dots, s^0_\mathcal{K}\rangle \in \mathcal{S}_\mathcal{P}$.

The defined Transition System (TS) of $\mathcal{P}$ is denoted as $\langle \mathcal{S}_\mathcal{P}, \rightarrow, s^0\rangle$.

Thus a global state of a system is basically a state vector consisting of $\mathcal{K}$ slots, corresponding to the $\mathcal{K}$ components that the model is partitioned into. A state slot is also denoted as a local state value. For convenience, the state vectors are implemented in the tool as vectors of integers, meaning that each local state value is given an integer value to identify with. So when handling state vectors, these are converted from a state vector $s_\ell = \langle s_{i0}, s_{i1}, \dots, s_{i\mathcal{K}}\rangle$ into a vector of integers $\underline{s_\ell} = \langle i_0, i_1, \dots, i_\mathcal{K}\rangle$.

A transition relation of the model is partitioned into a number of $\mathcal{M}$ transition groups and these are equivalent to the aforementioned partitions of the next-state function, denoted as $\mathcal{N}_e$. And the set of transition group names (or events) is equivalent to $\mathcal{E}$, with $\mathcal{M} = |\mathcal{E}|$. For convenience, the transition group names are identified in the tool by an integer in the range of 0 to $\mathcal{M} - 1$ and these integers will from now on be called transition group IDs.

3.2.2 Access to model through PINS

For access to the model PINS is used, which provides specific functionality for retrieving certain types of data from the model without having to deal with the underlying language used. The user can choose to start a verification process using a certain language module, after which it is selected as underlying

module for handling the transition relation of the model. As indicated earlier, the available language modules are ETF, DVE, SpinJa, NIPS, µCRL and mCRL2.
Access to language-specific operations of reading states and performing transitions from a certain language module is done via the header file spec-greybox.h.

A core feature of PINS is the dependency matrix of the model. This matrix consists of $\mathcal{M}$ rows and $\mathcal{K}$ columns, where the rows correspond to the transition groups and the columns to the state slots of the state vector. A matrix cell can be marked with the following symbols:

- Letter ‘r’: the state slot of the transition group is only read.
- Letter ‘w’: the state slot of the transition group is only written.
- Symbol ‘+’: the state slot of the transition group is both read as written.
- Symbol ‘–’: the state slot of the transition group is not taken into account.

A cell in the matrix which is marked with one of the first three symbols means that a transition group depends on a state slot, so the firing of an event needs to take into account the value of a certain local state value. A cell in the matrix which is marked with a ‘–’ symbol means that a transition group is independent from a certain state slot, where independent means [4]:

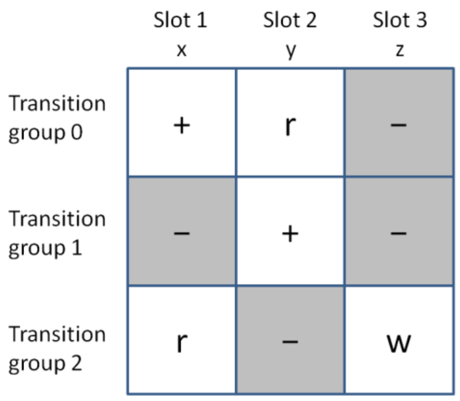
- The affected state slot is not modified when the transition of the given transition group occurs (e.g. no write occurs to this state slot).
- It does not matter what value the affected state slot has when the transition of the given transition group occurs (e.g. no read occurs from this state slot).

Also the notion of a projected state vector plays a role here and this means that for a given transition group, from a state vector only those state slots are taken into consideration if the corresponding cell in the dependency matrix is marked with a ‘+’, ‘r’ or ‘w’ symbol. So a projected state vector returns a subvector of the state slots of the given state vector for a given transition group. In the implementation each transition group is associated with a certain projection on the state vector (see section 3.3.2).

■ Running example 2 – PINS dependency matrix & projections

The three transition groups of example model 2 are already given in Figure 2.9 and when taking the model description into account, the PINS dependency matrix will be as shown in Figure 3.2.
For each transition group, one can derive the following projected state vectors:

- Transition group 0: subvector consisting of state slots 1 and 2.
- Transition group 1: subvector consisting of only state slot 2.
- Transition group 2: subvector consisting of state slots 1 and 3.



	Slot 1 x	Slot 2 y	Slot 3 z
Transition group 0	+	r	–
Transition group 1	–	+	–
Transition group 2	r	–	w

Figure 3.2: PINS dependency matrix for example model 2.

The PINS interface contains a number of functions related to the dependency matrix and the transition relation, and the most relevant are given in Table 3.2. These operations are available in the header file greybox.h (with concrete implementation in file greybox.c) and their names are simplified for readability reasons. Concerning the next-state operations, they require a call back function and context information. With this construct one can provide own implementations of how to handle the list of successor states.

PINS operation	Input arguments	Return value
get_dependency_matrix	- model	Dependency matrix from given model.
get_initial_state	- model	Indexed state vector of initial state from given model.
get_next_states_short	- model - transition group ID - source state (as projected indexed state vector) - call back function - context information	For a given model, enumerate the transitions of the given transition group ID starting from the source state. The call back function together with the context information is used to return a list of successor states (as a list of projected indexed state vectors).
get_next_states_long	- model - transition group ID - source state (as indexed state vector) - call back function - context information	Idem, but now using normal indexed state vectors for the states.
get_next_states_all	- model - source state (as indexed state vector) - call back function - context information	Idem, but it enumerates the transitions of every transition group.

Table 3.2: Some PINS operations and their arguments as defined in header file greybox.h (simplified).

■ Running example 2 – PINS operations

Imagine that we are at the current source state (1, 2, 0) for vector (x, y, z) of example model 2 and for the next-state operations the transition group with ID number 0 is taken. Then the outcomes are as given below in Table 3.3, where the input arguments for the call back function and context information of the last three operations have been neglected.

PINS operation	Input arguments	Return value
get_dependency_matrix	- example model 2	See Figure 3.2.
get_initial_state	- example model 2	(0, 0, 0)
get_next_states_short	- example model 2 - transition group ID no. 0 - projected source state (1, 2)	(2, 2)
get_next_states_long	- example model 2 - transition group ID no. 0 - source state (1, 2, 0)	(2, 2, 0)
get_next_states_all	- example model 2 - source state (1, 2, 0)	(2, 2, 0) with transition group 0 (1, 2, 1) with transition group 2

Table 3.3: Example evaluation of PINS operations for example model 2.

■ PINS wrapper: Regrouping

The PINS dependency matrix is obtained from the model specification which has a predetermined variable ordering. In most cases this would result in a PINS dependency matrix where transition groups span multiple state slots it depends on, and with possibly lots of state slots in between it does not depend on (the latter are also denoted as intermediate identity transitions). The PINS wrapper dealing with regrouping is capable of reordering the transition groups and the state slots in such a way that it attempts to minimize the distance between the lowest and highest state slots it depends on for the transition groups, where a cost function is used. This wrapper is not capable of finding the most optimal variable ordering, since this is an NP-complete problem [21]. It tries to find a suitable ordering with the lowest cost function it can manage.

3.3 LTSmin toolset – MDD encodings

First this section provides an overview of the available MDD data encodings in the LTSmin toolset, after which it goes into detail how MDD operations are accessed.

3.3.1 Available MDD libraries

In the reachability algorithms of the LTSmin toolset a number of available MDD data structure implementations can be used. The back-end layer of the LTSmin toolset currently contains five different types of such implementations using four different libraries, resulting in different underlying implemented encodings. The libraries used (which all contain a form of garbage collection) are:

- ATermDD library (vset_atermdd.c) [22]: a C library that uses an abstract data type (ATerm) for handling graph-like data structures and is capable of maximal subterm sharing (common subterms are shared to prevent storing duplicate subterms, which reduces memory usage).
- BuDDy library (vset_buddy.c) [17]: a fully-fledged BDD library written in C containing all standard BDD operations.
- libDDD library (vset_ddd.cpp) [15]: a C++ library which uses Data (which only stores integers) or Set Decision Diagrams with functions that represent relations, and with support for Saturation-type algorithms.
- listDD library (vset_listdd.c): an in-house developed MDD library written in C.

The underlying implementation structure of an MDD can be one of the following:

- In a graph-like data structure for MDDs (just as given in **Definition 2.2**), where MDD nodes contain local state values and from each one of them, an arc points to another MDD node.
- In a linked-list data structure for MDDs, where MDD nodes are represented as a linked list and each entry of the linked list contains a local state value, a link to another linked list and a link to the next entry of the linked list (if any). This data structure is also referred to as a List Decision Diagram (abbreviated to LDD) [3].

With the given descriptions of the four libraries and the implementation structure used for storing MDDs, Table 3.4 gives an overview of the concrete encodings of the MDD data structure available in the LTSmin toolset.

Option	Library used	Data structure	Explanation
list	ATermDD	Graph using linked-lists	Full-fledged implementation of LDDs with integer as node value and building on ATerm objects.
tree	ATermDD	Graph	Full-fledged implementation of MDDs with integer as node value and building on ATerm objects.
fdd	BuDDy	Graph	Wrapper around BuDDy library.
ddd	libDDD	Graph	Wrapper around libDDD library.
ldd	listDD	Graph using linked-list	Full-fledged implementation of LDDs with integer as node value.

Table 3.4: Available MDD data encodings in the LTSmin toolset.

■ Running example 2 – Next-state function as LDD

The LDD of event 1 of the second example model is given in Figure 3.3. A node is a linked list which is stretched out horizontally in a level. The three components in an entry of a linked list are given in the order as pointed out above (local state value, pointer to other linked list and pointer to next entry of linked list). And also the memory usage is optimized: instead of storing the redundant LDD node in level 4 (node with dashed borders), part of the linked list on the left is reused.

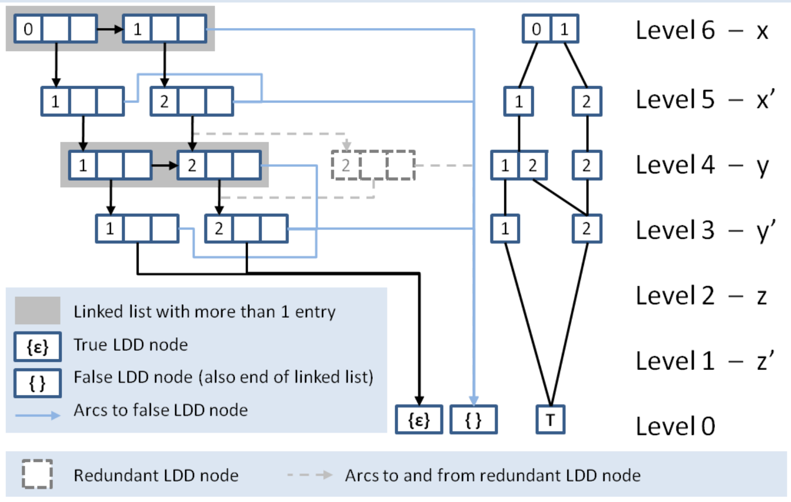


Figure 3.3: LDD (left) corresponding to MDD of event 1 of example model 2.

■ Running example 1 & 2 – Reachable state space as LDD

The LDD data structure can also be used for storing the reachable state space of the example models. In Figure 3.4 the reachable state space of example model 1 using an LDD and an MDD are depicted next to each other, and likewise for example model 2 in Figure 3.5.

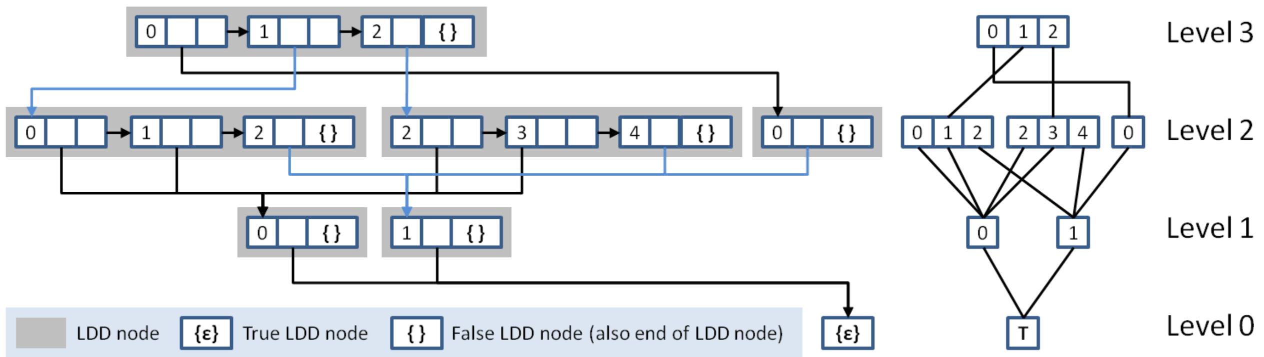


Figure 3.4: Reachable state space of example model 1 using LDD (left) and MDD (right).

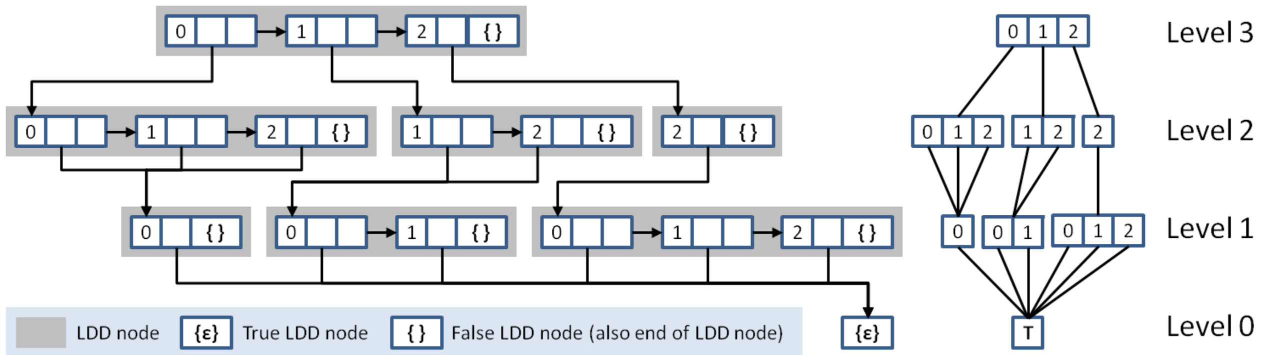


Figure 3.5: Reachable state space of example model 2 using LDD (left) and MDD (right).

3.3.2 Access to MDD operations

To be able to work with the MDD data structure implementations in the tool, the back-end layer is structured to distinguish among the data structure encodings and its concrete uses. It makes use of the following three types of data objects:

- Vector domain (vdom_t): this refers to the type of MDD data structure encoding to be used, namely the five mentioned earlier in Table 3.4 (list, tree, fdd, ddd and ldd). It then makes a correct reference to the actual MDD operations belonging to a specific encoding.
- Vector set (vset_t): this refers to the MDD implementation structure that is used for maintaining sets of state vectors and auxiliary data, e.g. it is used for storing the reachable state space.
- Vector relation (vrel_t): this refers to the MDD implementation structure that is used for maintaining (transition) relations and auxiliary data, e.g. it is used for storing the next-state function of a model.

The most crucial auxiliary data that is being stored in the vector set and vector relation is the projection on the state vector. This makes it possible to do operations on a subvector of the state vector, when certain state slots do count.

When considering the use of MDD operations, the current exploration algorithms only have access to high-level operations via the vector domain, for example union of, intersection of and equality matching between two MDD. The concrete (low-level) implementations depend on the type of MDD data structure encoding selected by the user.

In Table 3.5 and Table 3.6 an overview of the most relevant high-level MDD operations is given in simplified form. All these operations are defined in the header file vector_set.h (with concrete implementation in file vector_set.c).

Before starting a symbolic reachability analysis, the user is able to choose which vector domain (or MDD data structure encoding) is used, after which the appropriate vector set and vector relations and references to the high-level MDD operators are constructed. Hereby the header file vdom_object.h comes into play, which defines the possible high-level MDD operators for which a connection can be established with a particular associated MDD operator that is implemented in the chosen MDD data structure encoding.

High-level MDD operation	Input arguments	Return value
vset_create	- vector domain (as vdom_t) - length of state vector - projections of state vectors	New empty instance of MDD (as vset_t) with the number of levels corresponding to length of state vector and containing projections as auxiliary data.
vset_add	- MDD (as vset_t) - state (as indexed state vector)	MDD (as vset_t) with the given state added.
vset_is_empty	- MDD (as vset_t)	True if given MDD is empty and otherwise False.
vset_equal	- 2 MDDs (as vset_t)	True if the given MDDs are equal and otherwise False.
vset_copy	- MDD (as vset_t)	Copy of MDD (as vset_t).
vset_project	- source MDD (as vset_t) - projection list	Projected MDD (as vset_t) of source MDD using the given projection list.
vset_union	- 2 MDDs (as vset_t)	Union of given MDDs (as vset_t).
vset_intersect	- 2 MDDs (as vset_t)	Intersection of given MDDs (as vset_t).
vset_minus	- 2 MDDs (as vset_t)	Difference of given MDDs (as vset_t).
vset_enum	- MDD (as vset_t) - call back function - context information	Enumerate every (sub) state of given MDD and use call back function and context information to perform operations on these (sub) states.
vset_least_fixpoint	- source MDD (as vset_t) - list of relation MDDs (as vrel_t) - number of relation MDDs in list	MDD (as vset_t) that is the result of performing a least fix-point computation on the source MDD using the relation MDDs (for which there number is given).
vset_next	- source MDD (as vset_t) - relation MDD (as vrel_t)	Return an MDD (as vset_t) containing all successors that can be reached in one step from (sub) states in the given source MDD using the given relation MDD.
vset_prev	- source MDD (as vset_t) - relation MDD (as vrel_t)	Return an MDD (as vset_t) containing all predecessors that can reach (sub) states in the given source MDD in one step using the given relation MDD.

Table 3.5: Some high-level MDD operations and their arguments as defined in header file vector_set.h (simplified).

High-level MDD operation for transition groups	Input arguments	Return value
vrel_create	- vector domain (as vdom_t) - length of state vector - projections of state vectors	New empty instance of MDD (as vrel_t) with the number of levels corresponding to 2 × length of state vector (representing unprimed and primed levels) and containing projections as auxiliary data.
vrel_add	- relation MDD (as vrel_t) - source state (as indexed state vector) - successor state (as indexed state vector)	Relation MDD (as vrel_t) with the given source and successor states added (corresponding to the 'from' and 'to' states, respectively).

Table 3.6: Some high-level MDD operations and their arguments for transition groups as defined in header file vector_set.h (simplified).

3.4 Symbolic reachability analysis in LTSmin toolset

In the LTSmin toolset [4] one can use transition relations from general types of models, and a state is described by a vector consisting of state slots. The transition relation is split into so-called transition groups. The state slots of a state vector correspond to the partitioning into levels/components, and the transition group IDs to the events for the Saturation approaches. In the LTSmin toolset it is not required that the transition relation is decomposed into a Kronecker-consistent way. Furthermore the tool does not build the transition groups (of the transition relation) in advance, but in parallel next to generating the reachable state space as the domain of the state slots may be infinite in size. The transition groups are updated with new (sub) transitions by retrieving the newly found (sub) states from the reachable state space so far during an iteration of the reachability analysis and use these to calculate their successor (sub) states.

The symbolic exploration algorithms reside in spec-reach.c and some useful variables and operations are summarized in Table 3.7 and Table 3.8.

Global variable	**Explanation**
int N	The length of a state vector.
int nGrps	The total number of transition groups of the model.
proj_info *proj	Array variable storing information about the projections of the state vector per transition group.
vrel_t *group_next	Array variable storing the transition relation in individual transition groups.
vset_t *group_explored	Array variable storing the currently discovered set of states (of reachable state space) per transition group.
vdom_t domain	The type of MDD data structure encoding currently being used.

Table 3.7: Some global variables used by reachability tools as defined in file spec-reach.c (simplified).

High-level operation	**Input arguments**	**Explanation**
expand_group_next	- transition group ID - MDD (as vset_t)	Updates the transition group with given ID using the given MDD of reachable states discovered so far.

Table 3.8: Operation used by reachability tools as defined in file spec-reach.c (simplified).

Currently the LTSmin toolset contains the following symbolic reachability techniques:

- Variants of breadth first search (bfs-prev & bfs, as given in Listing 1.1 left and right, respectively);
- Variants of chaining (chain-prev & chain, as given in Listing 1.2 left and right, respectively);
- Variants of tool-specific Saturation-like approaches which use the previous two techniques as supporting traversal technique (sat-like & sat-loop);
- A tool-specific Saturation-like approach which performs least fix-point computations (sat-ddd).

The following text boxes will go into some of the Saturation-like approaches which will be used for the experiments (in chapter 5), namely the search techniques sat-like and sat-ddd.

■ Basic LTSmin version of Saturation-like search technique sat-like

The LTSmin toolset contains a traversal technique with the option name sat-like which is inspired by the Saturation approach as devised by G. Ciardo, but operates in a completely different way and the pseudo code of the basic version of this algorithm is given in Listing 3.1. Basically this algorithm starts at the terminal MDD nodes and works up the MDD tree until the root level at `nLvls - 1`. While the root level is not yet reached (line 19 in Listing 3.1), it will perform a state space search using a predetermined auxiliary traversal technique and a list of transition groups that start at the current working MDD level `k` (line 20 in Listing 3.1). The function `reach_proc` basically performs a traditional state space search, either breadth-first search or chaining over the transition groups that start at level `k`. If the resulting MDD from this state space search is the same as the one at the start of this search (line 21 in Listing 3.1), then the working MDD level `k` is increased by 1, indicating that the MDD tree until level `k` is 'saturated'. If the MDD has changed (line 23 in Listing 3.1), then the working MDD level `k` is set to a lower MDD level to see if more new states can be discovered using transition groups that start at lower topmost MDD levels. This is an attempt to 'saturate' lower levels of the MDD tree before continuing further up the MDD tree.

This algorithm is using an auxiliary traversal technique to 'saturate' the state space for a certain MDD level first before heading down the MDD to 'saturate' lower levels (using the auxiliary traversal technique) in case of an updated state space so far. This is different from the Saturation algorithm by G. Ciardo, where Saturation goes down the MDD immediately to start saturating the MDD from the bottom-up instead of sticking to a particular MDD level.

```
 1  // Total number of MDD levels.
 2  int nLvls
 3  
 4  // List (of length nLvls) containing list of transition groups that start at
 5  // the topmost MDD level, which is indicated by the index of the list.
 6  // E.g. index 1 of list corresponds to topmost MDD level 1, index 2 to topmost MDD
 7  // level 2, and so on. At index 2 of this list, a list of transition groups is
 8  // stored that start at topmost level 2.
 9  array<array<vrel_t>> topmostlevelGroups
10  
11  // MDD tree of initial state.
12  vset_t init
13  
14  // Auxiliary traversal technique used (either breadth-first search or chaining).
15  exploreStrategy order
16  
17  int k := 0
18  vset_t visited := init
19  while (k != nLvls) :
20    vset_t new_vis := reach_proc(order, visited, topmostlevelGroups[k])
21    if (visited == new_vis) :
22      k := k + 1
23    else :
24      k := go_to_lower_level(k)
25    visited := new_vis
```

```
 1  // Start a state space search using the given exploration strategy (breadth-first
 2  // search or chaining), the (initial) set of states and list of
 3  // transition groups.
 4  vset_t reach_proc(exploreStrategy order, vset_t set, array<vrel_t> trans_groups)
 5  
 6  // For a given MDD level number, return a lower MDD level number for 'saturating'
 7  // lower MDD levels of state space.
 8  int go_to_lower_level(int level)
```

Listing 3.1: Pseudo code of basic sat-like algorithm and accompanying function definitions.

■ Fix-point computational search technique sat-ddd

The option sat-ddd (for which its algorithm is given in Listing 3.2) calculates the fix-point on the currently discovered reachable state space using the incomplete transition groups. It repeats the fix-point computation (line 16 in Listing 3.2) multiple times in which the transition groups are updated with the latest version of the reachable state space outside the fix-point computation (lines 14 – 15 in Listing 3.2), until no more new states are discovered.

```
 1  // Total number of transition groups.
 2  int nGrps
 3
 4  // List of transition groups.
 5  array<vrel_t> group_next
 6
 7  // MDD tree of initial state.
 8  vset_t init
 9
10  vset_t visited := init
11  vset_t old_vis := { }
12  while (visited != old_vis) :
13    old_vis := visited
14    foreach (int i in 1 .. nGrps) :
15      expand_group_next(i, visited)
16    visited := vset_least_fixpoint(visited, group_next, nGrps)
```

```
The indices of arrays are numbered from
1 to length.
```

```
General Prebuilt Saturation is performed
inside the function vset_least_fixpoint
(which will be discussed in section 4.2.2).
```

Listing 3.2: Pseudo code of sat-ddd algorithm.

3.5 Summary of LTSmin toolset

The LTSmin toolset contains a layered architecture, separating language specific details from the actual verification techniques. The relations among the discussed parts of the LTSmin toolset with the MDD data structure encodings and access to the model through PINS, are given in Figure 3.6.

The most crucial part of the LTSmin toolset is the PINS interface, which defines a transition relation of a model in a unified way. The transition relation is partitioned into a number of transition groups to account for the different events that can occur in the model. The model itself is partitioned in a number of components. States are described using vectors with a size equal to the number of components of the model. The individual slots of these state vectors contain a local state value that a certain component of the model can assume. The transition groups affect certain components of the model and these are captured in the PINS dependency matrix.

Regarding the MDD data encodings, the LTSmin toolset provides access to four MDD libraries, namely ATermDD, BuDDy, LibDDD and ListDD. Next to the conventional MDD data structure as a graph-like structure, the linked list structure has been introduced which considers an MDD node as a linked list and the entries of the node are entries of this list. Each of these entries contains a pointer to an MDD node (another linked list) at a lower MDD level.
To have access to the functionality of the MDD libraries, data objects have been introduced for storing an MDD tree of the reachable state space, an MDD tree of a transition group and a domain object to point to the right MDD operations of a certain chosen MDD library.

The symbolic reachability algorithms in the LTSmin toolset are implemented in one file called spec-reach.c and the algorithms make use of the high-level MDD operations. Individual access to MDD nodes on a low level of the tool is not possible as to hide the implementation details of the different MDD libraries. The toolset contains a variety of symbolic reachability algorithms containing the traditional symbolic approaches as breadth-first search and chaining, and LTSmin versions of Saturation-like approaches.

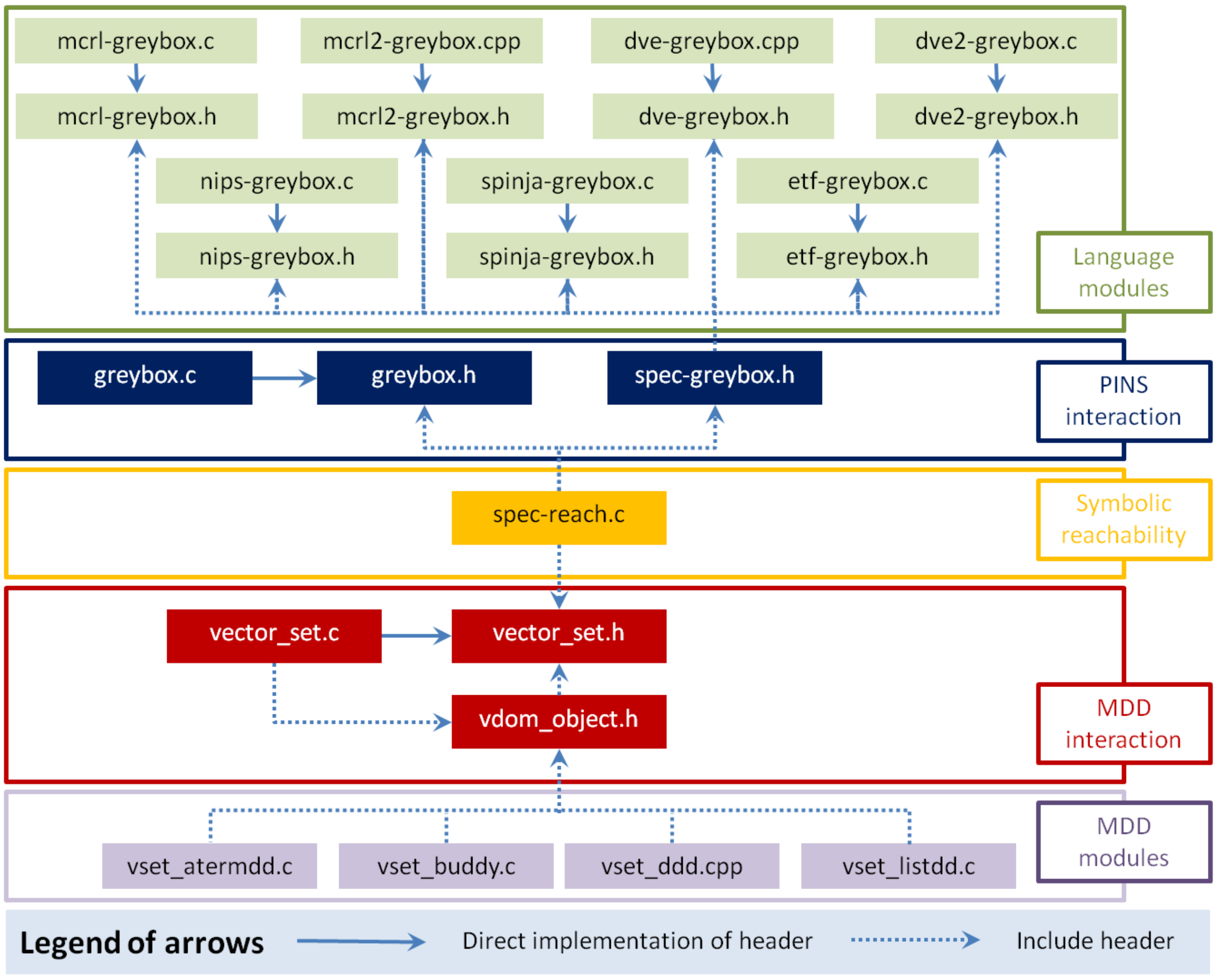


Figure 3.6: Pictorial overview of parts of layered architecture of LTSmin toolset concerning symbolic reachability tools.

4 Design & implementation of Saturation for LTSmin toolset

The design of Saturation for the LTSmin toolset is based on the analysis in chapters 2 and 3: the analysis from chapter 2 provides an overview of the different versions of the Saturation algorithm, and the analysis from chapter 3 gives an overview of the current possibilities of the LTSmin toolset, especially for the implementation of symbolic reachability tools. In this chapter both are taken into consideration when deciding:

- which Saturation approach to implement in the LTSmin toolset and
- how to implement the chosen algorithm in the LTSmin toolset.

First section 4.1 presents a requirements analysis for the design of Saturation. In section 4.2 some design challenges derived from the Saturation pseudo code are described with their implementation solutions. The design of the Saturation algorithm for the LTSmin toolset then needs to be implemented and during implementation some issues occurred which need to be tackled. These issues are discussed in section 4.3.

4.1 Requirements analysis for design of Saturation

The requirements analysis is split into two parts: section 4.1.1 gives the choice of the Saturation approach and the arguments leading to this choice, and section 4.1.2 gives solutions to issues regarding implementation of the chosen algorithm into the LTSmin toolset.

4.1.1 LTSmin-related requirements & design choices

To be able to make a choice of the Saturation approach to use, a small requirements analysis is performed to assess the needs of incorporating the Saturation algorithm into the LTSmin toolset. When taking the goal description and the LTSmin toolset into consideration a number of requirements can be inferred which need to be adhered to when incorporating the Saturation algorithm into the toolset. The requirements concerning the LTSmin toolset are given in Table 4.1.

R1.1	The use of PINS as intermediate interface to the language modules should be respected.
R1.2	The algorithm should be available as a reachability option next to the currently available symbolic reachability options of the LTSmin toolset.
R1.3	No restrictions should be imposed on the structure of the model and/or its transition relation.
R1.4	The transition relation (split into transition groups) is not known in advance and the algorithm should be able to compute it in parallel to state space generation.

Table 4.1: LTSmin-related requirements.

Requirement R1.1 is directly derived from the goal description and needs to be respected, because the PINS interface is the main feature of the toolset which separates language features from algorithm-specific details. Requirement R1.2 has a practical reason behind it; the user of the LTSmin toolset should be given the opportunity to choose the newly implemented reachability option as one of the possible options for generating the reachable state space. The last two requirements result from the analysis of the symbolic reachability algorithms currently available in the LTSmin toolset (as described in section 3.4).

The first three requirements dictate how the Saturation algorithm should be incorporated in the LTSmin toolset. The Saturation algorithm itself does not have any special needs regarding how the model is abstracted into a specification and/or language and can be implemented inside the existing architecture of the LTSmin toolset, just like other available symbolic traversal techniques. But the last two requirements do have an impact on the choice of Saturation approach to pick for implementation in the LTSmin toolset.

With the requirements clarified the choice of which Saturation algorithm to take for implementation can now be explained into two steps.

First the transition relation of a model is specified using a modelling language for which the LTSmin toolset has a front-end available and where PINS acts as the interface to the transition relation without having to deal with language-specific details. Although the transition relation is specified in a certain modelling language, this is abstracted away by PINS and this interface poses no restriction how the transition relation of a model is structured and/or partitioned. When considering the Saturation approaches, two main categories are discussed, namely the Kronecker and the General Saturation approaches. The former requires that the transition relation is decomposable in a Kronecker-consistent way, while the latter lifts this restriction. When considering requirement R1.3, it is obvious that the Kronecker Saturation approaches cannot be selected for implementation into the LTSmin toolset.
Although the Kronecker partitioning of the transition relation seems to resemble the partitioning in the PINS dependency matrix, they are in fact quite different. The Kronecker partitioning makes sure that the transition relation is decomposed in local transitions per MDD level and event, and each of these local transitions is independent of any other. Just like the PINS dependency matrix, it gives information about which state slots or MDD levels are affected by transition groups or events and this is the only aspect that they have in common.

Second the transition relation of a model is not known in advance when starting a reachability analysis in the LTSmin toolset. It is also not possible to prebuilt it before starting a reachability analysis and with this the local state space sizes (or variable domains) per MDD level can also not be calculated. This is due to the fact that the range of values from model variables is mostly not specified in the model and their range can therefore contain an infinite number of values. When considering requirement R1.4 and the fact that only General Saturation approaches are possible (due to requirement R1.3), it becomes clear that the only viable option that is left for implementation is the General Saturation approach that builds the transition relation during state space generation, namely General On-the-fly Saturation.

4.1.2 Saturation-related requirements & design solutions

Now the choice has fallen on General On-the-fly Saturation, there are still some issues that need to be tackled before it can be implemented in the LTSmin toolset.
From the analysis of the Saturation approaches it has become clear that the Saturation algorithm itself requires some special attention before it can be incorporated into the LTSmin toolset. These algorithm-specific requirements are given in Table 4.2.

R2.1	The algorithm needs to be able to access low-level MDD operations.
R2.2	The algorithm needs to have access to the individual transition groups.
R2.3	The algorithm needs to be able to detect the boundaries of the transition groups (namely the top- and bottommost level) for performing fix-point computations.
R2.4	The computation of the relational product of an MDD is part of the algorithm and needs to be incorporated (and adjusted) inside it.
R2.5	The algorithm needs to be able to store intermediate calculated results in different parts of the algorithm.

Table 4.2: Saturation-related requirements.

Requirement R2.1 is due to the fact that the Saturation algorithm needs to change individual MDD nodes and access the appropriate arcs pointing to other MDD nodes. Requirements R2.2 and R2.3 are important for performing the fix-point computations and their boundaries make sure the algorithm does not have to go through levels of the MDD which do not change. It gives an indication at which topmost level to start the process and at which bottommost level to stop the recursive calls to the relational product. Regarding the relational product, requirement R2.4 indicates that it needs to be able to use and adjust this function if any exist in the LTSmin toolset (otherwise it can be implemented from scratch with the call to the topmost Saturation function inside it). And finally the use of lookup tables is stressed in requirement R2.5, to prevent recalculations of intermediate calculated results.

With the second set of requirements ready, the design decisions for implementation of the algorithm in the LTSmin toolset are now the main focus.

The Saturation algorithm has a main structure in which multiple recursive calls are made, but it is impossible to implement this whole structure on a high level in the tool (e.g. in spec-reach.c). This is due to the fact that the algorithm requires operations on individual MDD nodes which are not accessible from a high level. Implementation on a high level would result in introducing low-level MDD functions to be incorporated in the vector domain, and it is more suitable to make use of the available architectural structure as it is. Also to be able to adhere to requirement R2.1 it is decided to implement the entire Saturation algorithm in an MDD library, but this has a downside as well. It requires re-implementing the whole algorithmic structure in every MDD library. For libraries with full-fledged implementations of MDD operations (as ATermDD and listDD) this would result in variations of the algorithmic structure. This is still manageable since an implementation in a particular library can function as example for implementation in other such libraries. The problem lies in implementing the algorithm inside wrapper libraries with a foreign source (as BuDDy and libDDD) that have a different structure than own (in-house developed) full-fledged implementations. It depends per wrapper library how easy changes can be made to incorporate a whole new algorithm in it.
Furthermore when implementing inside libraries with full-fledged implementations of MDD operations, it is possible to reuse and make adjustments to currently available functions in the LTSmin toolset. This is especially so for adjusting the functions that are used for calculating the relational product, which are part of the Saturation algorithm.
The initial focus of the implementation will lie on implementation of the Saturation algorithm inside libraries as ATermDD and listDD, which contain full-fledged implementations of MDD operations. The focus will also be on using the LDD data structure (where nodes are stored in a linked list structure), because of flexibility of switching between the ATermDD and listDD libraries. A major factor that contributed to this choice is the capability of accessing low-level MDD operations more easily, also by reusing and adjusting currently available functionality.

From a high-level perspective there still needs to be a way to access the Saturation algorithm (from spec-reach.c) and a suitable candidate is implementing the algorithm inside the function for performing fix-point computations (via the function `vset_least_fixpoint` from the domain object). When considering the structure of the Saturation algorithm, it is basically another version of a fix-point computation on an MDD, so the choice for implementing it inside this function is clear. Furthermore one of the parameters of this function is the list of transition groups and so the algorithm has access to them (as required by requirement R2.2). Now the transition groups are accessible for the Saturation algorithm, the boundaries (top- and bottommost levels) of these relations still need to be extracted, as stated by requirement R2.3. As explained in section 3.3.2, each transition group keeps a projection on the state vector which is a list of state slot values (MDD levels) that are affected by this transition group. From this projection the boundaries are easily extracted, which are basically the first and last entry of the projection list.

The final two requirements R2.4 and R2.5 are mainly affected by the location where the Saturation algorithm is going to be implemented. As indicated above, this will be done inside libraries with full-fledged implementations of MDD operations. In these libraries a function for calculating the relational product for an MDD is present, so it can be changed slightly to cope with the Saturation algorithm. Also implementing in such libraries gives access to other available features such as lookup tables.

4.1.3 Summary of design choices & solutions

Before discussing the design challenges, a summary of the design choices and solutions made to meet the requirements is given.

The requirements R1.1 and R1.2 are basic requirements that are only adhered to when starting to implement the Saturation algorithm into the LTSmin toolset.
To adhere to requirement R1.3, the choice is made to only take the General Saturation approaches into account. This is due to the fact that they do not pose any restriction on the (partitioning of the) transition relation, which is required by the Kronecker Saturation approaches. The latter requires that the transition relation is partitioned per MDD level and event that are independent from each other.
With the choice made for General Saturation approaches, the focus turned to adhering to requirement R1.4. The only viable choice here is to take the On-the-fly version of the General Saturation approaches, since this algorithm builds the transition relation in parallel with the reachable state space of the model. Building the transition relation in advance is not possible in the LTSmin toolset due to the possibility of encountering local state space sizes with spurious and infinite values.

When considering the requirements of the Saturation algorithm itself, requirement R2.1 is met by implementing the Saturation algorithm inside a full-fledged MDD library as ATermDD and listDD. This also affects requirements R2.4 and R2.5, because implementation on a low level makes it possible to reuse functions that deal with the relational product and make use of the possibilities in regard to memoization. For implementation purposes, the choice is further reduced to implementation in one of these MDD libraries, namely ATermDD.
To meet requirements R2.2 and R2.3, a solution is found by accessing the Saturation algorithm from a high level via the function `vset_least_fixpoint` (in spec-reach.c). This is due to the fact that Saturation is a fix-point computation. Furthermore this function also provides access to the transition groups, from which the bottom- and topmost levels can be calculated.

4.2 Algorithmic-dependent design challenges

The pseudo code of the Saturation algorithm as such needs a couple of adjustments before it can be used in the LTSmin toolset. Some of the constructs used in Saturation are being adjusted for efficiency reasons and for making it suitable to incorporate it into the toolset. The issues and challenges that are going to be discussed, are mainly related to the ATermDD library. Section 4.2.1 gives an overview of the adjustments made to the Saturation algorithm to let it work inside the LTSmin toolset.
From the pseudo code of the Saturation algorithm it is also noticeable that the structural difference between the General Prebuilt and General On-the-fly Saturation algorithm is quite minimal, namely the presence or absence of the `confirm` function. Since this difference is quite minimal, the decision is made

to provide both General Saturation approaches as symbolic reachability tools to the user. However, since General Prebuilt Saturation on its own is not usable in the LTSmin toolset (because the transition relations cannot be computed in advance), it is incorporated as part of an LTSmin version of a Saturation-like option in the LTSmin toolset, and section 4.2.2 goes into detail here. The General On-the-fly Saturation approach is the newly implemented option which is the subject of section 4.2.3.

To give a sense how the pseudo code of the Saturation algorithm has been altered to fit inside the LTSmin toolset, an updated version of this pseudo code is presented in Listing 4.1, and Listing 4.2 explains some of the new MDD operations used here. The changes are discussed in the following subsections.

```
// Start saturating MDD subtree in a certain level.
MDDnode saturate(int l, MDDnode p) :
  MDDnode t := findInLookupTable(SC[l], p)
  if (t != null) :
    return t
  t := saturateLocals(l, p)
  t := startFixPoint(l, t)
  insertInLookupTable(SC[l], p, t)
  return t

// Traverse entries of an MDD node and saturate each one.
MDDnode saturateLocals(int l, MDDnode p) :
  if (p == getMDDFalse() || p == getMDDTrue()) :
    return p
  MDDnode sat_p := saturate(l - 1, p.getCurrentArc())
  MDDnode next_p := saturateLocals(l, p.getNextMDDEntry())
  return constructMDD(p.getValue(), sat_p, next_p)

// Start fix-point computation on MDD subtree for every event that starts at
// the given topmost level.
MDDnode startFixPoint(int l, MDDnode t) :
  if (t == getMDDFalse() || t == getMDDTrue()) :
    return t
  MDDnode old_t := getMDDFalse()
  while (old_t != t) :
    old_t := t
    foreach (int grp in getGroupIDsOnTopLevel(l)) :
      expand_group_next(grp, t) // Update used by On-the-fly version
      expand_group_next()       // Update used by revised On-the-fly version
      MDDnode r := getTransitionGroupMDD(grp)
      projection proj := getTransitionGroupProjection(grp)
      t := applyFixpoint(t, r, grp, proj, l)
  return t

// Applying fix-point computation on MDD node using transition group MDD.
MDDnode applyFixpoint(MDDnode s, MDDnode r, int grp, projection prj, int l) :
  MDDnode t := s;
  while (s != getMDDFalse() && r != getMDDFalse()) :
    if (s.getValue() < r.getValue()) :
      s := s.getNextMDDEntry()
    elif (s.getValue() > r.getValue()) :
      r := r.getNextMDDEntry()
    else :
      MDDnode new_t := t
      while (new_t.getValue() != r.getValue()) :
        new_t := new_t.getNextMDDEntry()
      MDDnode u := transLevel(new_t.getCurrentArc(), r.getCurrentArc(), grp, prj, l)
      t := unionMDDs(l, t, u)
      s := s.getNextMDDEntry()
      r := r.getNextMDDEntry()
  return t
```

Global lookup table (per level)

■ SC : Saturation Cache

Global lookup tables (per level & group)

■ RC : Rel-prod Cache

```
// Start performing relational product on MDD subtree in a given level for
// finding new MDD subtree to saturate with.
MDDnode startRelProd(MDDnode s, MDDnode r, int grp, projection prj, int l) :
  if (prj.isProjEmpty()) :
    return s
  else :
    MDDnode t := findInLookupTable(RC[l][grp], (s, r))
    if (t != null) :
      return t
    if (proj.getCurrentProjLevel() == l) :
      t := applyRelProd(s, r, grp, prj, l)
    else :
      t := copyLevel(s, r, grp, prj, l)
    t := saturate(l, t)
    insertInLookupTable(RC[l][grp], (s, r), t)
    return t

// Applying the relational product operation on MDD node using transition group
// MDD.
MDDnode applyRelProd(MDDnode s, MDDnode r, int grp, projection proj, int l) :
  MDDnode t := getMDDFalse()
  while (s != getMDDFalse() && r != getMDDFalse()) :
    if (s.getValue() < r.getValue()) :
      s := s.getNextMDDEntry()
    elif (s.getValue() > r.getValue()) :
      r := r.getNextMDDEntry()
    else :
      MDDnode u := transLevel(t.getCurrentArc(), r.getCurrentArc(), grp, proj, l)
      t := unionMDDs(l, t, u)
      s := s.getNextMDDEntry()
      r := r.getNextMDDEntry()
  return t

// Construct new MDD node with entries of transition group MDD node (finalizing
// the relational product operation).
MDDnode transLevel(MDDnode s, MDDnode r, int grp, projection proj, int l) :
  if (r == getMDDFalse()) :
    return getMDDFalse()
  else :
    addNewValue(l, r.getValue()) // Used by revised On-the-fly version
    projection projNext := proj
    projNext.setProjToNextLevel()
    MDDnode newArc := startRelProd(s, r.getCurrentArc(), grp, projNext, l - 1)
    MDDnode newEntry := transLevel(s, r.getNextMDDEntry(), grp, proj, l)
    return constructMDD(r.getValue(), newArc, newEntry)

// Copy MDD node at current level, because the transition group MDD has
// no transitions stored at this level (they are identity transitions, which are
// not stored).
MDDnode copyLevel(MDDnode s, MDDnode r, int grp, projection proj, int l) :
  if (s == getMDDFalse()) :
    return getMDDFalse()
  else :
    MDDnode newArc := startRelProd(s.getCurrentArc(), r, grp, proj, l - 1)
    MDDnode newEntry := copyLevel(s.getNextMDDEntry(), r, grp, proj, l)
    return constructMDD(s.getValue(), newArc, newEntry)
```

Listing 4.1: Pseudo code of implemented General Prebuilt Saturation algorithm with additional code for On-the-fly version highlighted in light-blue and for revised On-the-fly version in dark-blue (which is discussed in chapter 6).

```
// Entries of an MDD node are stored in a linked list structure.

// Construct a new entry node with the given local state value, pointing to the
// given lower MDD node and the next entry of the current MDD node.
MDDnode constructMDD (int value, MDDnode lowerMDDNode, MDDnode nextEntryNode)

// Return local state value of the current entry of the MDD node.
int getValue()

// Return (lower) MDD node pointed to by arc at the current entry of the MDD node.
MDDnode getCurrentArc()

// Return next entry of MDD node starting from the current entry.
// If current entry is the final one, then it returns the False MDD node.
MDDnode getNextMDDEntry()
```
Listing 4.2: Function definitions used in pseudo code of implemented General Saturation algorithm.

4.2.1 Design-specific adjustments to Saturation algorithm

The original pseudo code in Listing 2.4 assumes the use of in-place updates for updating the MDD of the reachable state space and this makes it possible to create new empty MDD nodes, which grow during the Saturation process when new MDD nodes are created and pointed to. In Figure 4.1 the MDD operations for retrieving and (in-place) updating an entry of an MDD node are visualized to give a sense how they would operate when using the LDD data structure. In LTSmin no in-place updates are used and the creation of new MDD nodes involves constructing a new MDD node with a specific local state value and a link to another MDD node (on a lower level). In case of using the LDD data structure, one would also indicate a link to the next entry of the current MDD node.

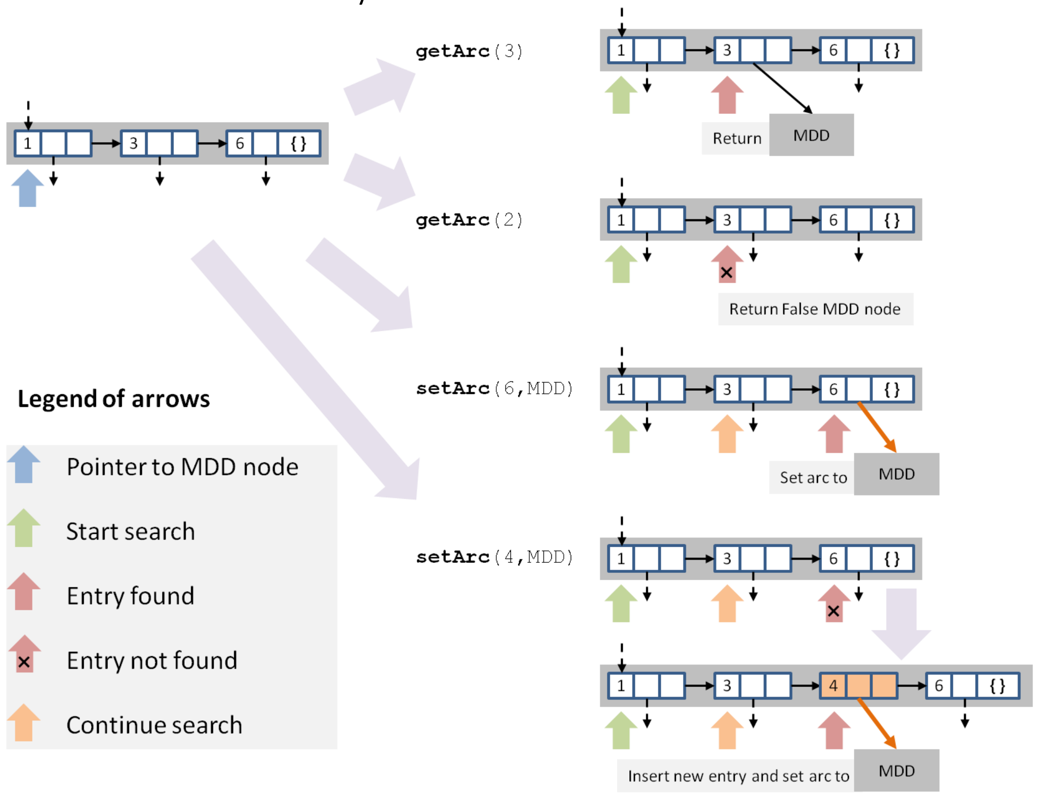


Figure 4.1: Low-level MDD operations used in original pseudo code of General Saturation algorithm in Listing 2.4.

Basically this means that in the original pseudo code the function calls for creating a new MDD node (the function `createEmptyMDD` as shown in Listing 2.2) can be removed and in-place updates of an arc inside an MDD node (the function `setArc`) is replaced by the function `constructMDD` (as shown in Listing 4.2) that constructs a new MDD node with the specified arguments (which also entails the introduction of recursive functions, more about this later in this section). And to prevent creating MDD nodes with a link to a False MDD node on a lower level, a check is performed before each construction process to see if the lower MDD node is not equal to the False MDD node (in case it is, no new MDD node is constructed). This feature is already present in the ATermDD library when constructing a new MDD node.

The functionality of calculating the relational product (function `relProd` in Listing 2.4) is part of the Saturation algorithm. It turns out that similar functionality is already present in the ATermDD library, but instead recursive functions are used and give a better time complexity than what is achieved when implementing these functions according to the original pseudo code of the Saturation algorithm in Listing 2.4 (as is explained in the text box below). There are also some other functions in the algorithm, which suffer from a bad time complexity and these are tackled as well using recursion. This has resulted in the following changes:

- The function `saturate` given in Listing 2.4 is split into the entry function `saturate` and the recursive function `saturateLocals` in Listing 4.1.
  (That is lines 12 – 13 of function `saturate` in Listing 2.4 are replaced by function `saturateLocals` in Listing 4.1.)
- The function `doFixPoint` given in Listing 2.4 is split into the entry function `startFixPoint` and the recursive functions `applyFixPoint` and `transLevel` in Listing 4.1.
  (That is lines 26 – 31 of function `doFixPoint` in Listing 2.4 are replaced by functions `applyFixPoint` and `transLevel` in Listing 4.1.)
- The function `relProd` given in Listing 2.4 is split into the entry function `startRelProd` and the recursive functions `applyRelProd`, `transLevel` and `copyLevel` in Listing 4.1.
  (That is lines 42 – 48 of function `relProd` in Listing 2.4 are replaced by functions `applyRelProd`, `transLevel` and `copyLevel` in Listing 4.1.)

In most cases of these recursive functions, the next entry of an MDD node is passed on into the next recursion until the end of the MDD node is reached and this is indicated by a False MDD node (see explanation in Figure 3.3).

Since the MDD encodings of the ATermDD library use fully-reduced MDDs for the transition relations, the Saturation algorithm also has to deal with the issue if there are skipped levels in the MDD tree, which are caused by intermediate identity transitions. For each transition group MDD a projection list is stored, indicating on which state slots (or MDD levels) a local transition occurs. To handle this issue, a modification inside the function dealing with the relational product is performed. Since the functionality dealing with the relational product is reused from the MDD library, the issue is dealt with by copying the current MDD node of the reachable state space for the skipped level and head to the next MDD nodes at lower levels. This is the function `copyLevel`, which is called within the function `startRelProd` (in line 64 in Listing 4.1).

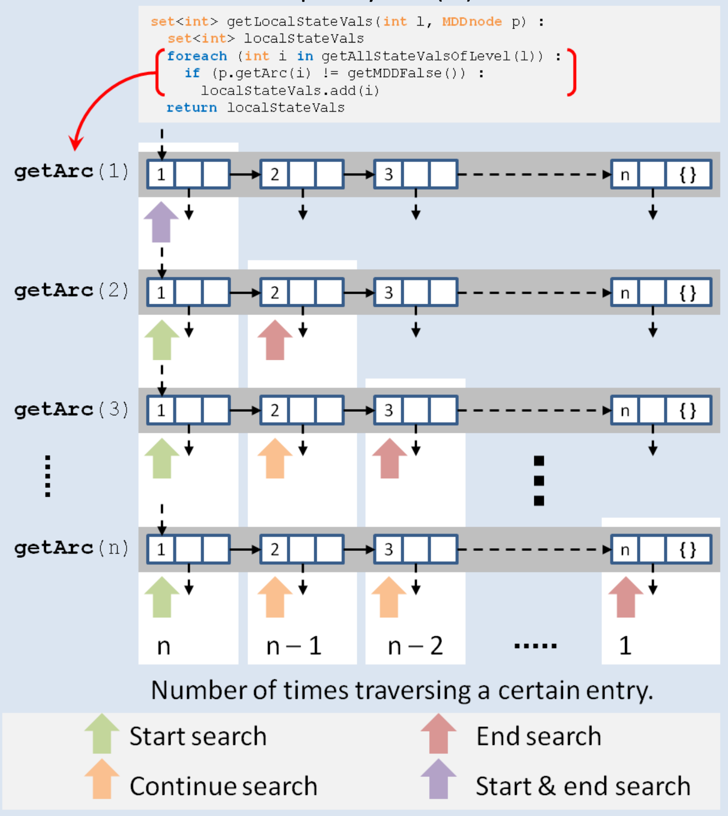
The original pseudo code also contains checks in the function `checkMDDnode` (as explained in Listing 2.2 and shown in lines 14 and 49 in Listing 2.4). These checks are performed to see if a similar MDD subtree has been constructed before and are no longer required, because the ATermDD library uses subterm sharing (of common nodes) and so this issue is dealt automatically inside the library.

Also the transition group MDDs used in the LTSmin toolset are not further split into smaller conjunct MDDs and that is not required for the Saturation algorithm. So the function `buildTransitionMDDFromConjuncts` as used in line 25 in Listing 2.4 is replaced by the function `getTransitionGroupMDD` as used in line 30 in Listing 4.1.

■ Time complexity analysis of Saturation pseudo code

When the MDD node is represented as a linked list structure, the pseudo code assumes that a reference to an MDD node is kept by pointing to the first entry of this linked list. Every time a specific local state value of the MDD node needs to be retrieved, it searches by starting from the first entry through the linked list until the entry is found (this is the `getArc` function in Figure 4.1). Updating an arc for a certain local state value in the MDD node also has to search for the correct entry in the linked list (this is the previously mentioned `setArc` function in Figure 4.1). So every time when one of these operations is performed, it starts its search from the first entry of the MDD node.

To assess how inefficient this is, the worst-case time complexity is evaluated at points in the original pseudo code algorithm in Listing 2.4 where such search iterations are taking place over entries in the linked list structure of the MDDs. When considering a domain of n local state values, it turns out that in the worst-case scenario an MDD node has all these n entries present (otherwise there are less than n iterations through the MDD node).
Consider the worst-case time complexity for the call to function `getLocalStateVals` in Listing 2.4. This occurs when the MDD node has all n entries present and needs to traverse each of them from the first entry of the linked list. So in the for-loop it first calls `getArc`(1), `getArc`(2), `getArc`(3), until `getArc`(n). So in total the first entry of the linked list is traversed n times, the second entry $n - 1$ times, the third entry $n - 2$ times, and so on (see Figure 4.2). This will result in $n + (n-1) + (n - 2) + \ldots 3 + 2 + 1 = ½n(n + 1)$ iterations, so a worst-case time complexity of $O(n^2)$.

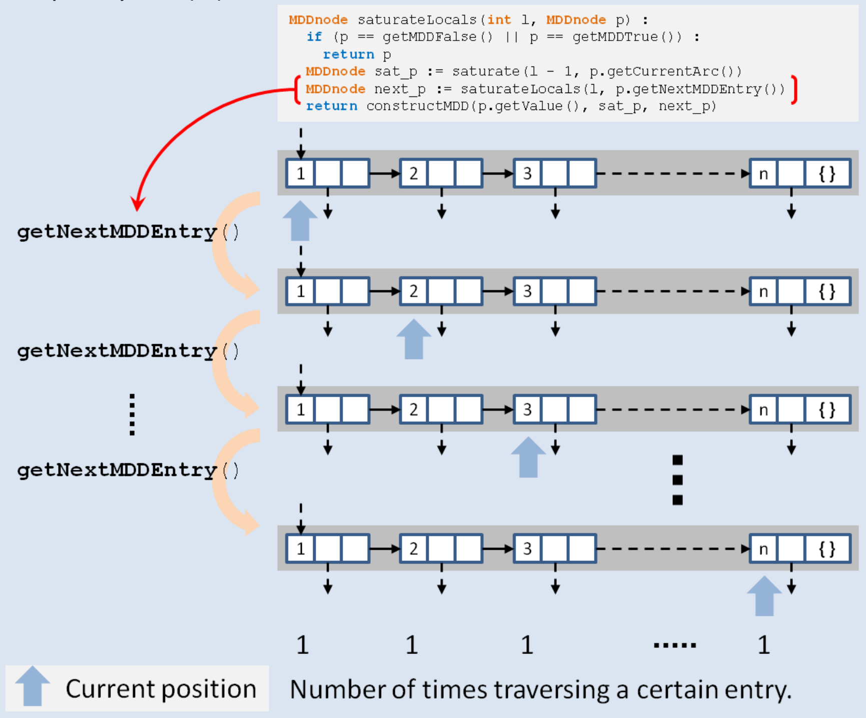


```
set<int> getLocalStateVals(int l, MDDnode p) :
  set<int> localStateVals
  foreach (int i in getAllStateValsOfLevel(l)) :
    if (p.getArc(i) != getMDDFalse()) :
      localStateVals.add(i)
  return localStateVals
```

Figure 4.2: Visualization of inefficient MDD node traversals with quadratic time complexity.

The for-loops iterating over n entries (as worst-case) in functions where `getArc` is called, have similar time complexity of $O(n^2)$. This accounts only for the point when one looks at finding and updating an entry in a linked list, without considering other function calls that may interfere. This accounts for lines 12 – 13 in function `saturate`, lines 27 – 31 in function `doFixPoint` and lines 44 – 48 in function `relProd` in Listing 2.4. In the latter two functions another outer for-loop is present, which gives rise to a worst-case time complexity of $O(n^3)$ (when not considering the inner-calls to `relProd`).

The recursive functions used in the MDD library are constructed in such a way that the entries of a linked list are traversed only once and this results in a worst-case time complexity of $O(n)$, as can be seen in Figure 4.3. Inside the functions where the fix-point computation is started (line 32 in Listing 4.1) and inside the computation of the relational product (line 62 in Listing 4.1), this results in a worst-case time-complexity of $O(n^2)$.



```
MDDnode saturateLocals(int l, MDDnode p) :
  if (p == getMDDFalse() || p == getMDDTrue()) :
    return p
  MDDnode sat_p := saturate(l - 1, p.getCurrentArc())
  MDDnode next_p := saturateLocals(l, p.getNextMDDEntry())
  return constructMDD(p.getValue(), sat_p, next_p)
```

Figure 4.3: Visualization of efficient MDD node traversals with linear time complexity.

4.2.2 Design-specific adjustments for LTSmin - General Prebuilt Saturation

For General Prebuilt Saturation to work inside the LTSmin toolset some additional functionality needs to be added to the ATermDD library where the algorithm is going to be implemented (inside the function that starts performing a fix-point computation in ATermDD library, namely the function called `set_least_fixpoint_list`).

In the initialization phase of the algorithm, the projection lists belonging to the transition groups are used to determine the transition group IDs per topmost level. These are used to determine which transition group starts at a certain topmost level when starting the fix-point computation. For this purpose a new global (array) variable is added in the MDD library, which stores a newly introduced structure type containing transition group IDs per topmost level. In the pseudo code in Listing 4.1 this is simplified to the function `getGroupIDsOnTopLevel` (line 27 in Listing 4.1) which returns a list of transition group IDs for

a certain level. The transition group ID is then used to obtain other data, such as the transition group MDD.
On the other hand, the bottommost levels per event (used to indicate when to stop the recursive calls to the relational product) are no longer needed, because this is solved in a different manner when reusing the relational product from the MDD libraries. Hereby it relies on the projection list for which a pointer is kept indicating at which entry of the projection list it currently is (the current level it is pointing to, is achieved with the function `getCurrentProjLevel` in line 61 in Listing 4.1). This pointer is increased each time it enters another level during the computation of the relational product (as mimicked with the function `setProjToNextLevel` in line 93 in Listing 4.1). When the end of the projection list is reached (the function `isProjEmpty` in line 55 in Listing 4.1), it knows it has reached the bottommost level of the transition group.

The functions `saturate` and `doFixPoint` from the original pseudo code in Listing 2.4 are revised into entry and recursive functions as shown in Listing 4.1 and these are the only new additional functions which need to be implemented in the ATermDD library; the rest can be reused with some slight adjustments.
Furthermore it is noticeable in the original pseudo code algorithm in Listing 2.4 that the two innermost for-loops in the function `doFixPoint` (lines 26 – 31 in Listing 2.4) are almost similar to the two for-loops in the function `relProd` (lines 43 – 48 in Listing 2.4). The difference lies in the following:

- In the function `doFixPoint` it reuses the state space computed so far from the start (referring to parameter `t` in line 19 in Listing 2.4), while in the function `relProd` an empty MDD is constructed for `t` (in line 42 in Listing 2.4).
- Inside the second for-loop of the function `doFixPoint` it uses the latest updated version of the computed state space (`t.getArc(i)` in line 30 in Listing 2.4), while the function `relProd` uses a fixed MDD (parameter `s` for `s.getArc(i)` in line 47 in Listing 2.4).

In the MDD library a similar function to `applyRelProd` from Listing 4.1 is already present and this is also the location where the issues mentioned above need to be tackled. The function `applyRelProd` (in Listing 4.1) is similar in functionality as the one in the original pseudo code inside the function `relProd` in Listing 2.4 and can be kept. But for `doFixPoint` from Listing 2.4 the function `applyRelProd` is copied and altered with the aforementioned issues in mind, resulting in function `applyFixPoint` in Listing 4.1.

To be able to work with General Prebuilt Saturation in the LTSmin toolset, a call needs to be made to the function `vset_least_fixpoint` in spec-reach.c with the MDD of the initial state, a complete set of the transition groups and the total number of these transition groups. But it is not possible to use this algorithm on its own in the LTSmin toolset, because the transition groups are not known in advance. This algorithm will be available as part of the option sat-ddd in the LTSmin toolset and results in an own variation on the Saturation algorithm.
When using this option with Saturation as fix-point computation, the Saturation process will build the reachable state space with the incomplete set of transition groups. Then the newly discovered states are retrieved from the intermediate resulting state space, which are used to update the transition groups. The Saturation process continues in a next iteration with the reachable state space so far and the updated set of transition groups, until no more new states are discovered and the final reachable state space is achieved. The pseudo code of this option is given in Listing 3.2.

4.2.3 Design-specific adjustments for LTSmin - General On-the-fly Saturation

To be able to let General On-the-fly Saturation work inside the LTSmin toolset, the following adjustments need to be incorporated on top of the changes made to General Prebuilt Saturation as discussed in section 4.2.2.

The difference with the Prebuilt version of the Saturation algorithm is the update process which is done in parallel to generating the reachable state space. In the function `startFixPoint` (that starts the fix-point computations as shown in Listing 4.1) additional functionality needs to be added to handle the updates of the transition groups. Unfortunately the update process from G. Ciardo [23] requires functionality for keeping track of the local state values that are discovered so far per MDD level. This is currently not available in the LTSmin toolset and the initial focus lies in devising an update process that fits with the LTSmin architecture, reusing available functionality as much as possible.
Every time when the latest version of lower parts of the reachable state space is changed (the condition in the while-loop in line 25 in Listing 4.1 evaluates to True), an update process is performed (highlighted in light-blue in line 28 in Listing 4.1) prior to calling the function `applyFixPoint` and the relational product inside it. In the update process the latest version of lower parts of the reachable state space is used to extract the new local states which are used to update one of the affected transition groups that start at the current topmost level. The updated transition group is then used in the next iteration of calling the function `applyFixPoint` . The pseudo code is given in Listing 4.3 and a visualization of this update process is presented in Figure 4.4.

```
// Storing MDD trees containing already visited states per transition group.
array<vset_t> group_explored

// Update transition group with given transition group ID and partial MDD tree.
void expand_group_next(int grpID, vset_t set) :
  context ctx := createContext(grpID, set)
  vset_t group_tmp := vset_project(set)
  vset_t old_group_explored := group_explored[grpID]
  group_explored[grpID] := vset_union(old_group_explored, group_tmp)
  group_tmp := vset_minus(group_tmp, old_group_explored)
  // Symbol * indicates a function pointer.
  vset_enum(group_tmp, * exploreModelAndUpdate, ctx)
```
Listing 4.3: Pseudo code of update process for updating a single transition group in implementation of General On-the-fly Saturation algorithm.

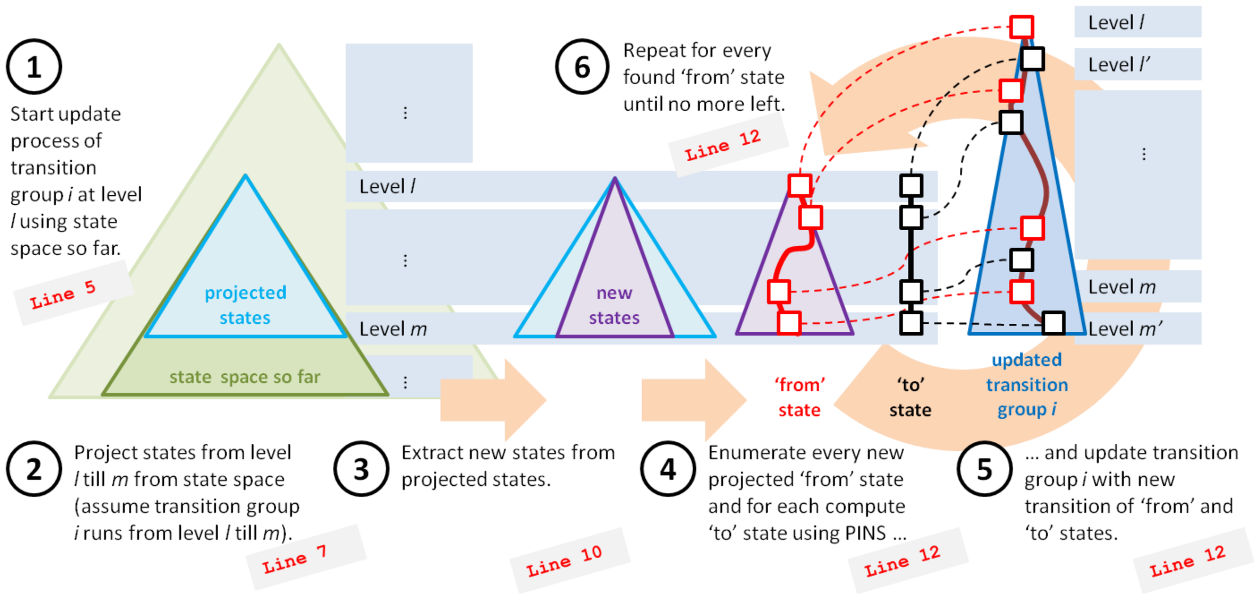

1
Start update process of transition group *i* at level *l* using state space so far.
Line 5
projected states
state space so far
Level *l*
Level *m*
new states
'from' state
'to' state
updated transition group *i*
Level *l*
Level *l'*
Level *m*
Level *m'*
6
Repeat for every found 'from' state until no more left.
Line 12
2
Project states from level *l* till *m* from state space (assume transition group *i* runs from level *l* till *m*).
Line 7
3
Extract new states from projected states.
Line 10
4
Enumerate every new projected 'from' state and for each compute 'to' state using PINS …
Line 12
5
… and update transition group *i* with new transition of 'from' and 'to' states.
Line 12

Figure 4.4: Pictorial overview of update process for transition group *i* with topmost level *l* (line numbers in figure refer to corresponding line numbers in Listing 4.3).

In the update process described by G. Ciardo all affected MDD conjuncts are updated with this newly found local state value and after the update, one is certain that all partitioned transition relations know what (local) transitions are possible with the newly found local state value under consideration.
In the version of the update process introduced here, it is not always certain that after one iteration of calling the relational product inside the function `startFixPoint` (in Listing 4.1), all new (sub) state vectors are discovered which involve a certain newly discovered local state value at a certain level. This is because certain (sub) state vectors can only be discovered via other transitions or via other parts of the reachable state space, as shown in Figure 4.5.

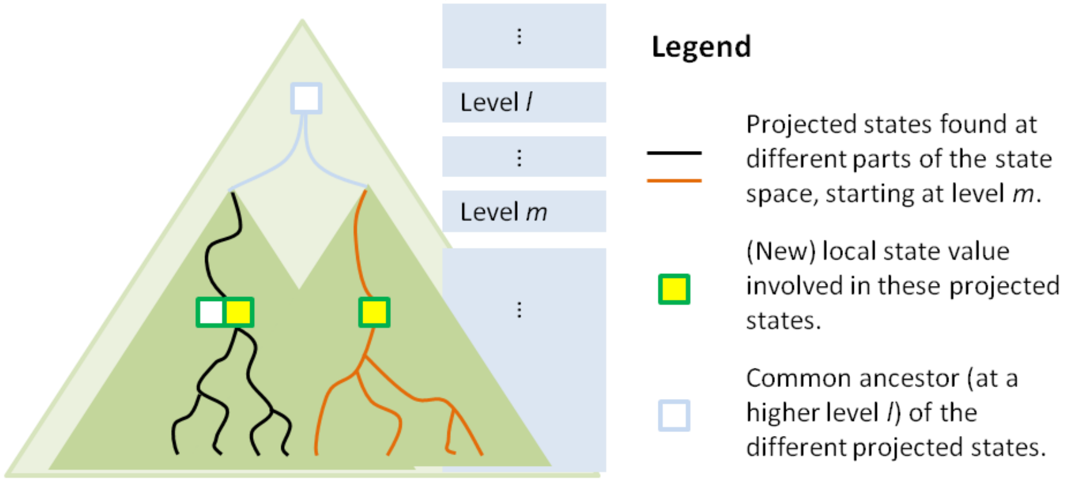

Level *l*
Level *m*
Legend
Projected states found at different parts of the state space, starting at level *m*.
(New) local state value involved in these projected states.
Common ancestor (at a higher level *l*) of the different projected states.

Figure 4.5: Different local state values located in different subtrees of reachable state space.

To work with General On-the-fly Saturation one needs to call the function `vset_least_fixpoint` in spec-reach.c with an MDD of the initial state, an empty set of transition groups and the total number of these transition groups. In the LTSmin toolset a new option will be introduced that will call the function `vset_least_fixpoint` once and is named sat-ciardo. Every time when it enters the while-loop in the function `startFixPoint` and begins to start this process for a certain transition group that starts at the current topmost level, the update process is initiated for the transition group before it heads into the function `applyFixPoint` and the recursive calls of the relational product.

4.3 Implementation of Saturation in LTSmin toolset

The design of Saturation for the LTSmin toolset as presented in the previous section is implemented in the tool, but in practice the presented design has to address some implementation-specific details as well. These issues will be discussed for the algorithm itself and for the LTSmin toolset in sections 4.3.1 and 4.3.2, respectively. When dealing with issues inside MDD libraries, the focus lies on the ATermDD library.

4.3.1 Implementation-specific adjustments for Saturation

In addition to the design of the Saturation algorithm, some issues related to the LTSmin toolset and especially inside the ATermDD library need to be addressed to be able to let the algorithm function properly.

In the LTSmin toolset the level numbers of an MDD tree are traversed in reverse order: the level with the root node is considered as level 0 and when traversing down the MDD tree, the level number increases until the terminal nodes, which occurs at level number $\mathcal{K}+1$. The algorithm can be easily adjusted to cope with this issue, simply by starting the Saturation process at level number 0 and every time when the level number is decreased by 1 in the original algorithm, one can change it into an increase by 1.

Another issue for consideration is the usage of the global lookup table for storing intermediate state space results during computation of the relational product. This lookup table uses an (MDD tree, MDD tree)-tuple as key and a single MDD tree as value. Storing a tuple of MDD trees as key is not possible, but this issue can be resolved quite easily by constructing a new data object containing the level number, the transition group ID, and furthermore two links pointing to the two separate MDD trees (see Figure 4.6 below). The newly constructed MDD tree will now function as the key for the lookup table.
In this case it is important to store both the level number and transition group ID in an MDD node, because one of the MDD trees being stored as part of the composite key is the MDD tree for parts of a particular transition group. Since the MDD encodings use fully-reduced MDDs for the transition groups, this can cause mismatches in the lookup table when for another event and same level number, a value is retrieved. When only using the level number in the composite key, there is the possibility that a match is found in the lookup table because of similar MDD parts for different transition groups (and causes the algorithm to think it has computed this partial result before). This issue is visualized in Figure 4.7.
Further consequences of using the transition group ID for the lookup table belonging to the relational product computations, is that the transition group ID itself needs to be known during these computations. Unfortunately the available functions in the MDD library do not keep up which transition group ID is currently being used and this has been resolved by introducing an extra parameter to the affected functions to keep a reference to the transition group ID (e.g. parameter `grp` in function `startRelProd`).

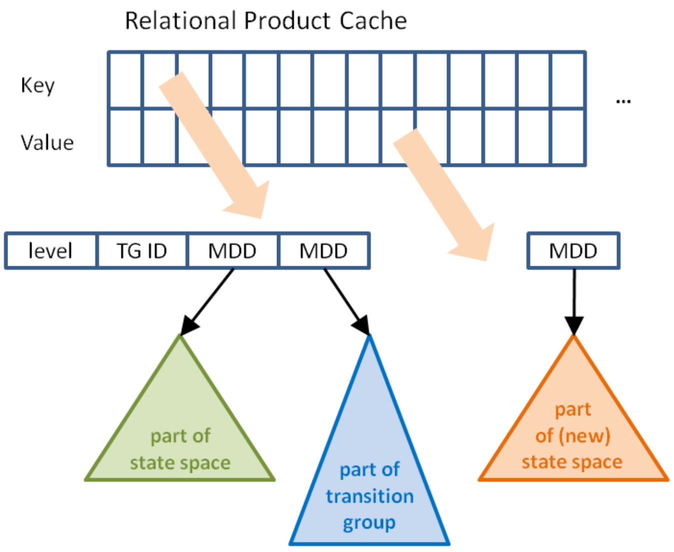
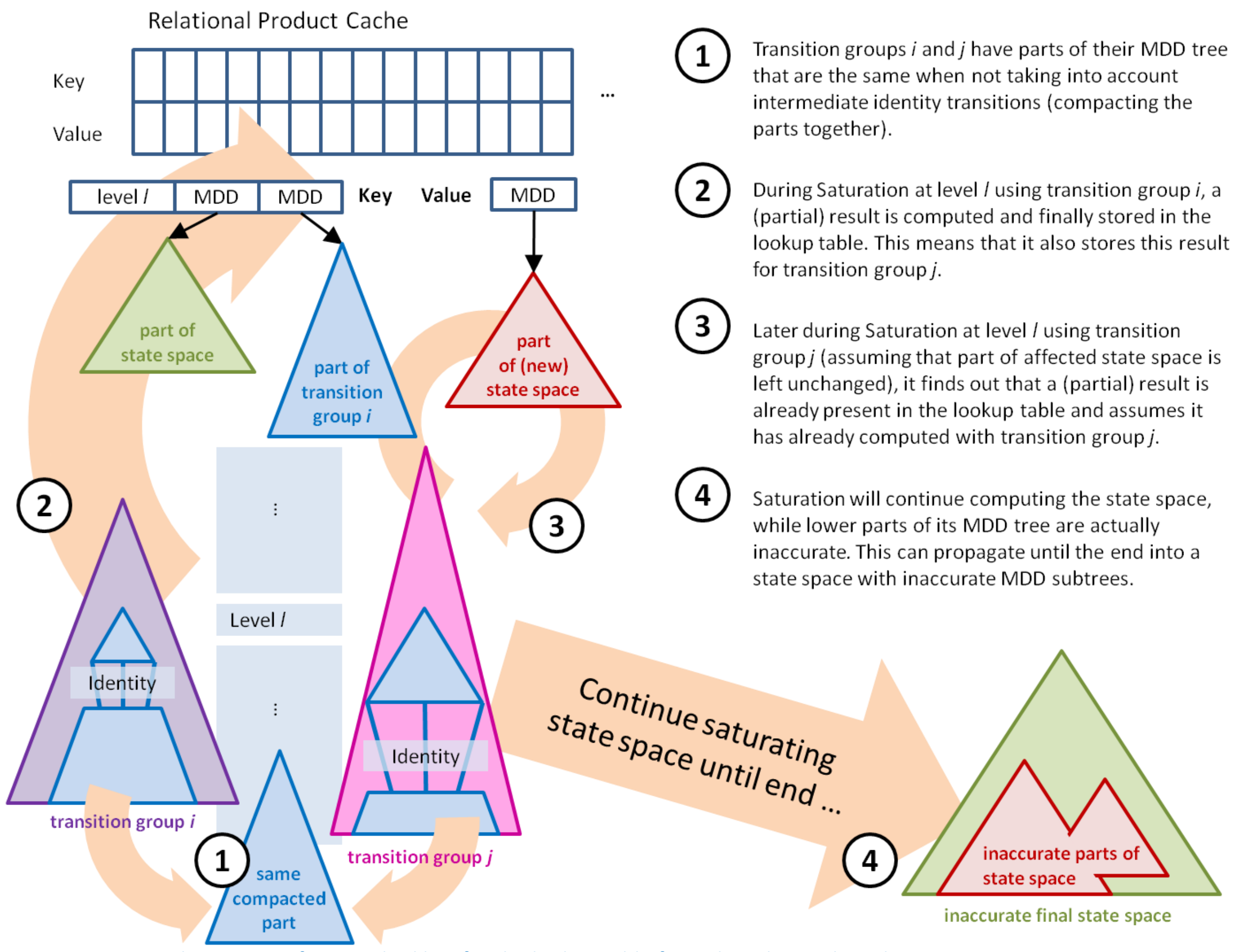


Figure 4.6: Visualization of the key and value used in the lookup table from the relational product.



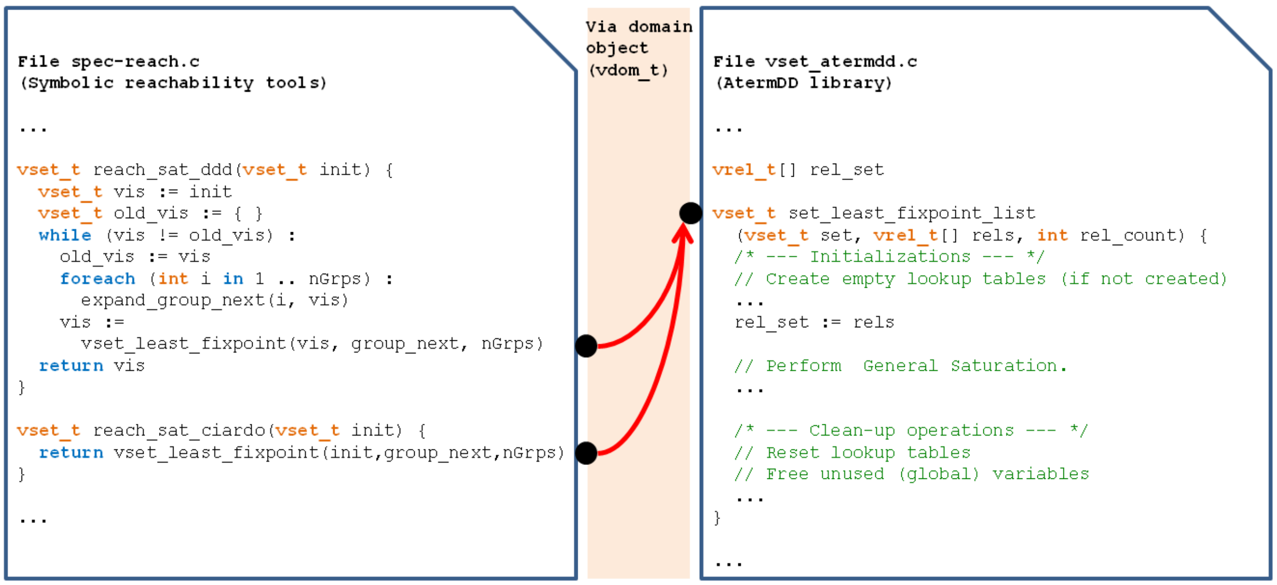
Relational Product Cache
Key
Value
...
level *l*
MDD
MDD
Key
Value
MDD
part of state space
part of transition group *i*
part of (new) state space
2
3
:
Level *l*
:
Identity
Identity
transition group *i*
transition group *j*
1
same compacted part
Continue saturating state space until end ...
4
inaccurate parts of state space
inaccurate final state space
1
Transition groups *i* and *j* have parts of their MDD tree that are the same when not taking into account intermediate identity transitions (compacting the parts together).
2
During Saturation at level *l* using transition group *i*, a (partial) result is computed and finally stored in the lookup table. This means that it also stores this result for transition group *j*.
3
Later during Saturation at level *l* using transition group *j* (assuming that part of affected state space is left unchanged), it finds out that a (partial) result is already present in the lookup table and assumes it has already computed with transition group *j*.
4
Saturation will continue computing the state space, while lower parts of its MDD tree are actually inaccurate. This can propagate until the end into a state space with inaccurate MDD subtrees.

Figure 4.7: Pictorial overview of using a bad key for the lookup table from the relational product.

When using the lookup tables as they are now, for the General Prebuilt Saturation algorithm inside the option sat-ddd it is important that the lookup tables are reset when the Saturation loop (in `vset_least_fixpoint` in spec-reach.c) is re-entered. This is due to the chance that parts of the state space so far have already been stored in the lookup table (and also due to the usage of incomplete keys). The algorithm thinks it has encountered a certain part of the state space before and will not revisit it, although its transition groups may have been updated with new transitions (outside the Saturation loop).

Finally some initialization and clean-up operations need to be added before and after the Saturation process in the MDD library. For the initialization a number of global lookup tables need to be created which are empty at the beginning. Also references to the transition groups that are passed to the function `vset_least_fixpoint` (in spec-reach.c) need to be stored and hereby a global variable is used. It is possible to pass it as a parameter when calling the functions related to the Saturation algorithm and use it inside the function performing the start of the fix-point computation, but this does not make the functions better readable.
And after the Saturation process is finished, some clean-up operations are performed by freeing allocated memory (which does not happen automatically in C).

A summary of the initialization and clean-up operations inside the MDD library is depicted using pseudo code in Figure 4.8. The functions `reach_sat_ddd` and `reach_sat_ciardo` represent the concrete implementation of options sat-ddd and sat-ciardo, respectively (more about this in the next section).


```
File spec-reach.c
(Symbolic reachability tools)

...

vset_t reach_sat_ddd(vset_t init) {
  vset_t vis := init
  vset_t old_vis := { }
  while (vis != old_vis) :
    old_vis := vis
    foreach (int i in 1 .. nGrps) :
      expand_group_next(i, vis)
    vis :=
      vset_least_fixpoint(vis, group_next, nGrps)
  return vis
}

vset_t reach_sat_ciardo(vset_t init) {
  return vset_least_fixpoint(init,group_next,nGrps)
}

...
```

```
File vset_atermdd.c
(AtermDD library)

...

vrel_t[] rel_set

vset_t set_least_fixpoint_list
  (vset_t set, vrel_t[] rels, int rel_count) {
  /* --- Initializations --- */
  // Create empty lookup tables (if not created)
  ...
  rel_set := rels

  // Perform  General Saturation.
  ...

  /* --- Clean-up operations --- */
  // Reset lookup tables
  // Free unused (global) variables
  ...
}

...
```
Figure 4.8: Passing on calls of vset_least_fixpoint in spec-reach.c to atermdd.c.

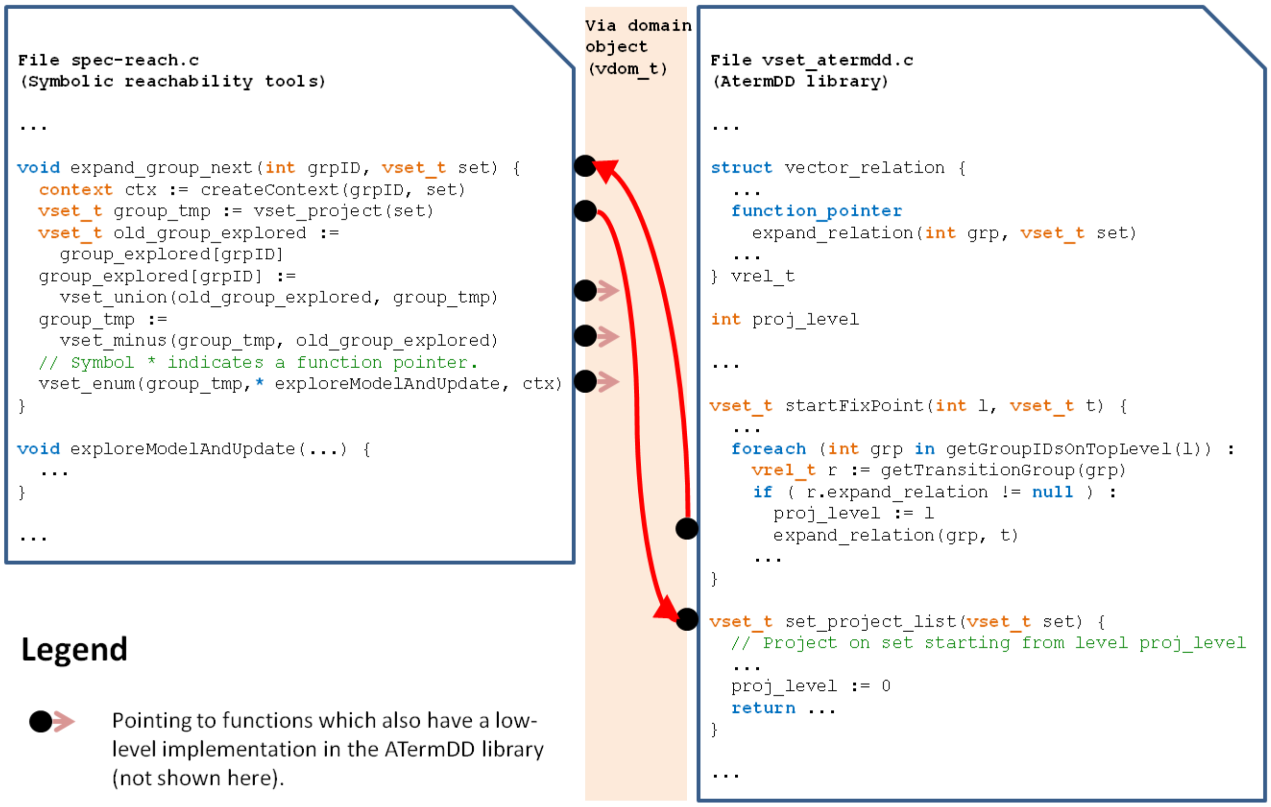
4.3.2 Implementation-specific adjustments for LTSmin toolset

The LTSmin toolset itself cannot cope with the Saturation algorithm unless some additional implementation-specific issues are resolved.

First of all a new reachability option should be introduced in spec-reach.c and for this purpose the option is given the name sat-ciardo. The available reachability options of the LTSmin toolset are extended with this new option to make sure it considers the option as legitimate when the user provides it.
Also a new high-level reachability function is added in spec-reach.c, named `reach_sat_ciardo`. Basically it has a simplified version of the function `reach_sat_ddd`, in which it calls the function `vset_least_fixpoint` only once, because the update process of the transition groups is handled within this function (also see Figure 4.8).

Another major issue is dealing with the update process of the transition groups. At a high level in spec-reach.c there already exists a function which is capable of updating the transition groups, namely `expand_group_next`. Unfortunately this function can only handle MDD sets in which all levels are present, but this is not the case when using the Saturation process in which transition groups are updated using the latest MDD tree that can start at a certain intermediate level. To be able to reuse the function `expand_group_next` and leave it intact as it is, some modifications have been made in the projection function inside the MDD library itself. An additional global variable is introduced indicating at which MDD level to start projecting and this variable is set at the point just before the `expand_group_next` function is called (this would happen between lines 27 and 28 in Listing 4.1, which is not shown in the pseudo code). When the projection function inside `expand_group_next` is called, it will use the set global variable to start projecting from a certain MDD level. This global variable is reset to 0 (indicating top level of MDD tree) at the end of the projection function to prevent miscomputations with the projection later on. The advantage of using such a construct is that spec-reach.c is left unchanged. But a disadvantage is that between the call to `expand_group_next` and the first call to the projection function, no calls to this projection function can be made. Currently this is not the case in `expand_group_next`, but in future modifications of this function it may become a problem if it involves the projection function.

To be able to call the function `expand_group_next` from within Saturation inside the MDD library, a function pointer to this function type is added in the vector relation structure. Together with this adjustment, a new vector set operation called `vrel_set_expand_relation` is introduced which initializes this function pointer for a certain transition group (for which also the vector domain is adjusted). The initialization of this function pointer occurs during the initialization phase of calling the reachability tool (in spec-reach.c).
Storage of the function pointer per transition group creates the possibility of storing different types of update functions per transition group, allowing more flexibility in future extensions of the tool.
Also since both option sat-ddd and sat-ciardo use the same Saturation algorithm implemented in the MDD library, additional checks inside the Saturation algorithm are inserted that check if the aforementioned function pointer for the transition groups is set. If this is the case, then the algorithm knows that option sat-ciardo is currently performing (where transition groups are updated within Saturation) and otherwise it is certain that option sat-ddd is running (where transition groups are updated outside Saturation).
A visualization of the discussed issues using pseudo code is depicted in Figure 4.9.


```
File spec-reach.c
(Symbolic reachability tools)

...

void expand_group_next(int grpID, vset_t set) {
  context ctx := createContext(grpID, set)
  vset_t group_tmp := vset_project(set)
  vset_t old_group_explored :=
    group_explored[grpID]
  group_explored[grpID] :=
    vset_union(old_group_explored, group_tmp)
  group_tmp :=
    vset_minus(group_tmp, old_group_explored)
  // Symbol * indicates a function pointer.
  vset_enum(group_tmp,* exploreModelAndUpdate, ctx)
}

void exploreModelAndUpdate(...) {
  ...
}

...
```

```
File vset_atermdd.c
(AtermDD library)

...

struct vector_relation {
  ...
  function_pointer
    expand_relation(int grp, vset_t set)
  ...
} vrel_t

int proj_level

...

vset_t startFixPoint(int l, vset_t t) {
  ...
  foreach (int grp in getGroupIDsOnTopLevel(l)) :
    vrel_t r := getTransitionGroup(grp)
    if ( r.expand_relation != null ) :
      proj_level := l
      expand_relation(grp, t)
    ...
}

vset_t set_project_list(vset_t set) {
  // Project on set starting from level proj_level
  ...
  proj_level := 0
  return ...
}

...
```
Via domain object (vdom_t)
Legend
Pointing to functions which also have a low-level implementation in the ATermDD library (not shown here).

Figure 4.9: Passing on calls related to update process of transition groups between spec-reach.c and atermdd.c.

Most low-level MDD operations in the ATermDD library depend on a single global lookup table for storing intermediate calculated results. This global lookup table is immediately reset after performing one of such MDD operations to prevent causing mismatches when using other MDD operations that rely on the same global lookup table as well. One such MDD operation is the union operator for taking the union of two MDDs (at a certain level). For General Prebuilt Saturation the usage of the global lookup table is not interrupted by other MDD operations which use this lookup table too. But for General On-the-fly Saturation this does happen, due to the usage of the update process of the transition groups, where for instance the projection operator comes into play. To cope with this issue, the union operator has been copied and renamed and this copy is only used by the Saturation process. Inside the copy of the union operator, it uses a newly introduced lookup table, especially for the purpose of storing intermediate calculated results obtained during the union operation. Although this solution is far from perfect (due to copying a function), this avoids tampering with the original union operator which is called by other functions in the MDD library.

4.4 Summary of design & implementation of Saturation

This chapter provides answers to the first two research questions of the project, concerning the Saturation approach to choose for implementation and the consequences for implementing it inside the LTSmin toolset.

"Different versions of the Saturation approach exist and which one of these is a good candidate for extending the symbolic reachability techniques of the LTSmin toolset?"

The deciding factors for choosing a suitable Saturation algorithm to implement are the requirements that indicate that no restrictions should be imposed on the structure of the model and/or its transition relation and that the transition relation should not be known in advance. It is obvious that the Kronecker Saturation algorithms do not fit into this picture, since they require that the transition relation is partitioned into a Kronecker matrix form. From the General Prebuilt and General On-the-fly Saturation algorithms the former still needs to be able to know the transition relation of the model beforehand, so the only choice left is the General On-the-fly Saturation algorithm.

"How can the chosen version of the Saturation approach be implemented in the LTSmin toolset and what are the consequences for the tool's architecture?"

The General On-the-fly Saturation algorithm does have certain requirements before it can function properly in the LTSmin toolset. The most important ones are the ability to access low-level MDD operations and the MDD levels when traversing the MDDs of the transition groups. And related to the use of MDDs is its necessity to store computed parts of the MDD tree of the reachable state space. Although there is the possibility to implement the entire structure of the Saturation algorithm on a high level (e.g. in spec-reach.c), the choice is made to implement the algorithm in the MDD library itself. This prevents introducing extra high-level access to MDD operations, which can cause much clutter in the code.
For now, the implementation is done inside MDD libraries with full-fledged implementations of MDD operations, and the initial focus lies on using ATermDD. This library contains MDD operations which can be easily reused within the framework of the Saturation algorithm.

Before the actual implementation it turns out that some issues need to be resolved related to the Saturation algorithm. From the original pseudo code it turns out that the algorithm makes use of in-place updates to update pointers that point to lower-level MDD nodes. In the LTSmin toolset no in-place updates are used and instead entirely new MDD nodes are constructed with the new node pointers. To incorporate this into the algorithm, an additional adjustment is introduced, namely replacing the iterative function calls with recursive function calls. This makes it possible to construct new MDD nodes at the level of the MDD node entries. Another advantage is a better time complexity. Fortunately the algorithm does not need to be redesigned from scratch with recursive function calls and parts of the algorithm's design are based on code already present in the MDD library. This code also takes into account the intermediate identity transitions that can occur in MDDs of the transition groups, since the MDD library makes use of fully-reduced MDDs.

For the implementation of the Saturation algorithm it turns out that the difference between the Prebuilt and On-the-fly version of the algorithm is small and therefore both versions are implemented in the LTSmin toolset.
The General Prebuilt Saturation algorithm cannot be used on its own (transition relation should be known in advance) and is now part of the option sat-ddd, which basically performs fix-point computations to compute the reachable state space interleaved with extension of the transition groups. The Saturation algorithm replaces the standard fix-point function in this reachability option and updates to the transition groups are performed outside the iteration of this fix-point computation.
For the General On-the-fly Saturation algorithm a new reachability option is added to the LTSmin toolset, named sat-ciardo. An issue to be dealt with here is that the update process of the transition groups as described by G. Ciardo [23] does not fit well inside the LTSmin architecture, since it requires additional changes. Therefore an own version of the update process of transition groups is devised, which makes use of the state space so far to update one of the affected transition groups that start at the current working MDD level.

Next to these design issues, during implementation some other issues occurred which need to be addressed and they are either related to the algorithmic structure of the Saturation algorithm or related to the LTSmin framework. The issues discussed also address the second research question.

Implementation-specific adjustments to the Saturation algorithm are not quite extensive and are easily dealt with. These involve handling the reverse numbering of the MDD levels used in the MDD library, and making use of another data object to be used as key for the lookup table involved in the relational product for purposes in regard to memoization.

Implementation-specific adjustments to the affected parts of the LTSmin toolset are also not too severe and mostly concern making changes to existing functionality. This is trivial for incorporating the new reachability option sat-ciardo as legitimate option and use the General Prebuilt Saturation algorithm as fix-point computation inside the option sat-ddd.
Furthermore the projection operation for projecting an MDD on a given projection list of MDD levels needs to be adjusted slightly, by making it possible to start the projection at a certain intermediate MDD level (and not only for the root MDD level as was the case). Also access to the function that performs the update process of the transition groups needs to be made available from inside the MDD library, which is done by adding an extra pointer reference to this function in the data object that encodes the transition group. Additionally, this pointer reference indicates if transition groups should be updated during Saturation (which is the case when the pointer reference is set).

Finally the usage of a single global lookup table in the ATermDD library for usage by different MDD operations to store intermediate calculated results is not a very good choice due to interference with what the General On-the-fly Saturation algorithm is doing. Although a crude solution has been given by copying the union operation of two MDDs, it is better to devise a more elaborate solution in the future.

5 Evaluation of Saturation in LTSmin toolset

With the implementation of the Saturation algorithm inside the LTSmin toolset, it is now possible to assess how the Saturation algorithms operate. Section 5.1 explains the setup of the experiments where different available reachability options are used and presents the results next to each other for comparison. Section 5.2 evaluates how the different reachability options perform and tries to explain why these perform as they do.

5.1 Experiments on Saturation in LTSmin toolset

The experimental setup for performing the experiments with the LTSmin toolset is the subject of the first part of this section, followed by the obtained experimental results.

5.1.1 Experimental setup using reachability tools from LTSmin

The implemented versions of the Saturation algorithms are compared with other available symbolic reachability options inside the LTSmin toolset. For testing purposes the reachability tool using the state-based specification language DVE is taken, for which a broad range of models are publicly available as part of the BEEM database [20]. The symbolic reachability tool using the DVE specification language comes in two versions:

- dve-reach tool: uses an interpreter to read the DiVinE models.
- dve2-reach tool: uses a compiled version of the DiVinE models (and pre-compiles DiVinE models if necessary).

For our purposes the dve2-reach tool is used, since using compiled versions of the DiVinE models works faster than interpreting them.

When calling a reachability tool a number of additional input arguments can be provided for which the user can specify a number of possibilities of how to perform the reachability analysis and under which conditions. For the experiments the following input arguments suffice:

- [--saturation]: the top-level saturation-like traversal technique to be used for generating the state space. For our purposes it suffices to look at the following possibilities: none, sat-like, sat-ddd and sat-ciardo. The first two options require a secondary auxiliary exploration technique to work with and the default option is none (no saturation-like strategy is used).
- [--order]: exploration strategy to be used as part of the saturation-like traversal technique that uses an auxiliary exploration technique. For our purposes a variation of breadth-first search (bfs-prev) and chaining (chain-prev) is used, since they only use the newest states in the next iteration of the exploration. Also with the set save-levels option (explained below) these are more likely to perform better in time than their counterparts bfs and chain. The default option is bfs-prev.
- [--save-levels]: saving discovered states which are found in the previous iteration of state space exploration. This option is only used in conjunction with the saturation option sat-like accompanied with order options bfs-prev or chain-prev, in which previously discovered states are not taken into account in next iterations of the exploration.
- [--regroup]: enabling reordering optimization on the PINS dependency matrix to improve variable ordering. For our purposes it suffices to take the option gs, which stands for Group Safely and is for most cases the most effective reordering option (for the full list of regrouping options, see [13]).

- [--vset]: type of vector set (or MDD data encoding) to be used for storing the reachable state space. Possible options are list, tree, fdd, ddd and ldd (as explained in Table 3.4). Since our main focus lies on design and implementation in the full-fledged MDD library ATermDD (the General Saturation algorithms are currently only implemented here) and especially the LDD data structure, option list is taken (which is also the default option).

For doing experiments it is important to keep the provided input arguments to the reachability tool consistent for all traversal techniques used. The input arguments provided to the dve2-reach tool are given in Table 5.1 (input argument [--vset] is not given, because the default option list is used for our experiments for which one does not have to explicitly provide it in the command). In the next few sections, the none-saturation options will be abbreviated to the no-sat options.
(Refer back to Listing 3.1 and Listing 3.2 for the pseudo code algorithms of the options sat-like and sat-ddd, respectively.)

Traversal technique description	**Input arguments used for dve2-reach tool**			
	--saturation=	--order=	--save-levels	--regroup=
Bread-first search with previously seen new states	none	bfs-prev		gs
Chaining with previously seen new states	none	chain-prev		gs
Saturation-like strategy with breadth-first search as auxiliary exploration technique	sat-like	bfs-prev	×	gs
Saturation-like strategy with chaining as auxiliary exploration technique	sat-like	chain-prev	×	gs
Fix-point computation strategy with implemented General Prebuilt Saturation	sat-ddd			gs
Implemented General On-the-fly Saturation	sat-ciardo			gs

Table 5.1: Input arguments used with dve2-reach tool for experiments.

The testing environment under which the experiments are performed, is a cluster machine from the FMT lab, with dual Intel E5335 CPUs and 24 GB RAM. By the time of testing, version 1.7 of the LTSmin toolset (with addition of Saturation) is used together with the tool memtime (version 1.3), which is capable of keeping track of total execution times of and total amounts of memory being used by software programs.

The models for the experiments are retrieved from the BEEM database [20] and a broad selection of 27 DiVinE models is taken into consideration with a state space of at least 10 million states (to challenge the capabilities of the traversal techniques). An overview of these DiVinE models and the size of their reachable state space is given in Table 5.2. When performing the experiments it is not always possible that the tool comes up with a result because of memory limitations or it simply takes too long to compute the reachable state space. The following limits are imposed when performing the experiments:

- Use a time limit of 9000 seconds (i.e. 2½ hours).
- No exact memory limit is imposed, but the available working memory of 24 GB on the cluster machine is considered as final memory limit.

The 27 DiVinE models chosen are only a subset of the DiVinE models found in the BEEM database. Lots of models have a state space with a size smaller than 10 million states and do not provide much effort for the traversal techniques, while there are also models which simply take too much time to come up with a result. From a first impression of the model's state space size as given by the BEEM database and estimated effort to come up with a result, these 27 models have been taken as starting point for the experiments.

Name	Abbrev.	Short description	Size of reachable state space
Communication protocols			
brp.5.dve	brp.5	Bounded retransmission protocol	17740267
brp.6.dve	brp.6		42728113
cambridge.7.dve	cam.7	Cambridge ring protocol	11465015
collision.4.dve	col.4	Collision avoidance protocol	41465543
firewire_link.5.dve	fir.5	Layer link protocol of the IEEE-1394	18553032
iprotocol.6.dve	ipr.6	Optimized sliding window protocol	41387484
iprotocol.7.dve	ipr.7		59794192
Mutual exclusion algorithms			
anderson.6.dve	and.6	Anderson's queue lock mutual exclusion algorithm	18206917
at.5.dve	at.5	Alur-Taubenfeld mutual exclusion algorithm	31999440
at.6.dve	at.6		160589600
bakery.6.dve	bak.6	Bakery mutual exclusion algorithm	11845035
lamport.7.dve	lam.7	Lamport's fast mutual exclusion algorithm	38717846
lamport.8.dve	lam.8		62669317
lamport_nonatomic.5.dve	lna.5	Lamport mutual exclusion protocol with non-atomic operations	95118524
lann.6.dve	lann.6	Lann leader election algorithm for token ring	144151628
lann.7.dve	lann.7		160025986
peterson.6.dve	pet.6	Peterson's mutual exclusion protocol for N processes	174495861
peterson.7.dve	pet.7		142471098
phils.6.dve	phi.6	Dining philosophers problem	14348906
phils.8.dve	phi.8		43046720
szymanski.5.dve	szy.5	Szymanski mutual exclusion protocol	79518740
Leader election algorithms			
leader_election.6.dve	le.6	Leader election algorithm on unidirectional ring	35777100
leader_filters.7.dve	lf.7	Leader election algorithm based on filters	26302351
Other protocols			
telephony.4.dve	tel.4	Telecommunication service	12291552
telephony.7.dve	tel.7		21960308
Planning & scheduling			
elevator_planning.2.dve	ele.2	Planning of elevator strategy under several constraints	11428767
schedule_world.3.dve	sch.3	Scheduling of machines for production	166649331

Table 5.2: DVE models used for experiments (with descriptions from BEEM database [20]).

Data collection	Unit	Data obtained from
Size of reachable state space	Number	LTSmin toolset
Final MDD nodes	Number	LTSmin toolset
Peak MDD nodes	Number	LTSmin toolset
Time stamp of intermediate state space count	Seconds (sec)	LTSmin toolset
State space size so far per intermediate state space count	Number	LTSmin toolset
MDD nodes so far per intermediate state space count	Number	LTSmin toolset
Reachability time (of algorithm only)	Seconds (sec)	LTSmin toolset
Total execution time (of reachability tool)	Seconds (sec)	Tool memtime
Maximum memory usage (of reachability tool)	KiloBytes (KB)	Tool memtime
Useful and useless MDD projections done	Number	LTSmin toolset
Time contribution of functions in option sat-ciardo	Percentage (%)	Tool GNU gprof

Table 5.3: Overview of data that is being collected in the experiments using reachability tools from LTSmin.

Table 5.3 gives an overview of the types of data being gathered and which tools are used in this regard.

The experiments are performed in a structured way in which the reachable state space is computed for the 27 DiVinE models. Each model is run 3 times for each of the 6 traversal techniques as given in Table 5.1. This is necessary due to small fluctuations in resulting time and memory values for which the median value is taken over the 3 runs. Hereby the original implementation of the options sat-ddd and sat-ciardo is used.
Also for obtaining data related to the reachable state space and the number of MDD nodes (also during the state space search), some functionality regarding statistical analysis needs to be incorporated into the code of the implemented options sat-ddd and sat-ciardo (this is already present for the other chosen reachability options), for which some quick fixes were introduced by reusing some statistical analysis functionality from spec-reach.c and calling them at the right locations during Saturation. This adjusted implementation of the options sat-ddd and sat-ciardo is only used to gather the additional data and therefore each model is also run again once with these adjusted options. Because the algorithms are deterministic, it is not necessary to run these tests 3 times.
For the number of MDD projections performed in each of the reachability options, some additional counters are introduced in the code (that affects each reachability option). Furthermore profiling is performed using GNU gprof (version 2.19), for which auxiliary data is gathered to see how certain functions in the reachability tool contribute to the total execution time (which is only performed for option sat-ciardo, as will be explained in section 5.2.1).
The test runs are performed on the cluster machine from the FMT lab and to perform these sequentially in an automatic way, a Bash script has been constructed that is provided to the cluster machine to perform the test runs in the background.

5.1.2 Experimental results of symbolic reachability algorithms from LTSmin

The raw data from the test runs contain much clutter and therefore the right data is extracted first and put in a Comma Separated Values (CSV) file for further data manipulation. The median of the time and memory values are taken, while the other data is not manipulated any further. For better visualizations and comparisons of the measurements, the results are put into plots.
In this section the main focus lies on presenting measurements regarding time, the peak and final MDD nodes and how the state space and its corresponding MDD size evolve during the reachability analysis. Furthermore this section will only give some observations that can be extracted from the results and explanations of these observations will be given in the next section.

The measurements regarding the maximum memory usage will not be presented prominently here and one is referred to appendix A for details. It turns out that the memory usage is highly dependent on the MDD library being used (in this case ATermDD library) and how intermediate results are stored, which can differ when using other MDD packages and/or techniques in regard to memoization. Still the next section will go briefly into the memory measurements.
A short summary of these results are as follows:

- The results reveal that both implemented options sat-ddd and sat-ciardo obtain a much higher maximum memory than the options no-sat and sat-like in the vast majority of the tested models. In such cases it can run to almost 200 times the amount of memory being used by the options no-sat or sat-like.
- Between the options sat-ddd and sat-ciardo the results reveal that the maximum memory usage seems to keep pace with each other for the vast majority of the tested models.

The results of the total execution times are given in Table 5.4, since this gives the user a sense how long a complete computation takes using a reachability tool with DVE models. For the majority of the models the reachability time is just a few seconds less than the total execution time and only for those few models where the total execution times do differ significantly from the reachability times, as is indicated in the table (with * and **). These are mainly caused by heavy computations during pre-processing of the tool, such as pre-compiling the DVE models and computing the PINS dependency matrix for large models (which will become clear in the next section). Additionally, Table 5.5 gives an overview how much each of the reachability options perform as best or as worst (with lowest or highest total execution time, respectively). From these results one can observe that options sat-like with chain-prev and sat-ddd are the ones that perform the best for one out of three models, and option bfs-prev is the one that performs worst for almost one out of two models. But for option sat-ciardo the results are quite mixed.

Model	**State space size**	**Total execution time (median values in sec)**					
		bfs-prev	**chain-prev**	**sat-like bfs-prev**	**sat-like chain-prev**	**sat-ddd**	**sat-ciardo**
and.6	18206917	35.42	94.07	**33.02**	33.03	143.92	**196.64**
at.5	31999440	99.75	79.66	66.57	62.26	**55.54**	**261.68**
at.6	160589600	460.18	295.73	563.32	438.96	**292.38**	**6937.73**
bak.6	11845035	259.35	103.69	195.41	**65.14**	1259.2	**2692.94**
brp.5	17740267	**260.04**	107.18	208.12	188.38	32.72	**29.12**
brp.6	42728113	**620.36**	231.38	503.09	411.86	125.19	**75.15**
cam.7	11465015	1699.6	823.41	1546.39	1420.16	**629.4**	**8989.21**
col.4	41465543	**514.4**	383.19	45.43	49.83	**3.1****	9.11
ele.2	11428767	**69.54**	32.52	1.7	1.7	1.6**	**1.5****
fir.5	18553032	**616.46***	175.3*	123.07*	108.27*	**85.06***	151.65*
ipr.6	41387484	**1694.09**	305.54	271.21	222.69	82.06	**15.51**
ipr.7	59794192	**2155.44**	632.58	453.37	378.02	293.4	**65.27**
lam.7	38717846	**397.22**	150.93	29.42	**21.72**	77.86	92.97
lam.8	62669317	**511.58**	426.25	69.44	**52.53**	189.83	225.76
lna.5	95118524	**81.04***	57.07*	17.41*	**13.21***	18.41*	42.24*
lann.6	144151628	2032.26	757.77	581.13	378.94	**124.08**	**2280.5**
lann.7	160025986	**316.68**	138.41	41.54	38.12	**4.8**	15.11
le.6	35777100	**496.47***	14.21*	12.61*	12.41*	16.51*	**9.21***
lf.7	26302351	218.42	83.16	82.85	**82.15**	**2464.58**	2379.49
pet.6	174495861	169.19	**293.12**	48.53	**23.71**	74.65	151.91
pet.7	142471098	851.89	344.26	347	**267.45**	1894.09	**3768.49**
phi.6	14348906	**1.9****	1.4**	1.1**	1**	**1****	1**
phi.8	43046720	**2.3****	1.6**	1.2**	1.1**	**1.1****	1.1**
sch.3	166649331	**274.75**	30.24	3	**2.3****	3.4	13.51
szy.5	79518740	534.79	239.08	117.77	**64.44**	1041.64	**1729.29**
tel.4	12291552	59.93	37.33	43.92	**21.91**	115.5	**239.85**
tel.7	21960308	150.25	89.61	36.92	**30.02**	269.58	**699.88**
Legend of symbols							
*	Given total execution times (for all options) is larger than reachability time with:						
	± 15 seconds for fir.5	± 6 seconds for lna.5	± 10 seconds for le.6				
**	Reachability time is 1.5 to 2 times smaller than given total execution time.						
red	Worst performing option for a certain model.						
green	Best performing option for a certain model.						

Table 5.4: Experimental time results using the reachability tools from LTSmin on the 27 DVE models.

	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
# times as fastest	0	0	1	11	9	6
# times as slowest	15	1	0	0	1	10

Table 5.5: Summary of the best and worst performing reachability options regarding time performance.

To give an impression how the newly implemented Saturation algorithms in the options sat-ddd and sat-ciardo perform, the time measurements have been plotted against one of the other options, namely the options no-sat and sat-like. First the time results of option sat-ciardo is plotted out against the options no-sat and sat-like in Figure 5.1 and Figure 5.2 respectively, followed by the time results of option sat-ddd against the options no-sat and sat-like in Figure 5.3 and Figure 5.4 respectively. Then option sat-ciardo is plotted against option sat-ddd in Figure 5.5.

In all these plots, the plot on the left gives a sense where every result is located in a larger picture. The plot in the upper right corner is a magnification of a small part of the leftmost plot and the plot on the lower right corner is on its turn a magnification of the plot on the upper right corner. One should be aware that with these magnifications the horizontal and vertical axes scale as well.
Furthermore a diagonal is added in these plots to show that the two reachability options under comparison have the same time result. When a time result falls beneath the diagonal line in Figure 5.1, Figure 5.2 and Figure 5.5, then this means that option sat-ciardo computes faster than the other reachability option. Likewise for time results in Figure 5.3 and Figure 5.4, in which case option sat-ddd is performing faster than the other reachability option.

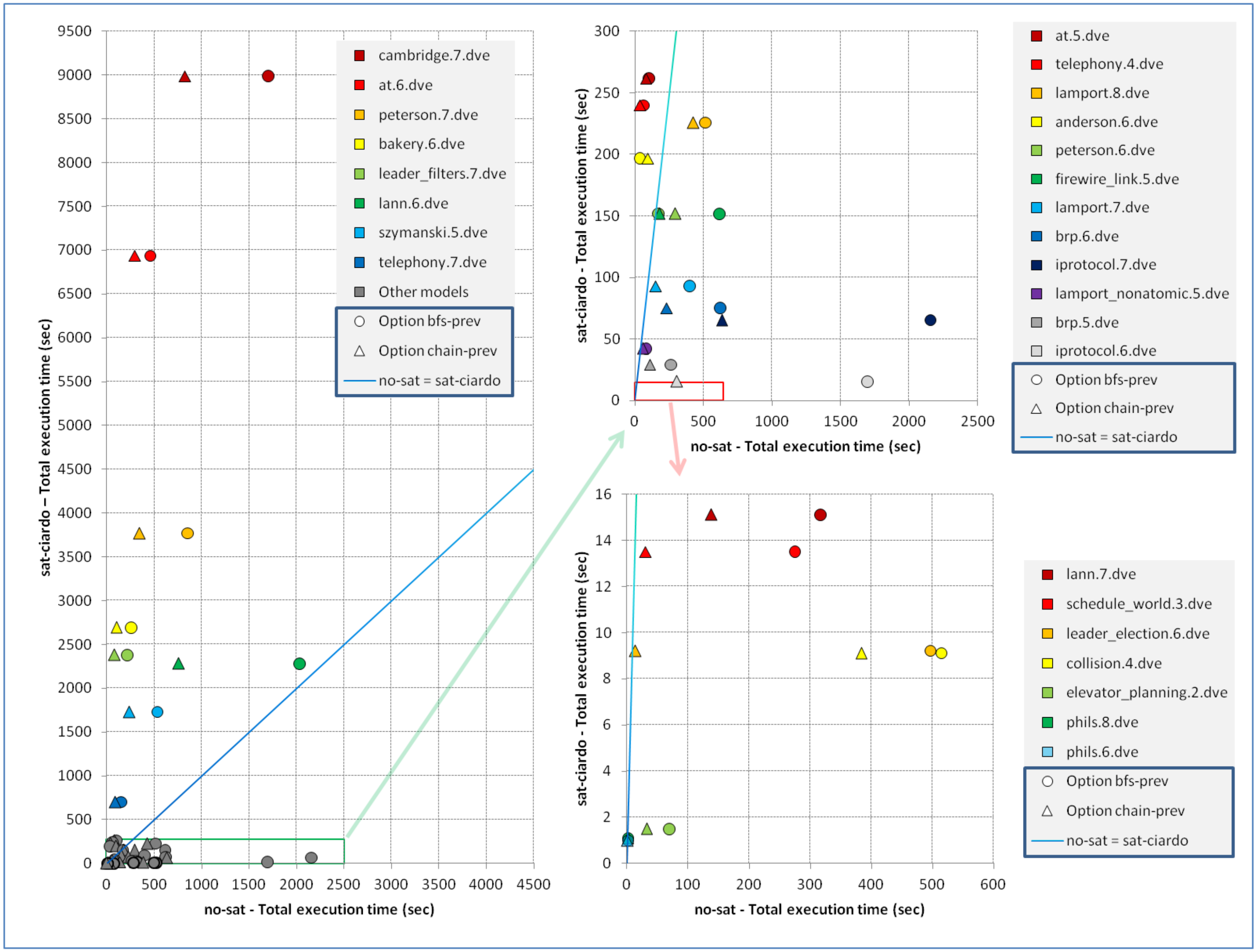


Figure 5.1: Plot comparing time values between options no-sat and sat-ciardo.

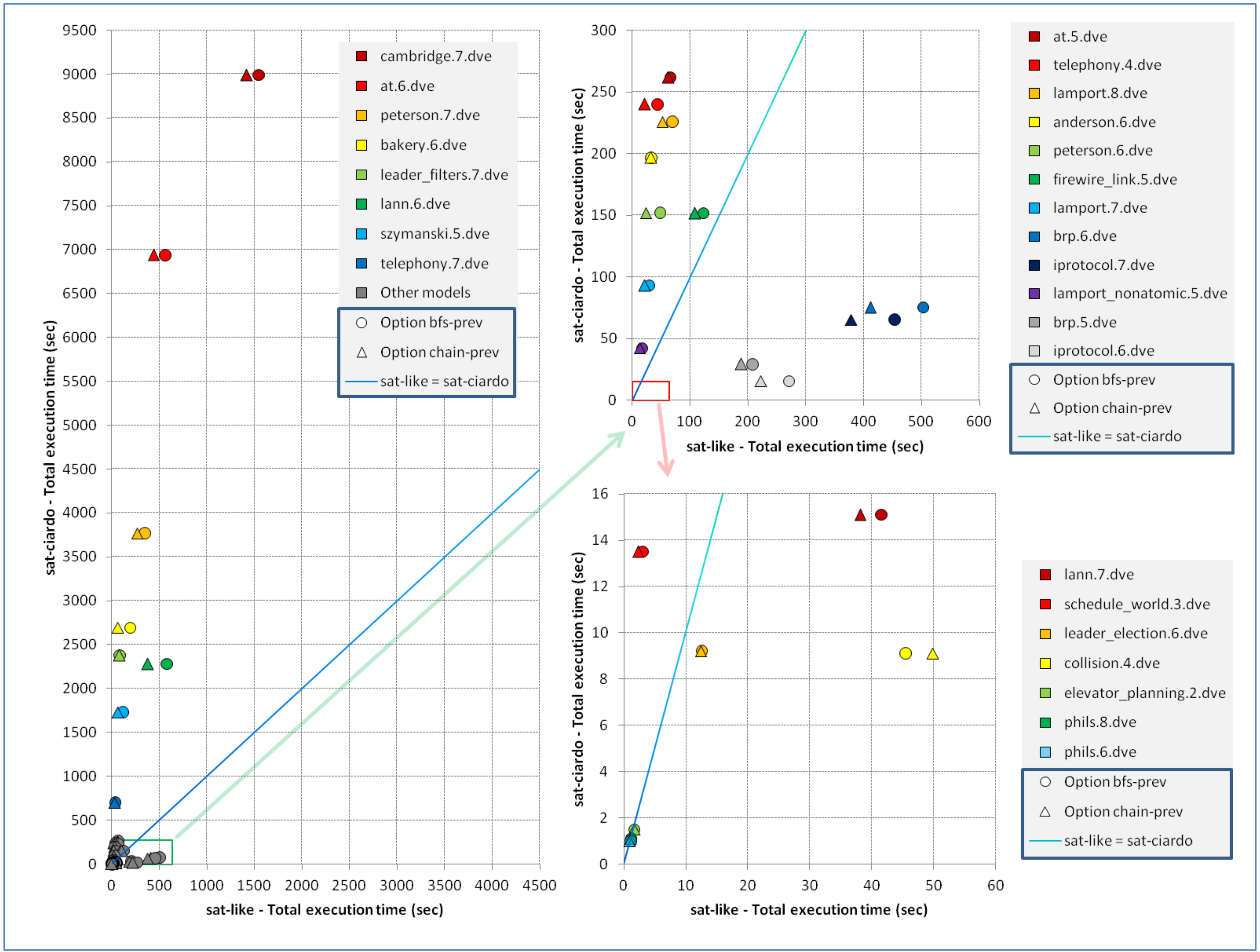


Figure 5.2: Plot comparing time values between options sat-like and sat-ciardo.

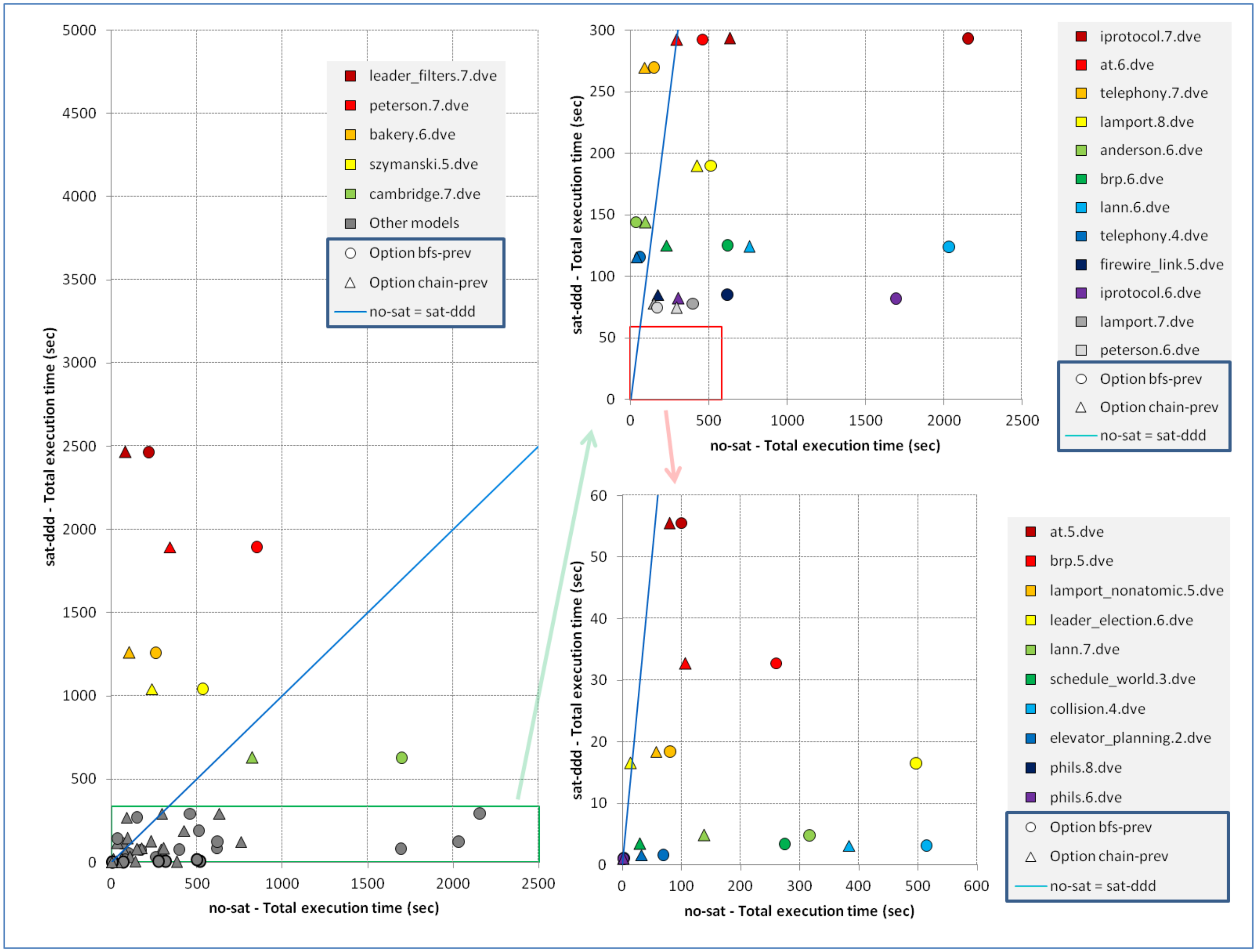


Figure 5.3: Plot comparing time values between options no-sat and sat-ddd.

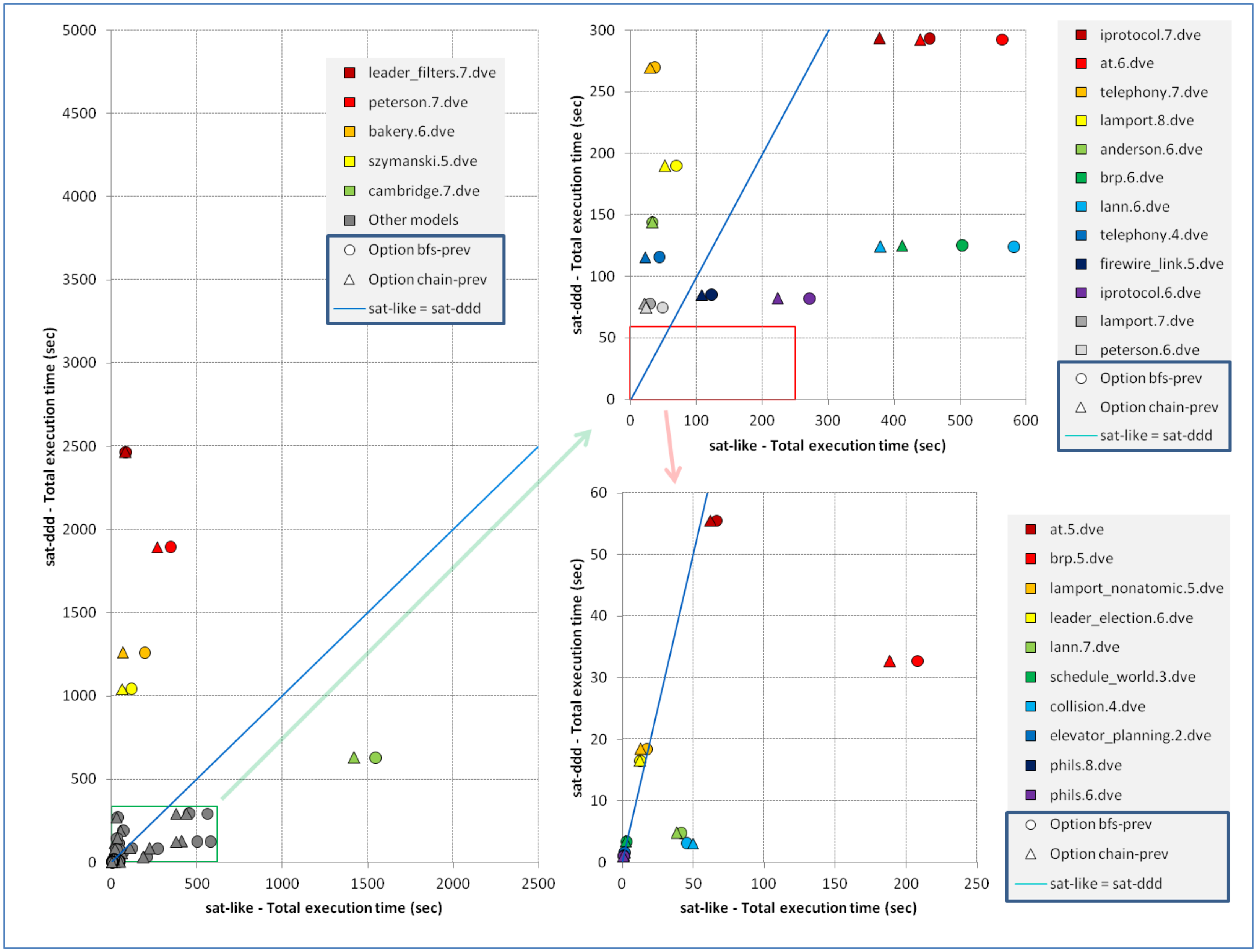


Figure 5.4: Plot comparing time values between options sat-like and sat-ddd.

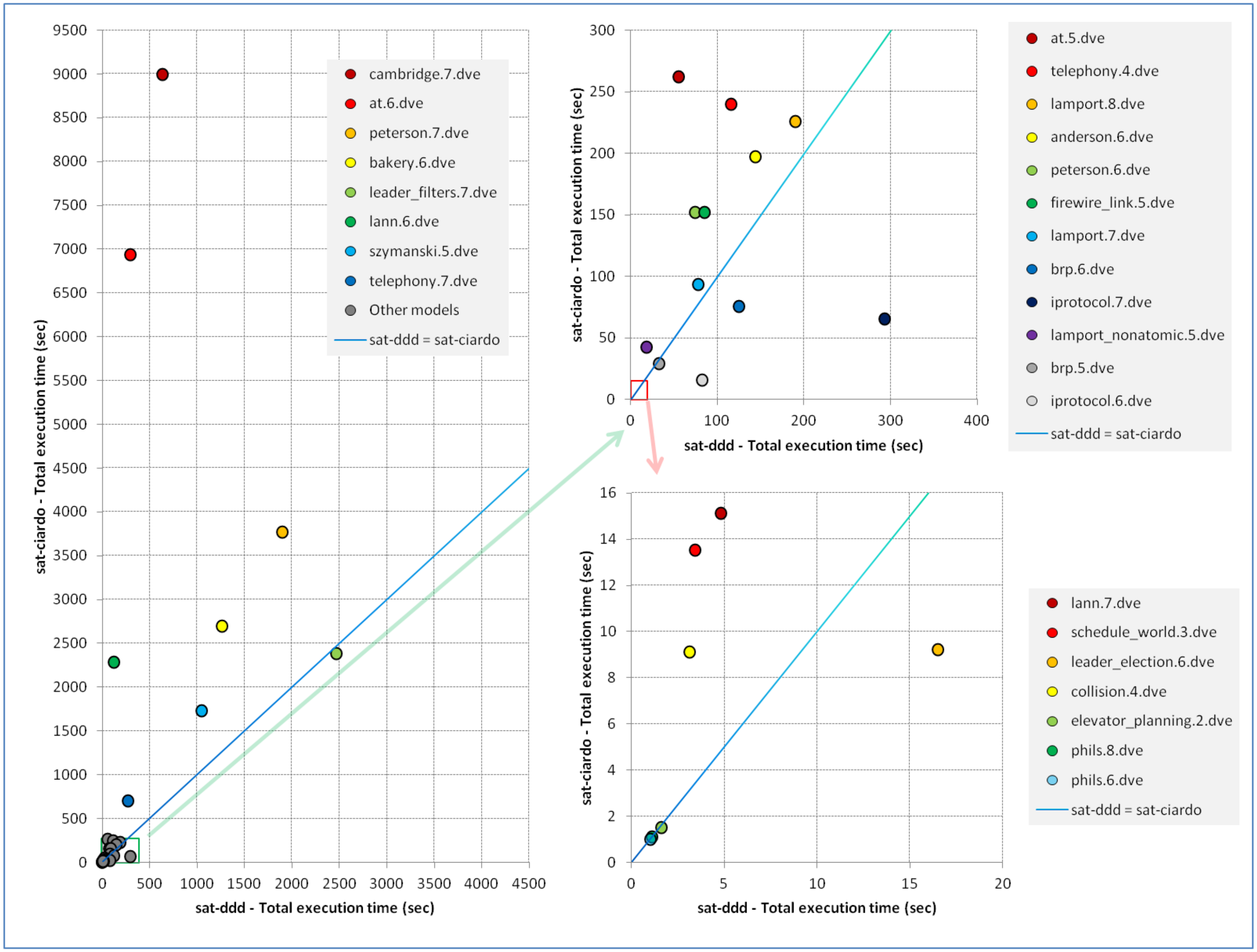


Figure 5.5: Plot comparing time values between options sat-ddd and sat-ciardo.

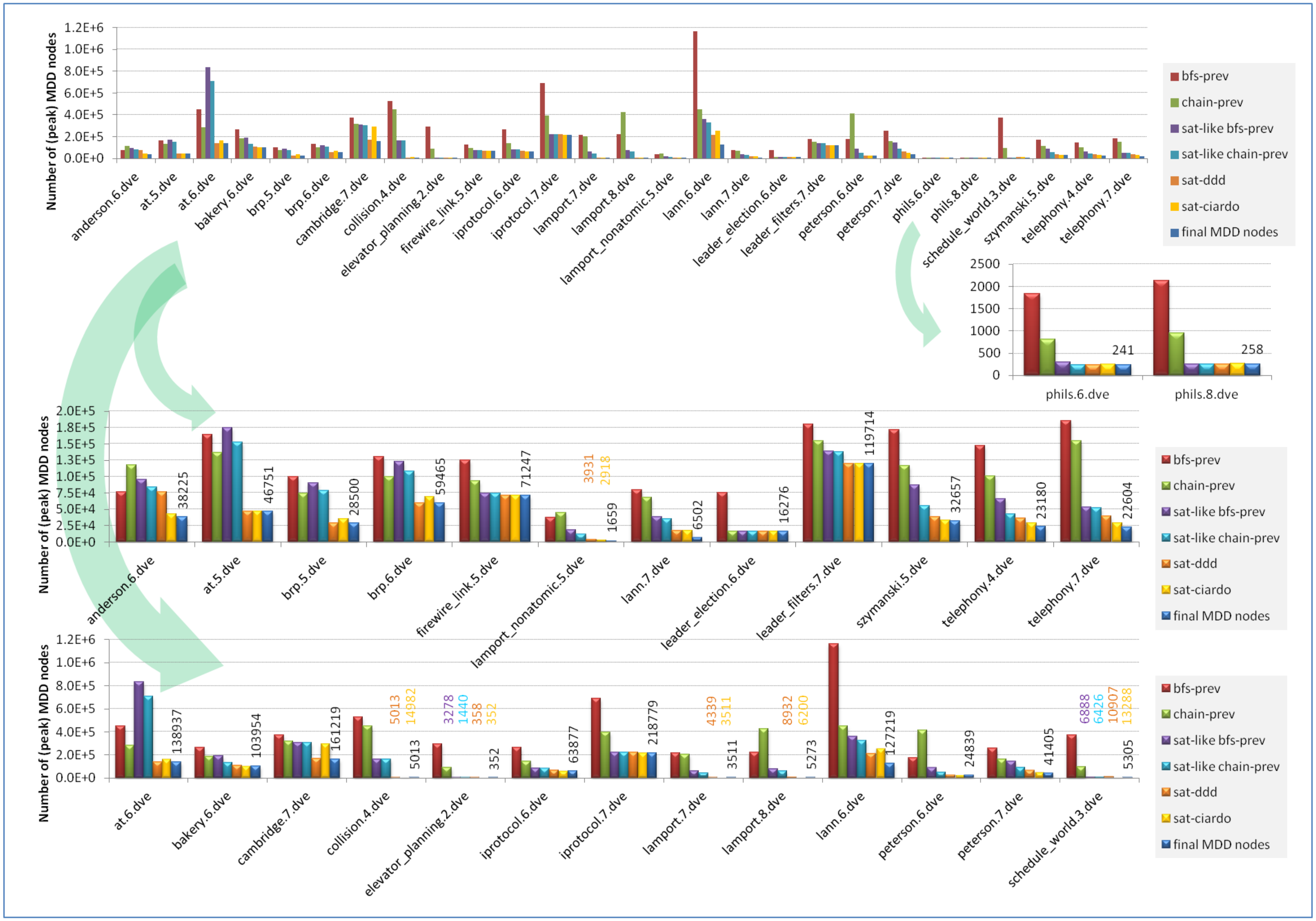


Figure 5.6: Bar charts of peak MDD node values for each reachability option per DVE model.

For the results of the peak MDD nodes, the results are plotted in bar charts (with the final MDD nodes given as the rightmost column per model) and this provides a better overview of how the different reachability options perform next to each other, as shown in Figure 5.6. To give an impression how the reachable state space and the size of the MDD evolve over time, the evolution plots of a selection of DVE models are given for each reachability option in Figure 5.9 until Figure 5.11. One should be aware that the horizontal time axis differs per reachability option and in these plots the reachability times are shown (not the total execution times). In appendix B the evolution plots of the other DVE models can be found.

Counting the MDD nodes and the state space size happens at the start of each main loop of the reachability algorithms no-sat, sat-like and sat-ddd. But for the option sat-ciardo this counting process is slightly different and this is due to how this algorithm builds the MDD. The MDD up to a certain level is built and levels above this level until the root level do not contain any processed MDD nodes. To account for this fact, the counting process is done at the point when the algorithm is starting to saturate by (re)entering the topmost working MDD level and this is visualized in Figure 5.7. Hereby the count of the state space size actually counts the number of substates found in the MDD starting at the current topmost working level.
For option sat-ddd the counting process takes place when it starts saturating by (re)entering root level 0 of the MDD tree. This is due to re-entering the Saturation algorithm a number of times (after updating the transition groups), for which all MDD levels contain processed MDD nodes. A visualization of this process is shown in Figure 5.8.

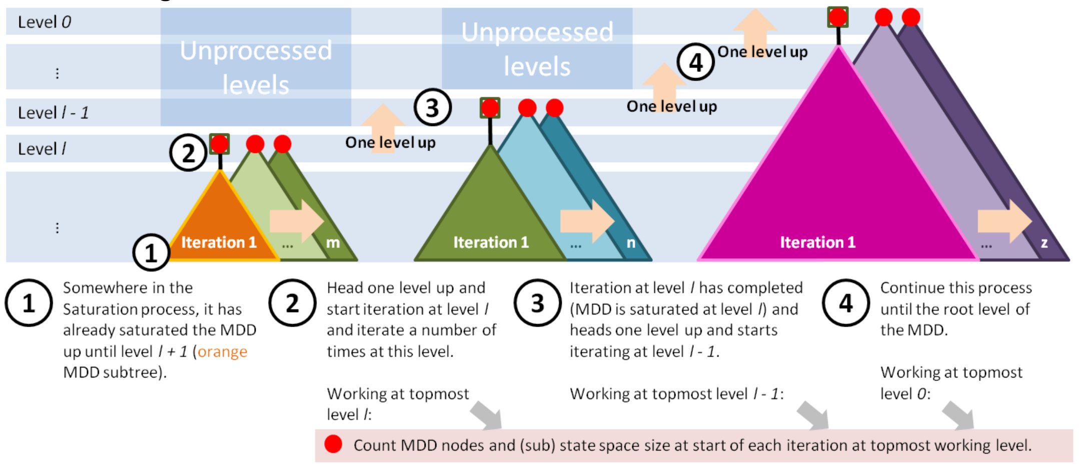

Level 0
⋮
Level l - 1
Level l
⋮
Unprocessed levels
Unprocessed levels
Iteration 1
...
m
One level up
Iteration 1
...
n
One level up
One level up
Iteration 1
...
z
1
2
3
4
Somewhere in the Saturation process, it has already saturated the MDD up until level l + 1 (orange MDD subtree).
Head one level up and start iteration at level l and iterate a number of times at this level.
Iteration at level l has completed (MDD is saturated at level l) and heads one level up and starts iterating at level l - 1.
Continue this process until the root level of the MDD.
Working at topmost level l:
Working at topmost level l - 1:
Working at topmost level 0:
● Count MDD nodes and (sub) state space size at start of each iteration at topmost working level.

Figure 5.7: Visualization of state and MDD count process used in option sat-ciardo.

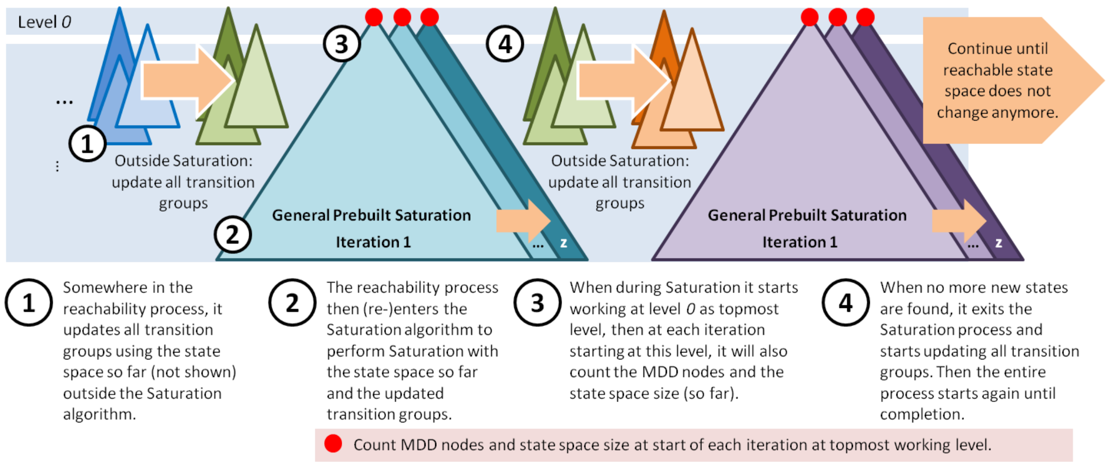
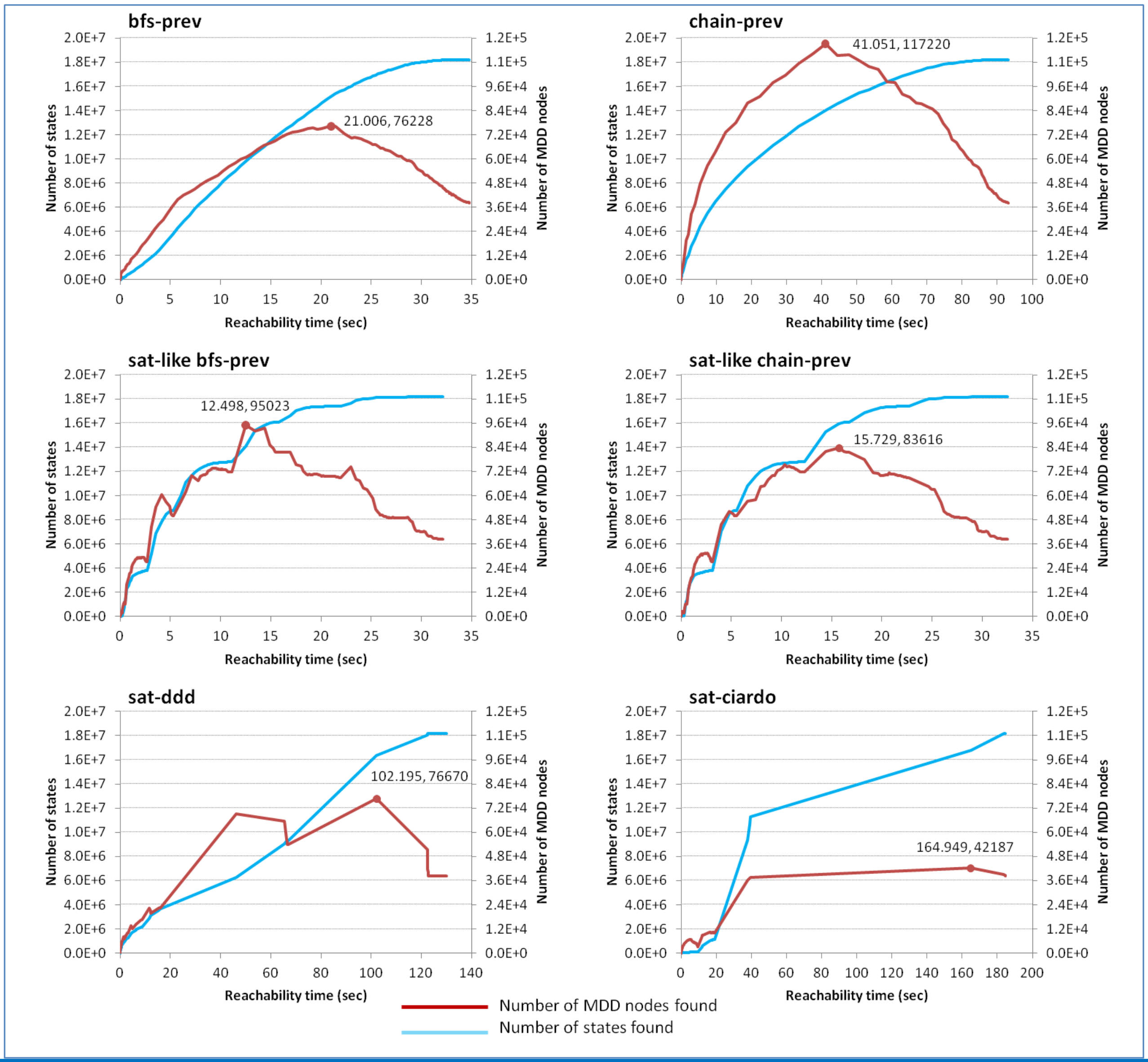
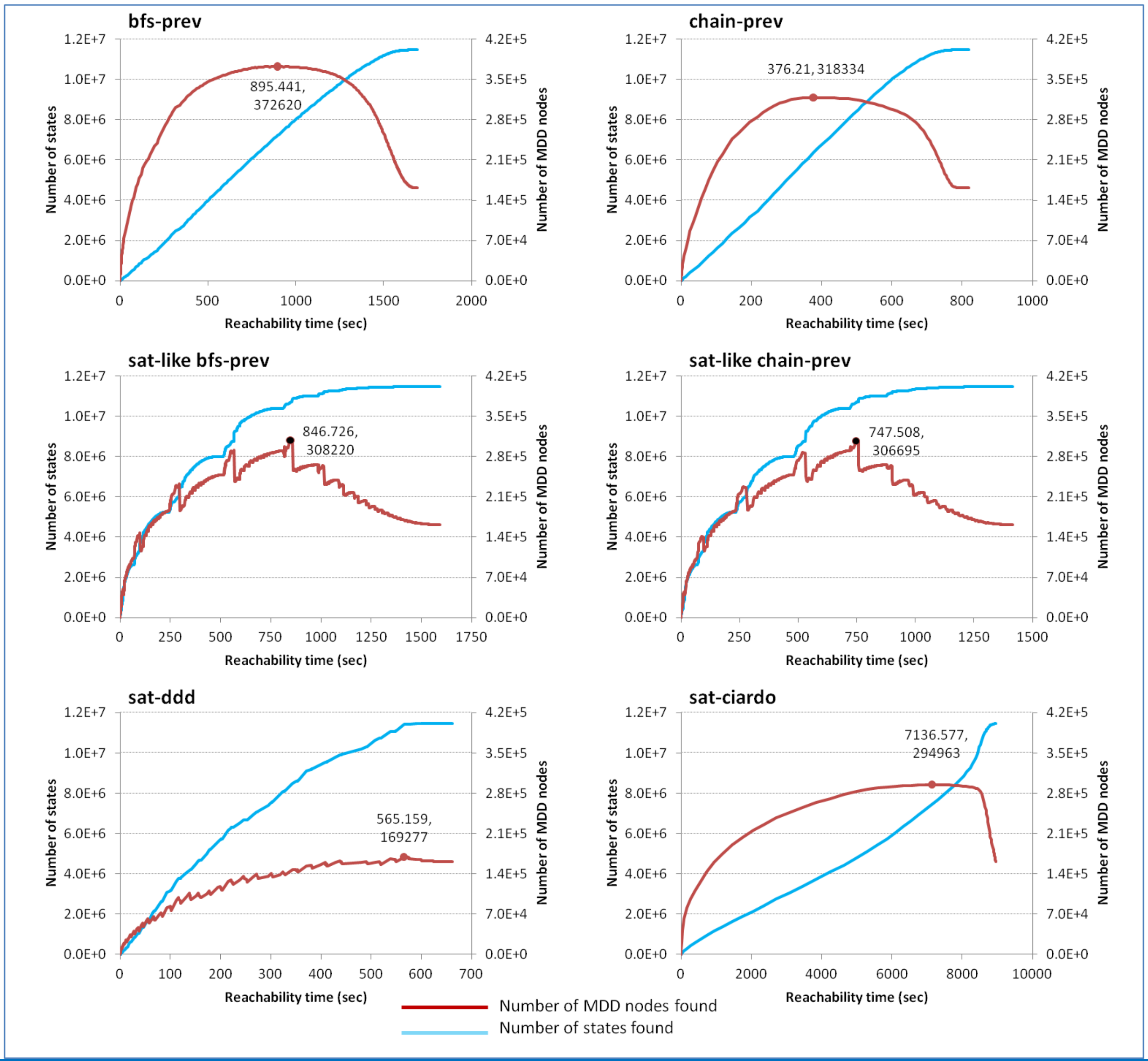

Level 0
...
⋮
1
Outside Saturation: update all transition groups
2
3
General Prebuilt Saturation
Iteration 1
...
z
4
Outside Saturation: update all transition groups
General Prebuilt Saturation
Iteration 1
...
z
Continue until reachable state space does not change anymore.
Somewhere in the reachability process, it updates all transition groups using the state space so far (not shown) outside the Saturation algorithm.
The reachability process then (re-)enters the Saturation algorithm to perform Saturation with the state space so far and the updated transition groups.
When during Saturation it starts working at level 0 as topmost level, then at each iteration starting at this level, it will also count the MDD nodes and the state space size (so far).
When no more new states are found, it exits the Saturation process and starts updating all transition groups. Then the entire process starts again until completion.
● Count MDD nodes and state space size at start of each iteration at topmost working level.

Figure 5.8: Visualization of state and MDD count process used in option sat-ddd.



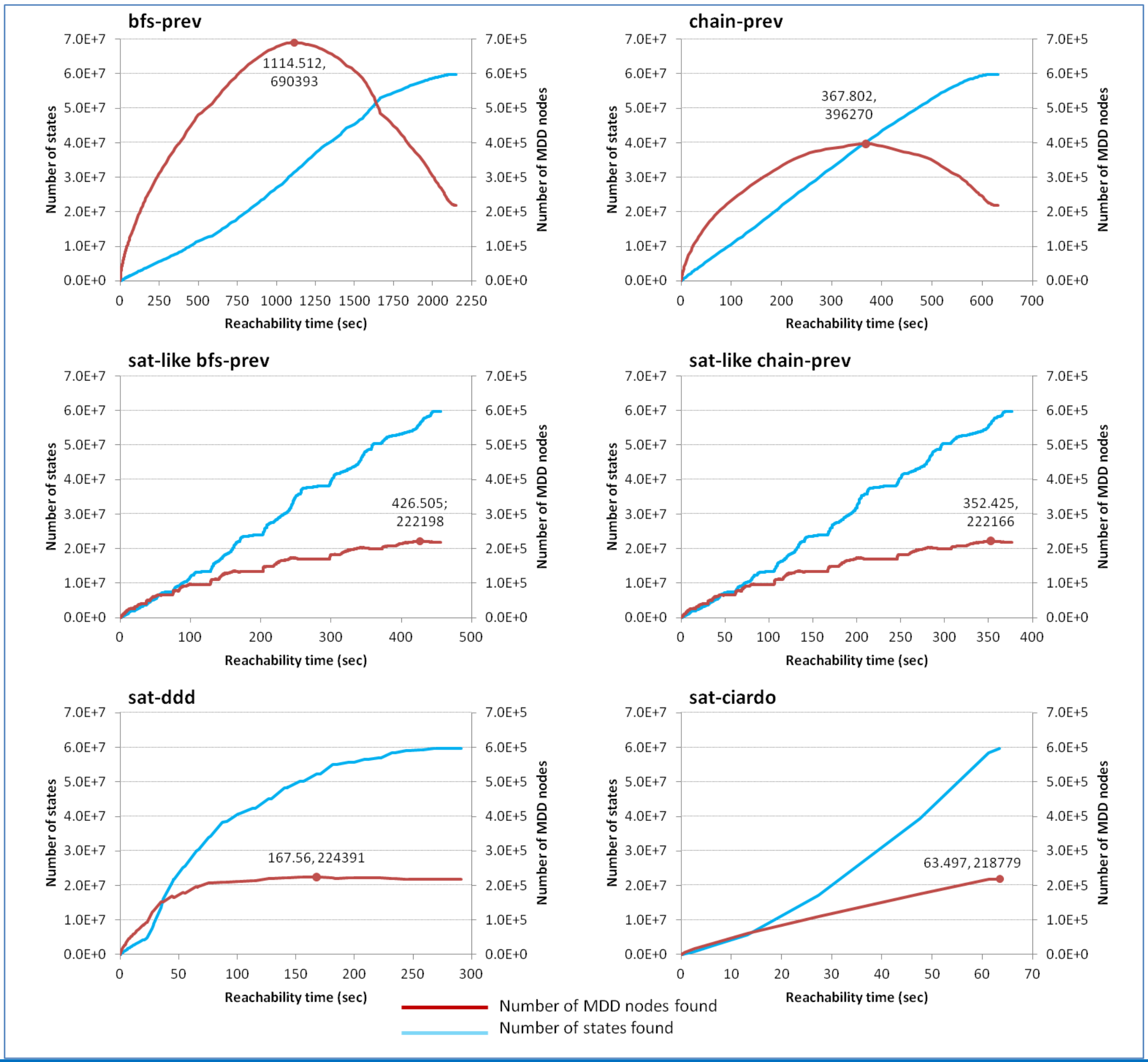
Final state space size	Final MDD node size	Fastest reachability option
18206917	38225	sat-like bfs-prev

Figure 5.9: State space evolution plots of DVE model anderson.6.



Final state space size	Final MDD node size	Fastest reachability option
11465015	161219	sat-ddd

Figure 5.10: State space evolution plots of DVE model cambridge.7.



Final state space size	Final MDD node size	Fastest reachability option
59794192	218779	sat-ciardo

Figure 5.11: State space evolution plots of DVE model iprotocol.7.

5.2 Analysis of Saturation performance

This section will discuss the obtained results by going into the different aspects of time performance, state space evolution and memory performance separately.

5.2.1 Time performance of Saturation

The time performance of the options sat-ddd and sat-ciardo seems to show some mixed results when compared to the options no-sat and sat-like. Both option sat-ciardo and sat-ddd are more likely to outperform the option no-sat than the option sat-like. This is especially noticeable for models which are not very time-demanding to come up with a result.

When comparing the time results between options sat-ddd and sat-ciardo, it turns out that in every two out of three tested models option sat-ddd performs faster in time than option sat-ciardo.

The main causes behind these results in relation to other reachability options seem to lie in the following:

- option sat-ciardo is suffering from updating transition groups slowly and performing lots of unnecessary MDD projections, and
- option sat-ddd is suffering from a gradual decrease in progress over time, but keeps the number of MDD projections as small as possible.

Option sat-ciardo tries to update the transition group at the current working MDD level during the Saturation process. Taking a closer look in the progression of the algorithm it turns out that it only looks for new (sub) states locally in the MDD and therefore can only find a few new (sub) states at a time, if any is found. Also each time when trying to update the affected transition group, it performs the projection operation on the MDD of the (sub) state space seen so far, which is a large burden on the time performance and unnecessary when the affected transition group is not updated at all. Performing such projections where the affected transition group is updated with new (sub) transitions, will be denoted as useful projections. In Table 5.6 one can compare the number of useful and total projection operations made per reachability option, and it is obvious that option sat-ciardo has a high percentage of useless projections (also compared to other options as summarized in Table 5.7). Also a visualization of this for three models is given in the bar charts in Figure 5.12 (for the other models, see appendix C). This problem is one of the reasons why for certain models option sat-ciardo is always performing slower compared to options no-sat and sat-like. For the models where the option sat-ciardo is always performing faster, it turns out that the options no-sat and sat-like need to overcome a high peak MDD node value, where also lots of iterations in the algorithm are needed to find all reachable states.
These findings are also supported by using profiling on the reachability option sat-ciardo for a couple of models which do not require much effort (in time) to compute a result, as shown in Table 5.8 and Table 5.9. In Table 5.8, for three types of functions the percentage of total execution time is given used by the function and next to it the rank of the function in the list of all time-demanding functions recorded by profiling (for projection and construction of MDD nodes, the results of multiple accompanying functions are accumulated). The projection operations take a high rank in the function list, in most cases it turns out that emptying the lookup tables is the most demanding operation, which is one of the operations being performed at the end of the projection operation. For completeness the final two columns of Table 5.8 also indicate how time-demanding it is for the algorithm to construct new MDD nodes.

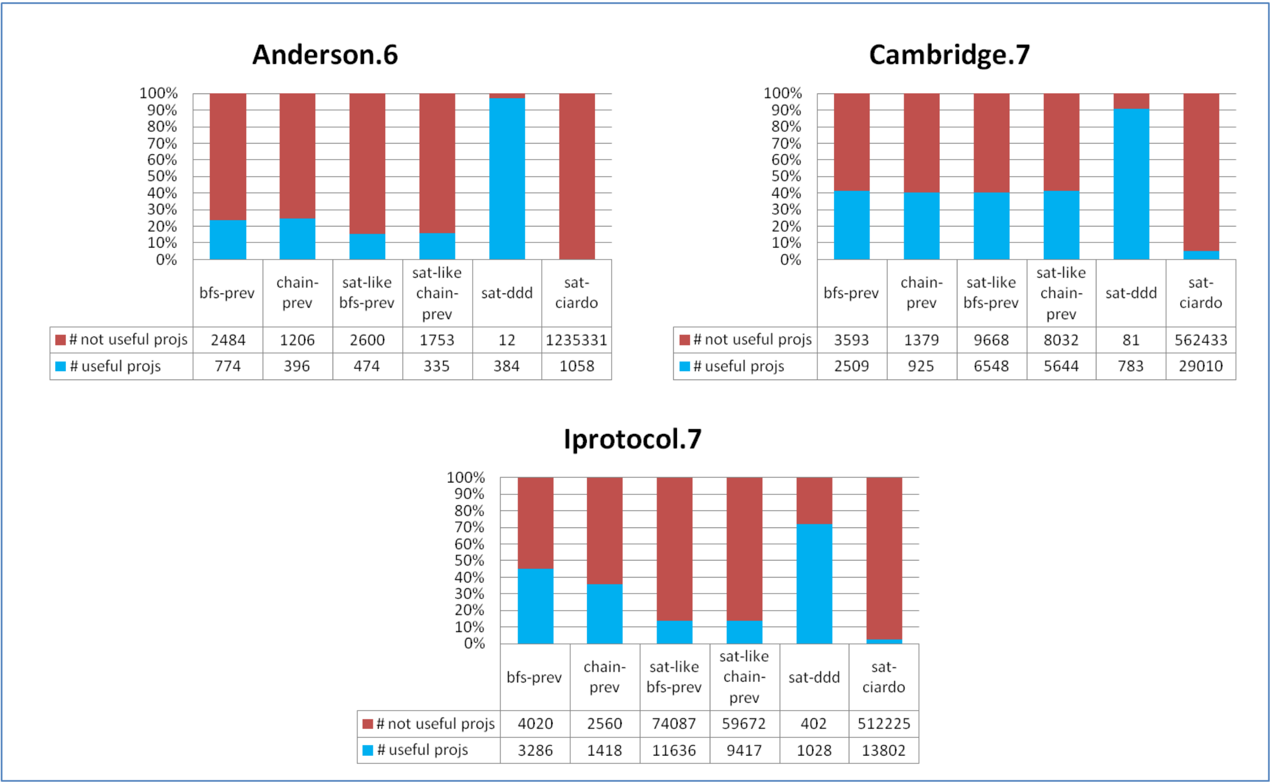
On the other hand, option sat-ddd only updates the transition groups outside the Saturation algorithm. This turns out to be more efficient, because newly found (sub) states are discovered globally taking into account every level of the MDD tree of the state space seen so far. This also makes sure that the number of useless projections made (while trying to update a certain transition group) is kept to a minimum, as is also noticeable in Table 5.6, Table 5.7 and the bar charts in Figure 5.12. But there is a downside to performing updates outside the Saturation loop, as noted for some models where option sat-ddd is always performing slower than the options no-sat and sat-like: it may start well at the beginning by discovery lots of new states, but the number of new states found diminishes over time and nearing the end only a small number of new states are found per iteration of the Saturation process, while re-computing and storing partial results again. But the positive side to this aspect is that the peak MDD node value lies closer to the final MDD node value and it might be the main aspect why option sat-ddd is always performing faster for some models compared to options no-sat and sat-like.

Model	bfs-prev			chain-prev			sat-like bfs-prev			sat-like chain-prev			sat-ddd			sat-ciardo		
	useful projs	total projs	% of total projs	useful projs	total projs	% of total projs	useful projs	total projs	% of total projs	useful projs	total projs	% of total projs	useful projs	total projs	% of total projs	useful projs	total projs	% of total projs
and.6	774	3258	23.76	396	1602	24.72	474	3074	15.42	335	2088	16.04	384	396	96.97	1058	1236389	0.09
at.5	481	1326	36.27	283	702	40.31	422	3514	12.01	278	2288	12.15	281	312	90.06	652	2049081	0.03
at.6	531	2470	21.50	320	1482	21.59	455	5664	8.03	309	3630	8.51	260	286	90.91	747	13098248	0.01
bak.6	1447	3668	39.45	636	1428	44.54	2106	9036	23.31	905	3588	25.22	583	756	77.12	2343	7988570	0.03
brp.5	2811	11902	23.62	1527	5940	25.71	2868	20490	14.00	2183	14178	15.40	779	1408	55.33	4574	228266	2.00
brp.6	3697	14498	25.50	1996	7238	27.58	3765	24712	15.24	2825	16836	16.78	945	1848	51.14	8659	346009	2.50
cam.7	2509	6102	41.12	925	2304	40.15	6548	16216	40.38	5644	13676	41.27	783	864	90.63	29010	591443	4.90
col.4	764	4872	15.68	388	2581	15.03	508	5924	8.58	356	4350	8.18	401	609	65.85	723	118126	0.61
ele.2	131	243	53.91	72	126	57.14	102	265	38.49	85	195	43.59	43	45	95.56	100	1518	6.59
fir.5	18349	43188	42.49	8270	17700	46.72	20673	111744	18.50	15400	82575	18.65	7307	9381	77.89	41221	1014049	4.06
ipr.6	3348	11804	28.36	1544	6630	23.29	8244	82653	9.97	6471	63430	10.20	989	1352	73.15	7384	180573	4.09
ipr.7	3286	7306	44.98	1418	3978	35.65	11636	85723	13.57	9417	69089	13.63	1028	1430	71.89	13802	526027	2.62
lam.7	655	3465	18.90	375	1435	26.13	575	5686	10.11	382	3250	11.75	370	455	81.32	431	1392076	0.03
lam.8	750	6480	11.57	434	2920	14.86	680	12940	5.26	436	8104	5.38	425	560	75.89	512	2886762	0.02
lna.5	1429	6400	22.33	776	1920	40.42	1435	12109	11.85	960	6967	13.78	1152	1200	96.00	1210	904992	0.13
lann.6	538	5656	9.51	137	2324	5.90	353	9329	3.78	202	4781	4.23	288	420	68.57	560	8674697	0.01
lann.7	427	10890	3.92	137	3300	4.15	310	12176	2.55	223	7888	2.83	228	540	42.22	288	256483	0.11
le.6	4973	21978	22.63	1413	2574	54.90	2025	3804	53.23	1513	3083	49.08	2249	4851	46.36	1412	9883	14.29
lf.7	1194	1728	69.10	448	672	66.67	959	3432	27.94	603	1959	30.78	522	624	83.65	4458	4856870	0.09
pet.6	517	1620	31.91	235	700	33.57	548	3380	16.21	325	1866	17.42	275	320	85.94	550	814164	0.07
pet.7	1395	4425	31.53	745	2275	32.75	1496	13375	11.19	1052	9356	11.24	539	800	67.38	5997	9187070	0.07
phi.6	135	1290	10.47	76	870	8.74	135	800	16.88	79	648	12.19	120	120	100.00	118	508	23.23
phi.8	144	1472	9.78	81	992	8.17	144	803	17.93	85	661	12.86	128	128	100.00	126	546	23.08
sch.3	239	782	30.56	135	272	49.63	241	1192	20.22	181	698	25.93	130	136	95.59	411	94497	0.43
szy.5	3025	9480	31.91	1511	3780	39.97	1762	20387	8.64	1359	13683	9.93	1719	2040	84.26	2082	9907129	0.02
tel.4	460	1224	37.58	275	672	40.92	1146	4436	25.83	674	2532	26.62	300	432	69.44	2574	911409	0.28
tel.7	728	2100	34.67	413	980	42.14	1807	9414	19.19	1100	5464	20.13	396	504	78.57	4042	1914874	0.21

Table 5.6: Statistics showing the number of useful and total MDD projections made using the reachability tools from LTSmin on the 27 DVE models.

	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
Average % of useful projs	28.63	32.27	17.35	17.92	78.21	3.32

Table 5.7: Summary of the average percentage of useful MDD projections made per reachability option.



Anderson.6

	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	2484	1206	2600	1753	12	1235331
🟦 # useful projs	774	396	474	335	384	1058

Cambridge.7

	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	3593	1379	9668	8032	81	562433
🟦 # useful projs	2509	925	6548	5644	783	29010

Iprotocol.7

	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	4020	2560	74087	59672	402	512225
🟦 # useful projs	3286	1418	11636	9417	1028	13802

Figure 5.12: Bar charts showing the contribution of useful and useless MDD projections using the reachability tools from LTSmin for 3 DVE model.

Model	MDD projection functions (vset_project, set_projection_list, set_project_2)		Function for emptying a lookup table (ATtableReset)		Functions for constructing an ATerm (MDD) node (ATmakeAppl functions)	
	Accumulated % of total time used	1st occurring rank in function list	% of total time used	Rank in function list	Accumulated % of total time used	1st occurring rank in function list
brp.5	4.35	3rd	58.49	1st	15.89	2nd
col.4	11.57	2nd	41.94	1st	20.25	3rd
ele.2	< 0.01	56th	-	-	0.30*	21st
ipr.6	6.33	4th	13.61	2nd	35.82	1st
lann.7	7.38	2nd	46.55	1st	17.62	3rd
le.6	0.11	18th	0.50	12th	0.26	19th
phi.6	< 0.01	45th	-	-	-	-
phi.8	< 0.01	46th	14.29	6th	-	-
sch.3	5.34	6th	22.95	1st	24.23	3rd
Legend of symbols						
-	No data available from profiling		*	Only 1 ATmakeAppl function recorded by profiling		

Table 5.8: Statistics from profiling using option sat-ciardo.

Model	Explanation most demanding function (at 1st rank in list)	% of time affected
ele.2	Function used to check if PINS dependency matrix is set.	25.00
le.6	Function used to check if PINS dependency matrix is set.	34.90
phi.6	Function used to check if a bit vector is set.	50.00
phi.8	Function used to check if PINS dependency matrix is set.	28.57

Table 5.9: Most demanding function for DVE models from Table 5.8.

Also Table 5.9 explains why the total execution time for some models differs much from the reachability time and is as expected, caused by the initialization of the model for the reachability analysis.

Other aspects that influence the time performance of options sat-ddd and sat-ciardo are the variable ordering used in the PINS dependency matrix and the number of (reachable) transitions found for certain transition groups for some models. It should be noted that this issue also applies to other reachability options from the LTSmin toolset.
A bad variable ordering results in transition groups that affect levels which are located far apart from each other, meaning that the Saturation algorithm still has to go through the intermediate identity transitions (and possibly starting to saturate with transition groups that start at this level).
And transition groups with a large number of reachable transitions can cause an explosion of new (sub) states that need to be added to the state space and traversed.

From the time plots between sat-ddd and sat-ciardo (as given in Figure 5.5) it is also noticeable that for the DVE models iprotocol.6.dve, iprotocol.7.dve and leader_election.6.dve, option sat-ciardo performs faster than option sat-ddd. There seems to be a pattern here underlying these results: in the PINS dependency matrices the transition groups are put here in such a way that the affected MDD levels are grouped together in blocks that together resemble a staircase pattern. This is clearly visible for the PINS dependency matrices of the DVE models iprotocol.6.dve and iprotocol.7.dve in Table 5.10 (the PINS dependency matrix for the DVE model leader_election.6.dve contains 99 transition groups and 235 MDD levels and is too large to be shown here).

DVE model iprotocol.6	**DVE model iprotocol.7**
26 transition groups (rows)	26 transition groups (rows)
37 MDD levels (columns)	41 MDD levels (columns)
`w+++++++++++r+-+---------------------`	`w+++++++++++++r+-+-----------------------`
`w+----------++r+---------------------`	`w+------------++r+-----------------------`
`-+-----------------------------------`	`-+---------------------------------------`
`--rrrrrrrrrrrr-+---------------------`	`--rrrrrrrrrrrrrr-+-----------------------`
`--wwwwwwwwww-r++w--------------------`	`--wwwwwwwwwwww-r++w----------------------`
`------------+r-+-----------w+w-------`	`--------------+r-+-------------w+w-------`
`-------------r-+rwwwwwwwwww-+w-------`	`---------------r-+rwwwwwwwwwwww-+w-------`
`--------------r++++++++++++-+w-------`	`----------------r++++++++++++++-+w-------`
`--------------r++rrrrrrrrrr----------`	`----------------r++rrrrrrrrrrrr----------`
`--------------r+r--------------------`	`----------------r+r----------------------`
`--------------r+-rrrrrrrrrr-+w-------`	`----------------r+-rrrrrrrrrrrr-+w-------`
`--------------r+-rrrrrrrrrr----------`	`----------------r+-rrrrrrrrrrrr----------`
`--------------w+------------+r-------`	`----------------w+--------------+r-------`
`---------------++wwwwwwwwww----------`	`-----------------++wwwwwwwwwwww----------`
`---------------+w-------------+------`	`-----------------+w---------------+------`
`---------------+-----------r+w-------`	`-----------------+-------------r+w-------`
`----------------------------+w-+rrr--`	`--------------------------------+w-+rrr--`
`----------------------------+w-+rr---`	`--------------------------------+w-+rr---`
`----------------------------+w-++----`	`--------------------------------+w-++----`
`----------------------------+r-+--w--`	`--------------------------------+r-+--w--`
`----------------------------+--------`	`--------------------------------+--------`
`------------------------------++-----`	`----------------------------------++-----`
`-------------------------------+rrw++`	`-----------------------------------+rrw++`
`-------------------------------+r+r--`	`-----------------------------------+r+r--`
`-------------------------------+rr---`	`-----------------------------------+rr---`
`-----------------------------------+-`	`---------------------------------------+-`

Table 5.10: PINS dependency matrices for DVE models iprotocols.6 and iprotocols.7.

The reason why option sat-ciardo performs well with such dependency matrices is likely due to its ability to saturate MDD nodes at the lower levels in their final form by updating transition groups with new (sub) states which are locally found. This makes sure that the algorithm does not have to traverse these lower levels again, since it knows they will remain unchanged (with partial results stored in lookup tables).

Option sat-ddd is struggling with this since new states for the update process is only found outside the Saturation process at a global view of the MDD.

5.2.2 State space evolution of Saturation

For one thing the Saturation algorithm is true to its name and this can be seen from the results regarding the state space evolution over time.
The bar charts in Figure 5.6 reveal a trend across the reachability options taken into consideration. For more than two thirds of the models the largest peaks occur with option no-sat, where breadth-first search mostly obtains a higher peak than chaining. The second-largest peaks are obtained by the option sat-like for almost two thirds of the models, where the auxiliary traversal technique breadth-first search also seems to be the one which obtains a higher peak than chaining. Finally the options sat-ddd and sat-ciardo are the ones with the lowest peak values which also lie closest to the final number of MDD nodes for almost every model.
Looking at the evolution plots, it turns out that for options sat-ddd and sat-ciardo the peak occurs more in the second half of the state space computation time, while this fluctuates more around the middle of the computation time for options no-sat and sat-like. It is also visible that the option sat-ddd seems to be resembling the results of the option sat-like, which is characterised by the occurrences of multiple (sharp) peaks during state space generation. But unlike option sat-like, the peak values in option sat-ddd are not as extreme.

The results confirm the fact that the Saturation algorithm attempts to bring the MDD nodes in their final form as early as possible. This happens by saturating MDD nodes at lower levels first before continuing one level up until the root level and this has resulted in the relatively low peak MDD node value which lie close to the final MDD node value. And the fact that the peak value is not always equal to the final value is due to characteristics of the model under consideration. Certain transition groups that start at a certain level can cause an explosion of newly found states and to cope with this issue, extra unsaturated MDD nodes are inserted to keep pace with it.

For the option sat-ciardo, in the ideal situation the number of MDD nodes grows steadily when starting to saturate at a new higher level. Sometimes this causes (small) local peaks and/or sharp rises, but steady growth is maintained until the final MDD size is reached. In the worst situation it just starts to find lots of new MDD nodes at the uppermost or root level. This is accompanied with a sharp rise to a peak lying way off the final MDD size. The MDD size then decreases until the final MDD size is reached at a lower value. The latter is mainly caused by the variable ordering being used for the PINS dependency matrix in which certain transition groups span lots of levels and/or by the nature of certain transition groups of the model that contain lots of transitions for which lots of new states can be discovered.

But still there are models where options sat-ddd and/or sat-ciardo perform badly with regard to peak values of MDD nodes.
For the model cambridge.7.dve the evolution plot of option sat-ciardo resembles the plots of options no-sat and sat-like, with a flat rounded peak and for all these reachability options the peak values lie quite close to each other. Only option sat-ddd has a plot that almost shows the ideal situation of an increasing line with a peak value lying closer to the final value. For this model it turns out that the PINS dependency matrix is not in a very optimal shape, where lots of intermediate identity transitions occur and lots of (adjacent) levels are affected. The transition groups also contain lots of transitions (some have 100000 transitions), causing the algorithm to start discovering new states and accompanying transitions to be added to the transition groups.

For the model schedule_world.3.dve the peak values in the evolution plots for options sat-ddd and sat-ciardo are higher than option sat-like. The cause lies in the transition group which starts at a particular level high in the MDD tree, which spans lots of levels in the PINS dependency matrix and causes an explosion of new states to be found.

5.2.3 Memory performance of Saturation

When looking at the memory performance, the options sat-ddd and sat-ciardo perform quite badly compared to the available options no-sat and sat-like in the LTSmin toolset. Also compared to each other, they almost seem to use equal amounts of memory, in which some results deviate much from each other: it turns out that when the maximum memory value is lower for option sat-ciardo, then its peak MDD value is also lower compared to sat-ddd and vice versa (more MDD nodes means more memory is needed to store them in the lookup tables and vice versa).

For option sat-ciardo the fact is that it keeps lookup tables throughout the computation of the state space without emptying them in between due to memoization (the lookup tables always grow in size). Also when the state space grows over time, new entries are inserted in the lookup tables which cause the memory usage to accumulate.
For option sat-ddd the lookup tables are emptied every time it exits the call to the General Prebuilt Saturation algorithm and therefore needs to recalculate and store partial results again when entering this algorithm. The combination of storing (parts of) previous partial results again and newly built parts of the state space (computed with the updated transition groups) can therefore result in the same amount of memory as achieved by option sat-ciardo.

It should be remarked that these findings are also caused by the type of MDD library used and in this case the ATermDD library and its capabilities in regard to memoization. Using other MDD libraries may result in other (hopefully better) findings.

5.3 Summary of evaluation of Saturation

Using the dve2-reach tool from the LTSmin toolset, experiments have been conducted on a selection of 27 DVE models from the BEEM database. Hereby the implemented General Prebuilt and General On-the-fly Saturation algorithms inside the options sat-ddd and sat-ciardo respectively, have been compared with the other available reachability options from the LTSmin toolset. These consist of the traditional traversal techniques breadth-first search and chaining, and an LTSmin version of a Saturation-like algorithm which uses the traditional traversal techniques as secondary technique, namely option sat-like. From the experiments results are gathered regarding time and memory performance, but also how the state space evolves over time. These results provide an answer to the third research question:

"How does the chosen Saturation approach perform in the LTSmin toolset compared to other symbolic reachability techniques in the tool?"

The implemented General Saturation algorithms in the options sat-ddd and sat-ciardo are true to their name. The results regarding the evolution plots and the peak MDD node size reveal that the peaks are situated more in the second half of the computation time and compared to the options no-sat and sat-like, the peak values are reduced dramatically for the majority of the tested models. This is due to the nature of how the Saturation algorithms build up the entire state space, by saturating lower MDD levels as

soon as possible before continuing with upper MDD levels until the root level. Although the evolution plots do occasionally show peaks for options sat-ddd and sat-ciardo, these mostly depend on the model and the variable ordering being used in the PINS dependency matrix.
But this turns out to have a downside to the memory performance, which shows a dramatic increase compared to the other tested reachability options (when using the ATermDD library). This is caused by the extensive use of storing intermediate results of the state space and due to the fact that lookup tables are not emptied during the Saturation process (caused by memoization).

A disappointing fact that is visible from the results is the bad performance in time for more complex models when compared to the other tested reachability options. The main cause seems to lie in the update process for option sat-ciardo and the gradual decrease in progress for option sat-ddd. In option sat-ciardo the algorithm only has a local view on the parts of the state space it works on, resulting in discovering only a small number of new (sub) states when attempting to start the update process of the affected transition groups. In this update process it already starts performing the projection operation on the state space so far, which is quite inefficient in time (especially when the transition group is not updated with new (sub) transitions). With the option sat-ddd the algorithm has a more global view of the state space so far resulting in a more accurate update process of the transition groups with less useless projections. But every time it re-enters the General Prebuilt Saturation algorithm it needs to recalculate previous intermediate results again together with constructing new parts of the state space. This can cause the algorithm to discover new states at the beginning of the process, but nearing the final size of the state space it starts to require more effort to find new states.

Between options sat-ciardo and sat-ddd it seems that the latter seems to be a more promising algorithm to go on with when looking at the time performance. With regard to memory performance there is no real significant difference between these two options. Comparing these two options also revealed the importance of the structure of the PINS dependency matrix to their time performance. It turns out that when the PINS dependency matrix reveals a blocked staircase pattern (of MDD levels affected by transition groups), then option sat-ciardo is a better alternative than option sat-ddd. A possible explanation is that option sat-ciardo can saturate lower MDD levels earlier into their final form, resulting in processing upper MDD levels only (caused by the local view on parts of the state space so far). Option sat-ddd only has a global view on the state space so far when starting the update process of the transition groups, which causes it to revisit every state again because of resetting the lookup tables.

6 Improvement on Saturation in LTSmin toolset

The implemented version of General On-the-fly Saturation in the option sat-ciardo shows mixed results. Especially its time performance is not as promising as hoped for and therefore an improvement is given to this algorithm by tackling its update process of the transition groups. This chapter first goes into the design and implementation of a new version of option sat-ciardo, which is followed by an evaluation, including comparisons with the results obtained previously.

6.1 Design & implementation of improvement of Saturation

The problem with the implemented version of the General On-the-fly Saturation algorithm lies in the update process of the transition groups, as indicated in section 5.2.1. The algorithm tries to update transition groups as soon as possible, and results in only having a localized view of parts of the state space so far (and therefore resulting in finding new (sub) states sparingly). To improve this update process, it is of great importance to reduce the number of times it attempts to start an update process for a certain transition group (and therefore reducing the number of useless MDD projections).

One possibility to tackle this issue can be found in the update process being used in the Saturation algorithm by G. Ciardo [23]. That version of the General On-the-fly Saturation algorithm collects newly found values per level during Saturation. When the algorithm starts to perform the fix-point computations with an event, it first takes every new value found (if any) and uses this to update every event that affects the level at which the value is found. After updating every event with the new value at a given level, every affected event has knowledge of this value and no successive updates for the same value at the given level are needed. Furthermore the known values that each level can possess are stored (per level) and after each update process, the new value together with the level number where it was found, are stored. Although in the papers of G. Ciardo no concrete implementation details are given on this update process, the procedure assumed to take place is visualized in Figure 6.1 (which describes it from an implementation perspective inside the LTSmin toolset). It turns out that using this update process one can encounter unreachable (sub) states for which also (sub) transitions are computed that are used to update the transition groups.

For the design of the improved version of the Saturation algorithm, the previously defined requirements as given in Table 4.1 and Table 4.2 still apply. But now the update process of the transition groups is being tackled, some additional assumptions are taken into consideration and are shown in Table 6.1.

A1.1	The algorithm can produce unreachable (sub) states for the update process of the transition groups and they are recognized as valid (sub) state vectors by the language modules used in the LTSmin toolset.
A1.2	The language modules used in the LTSmin toolset are able to calculate successor (sub) states from unreachable (sub) states.
A1.3	The data object used for storing the transition groups (vrel_t) is able to store unreachable (sub) states.

Table 6.1: Assumptions made for redesign of update process of General On-the-fly Saturation algorithm.

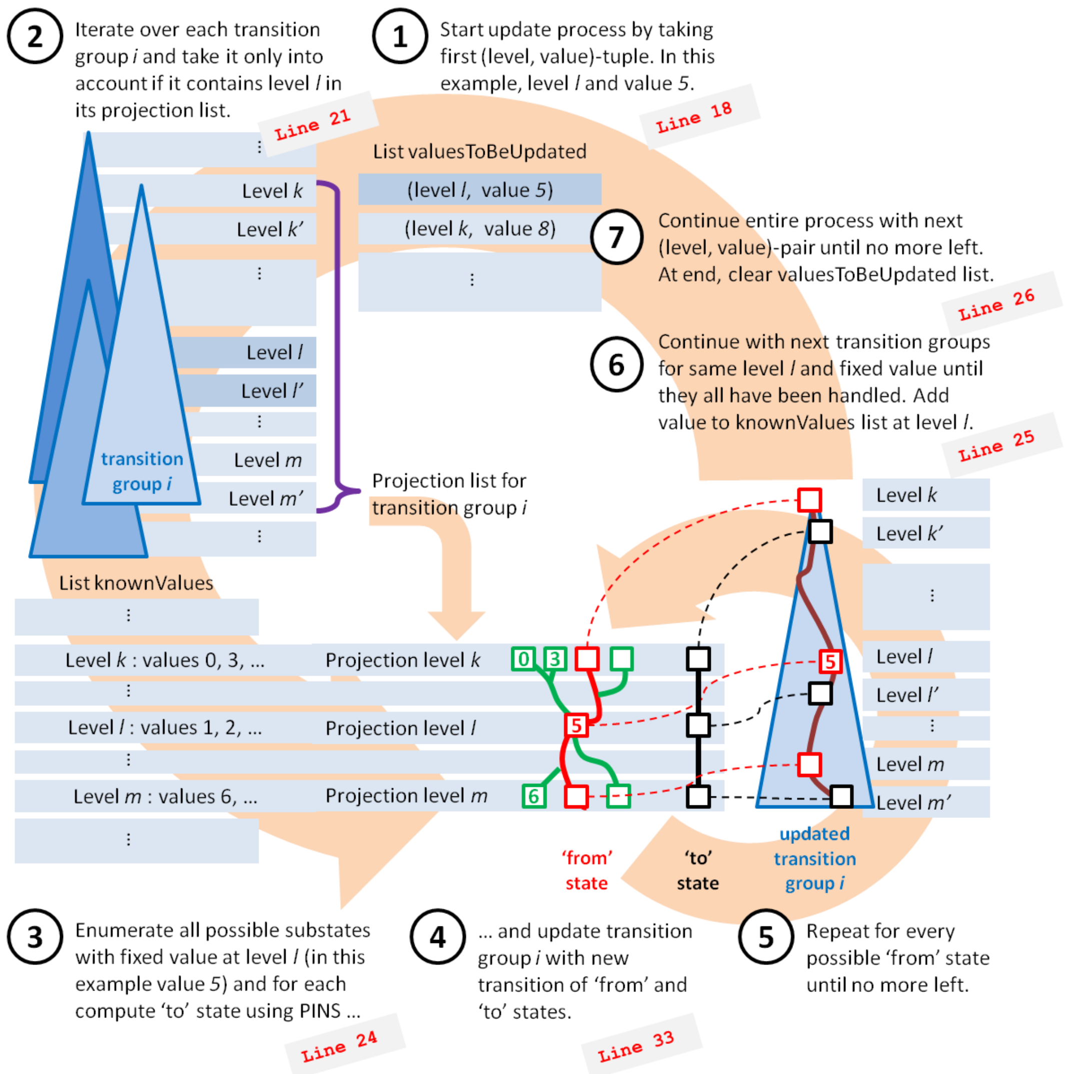
With these assumptions in mind the design of the algorithm is further improved, without actually checking that they are valid.

For the implementation in the LTSmin toolset the structure of the Saturation algorithm is left unchanged and only some adjustments need to be incorporated in the affected MDD library. An overview of the adjustments in pseudo code is given in Listing 6.1.
First, two new global array variables are introduced to store the known values per MDD level and the newly found values with the MDD level number where it was found. These are the arrays `knownValues` and `valuesToBeUpdated` in lines 2 and 6 in Listing 6.1, respectively.
Second, a function `addNewValue` (line 10 in Listing 6.1) is introduced to check if a certain value and level pair has been encountered before. If it does not, then it concerns a new value and the value and its associated level number are appended into the array `valuesToBeUpdated`. This function is called during the recursive calls of the relational product and more precisely inside the function `transLevel` as highlighted in dark-blue in line 91 in Listing 4.1 (at the moment it starts to head one MDD level down with the new value).
Third, the function `expand_group_next` inside the function `startFixPoint` needs to be replaced with a function that enumerates all possible (sub) states with a certain fixed new value and associated MDD level number (highlighted in dark-blue in line 29 in Listing 4.1). This is used to compute new successor (sub) states for updating transition groups with the given MDD level number. Iterating over the newly found values and affected transition groups to be updated, is done in the revised function `expand_group_next` in line 17 in Listing 6.1. The enumeration of all possible (sub) states with the values from the array `knownValues`, a fixed value and associated MDD level number is performed inside function `enumStateVectors` in line 30 in Listing 6.1. The actual update of the transition group by finding a successor (sub) state and adding the transition to it, is performed inside function `explore_cb` (line 33 in Listing 6.1), which is an existing function located in spec-reach.c. To be able to access this function from within the MDD library, a function pointer to this function is introduction in the structure of the vector relation and is set during the initialisation phase for each transition group.

When looking at the revised version of the update process of the transition groups, it is noticeable that it does not use the calculated state space so far for the update process. This also means that no projections need to be done on the MDD of the calculated state space so far, resulting in a projection-free update process of the transition groups.

```
 1  // All discovered valid values per level.
 2  array<array<int>> knownValues
 3
 4  // Newly discovered (level, value)-tuples that still need to be used to update
 5  // transition groups.
 6  array<(int, int)> valuesToBeUpdated
 7
 8  // If value found at certain level has not been discovered before, then append to
 9  // list of values that still need to be updated (for updating transition groups).
10  void addNewValue (int level, int value) :
11    if (!(value in knownValues[level] || (level, value) in valuesToBeUpdated)) :
12      valuesToBeUpdated.append((level, value))
13
14  // For all newly found values, update only those transition groups that contains
15  // affected level by enumerating all possible (sub) states with at level the given
16  // value.
17  void expand_group_next() :
18    foreach ((int level, int value) in valuesToBeUpdated) :
19      foreach (int grp in 0 .. nGrps) :
20        projection proj := getTransitionGroupProjection(grp)
21        if (level in proj) :
22          array<int> srcState[proj.length]
23          contextInfo ctx := constructNewContextInfo()
24          enumStateVectors(ctx, level, value, srcState, 0, proj)
25      knownValues[level].append(value)
26    valuesToBeUpdated.clear()
27
```

```
28 // Enumerate all possible substates with given value at level and for each (sub)
29 // state, call callback function explore_cb to update certain transition group.
30 void enumStateVectors(contextInfo ctx, int level, int value, array<int> src,
31                       int idx, projection prj) :
32   if (prj.isProjEmpty()) :
33     explore_cb(ctx, src)
34   elif (prj.getCurrentProjLevel() == level) :
35     src[idx] := value
36     projection projNext := proj
37     projNext.setProjToNextLevel()
38     enumStateVectors(ctx, level, value, src, idx + 1, projNext)
39   else :
40     foreach (int val in knownValues[prj.getCurrentProjLevel()]) :
41       src[idx] := val
42       projection projNext := proj
43       projNext.setProjToNextLevel()
44       enumStateVectors(ctx, level, value, src, idx + 1, projNext)
45
46 // Callback function for finding successor of given source state and update
47 // transition group with the computed transition from source state to successor.
48 void explore_cb(contextInfo ctx, array<int> src)
```
Listing 6.1: Pseudo code of update process for updating all affected transition groups in redesign of implementation of General On-the-fly Saturation algorithm.



2 Iterate over each transition group *i* and take it only into account if it contains level *l* in its projection list.
Line 21
1 Start update process by taking first (level, value)-tuple. In this example, level *l* and value *5*.
Line 18
List valuesToBeUpdated
(level *l*, value *5*)
(level *k*, value *8*)
7 Continue entire process with next (level, value)-pair until no more left. At end, clear valuesToBeUpdated list.
Line 26
6 Continue with next transition groups for same level *l* and fixed value until they all have been handled. Add value to knownValues list at level *l*.
Line 25
Level *k*
Level *k'*
Level *l*
Level *l'*
Level *m*
Level *m'*
transition group *i*
Projection list for transition group *i*
List knownValues
Level *k* : values 0, 3, ...
Level *l* : values 1, 2, ...
Level *m* : values 6, ...
Projection level *k*
Projection level *l*
Projection level *m*
0
3
5
5
6
'from' state
'to' state
updated transition group *i*
3 Enumerate all possible substates with fixed value at level *l* (in this example value *5*) and for each compute 'to' state using PINS ...
Line 24
4 ... and update transition group *i* with new transition of 'from' and 'to' states.
Line 33
5 Repeat for every possible 'from' state until no more left.

Figure 6.1: Pictorial overview of update process for all affected transition groups (line numbers in figure refer to corresponding line numbers in Listing 6.1).

6.2 Evaluation of improvement on Saturation

First the setup of the experiments and the obtained results using the revised version of the General On-the-fly Saturation algorithm are presented. This is then followed by an analysis of the obtained results.

6.2.1 Experiments on improved Saturation

The changes to the adjusted Saturation algorithm have only affected the option sat-ciardo from the LTSmin toolset and therefore only this reachability option is re-run with the same arguments as before. The other algorithms are not affected by the changes made and their results are reused for comparison. For the experiments the same setup is used as described in section 5.1.1 and is briefly summarized here:

- The dve2-reach tool of the LTSmin toolset is used to read DVE models.
- The input arguments are --saturation=sat-ciardo --regroup=gs.
- Experiments are performed on a cluster machine with dual Intel E5335 CPUs and 24 GB RAM.
- The same set of 27 DVE models is taken from the BEEM database (see Table 5.2).
- A time limit of 9000 seconds (i.e. 2½ hours) and a memory limit of 24 GB are used.
- The same set of data regarding time, memory, (intermediate) state space count is collected, except the number of MDD projections done and profiling (see Table 5.3).
- Each model is run 3 times and the median value of the time and memory values are taken. A Bash script is constructed to perform the test runs automatically in a sequential way.
- Each model is run once again for gathering statistical data regarding state space evolution.

The raw data is converted into CSV files for further data manipulation and the main focus lies on presenting measurements regarding time, and how other data differ from the previous version of the option sat-ciardo and the other reachability options. From now on the revised version of option sat-ciardo is called option new sat-ciardo.

During experimentation of option new sat-ciardo, it turns out that for three DVE models no result can be computed due to occurring errors (e.g. segmentation faults) or infinite loops. Assumption A1.1 does not hold for the DVE language module. The problem is that the code generated by DiVinE, does not contain checks on the bounds of arrays, which is actually being prescribed by the language semantics of DVE. By passing on combinations of state vector values to the language module that normally do not occur in the model (such as unreachable states), array-out-of-bounds errors occur. This causes parts of the call-stack being overwritten, resulting in segmentation faults and infinite loops.
This error may have consequences for the obtained results regarding the time and memory performance: the resulting time and memory values do not have to be accurate and may be affected by the error. For the three affected DVE models no results will be presented in the tables and plots that follow.

In appendix A the measurements regarding the maximum memory usage can be found, which will not be presented prominently here. Just as before, the memory usage is highly dependent on the MDD package being used and how memoization is performed. A short summary of these results are as follows:

- The results reveal that option new sat-ciardo obtains a much higher maximum memory than the options no-sat and sat-like in the vast majority of the tested models. In such cases it can run to over 150 times the amount of memory being used by the options no-sat or sat-like.
- Between the options new sat-ciardo and sat-ddd the results reveal that the maximum memory usage seems to keep pace with each other for the vast majority of the tested models. This also accounts when comparing option new sat-ciardo and the previous version of option sat-ciardo.

The time measurements can be found in the final column of Table 6.2 (for which previous time results are copied for convenience reasons) and a summary of the number of best and worst performance is given in Table 6.3.

Model	State space size	Total execution time (median values in sec)						
		bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo	new sat-ciardo
and.6	18206917	35.42	94.07	**33.02**	33.03	143.92	**196.64**	36.62
at.5	31999440	99.75	79.66	66.57	62.26	55.54	**261.68**	**36.62**
at.6	160589600	460.18	295.73	563.32	438.96	**292.38**	**6937.73**	327.11
bak.6	11845035	259.35	103.69	195.41	**65.14**	1259.2	**2692.94**	319.8
brp.5	17740267	**260.04**	107.18	208.12	188.38	32.72	29.12	**4.2**
brp.6	42728113	**620.36**	231.38	503.09	411.86	125.19	75.15	**9.71**
cam.7	11465015	1699.6	823.41	1546.39	1420.16	**629.4**	**8989.21**	×
col.4	41465543	**514.4**	383.19	45.43	49.83	3.1**	9.11	**2.7****
ele.2	11428767	**69.54**	32.52	1.7	1.7	1.6**	**1.5****	3
fir.5	18553032	**616.46***	175.3*	123.07*	108.27*	85.06*	151.65*	**77.95***
ipr.6	41387484	**1694.09**	305.54	271.21	222.69	82.06	**15.51**	68.94
ipr.7	59794192	**2155.44**	632.58	453.37	378.02	293.4	**65.27**	513.77
lam.7	38717846	**397.22**	150.93	29.42	21.72	77.86	92.97	**19.22**
lam.8	62669317	**511.58**	426.25	69.44	52.53	189.83	225.76	**42.93**
lna.5	95118524	**81.04***	57.07*	17.41*	13.21*	18.41*	42.24*	**9.61***
lann.6	144151628	2032.26	757.77	581.13	378.94	124.08	**2280.5**	**96.06**
lann.7	160025986	**316.68**	138.41	41.54	38.12	4.8	15.11	**4**
le.6	35777100	**496.47***	14.21*	12.61*	12.41*	16.51*	**9.21***	455.03*
lf.7	26302351	218.42	83.16	82.85	**82.15**	**2464.58**	2379.49	231.56
pet.6	174495861	169.19	**293.12**	48.53	**23.71**	74.65	151.91	32.02
pet.7	142471098	851.89	344.26	347	**267.45**	1894.09	**3768.49**	328.51
phi.6	14348906	**1.9****	1.4**	1.1**	1**	1**	1**	**1****
phi.8	43046720	**2.3****	1.6**	1.2**	1.1**	1.1**	1.1**	**1****
sch.3	166649331	**274.75**	30.24	3	**2.3****	3.4	13.51	4.4
szy.5	79518740	534.79	239.08	117.77	**64.44**	1041.64	**1729.29**	132.68
tel.4	12291552	59.93	37.33	43.92	**21.91**	115.5	**239.85**	××
tel.7	21960308	150.25	89.61	36.92	**30.02**	269.58	**699.88**	××
Legend of symbols								
*	Given total execution times (for all options) is larger than reachability time with:							
	± 15 seconds for fir.5	± 6 seconds for lna.5	± 10 seconds for le.6					
**	Reachability time is 1.5 to 2 times smaller than given total execution time.							
×	No data due to (possible) infinite looping.							
××	No data due to error (e.g. segmentation fault).							
red	Worst performing option for a certain model.							
green	Best performing option for a certain model.							

Table 6.2: Experimental time results using the reachability tools from LTSmin on the 27 DVE models, including option new sat-ciardo.

	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo	new sat-ciardo
# times as fastest	0	0	1	8	2	4	12
# times as slowest	15	1	0	0	1	10	0

Table 6.3: Summary of the best and worst performing reachability options regarding time performance, including option new sat-ciardo.

Also, in Table 6.4 an overview of differences with the previous version of option sat-ciardo regarding the MDD nodes and state space evolution is given, only for models where changes occur (in the other models the evolution of the state space size and MDD nodes are unaffected). The third and fourth columns indicate the difference in peak MDD node value with the previous version of option sat-ciardo, for which the absolute and relative difference (in percentage) is given, respectively. The fifth column indicates the difference in number of iterations at the topmost working MDD level with the previous version of option sat-ciardo. Finally the MDD level at which a difference is noticeable in node and/or state space count is given in the final column.
To get a sense how the data in Table 6.4 is extracted, part of the state and node count per iteration is given for DVE model anderson.6.dve in Table 6.5. Here only the state and node count is presented for the iterations that start at the topmost working MDD level 0, since the state and node count in lower MDD levels show no difference between both options of sat-ciardo. One can notice that the peak MDD node value is decreased with –7.37 % when using option new sat-ciardo (which is reached after the 4th iteration in both options). The number of iterations at topmost working MDD level 0 is one less for option new sat-ciardo and the difference in state and node count are only found in this topmost working MDD level (as indicated with the symbol $^+$).

Regarding the evolution of the state space and MDD tree, it turns out that in the majority of the DVE models this remains unchanged compared to the previous version of the option sat-ciardo (apart from the time it spends on the models). In those cases where it does change (mostly indicated by the decrease of the number of iterations as shown in Table 6.4), the differences compared to the previous version of option sat-ciardo is quite small in terms of percentage.

Model	State space size	Difference with previous version of sat-ciardo regarding			
		peak MDD node value		number of iterations at topmost working MDD level	level where node / state count differs
and.6	18206917	–3108 nodes	–7.37 %	–1 iteration at level 0	At level 0
bak.6	11845035	–34 nodes	–0.03 %	-	At level 0
ipr.7	59794192	-	-	–2 iterations at level 0	At level 0
lf.7	26302351	-	-	–4 iterations each at levels 0 & 2	-
pet.6	174495861	–212 nodes	–0.83 %	–1 iteration at level 0	At level 0
pet.7	142471098	–126 nodes	–0.24 %	–12 iterations at level 0	At level 0
szy.5	79518740	+3640 nodes	11.13 %	–13 iterations at level 0	At level 0
Legend of symbols					
-	No change.				

Table 6.4: Statistics showing the difference in state space evolution of option new sat-ciardo compared to the previous version of option sat-ciardo (only for DVE models where a difference is noticed).

Using option sat-ciardo				Using option new sat-ciardo			
Topmost working MDD level	Iteration number	State space size	Number of MDD nodes	Topmost working MDD level	Iteration number	State space size	Number of MDD nodes
0	0	1094632	10042	0	0	1094632	10042
	1	9337366	36064		1	9337366	36064
	2	10626838	37203		2	10626838	37203
	3⁺	11305510	37590		3⁺	11455361	37624
	4⁺	16795175	42187*		4⁺	18150989	39079*
	5⁺	18183714	38901		5⁺	18190259	38938
	6⁺	18201234	38766		6⁺	18203429	38789
	7⁺	18206004	38633		7⁺	18206519	38661
	8⁺	18206842	38380		8⁺	18206907	38367
	9⁺	18206915	38255		9	18206917	38225
	10	18206917	38225				
Legend of symbols							
*	Peak MDD node value.						
⁺	At these iterations the state and node count differs between both options.						

Table 6.5: Part of state space evolution of DVE model anderson.6 per iteration of Saturation at topmost working MDD level.

Furthermore Figure 6.2 shows plots where the time values of option new sat-ciardo is plotted against the time values from option no-sat and option sat-like (left and right plots, respectively). In Figure 6.3 the plots are shown against the time values from option sat-ddd and the previous version of option sat-ciardo.

In these plots, the plot on top gives a sense where every result is located in a larger picture. The plots on the bottom are magnifications of a small part of the upper plots. One should be aware that with these magnifications the horizontal and vertical axes scale with it as well.
Furthermore a diagonal line is added in these plots to show that the two reachability options under comparison have the same time result. When a time result falls beneath the diagonal line, then this means that option new sat-ciardo computes faster than the other reachability option.

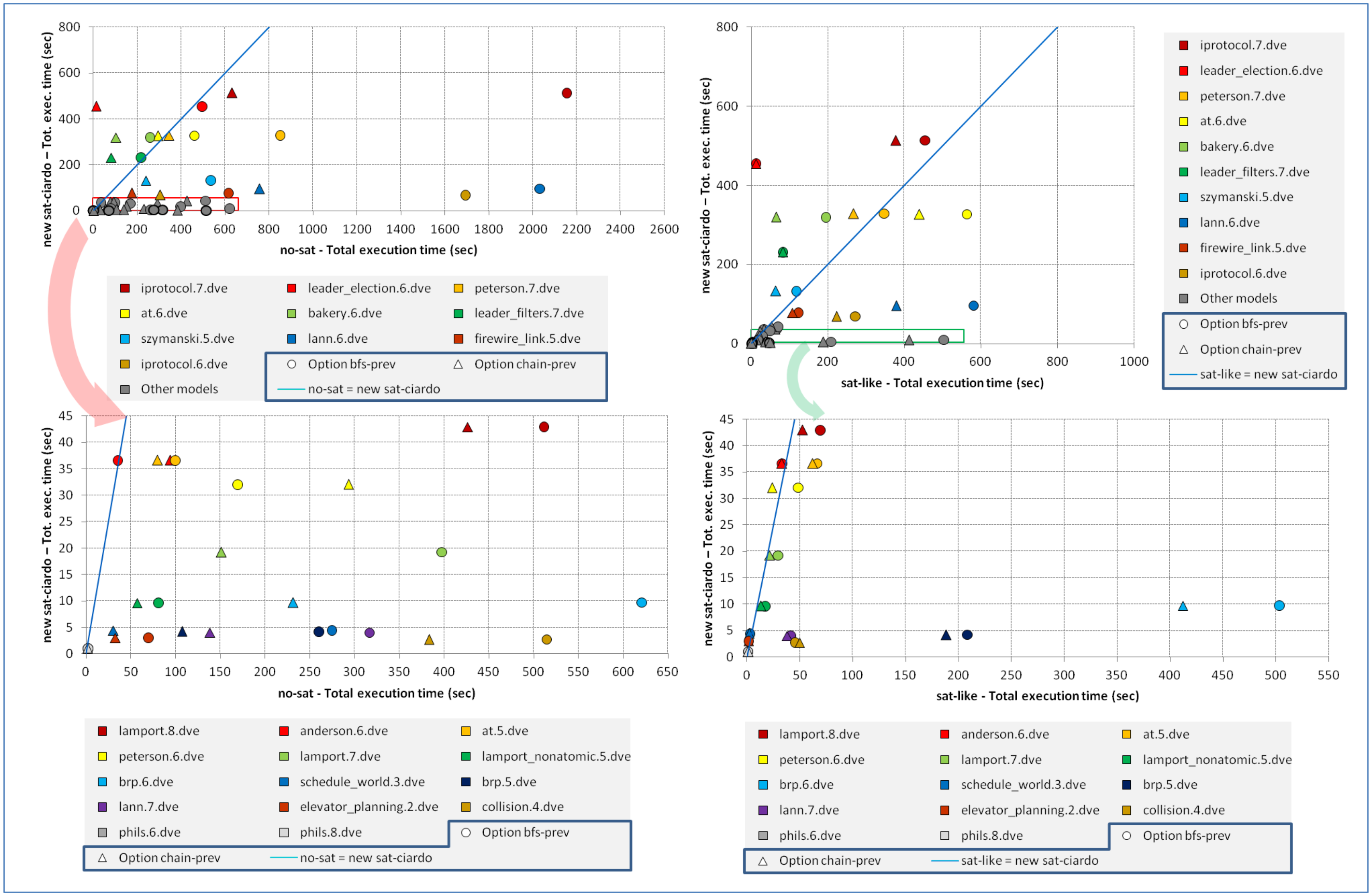


Figure 6.2: Plots comparing time values between options no-sat and new sat-ciardo (left), and between options sat-like and new sat-ciardo (right).

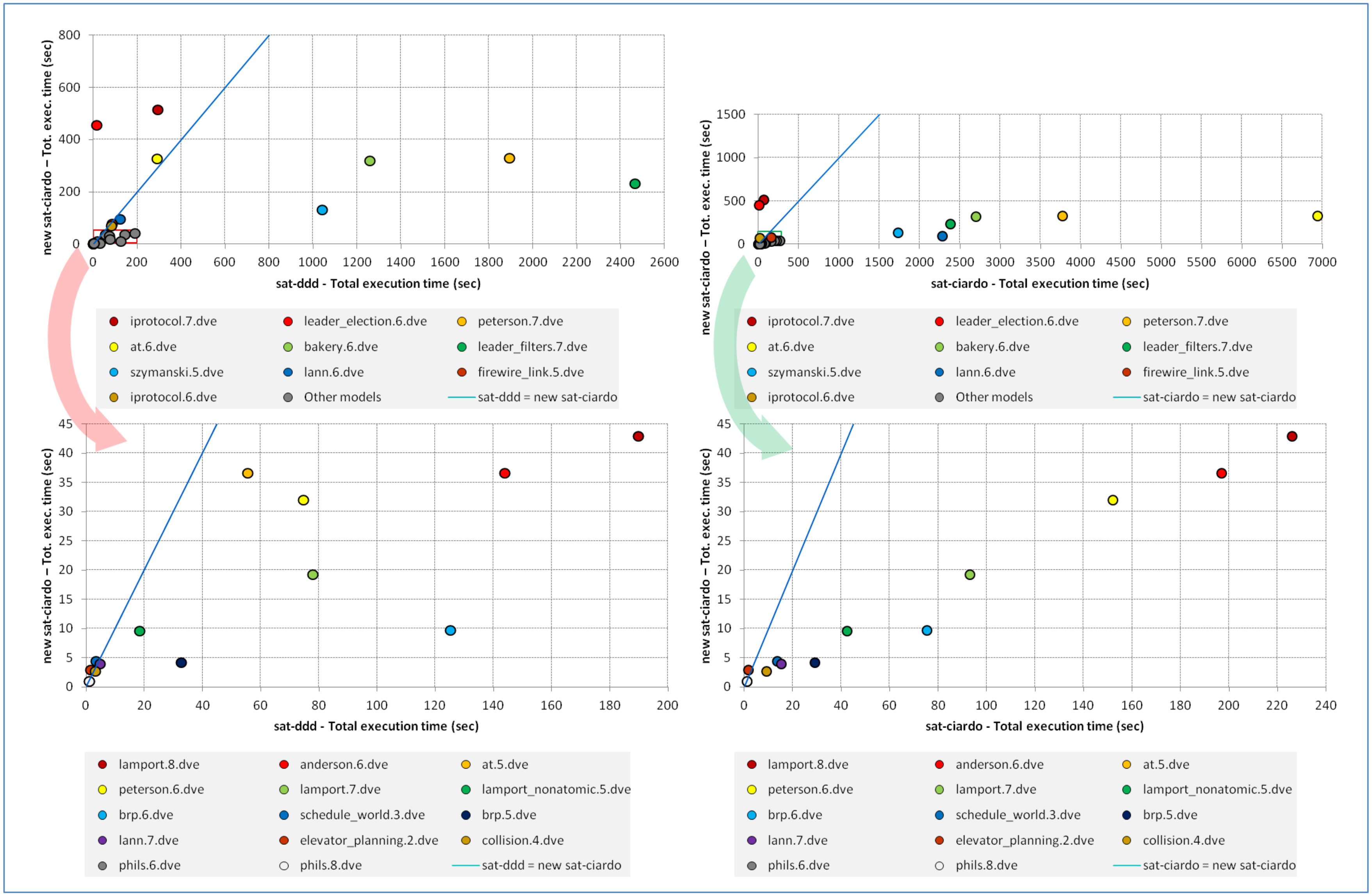


Figure 6.3: Plots comparing time values between options sat-ddd and new sat-ciardo (left), and between options sat-ciardo and new sat-ciardo (right).

6.2.2 Analysis of improved Saturation algorithm

When considering time performance, the results now clearly show that option new sat-ciardo is a good competitor to all the other reachability options, especially the previous version of option sat-ciardo. This is thanks to eliminating the MDD projection operations during updates of the transition groups. The number of updates to be performed is now kept to a minimum by only starting it when a new value is discovered at a certain level and updating every affected transition group immediately. A downside to this update process is that transition groups now also store lots more unreachable (sub) transitions and (sub) states, but it seems that these do not affect time performance in a bad way. Table 6.6 shows the minimum and maximum growth in MDD size and number of transitions that can occur per model (focused on absolute growth for which relative growth in percentage is calculated), relatively to the MDD size and number of transitions as recorded using the other reachability options and the previous version of option sat-ciardo. What can be observed from these results is that when an absolute maximum growth is seen that in terms of percentage is quite large as well, then one is almost certain that option new sat-ciardo will require more effort in time to calculate the reachable state space (as is especially noticeable for DVE models iprotocol.7.dve and leader_election.6.dve compared to options sat-like, sat-ddd and the previous version of sat-ciardo (see Table 6.2 for exact times, and Figure 6.2 and Figure 6.3 for comparisons)).

Model	**Number of transition groups (using safe regrouping)**	**Number of MDD levels**	**Number of MDD nodes growth for transition groups in new sat-ciardo (relative to previous sat-ciardo)**				**Number of transitions growth for transition groups in new sat-ciardo (relative to previous sat-ciardo)**			
			Minimum (abs.) growth		**Maximum (abs.) growth**		**Minimum (abs.) growth**		**Maximum (abs.) growth**	
			abs.	**%**	**abs.**	**%**	**abs.**	**%**	**abs.**	**%**
and.6	18	19	0	0	790	467.46	0	0	1582	8788.89
at.5	26	14	-26	-28.89	22	42.31	0	0	243	3.23
at.6	26	14	-24	-27.27	17	29.82	0	0	86	96.63
bak.6	28	20	-127	-28.10	5622	333.65	0	0	28258	992.90
brp.5	22	18	-354	-26.70	8	44.44	0	0	8651	1418.20
brp.6	22	18	-834	-28.90	8	44.44	0	0	28426	2082.49
col.4	29	22	-1	-2.33	45	409.09	-4*	-20*	214	10700
ele.2	9	35	-2	-1.35	217	59.94	0	0	1134	1157.14
fir.5	177	50	-88	-48.35	222	1009.09	0	0	13315	6370.81
ipr.6	26	37	-4583	-86.42	5156	433.28	0	0	432753	18360.33
ipr.7	26	41	-10991	-91.94	26265	1157.56	0	0	3161747	37227.68
lam.7	35	17	-1	-9.09	75	51.02	0	0	52	650
lam.8	40	17	-1	-9.09	75	51.02	0	0	52	650
lna.5	80	30	-4	-20	19	146.15	0	0	22	1100
lann.6	28	20	-14	-26.92	0	0	0	0	15	100
lann.7	30	25	-10	-27.78	0	0	0	0	8	100
le.6	99	235	0	0	77943	34795.98	0	0	2147026	15335900
lf.7	24	30	-422	-44.75	246	106.49	0	0	75819	3906.18
pet.6	20	20	3	27.27	798	170.88	3	100	7328	587.18
pet.7	25	25	-883	-76.58	9293	636.07	4	133.33	53589	2013.87
phi.6	30	30	1	10	1	10	1	50	1	50
phi.8	32	32	1	10	5	83.33	1	50	2	200
sch.3	34	34	-117	-73.58	3	14.29	0	0	3602	729.15
szy.5	60	25	1	5	1231	265.30	1	100	5583	352.24
Legend of symbols										
*	Not known why number of transitions decreases.									

Table 6.6: Statistics showing the minimum and maximum growth of transition groups of 24 DVE models for option new sat-ciardo compared to previous version of option sat-ciardo.

For the evolution of the state space no large differences can be detected and this is due to the fact that the Saturation algorithm is not changed with regard to its structure and therefore builds the state space in exactly the same way up for seven out of ten models (when only considering the non-error models). But for seven DVE models the number of iterations at the topmost working MDD level and/or the peak MDD nodes value does decrease in number. This is probably thanks to the update process of the transition groups. The transition groups are now updated with new (sub) transitions and (sub) states that have not been discovered yet (or will not be discovered because they are unreachable) and this makes sure that the Saturation algorithm can use these new transitions when it heads into the relational product that follow. For seven models this has resulted in building up the state space slightly more efficiently with a lower peak MDD node value and/or with an accompanying lower number of iterations (as already visualized with an example in Table 6.5).

When looking at the memory performance, the algorithm still performs quite badly with regard to options no-sat and sat-like. This is due to the fact that the lookup tables are still not cleared during Saturation. Also the results reveal that no real difference in memory results exist with option sat-ddd and the previous version of option sat-ciardo. This is not so surprising, because the algorithm computes the state space in (almost) exactly the same way.

6.3 Summary of improvement on Saturation

The previous version of option sat-ciardo has a bad time performance for more complex models and the problem lies in the update process of the transition groups. With inspiration from the update process used by G. Ciardo [23] the option sat-ciardo is revised in this respect.
In the improved version of the update process, the known values that can occur per MDD level are stored. During Saturation newly discovered values (and the MDD level where it was found) are collected and they are updated when a new fix-point computation is entered. For every transition group that contains the affected MDD level, all possible enumerations of (sub) states are taken with the new fixed value at the associated MDD level. Hereby the list of known values per level is used to construct all enumerations of (sub) state values.

Changes in this improved version of the Saturation algorithm only occur at the function that performs the update process of the transition groups, which now includes a function to enumerate all possible (sub) states. Also during the recursive calls to the relational product inside the Saturation process, a check is performed to find out if a certain value at its associated MDD level is new. If so, then it is added to the list of newly found values to be used in the update process. Furthermore these changes mainly occur inside the MDD library.

The improved implementation of option sat-ciardo has undergone the same tests using the dve2-reach tool and the 27 models from the BEEM database. Also the same results regarding time and memory performance and state space evolution are collected for analysis. But due to an error caused by generating code with DiVinE, three models cannot come up with a result and it is uncertain if (some) results obtained using the other models are accurate.

The results from the improved version of option sat-ciardo reveal no large changes in state space evolution and peak MDD node values. They are for seven out of ten tested models unchanged (when only considering the non-error models). For some models where this state space evolution has changed, the algorithm requires less iterations at the topmost working MDD level and/or it achieves a lower peak MDD

value. This is due to finding reachable transitions ahead of time during updates of the transition groups causing the algorithm to traverse the MDD of the state space so far slightly differently.
The memory performance of the algorithm remains bad and this is still caused by the lookup tables that are kept throughout the Saturation process.

The time performance of the General On-the-fly Saturation algorithm has now improved significantly for almost every model and also compared to the other tested reachability options. The absence of projection operations on MDDs has benefitted the algorithm greatly and reduced the number of updates for the transition groups. Although unreachable (sub) states and (sub) transitions are now stored in the transition groups, causing some transition group MDDs to grow largely in size, this has no significant effect on the time performance.

7 Conclusion & recommendations for Saturation in LTSmin

In the final chapter the main conclusions, discussions and recommendations for future work are given for the Saturation algorithm in the LTSmin toolset.

7.1 Final conclusion on Saturation in LTSmin

An adaptation and implementation of Saturation as devised by G. Ciardo, is provided in the LTSmin toolset as traversal technique for computing the reachable state space of a system's model. The improved version of the General On-the-fly Saturation algorithm (in option sat-ciardo) and the fix-point computation using the General Prebuilt Saturation algorithm (in option sat-ddd) turn out to be the best choice to traditional traversal techniques as breadth-first search and chaining. Especially the former algorithm is performing much better than the latter. Furthermore they are keeping up nicely with the LTSmin version of a Saturation-like algorithm (the option sat-like). A bottleneck undermining the performance of the General Saturation algorithms is either the update process of the transition groups for the first version of option sat-ciardo, or the gradual decrease in progress for option sat-ddd.
Updating a transition group as soon as possible during General On-the-fly Saturation as implemented in the first version of option sat-ciardo, results in entering the update process too often where in the majority of the cases no actual update of the transition group is performed. With inspiration from the update process by G. Ciardo an improved version of the General On-the-fly Saturation algorithm is provided which is only altered in its update process of the transition groups. The improved version of the algorithm uses the discovered domain values per MDD level to enumerate possible state values to be used for updating all affected transition groups. This version of the algorithm gives much better performance in time.
For the Prebuilt version of the algorithm as implemented as part of option sat-ddd where the update process occurs outside the Saturation process, the slow progress nearing the end of the final state space size is due to rediscovering visited parts of the state space again (since lookup tables containing intermediate results are deleted at the end of the Saturation process). This is becoming problematic for time-demanding models.
One aspect of Saturation is definitely confirmed: the ability of the Saturation algorithm to reduce the peak MDD node value drastically compared to the mentioned traversal techniques in the LTSmin toolset. But unfortunately this has come with a price of a high memory usage, which is blamed by the type of MDD library used here.

From the results of using option sat-ddd and the improved version of option sat-ciardo it is clearly noticeable that the Saturation approach is one of the best choices for calculating the reachable state space of a model compared to the traditional traversal techniques breadth-first search and chaining. This is visible in the better time performance and lower values achieved for the peak MDD node. Compared to the option sat-like, the results of both options sat-ddd and new sat-ciardo do not give a clear winner regarding time performance. This depends very much on the transition system of the model and how the accompanying PINS dependency matrix and variable ordering is set up. The final aspects actually apply for every traversal technique being used within the LTSmin toolset and are therefore next to the traversal technique a major contributor to the performance of a reachability analysis.
Furthermore the original Saturation approach as devised by G. Ciardo was originally meant for Petri-Net models, but by implementing it into the LTSmin toolset it proved itself as a reliable traversal technique for models written with explicit model checking in mind, as is the case for the DVE models from the BEEM database.

7.2 Final discussion on Saturation in LTSmin

Some issues from the project deserve special attention for future adaptations of the implemented Saturation algorithms.

During the design phase of the project, the choice was made to provide an implementation in one of the available fully-fledged MDD libraries and this gives the possibility to access low-level MDD operations, which are necessary for the Saturation algorithm. For future extensions of the algorithm using other MDD libraries, this means re-implementing it again in these other MDD libraries. This is not very practical and requires re-adapting the algorithm for each MDD library. Also the improved version of the General On-the-fly Saturation algorithm (option sat-ciardo) is the only traversal technique without an algorithmic structure implemented on a high level as is the case for other available traversal techniques in the LTSmin toolset (which have a high-level structure in the file spec-reach.c).
Looking back at the current implementation inside the ATermDD library, no real difficulties were encountered. Parts of the Saturation algorithm did not have to be implemented from scratch and parts of its functionalities can reuse existing functionality from the MDD library, which is the case for the union operator and the relational product for MDDs. There were also some minor issues with access to functionality which lie outside the scope of the MDD library. But thanks to the setup of the LTSmin architecture and the domain object (indicating which MDD library is being used as underlying MDD data structure), it is not too hard to provide access to functions at a high level (located in spec-reach.c).
A more suitable solution for the Saturation algorithm is to provide an implementation of the algorithm on a high level, and calling low-level MDD operations via the domain object. This also requires special attention to what extra operations need to be provided on a high level to support the Saturation algorithm (and those that need to be accessible from spec-reach.c). From the Saturation algorithm this would mean providing direct access to low-level MDD operations as creating individual MDD nodes from a high level and it remains a question if such direct access from a high level is desirable.

An issue that arose from the evaluation is a problem in the code generator of DiVinE, when using the improved version of option sat-ciardo with unreachable state values (as assumed for the redesign of the update process of the transition groups). The code generated by DiVinE, does not contain checks on the bounds of arrays, which is actually being prescribed by the language semantics of DVE. Invalid or unreachable state values can cause errors by overwriting the call-stack, resulting in segmentation faults or infinite loops.
Although this code generator is being used by the LTSmin toolset, the problem should be addressed by the developers of DiVinE. From the improved results with the improved version of option sat-ciardo, this version is more promising for incorporation into the LTSmin toolset. This is because it does not need to use the computed reachable state space so far for updating transition groups, which limits the use of heavy MDD operations drastically when performing a reachability analysis. So all language modules in use by the LTSmin toolset need to be assessed further with a stronger assumption on using unreachable state values and might lead to readjusting the interface with the language modules to deal with this issue.

Another issue resulting from the evaluation is that in the first version of option sat-ciardo, it was noticeable that it performs best when the PINS dependency matrix of a model resembles a staircase pattern for certain tested models (namely for the DVE models iprotocol.6.dve, iprotocol.7.dve and leader_election.6.dve). But with the improved version of option sat-ciardo it turns out that for these same set of models the time performance actually becomes worse. A first clue to this issue is given when looking at the maximum MDD growth of transition groups of these models, which suddenly increase dramatically compared to the first version of option sat-ciardo. Probably updating the transition groups with lots of unnecessary (sub) transitions is a major issue here, which is no problem for the first version of option sat-ciardo where only reachable (sub) transitions are stored. It still remains an issue to be investigated further with more models that have their PINS dependency matrix put in a staircase pattern. Also it remains to be investigated if this issue correlates with the problem found in the code generator of DiVinE, as indicated above.

Finally it should be remarked that during evaluation of the Saturation algorithms, the most striking issue is the occurring “memory explosion” due to memoization. Although no high importance is given to memory performance, this can still pose a problem when attempting to calculate the reachable state space of a large and complex model. One can think of using intelligent versions of “lossy” lookup tables, where invalid intermediate MDD results are thrown away, because it is no longer possible to retrieve it in the MDD of the reachable state space being built.

7.3 Future directions for Saturation in LTSmin

The current implementation of the General Saturation algorithms provides a good starting point for possible future improvements. Apart from the issues discussed above, this section will go into new possibilities which are worth investigating for improving the performance of the Saturation algorithm and possibly future releases of the algorithm in the LTSmin toolset.

For the General Prebuilt Saturation algorithm which is part of the option sat-ddd, the main issue is re-entering the Saturation process with the calculated state space so far and revisiting it again, due to emptying the lookup tables of intermediate results of the state space at the end of a Saturation process. To prevent it to revisit same parts of the state space over and over again, it is an idea to re-enter each iteration of the Saturation process only with the newest states of the calculated state space so far. This reduces the revisits of same states, since they are already handled and hopefully this saves time. It is worth investigating this solution, since the evaluation revealed that option sat-ddd is the only reachability option that has a relatively high average percentage of useful MDD projections performed, namely approximately 78 %, compared to all other reachability options.

For the General On-the-fly Saturation algorithm as implemented in option sat-ciardo, the improved version already provides a much better time performance. During the project a completely different direction for the Saturation process has been mentioned and arose from work done in a previous Master project that introduced “guards” into the LTSmin toolset [19]. The point is to incorporate conjunctive partitioning into the transition groups which is capable of distinguishing between enabling and updating conjuncts within a transition group or event (as briefly explained in section 2.3). The “guards” come into play when trying to assess if enabling conjuncts of a particular transition group is evaluated positively (e.g. to true). If this is the case, then the transition is taken by evaluating the updating conjuncts of the transition group. For this to work, one needs to readjust the data objects dealing with the transition groups, namely the vector relation and make adjustments in the functions dealing with the relational product of the Saturation process.

The final three directions for improvement can be implemented for both General Saturation algorithms.

The first direction is the order in which a transition group is taken from the list of transition groups that start at a certain topmost MDD level when starting a fix-point computation (e.g. line 27 in Listing 4.1). Currently the algorithm simply takes the transition groups in the order given as they are inserted in this list. The number of MDD levels that each transition group affects, differs individually and the question arises if this has influence on the performance of the algorithm. A suggestion is that it might be convenient to order the transition groups in the list in decreasing number of MDD levels that it affects. In addition to this, another way is to sort the rows of the PINS dependency matrices (which correspond to the transition groups) in decreasing number of MDD levels that it affects. The fix-point computation will then be started with transition groups that run over lots of MDD levels first and it is assumed that this will cause lower parts of the reachable state space to become saturated earlier, and therefore reducing effort in calculating it. This is still a hypothesis and needs further investigation.

The second direction is to look at the possibility of making use of the information stored in the PINS dependency matrix of a certain model. As explained in section 3.2.2 each matrix cell can contain symbols indicating if a certain state slot of a transition group is only read, only written, both read and written or none of these actions. Currently the Saturation algorithms do not make a distinction among the first three types of symbols (it does not take into account if a certain state slot is only read or only written). It might be worthwhile to investigate the possibilities to distinguish these types of symbols in the PINS dependency matrix, which might result in handling MDD levels more efficiently.
This direction should also be seen in light of the aforementioned proposal to use “guards”, because enabling conjuncts are represented by state slots that are only read and updating conjuncts are those state slots that can either be only written or both read and written.

The third direction lies in adjusting the metric that is being used for calculating the best variable ordering resulting in a PINS dependency matrix where affected MDD levels lie more closely together. This prevents transition groups to run over a large range of MDD levels with intermediate identity transitions in between, which results in a bad performance of the Saturation algorithm. Generally, the metric function tries different variable orderings and for each a value is calculated that indicates how closely located the affected MDD levels lie next to each other in the PINS dependency matrix. A low value indicates a good variable ordering and the computation is performed multiple times until it assumes that it has found the lowest value with accompanying variable ordering.
For this purpose it is possible to change the metric to the one devised by G. Ciardo in [21]. In this case the Saturation algorithms are left unchanged and one only has to change the metric function that is being used for computing the best variable ordering.

Bibliography

[1] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, P. Šimeček. DIVINE – A tool for distributed verification. In *Proc. 18th International Conference on Computer Aided Verification (CAV 2006)*, August 2006, Seattle, Washington, USA. pp. 278-281 (*Lecture Notes in Computer Science*, vol. 4144), 2006.

[2] S.C.C. Blom, J.R. Calamé, B. Lisser, S. Orzan, J. Pang, J.C. van de Pol, M.T. Dashti and A.J. Wijs. Distributed Analysis with µCRL: a Compendium of Case Studies. In *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, March – April 2007, Braga, Portugal. pp. 683-689 (*Lecture Notes in Computer Science*, vol. 4424), 2007.

[3] S.C.C. Blom and J.C. van de Pol. Symbolic Reachability for Process Algebras with Recursive Data Types. In *Proc. 5th International Colloquium on Theoretical Aspects of Computing - ICTAC 2008*, September 2008, Istanbul, Turkey. pp. 81-95 (*Lecture Notes in Computer Science*, vol. 5160), 2008.

[4] 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 (CTIT), University of Twente, Enschede, The Netherlands, 2009.

[5] S.C.C. Blom, J.C. van de Pol and M. Weber. LTSmin: Distributed and Symbolic Reachability. In *Proc. 22nd International Conference on Computer Aided Verification (CAV 2010)*, July 2010, Edinburgh, UK. pp. 354-359 (*Lecture Notes in Computer Science*, vol. 6174), 2010.

[6] M.Y. Chung, G. Ciardo, R.L. Jones III, R.M. Marmorstein, A.S. Miner, R.I. Siminiceanu and A. Yu. SMART, Stochastic Model checking Analyzer for Reliability and Timing, Version 1.1. User manual, Department of Computer Science and Engineering, University of California, Riverside, California, USA, 2006. Website of manual (retrieved in March 2011): http://www.cs.ucr.edu/~ciardo/SMART/SMARTman.pdf.

[7] G. Ciardo, G. Luettgen and A.S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, vol. 31(1). pp. 63-100, 2007.

[8] G. Ciardo, G. Luettgen and R. Siminiceanu. Saturation: an efficient iteration strategy for symbolic state-space generation. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, April 2001, Genova, Italy. pp. 328-342 (*Lecture Notes in Computer Science*, vol. 2031), 2001.

[9] G. Ciardo, R. Marmorstein and R. Siminiceanu. Saturation unbound. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, April 2003, Warsaw, Poland. pp. 379-393 (Lecture Notes in Computer Science, vol. 2619), 2003.

[10] G. Ciardo, R. Marmorstein and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 8(1). pp. 4-25, 2006.

[11] G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Proc. 13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, October 2005, Saarbrücken, Germany. pp. 146-161 (*Lecture Notes in Computer Science*, vol. 3725), 2005.

[12] M. de Jonge and T.C. Ruys. The SpinJa model checker. In *Proc. 17th International SPIN Workshop on Model Checking Software (SPIN 2010)*, September 2010, Enschede, The Netherlands. pp. 124-128 (*Lecture Notes in Computer Science*, vol. 6349), 2010.

[13] Formal Methods and Tools (FMT). dve2-reach(1) Manual Page. Website (retrieved in December 2011): http://fmt.cs.utwente.nl/tools/ltsmin/dve2-reach.html

[14] J.F. Groote, J.J.A. Keiren, A.H.J. Mathijssen, B. Ploeger, F.P.M. Stappers, C. Tankink, Y.S. Usenko, M.J. van Weerdenburg, J.W. Wesselink, T.A.C. Willemse and J. van der Wulp. The mCRL2 Toolset. In *Proc. 1st International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, July 2008, Paphos, Cyprus, 2008.

[15] A. Hamez, Y. Thierry-Mieg and F. Kordon. Hierarchical Set decision Diagrams and Automatic Saturation. In *Proc. 29th International Conference on Applications and Theory of Petri Nets (PETRI NETS 2008)*, June 2008, Xi'an, China. pp. 211-230 (*Lecture Notes in Computer Science*, vol. 5062), 2008. Website of library package (retrieved in September 2011): http://move.lip6.fr/software/DDD/.

[16] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. 7th IFIP WG6.1 International Conference on Formal Description Techniques (FORTE 1994)*, October 1994, Berne, Switzerland. pp. 197-211, 1995.

[17] J. Lind-Nielsen. BuDDy – A Binary decision Diagram Package. Website of library package (retrieved in September 2011): http://sourceforge.net/projects/buddy/.

[18] A.S. Miner. Saturation for a General Class of Models. *IEEE Transactions on Software Engineering*, vol. 32(8). pp. 559-570, 2006.

[19] E. Pater. Partial Order Reduction for PINS. Master thesis, Formal Methods and Tools (FMT), University of Twente, Enschede, The Netherlands, 2011. Website of thesis (retrieved in January 2012): http://fmt.cs.utwente.nl/education/master/17/

[20] R. Pelánek. BEEM: BEnchmarks for Explicit Model Checkers. In *Proc. 14th International SPIN Workshop on Model Checking Software (SPIN 2007)*, July 2007, Berlin, Germany. pp. 263-267 (*Lecture Notes in Computer Science*, vol. 4595), 2007. Website (retrieved in August 2011): http://anna.fi.muni.cz/models/.

[21] R. Siminiceanu and G. Ciardo. New metrics for static variable ordering in decision diagrams. In *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*, March – April 2006, Vienna, Austria. pp. 90-104 (*Lecture Notes in Computer Science*, vol. 3920), 2006.

[22] M.G.J. van den Brand and P. Klint. ATerms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, vol. 49(1). pp. 55-64, 2007. Website of library package (retrieved in September 2011): http://www.meta-environment.org/Meta-Environment/ATerms.

[23] M. Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In *Proc. 35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2009)*, February 2009, Spindleruv Mlyn, Czech Republic. pp. 582-594, 2009.

[24] M. Weber. An embedded virtual machine for state space generation. In *Proc. 14th International SPIN Workshop on Model Checking Software (SPIN 2007)*, July 2007, Berlin, Germany. pp. 168-186 (*Lecture Notes in Computer Science*, vol. 4595), 2007.

[25] E.W. Weisstein. Kronecker Product. MathWorld – A Wolfram Web Resource. Website (retrieved in October 2011): http://mathworld.wolfram.com/KroneckerProduct.html.

Saturation for LTSmin

Appendices

Final project (course code 192199978)
Course year 2011 – 2012

Tien Loong Siaw (s0045217)

Master Computer Science
Track Software Engineering

Graduation committee:
- prof. dr. J.C. van de Pol
- dr. J. Ketema
- prof. dr. ir. A. Rensink

Department of Formal Methods and Tools (FMT)
University of Twente

Enschede, Wednesday 15th of February 2012

UNIVERSITY OF TWENTE.

Formal Methods & Tools

A Saturation test results regarding memory performance

In this appendix the test results regarding memory performance is presented. In Table A. 1 the results are given showing the maximum memory values for each of the 27 DVE models per tested reachability option. A summary of these memory result are given in Table A. 2. It should be remarked that the maximum memory results are dependent on the MDD library being used, in this case the ATermDD library.

Model	State space size	Maximum memory usage (median values in KB)						
		bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo	new sat-ciardo
and.6	18206917	31916	79564	55128	71840	2381472	1036392	1274584
at.5	31999440	113360	135444	100516	103768	784740	1318980	1186712
at.6	160589600	315660	377732	1010848	1224884	4537800	9588340	8671452
bak.6	11845035	168308	159564	135772	78680	11350580	8384260	11342784
brp.5	17740267	92632	46840	80272	62592	142900	176260	154680
brp.6	42728113	186872	63844	152792	99708	358636	398032	362080
cam.7	11465015	1147344	584592	958592	579312	3170760	8391940	×
col.4	41465543	431772	424604	56348	59868	46796	57044	84156
ele.2	11428767	265232	185884	15336	15628	11656	10276	13924
fir.5	18553032	136396	65624	34924	34860	197388	223080	183744
ipr.6	41387484	438432	99852	36276	36912	455344	198412	227800
ipr.7	59794192	1833592	319048	65156	64068	1346752	650256	955748
lam.7	38717846	177128	201680	41008	37552	1426404	738132	643172
lam.8	62669317	190952	990836	53188	45200	3314664	1495296	1338596
lna.5	95118524	29312	52004	20604	20784	217668	274752	213788
lann.6	144151628	4501196	1083116	327228	499764	2399748	3657652	3021900
lann.7	160025986	76060	70536	30224	29168	114644	131172	114288
le.6	35777100	57384	21440	20820	21528	33944	23372	2060788
lf.7	26302351	175368	158548	95944	97560	10865808	7731500	8147268
pet.6	174495861	100888	869128	52380	34608	1218972	993480	1143568
pet.7	142471098	225672	190944	131656	96400	13277304	10494116	10330492
phi.6	14348906	15632	18688	8372	5064	5060	5064	5048
phi.8	43046720	15708	18824	7896	6956	6536	6756	4340
sch.3	166649331	1793212	92104	19028	18840	98468	195572	184748
szy.5	79518740	129300	132076	53388	34844	6540184	5991024	5417616
tel.4	12291552	73476	52404	42248	28672	799316	741352	××
tel.7	21960308	103272	105744	31196	30628	1562928	1718724	××

Legend of symbols			
×	No data due to (possible) infinite looping.	red	Worst performing option for a certain model.
××	No data due to error (e.g. segmentation fault).	green	Best performing option for a certain model.

Table A. 1: Experimental maximum memory results using the reachability tools from LTSmin on the 27 DVE models, including option new sat-ciardo.

	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo	new sat-ciardo
# times as best	2	2	6	13	1	1	2
# times as worst	5	2	0	0	10	9	1

Table A. 2: Summary of the best and worst performing reachability options regarding maximum memory performance, including option new sat-ciardo.

On the following pages some plots are presented where reachability options are compared with each other. In Figure A. 1 till Figure A. 3 option sat-ciardo (at vertical axis) is compared with the options no-sat, sat-like and sat-ddd, respectively. In Figure A. 4 and Figure A. 5 this is done for option sat-ddd (vertical axis) versus options no-sat and sat-like. Finally in Figure A. 6 till Figure A. 9 plots are presented, comparing option new sat-ciardo with options no-sat, sat-like, sat-ddd and the previous version of sat-ciardo, respectively.

In all these plots, the plot on the left gives a sense where every result is located in a larger picture. The plot in the upper right corner is a magnification of a small part of the leftmost plot and the plot on the lower right corner is on its turn a magnification of the plot on the upper right corner. One should be aware that with these magnifications the horizontal and vertical axes scale as well.
Furthermore a diagonal is added in these plots to show that the two reachability options under comparison have the same time result. When a time result falls beneath the diagonal line in Figure A. 1 until Figure A. 3, then this means that option sat-ciardo computes with less maximum memory than the other reachability option. Likewise for memory results in Figure A. 4 and Figure A. 5, in which case option sat-ddd is performing better than the other reachability option and for memory results in Figure A. 6 until Figure A. 9 for option new sat-ciardo.

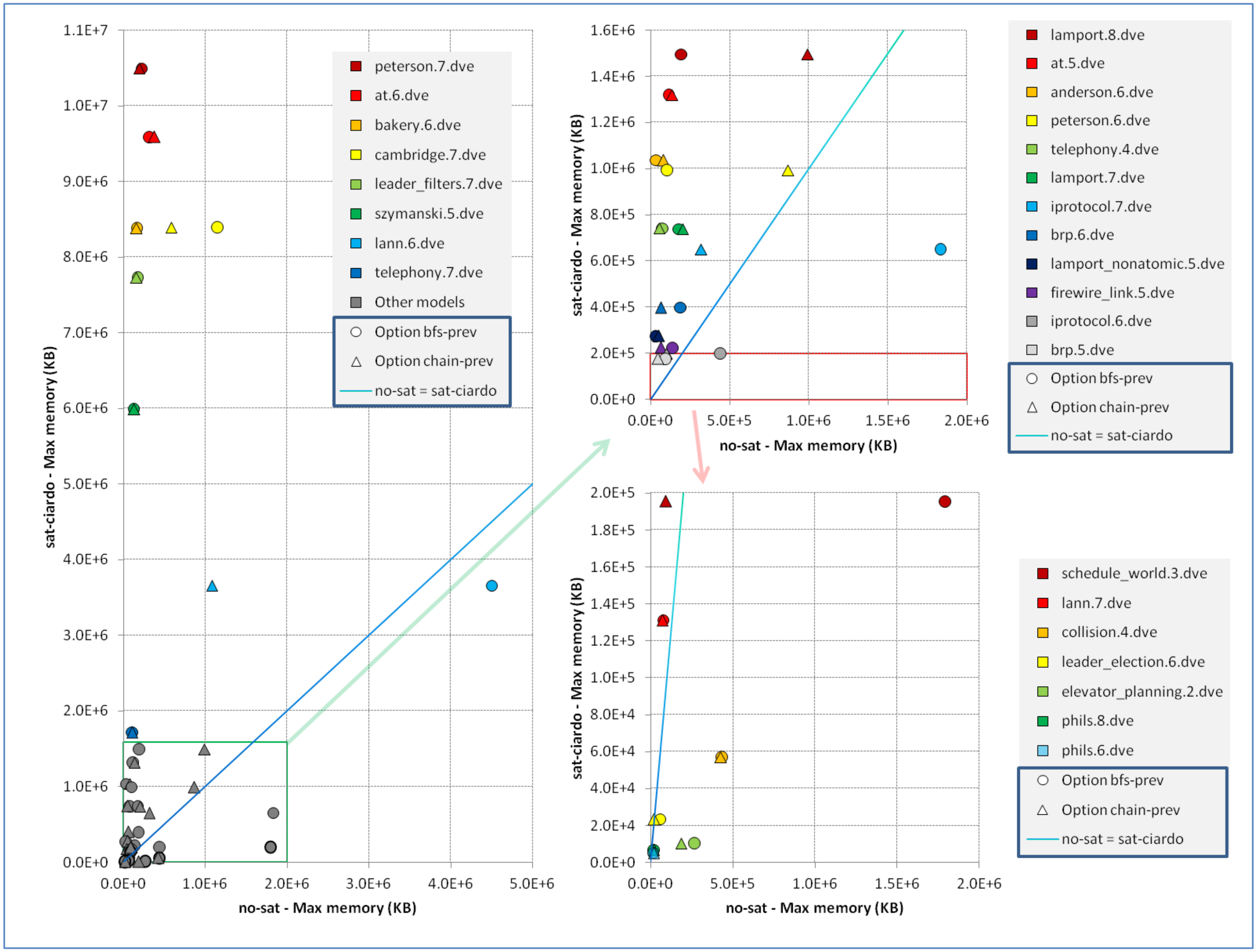


Figure A. 1: Plot comparing maximum memory values between options no-sat and sat-ciardo.

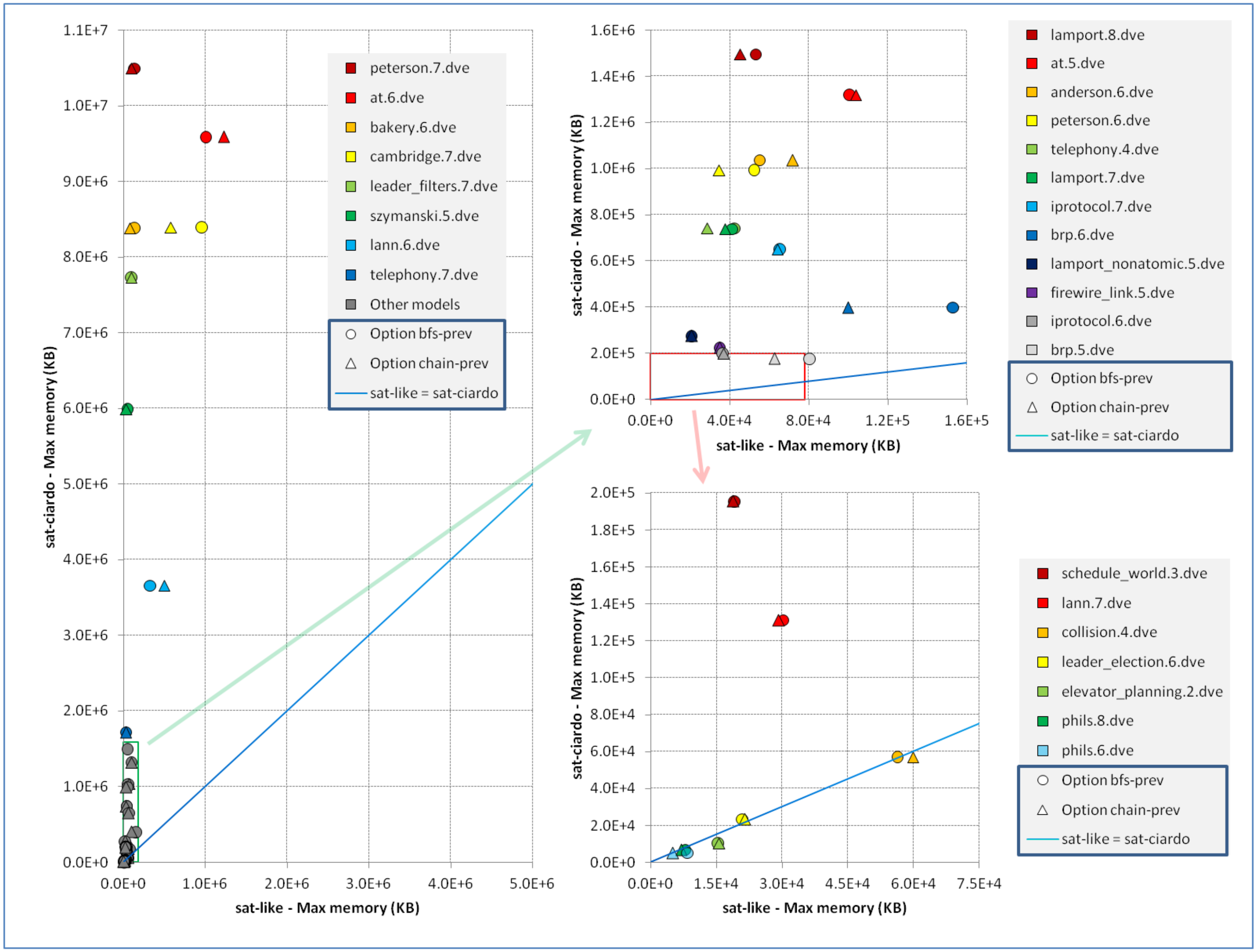


Figure A. 2: Plot comparing maximum memory values between options sat-like and sat-ciardo.

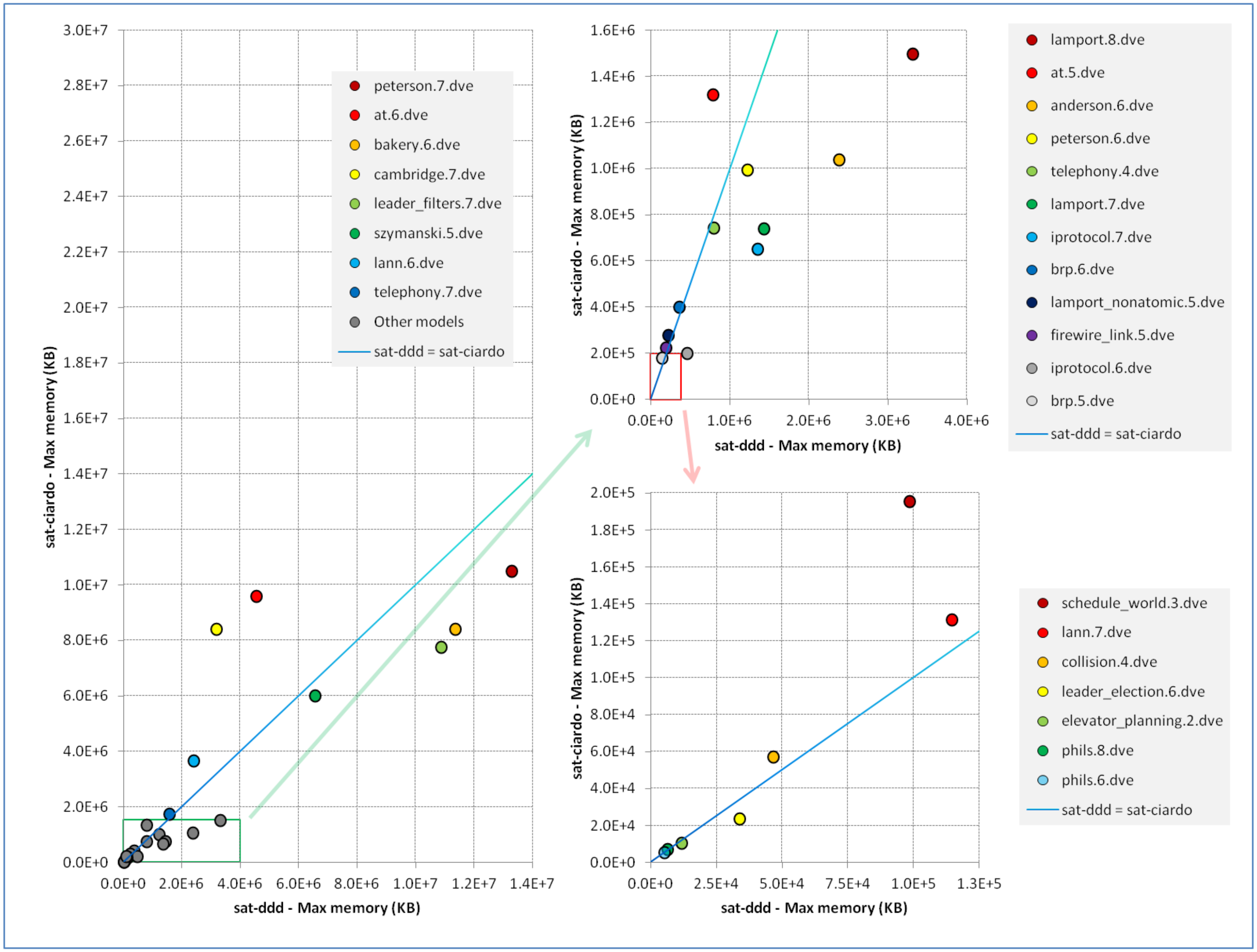


Figure A. 3: Plot comparing maximum memory values between options sat-ddd and sat-ciardo.

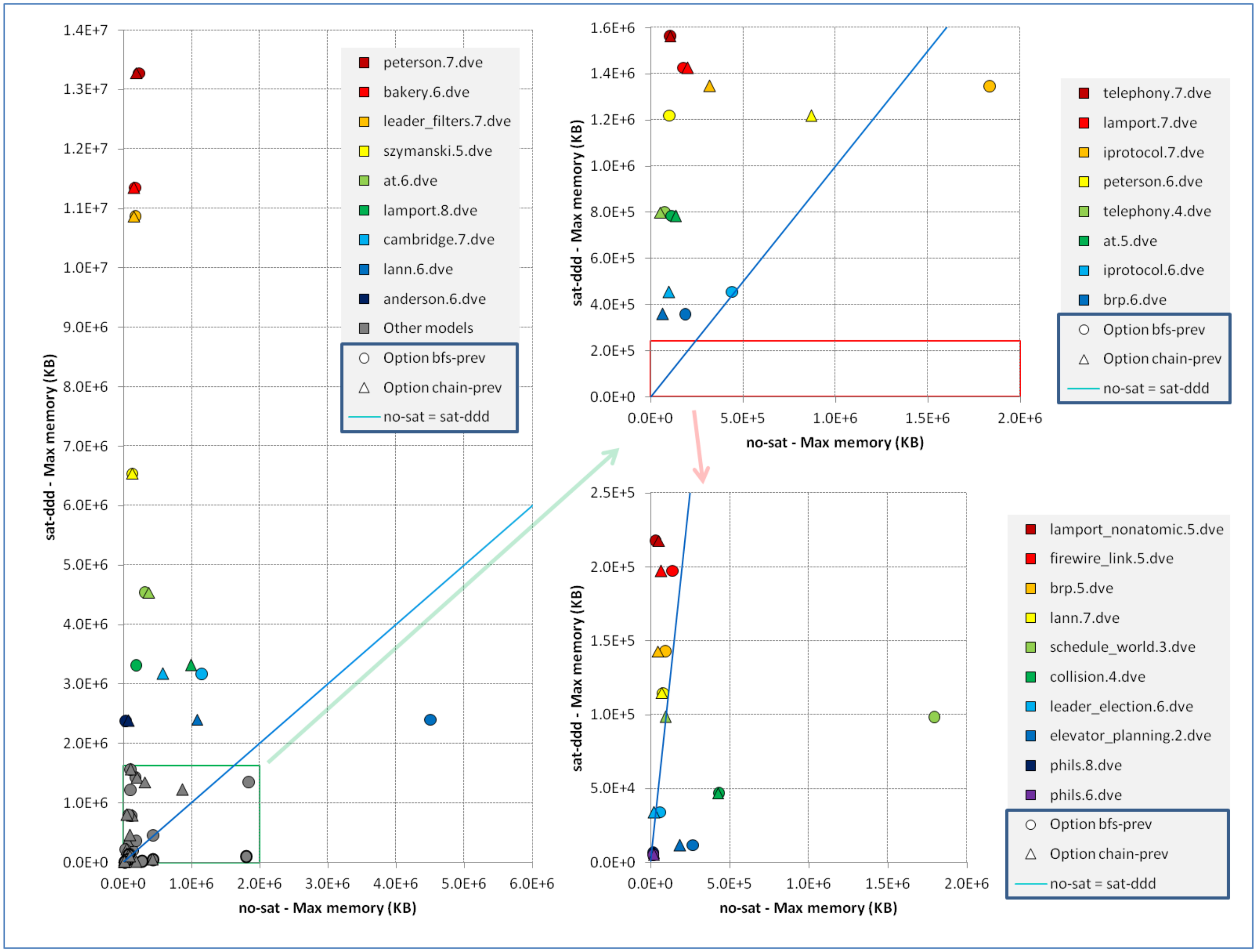


Figure A. 4: Plot comparing maximum memory values between options no-sat and sat-ddd.

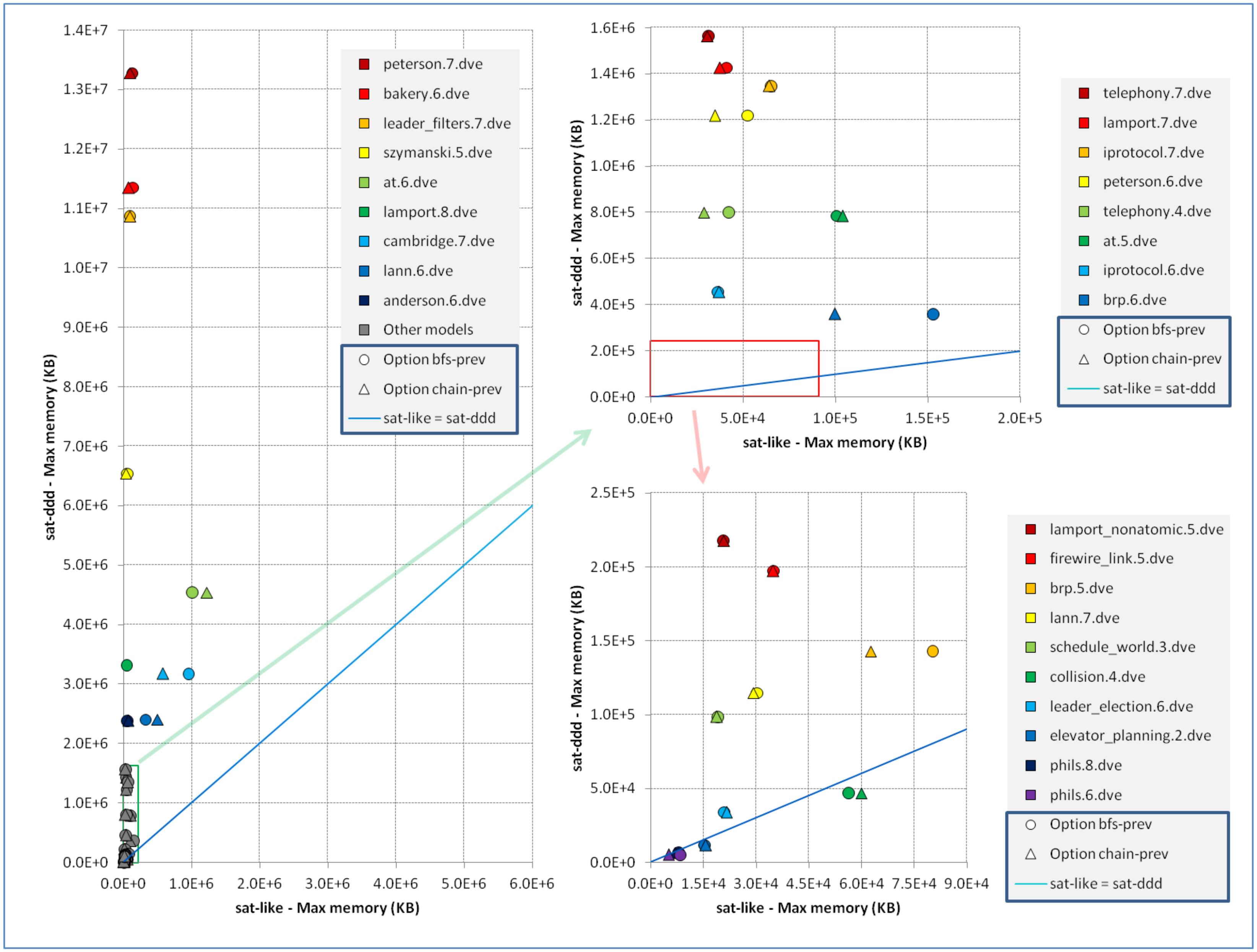


Figure A. 5: Plot comparing maximum memory values between options sat-like and sat-ddd.

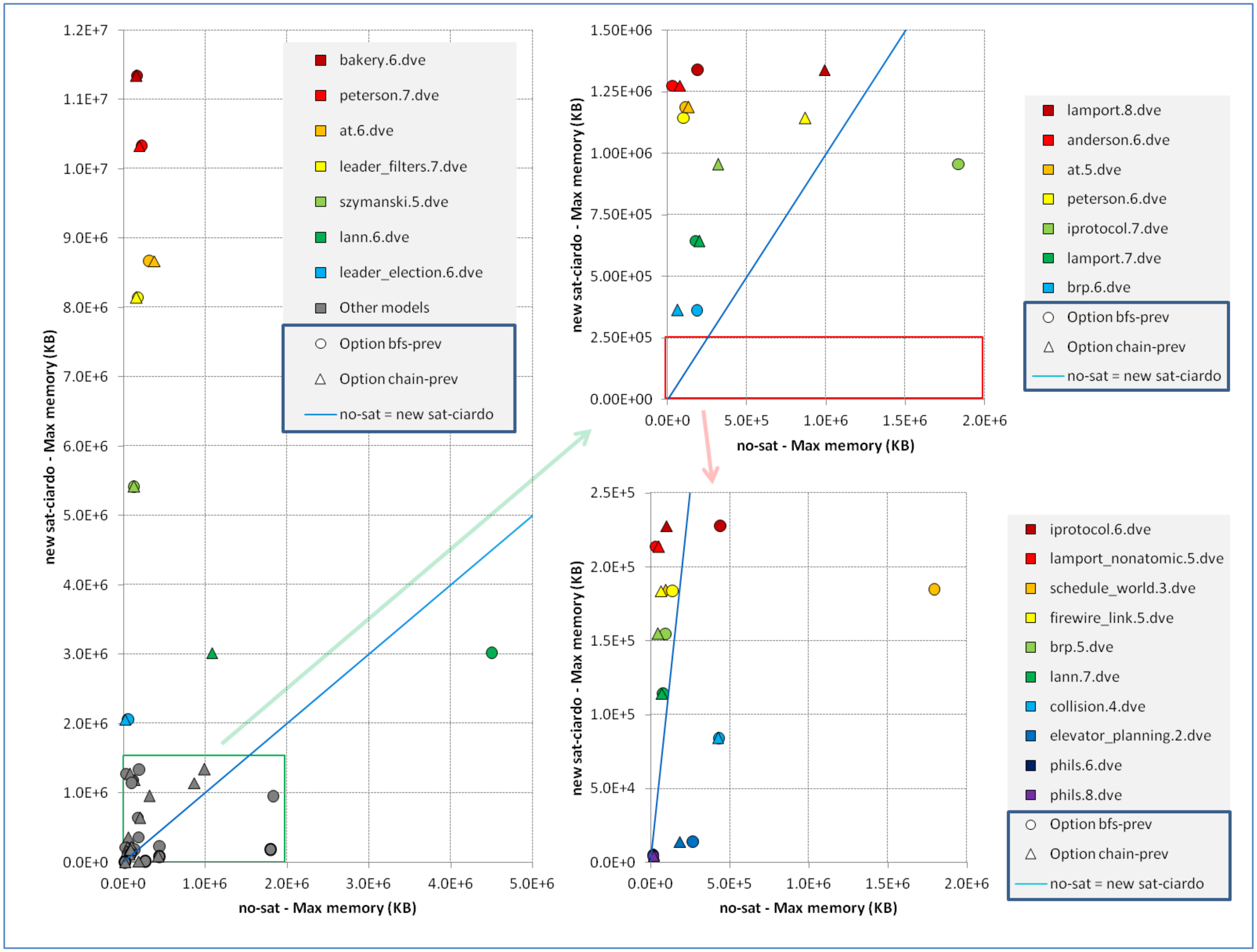


Figure A. 6: Plot comparing maximum memory values between options no-sat and new sat-ciardo.

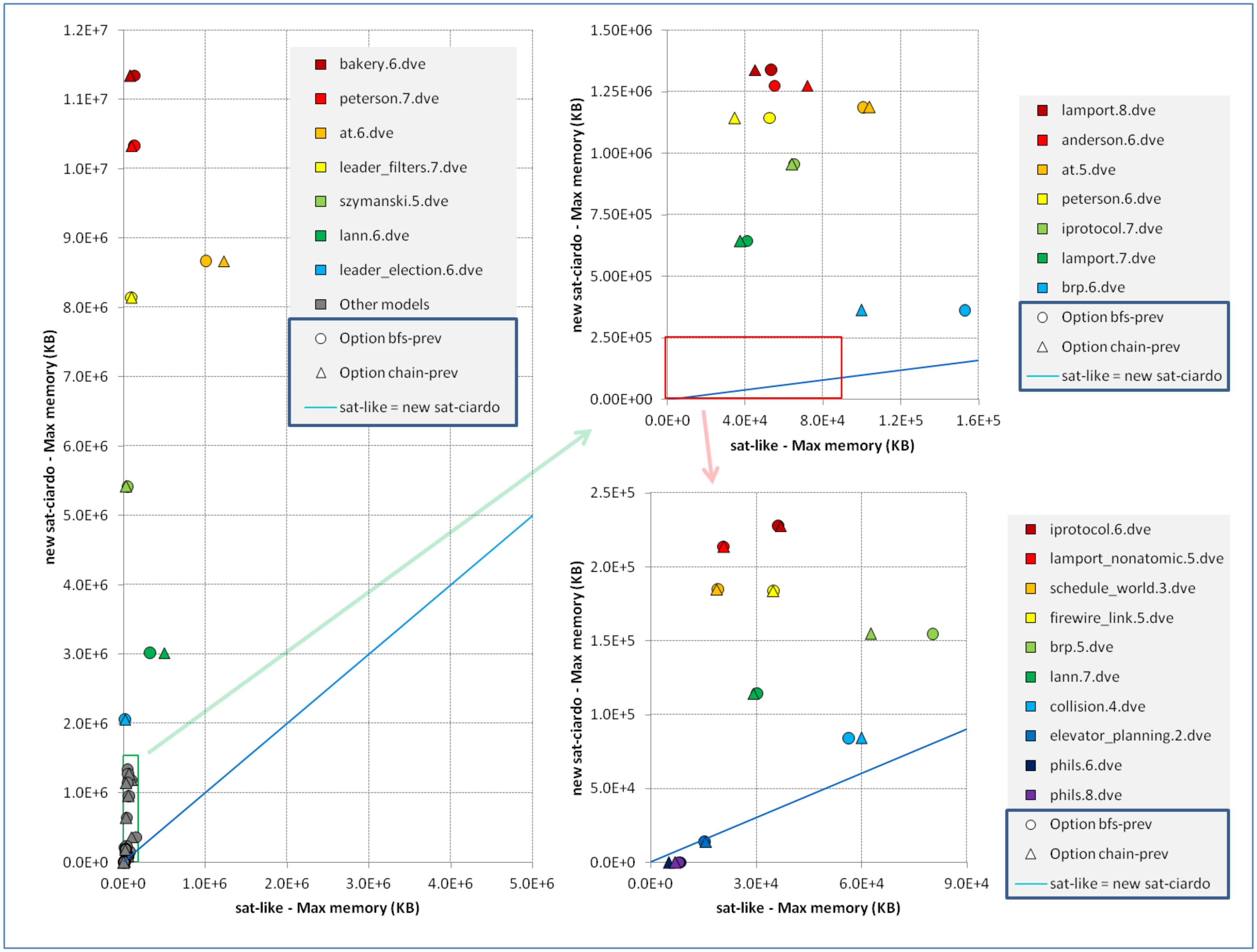


Figure A. 7: Plot comparing maximum memory values between options sat-like and new sat-ciardo.

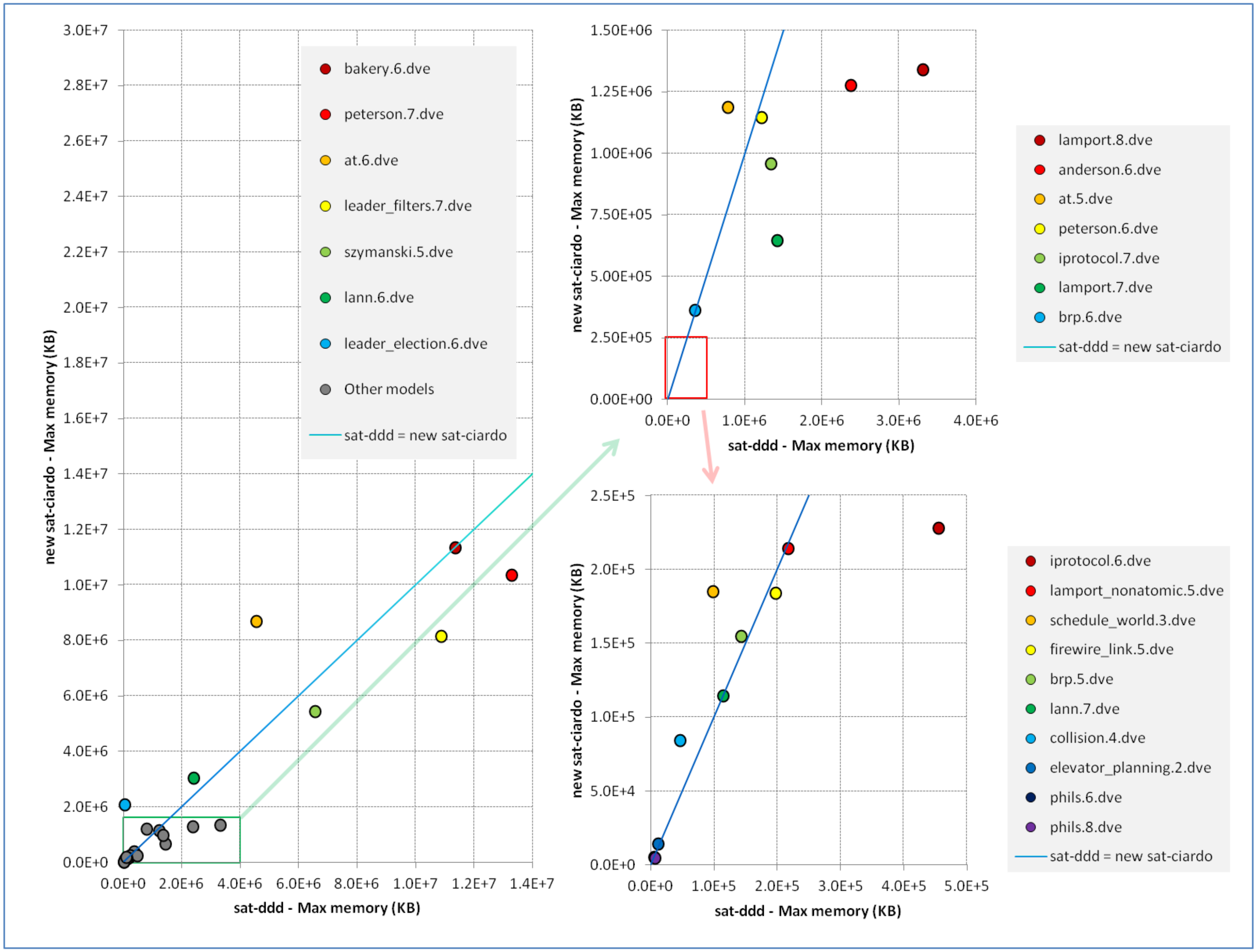


Figure A. 8: Plot comparing maximum memory values between options sat-ddd and new sat-ciardo.

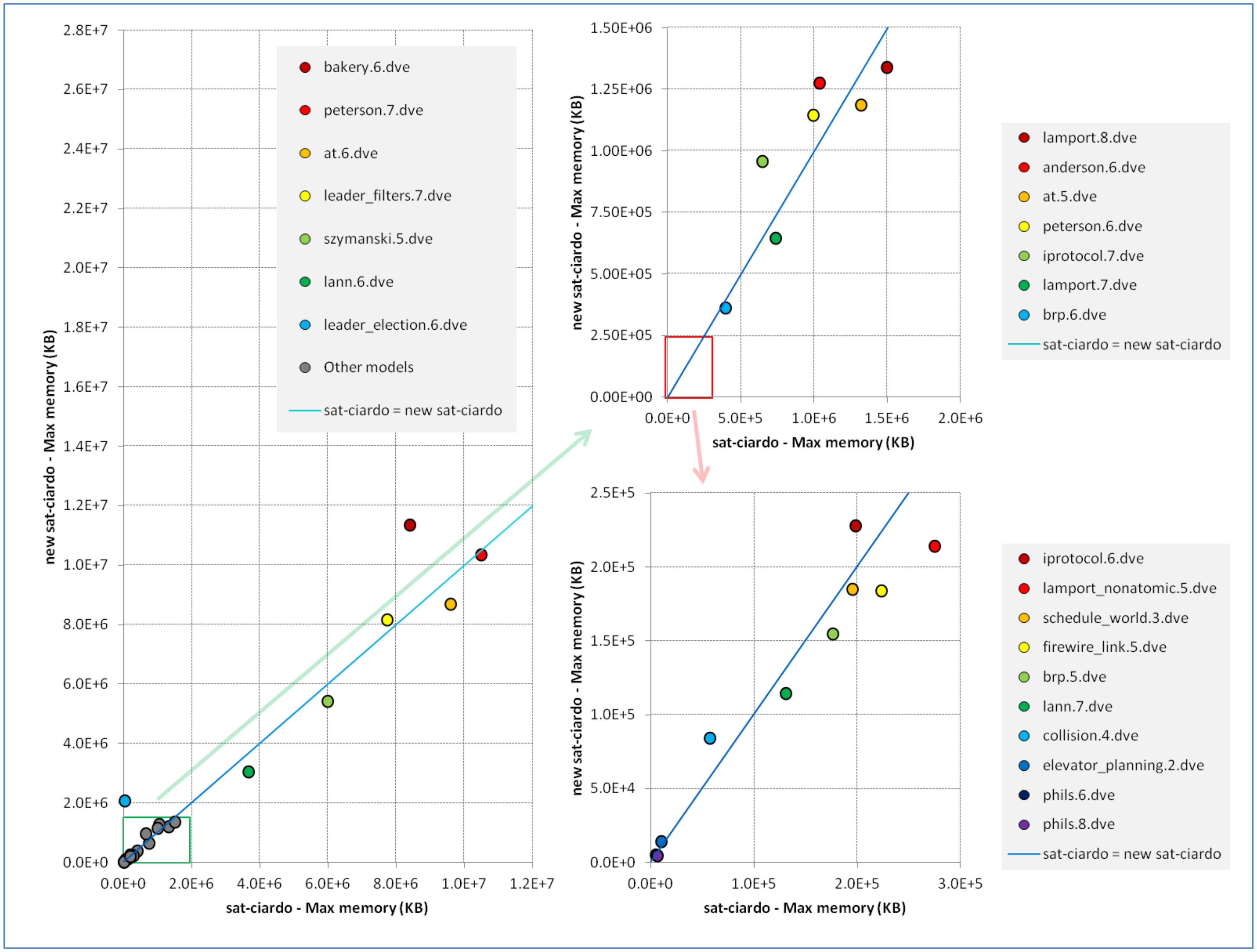
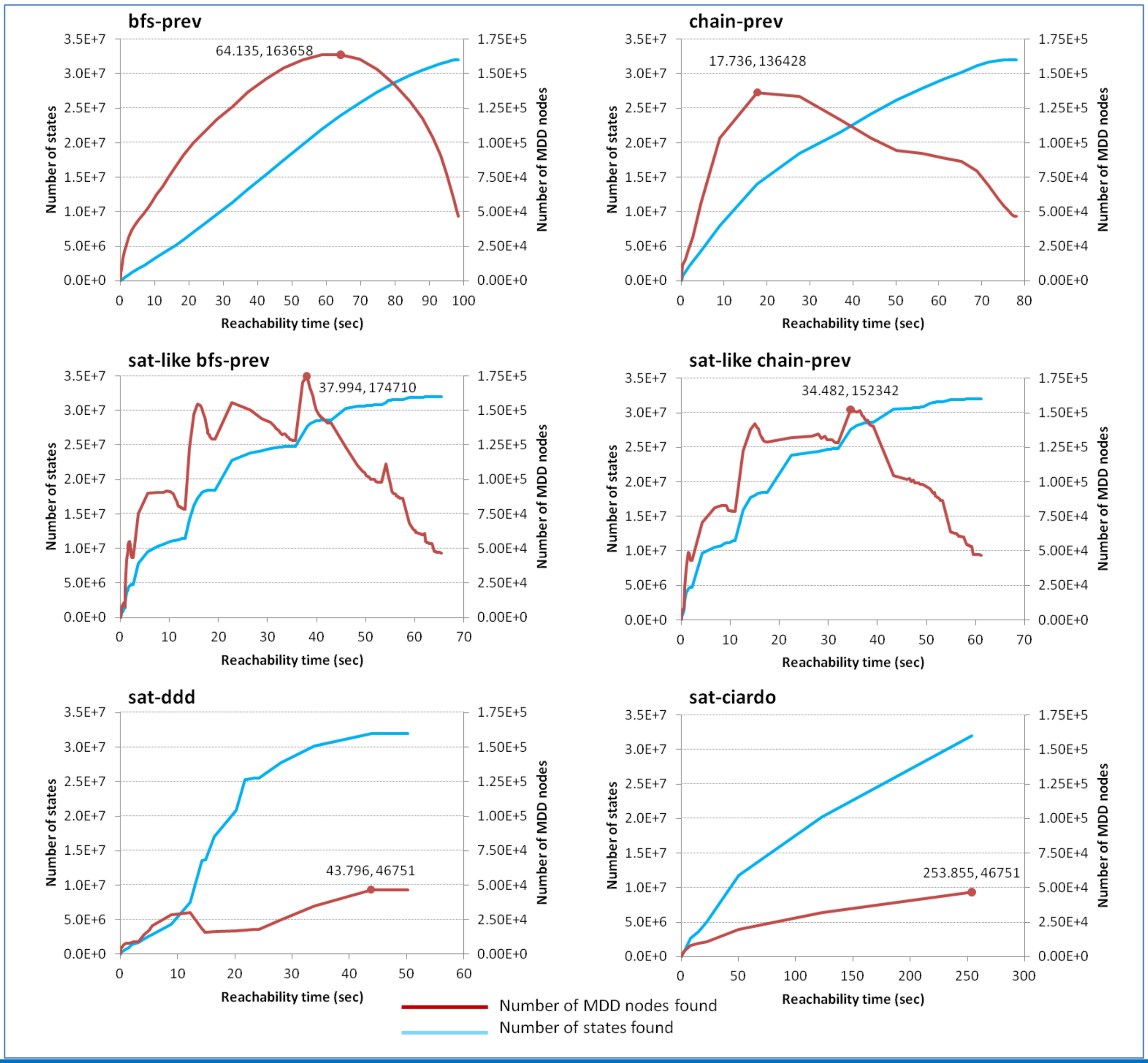


Figure A. 9: Plot comparing maximum memory values between options sat-ciardo and new sat-ciardo.

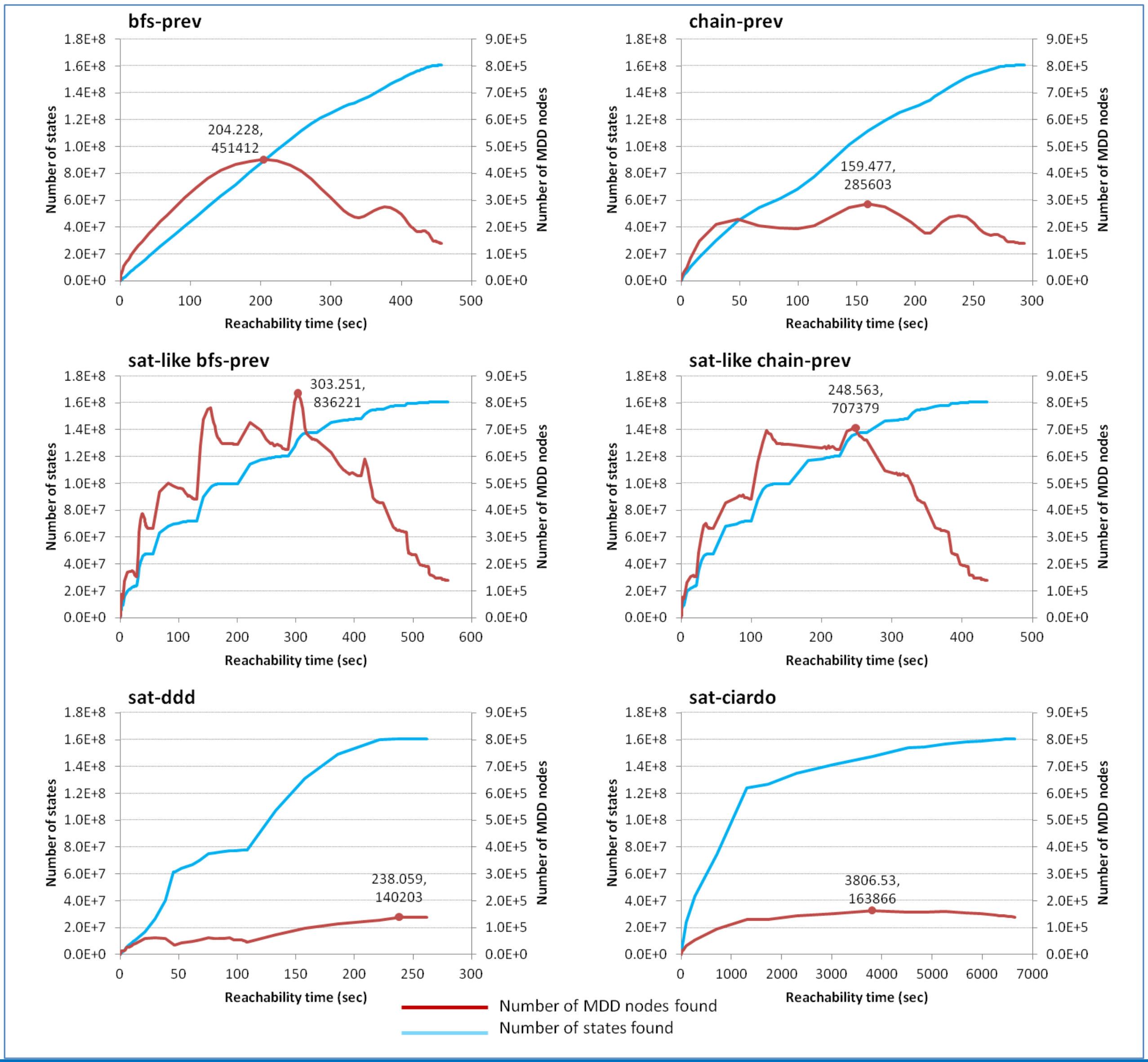
B Saturation evolution plots

In this appendix the rest of the evolution plots are shown per DVE model, to give an impression how the number of MDD nodes and the state space size evolves over time. In some plots one should be aware that the vertical axis for the number of MDD nodes can differ (this is done for visibility reasons). And also the horizontal time axis differs for each reachability option in the set of evolution plots of a certain model.



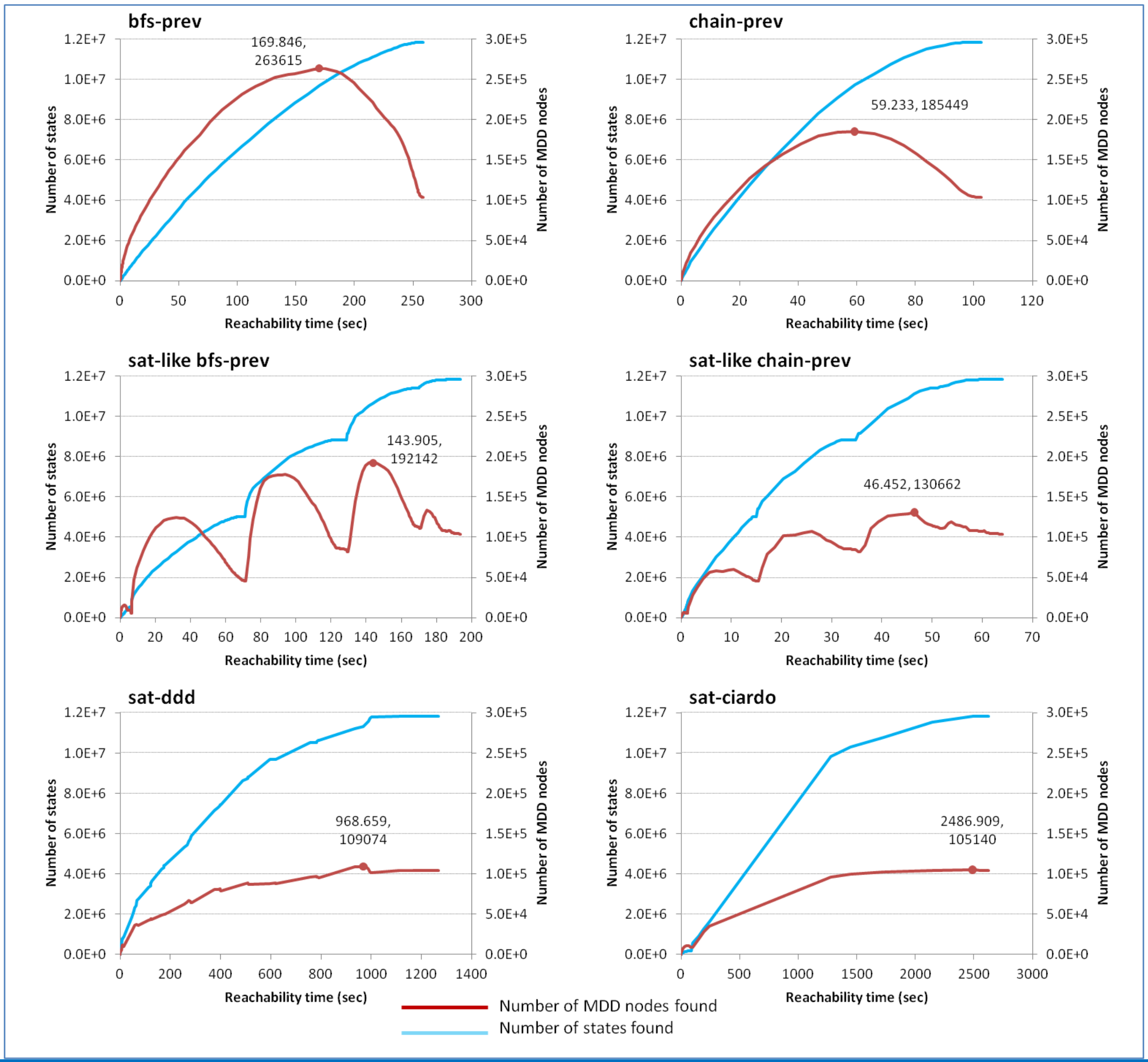
Final state space size	Final MDD node size	Fastest reachability option
31999440	46751	sat-ddd

Figure B. 1: State space evolution plots of DVE model at.5.



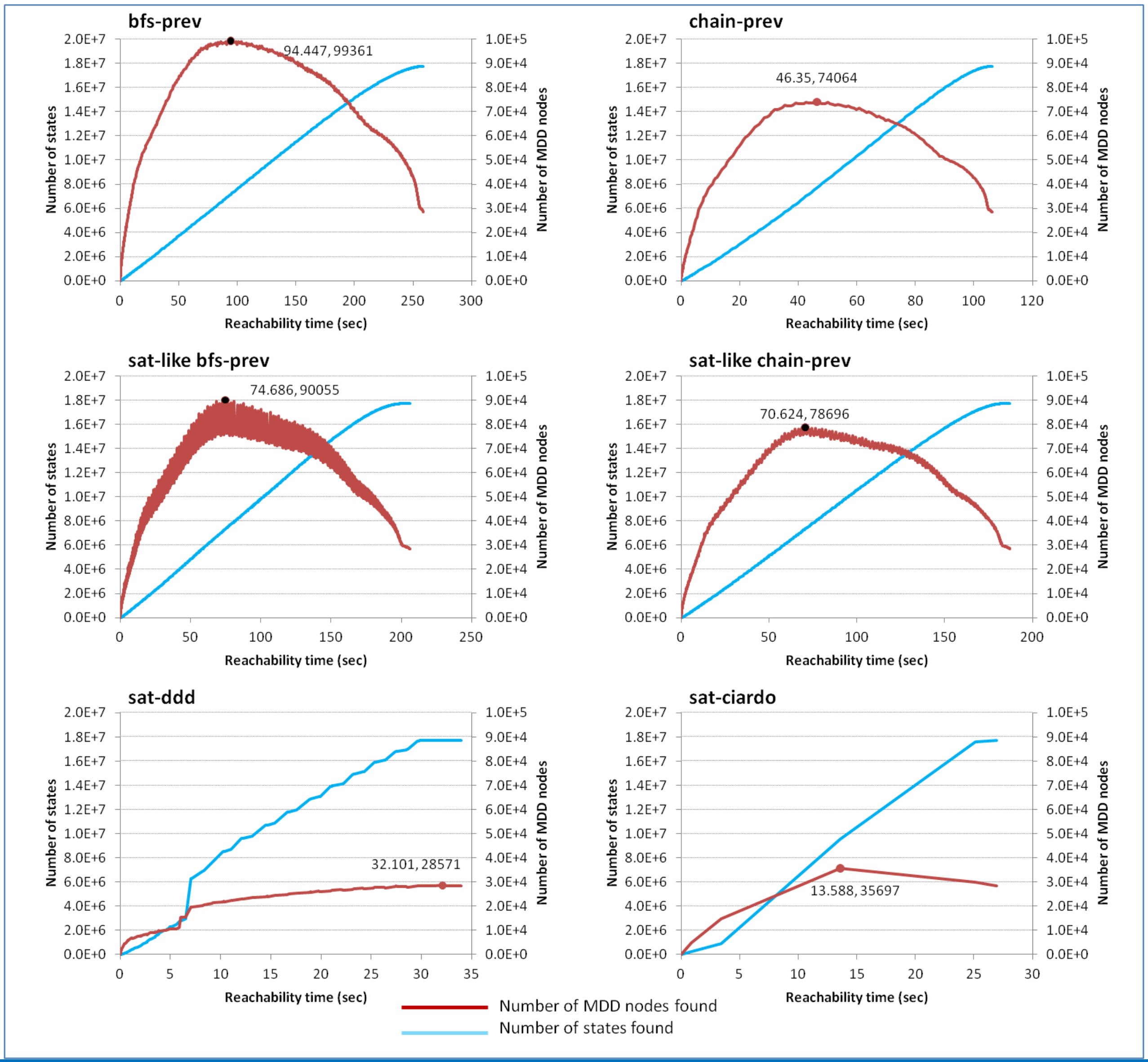
Final state space size	Final MDD node size	Fastest reachability option
160589600	138937	sat-ddd

Figure B. 2: State space evolution plots of DVE model at.6.



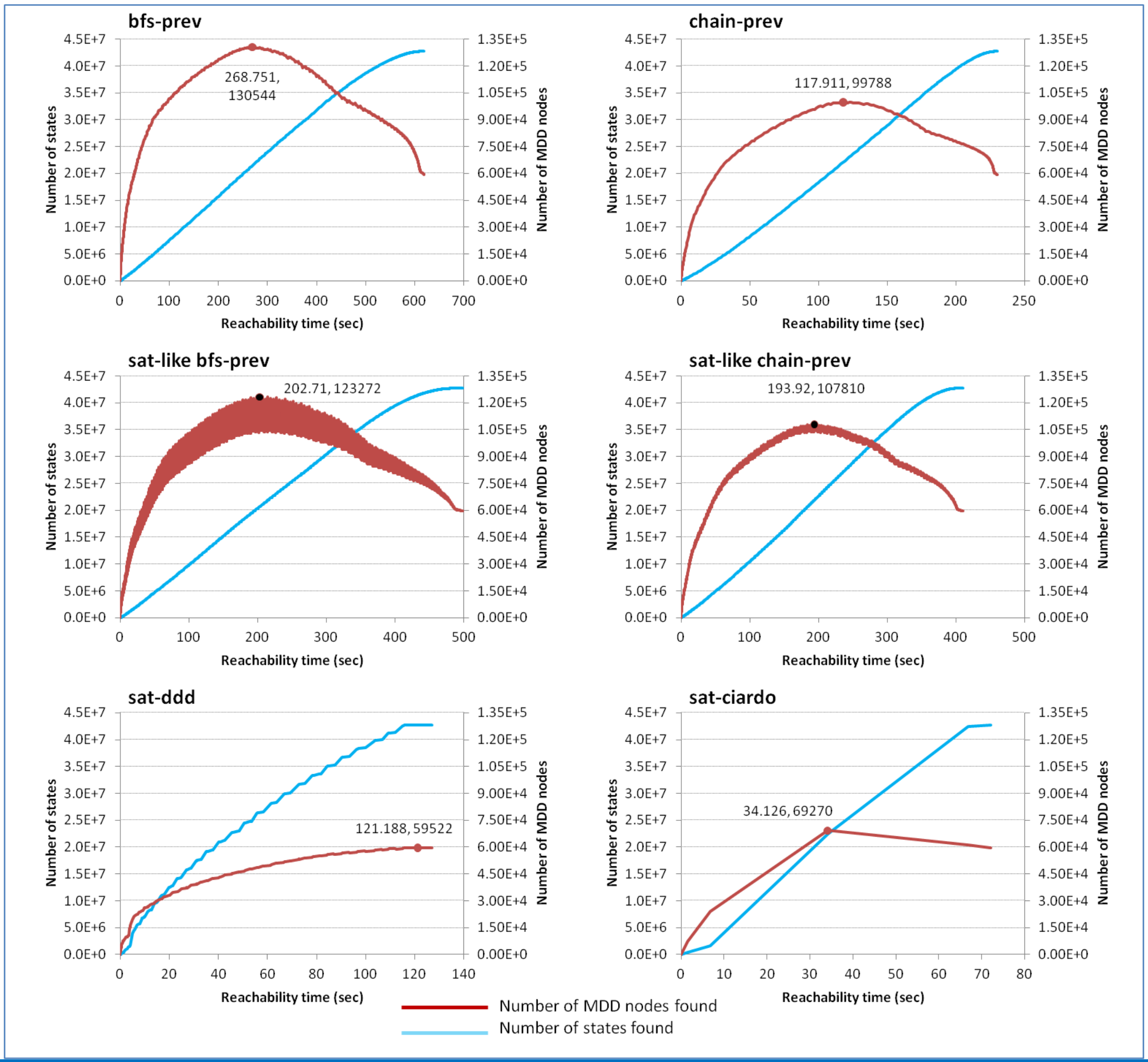
Final tate space size	Final MDD node size	Fastest reachability option
11845035	103954	sat-like chain-prev

Figure B. 3: State space evolution plots of DVE model bakery.6.



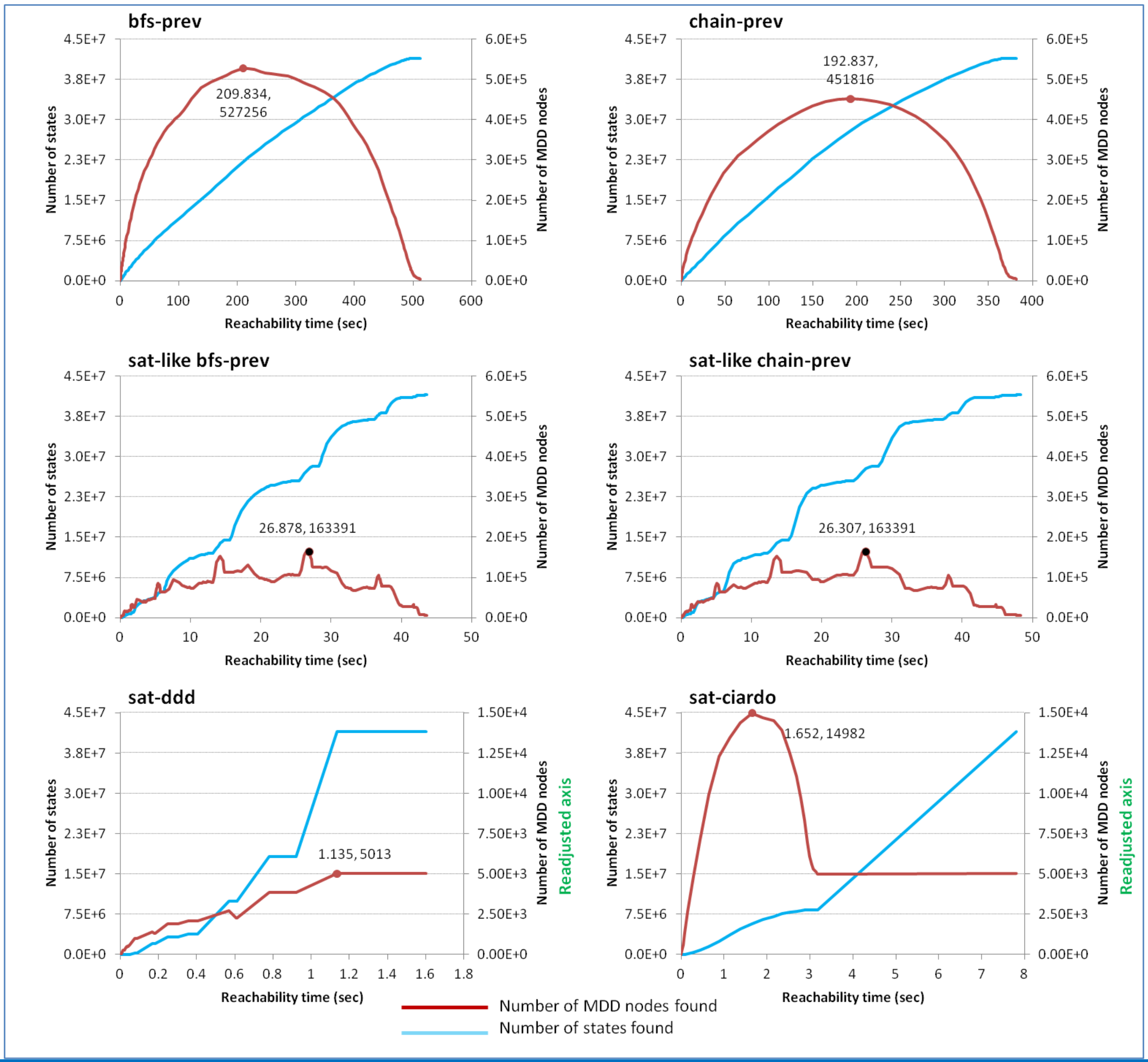
Final state space size	Final MDD node size	Fastest reachability option
17740267	28500	sat-ciardo

Figure B. 4: State space evolution plots of DVE model brp.5.



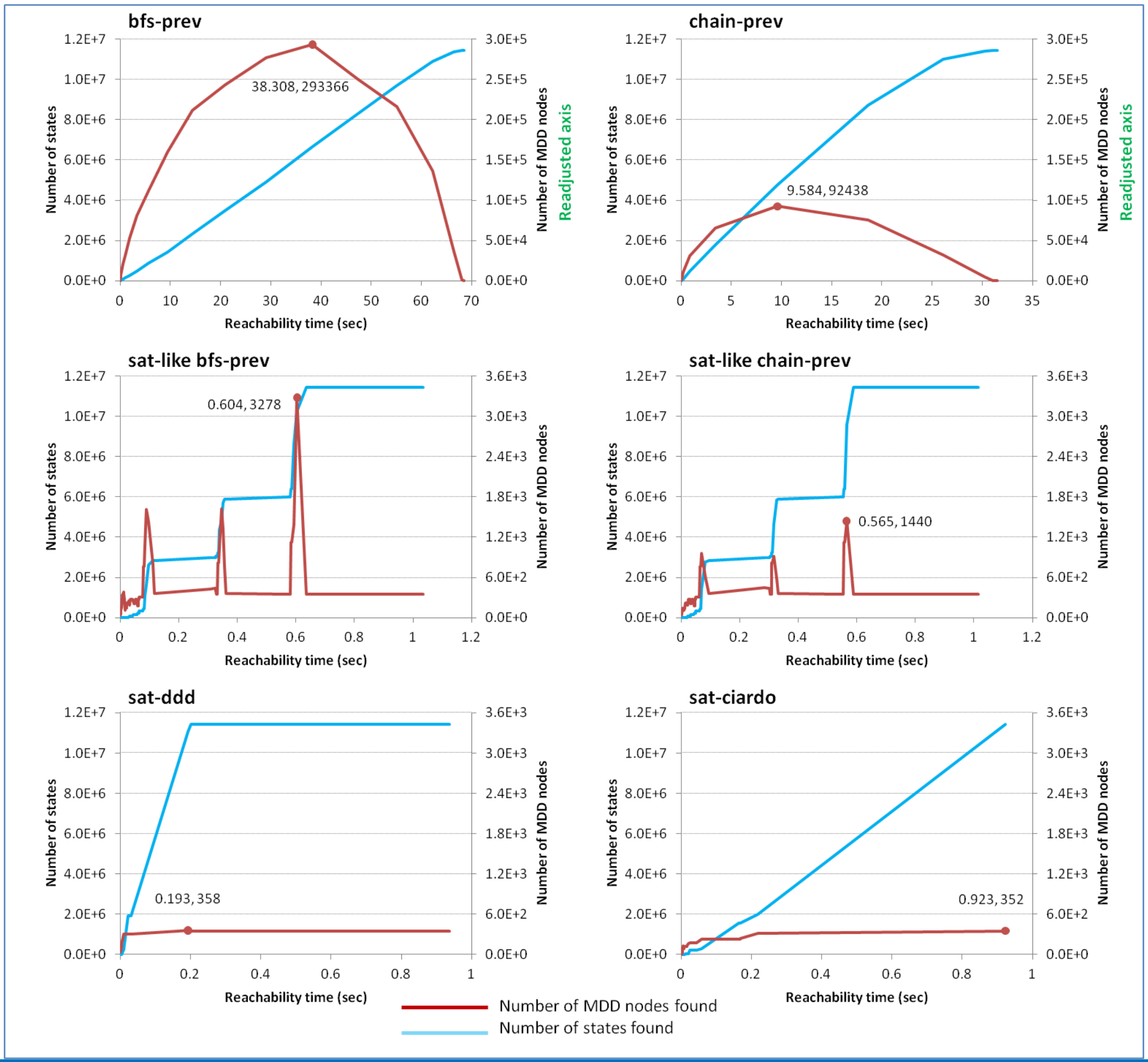
Final state space size	Final MDD node size	Fastest reachability option
42728113	59465	sat-ciardo

Figure B. 5: State space evolution plots of DVE model brp.6.



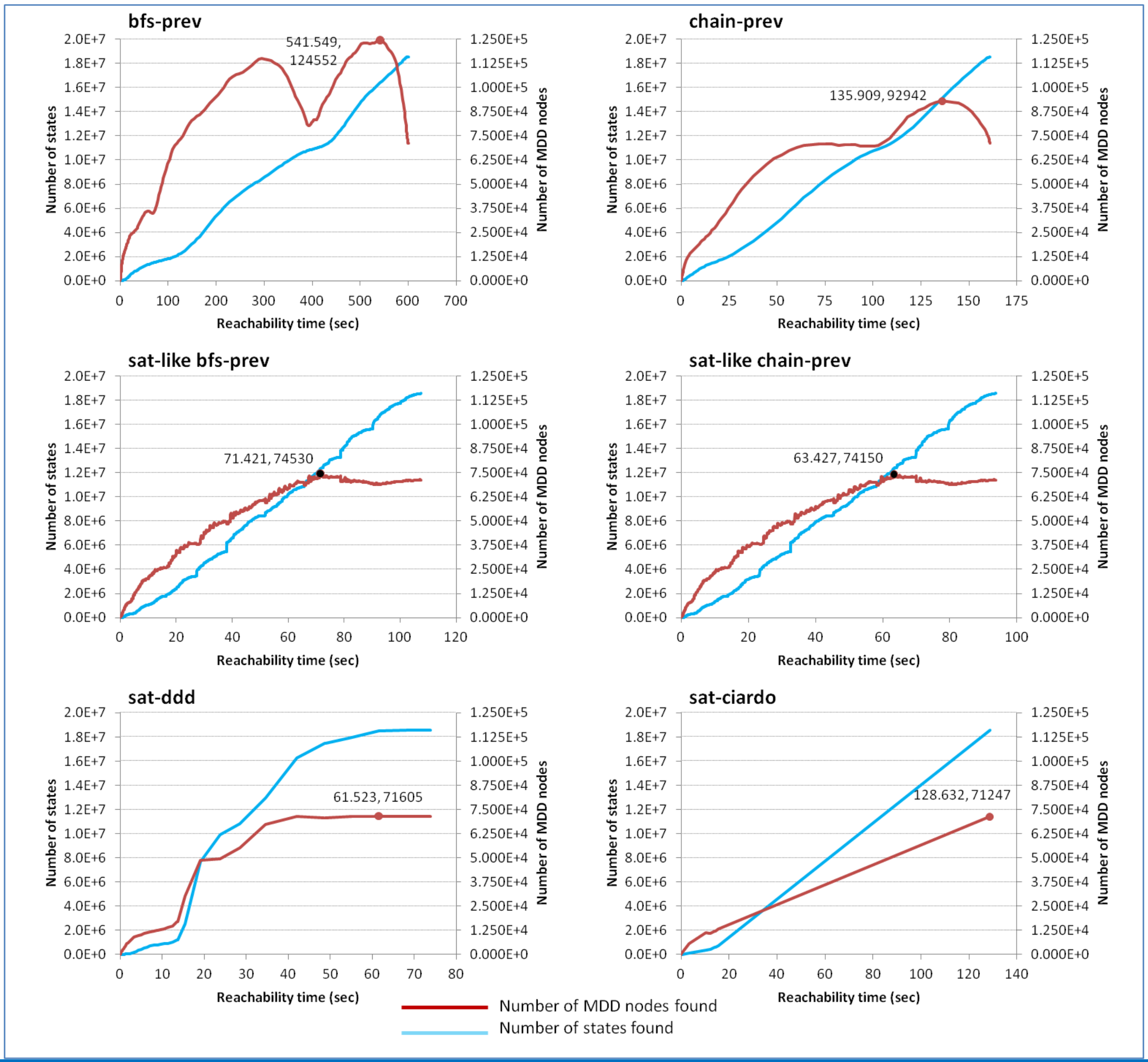
Final state space size	Final MDD node size	Fastest reachability option
41465543	5013	sat-ddd

Figure B. 6: State space evolution plots of DVE model collision.4.



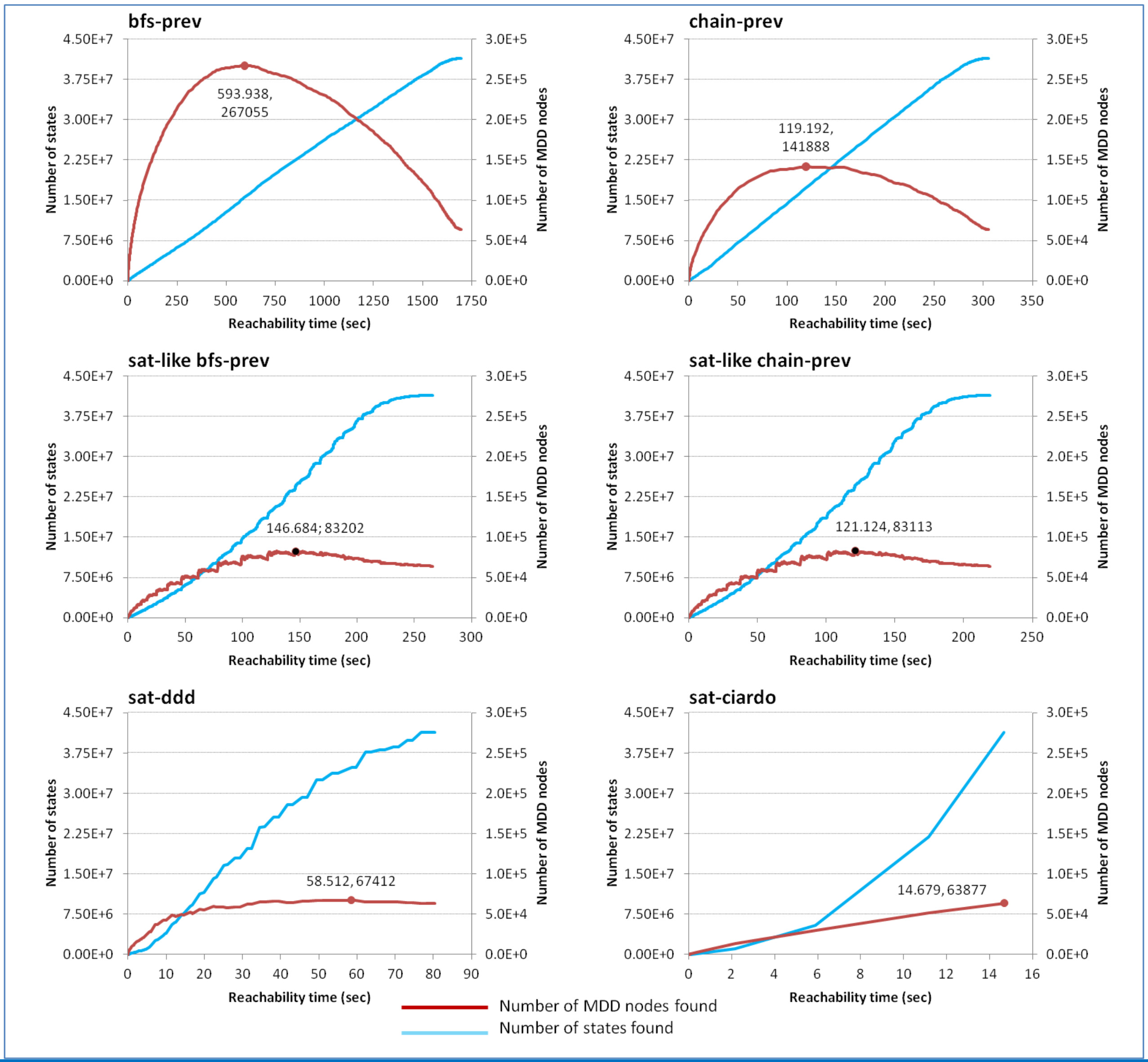
Final state space size	Final MDD node size	Fastest reachability option
11428767	352	sat-ciardo

Figure B. 7: State space evolution plots of DVE model elevator_planning.2.



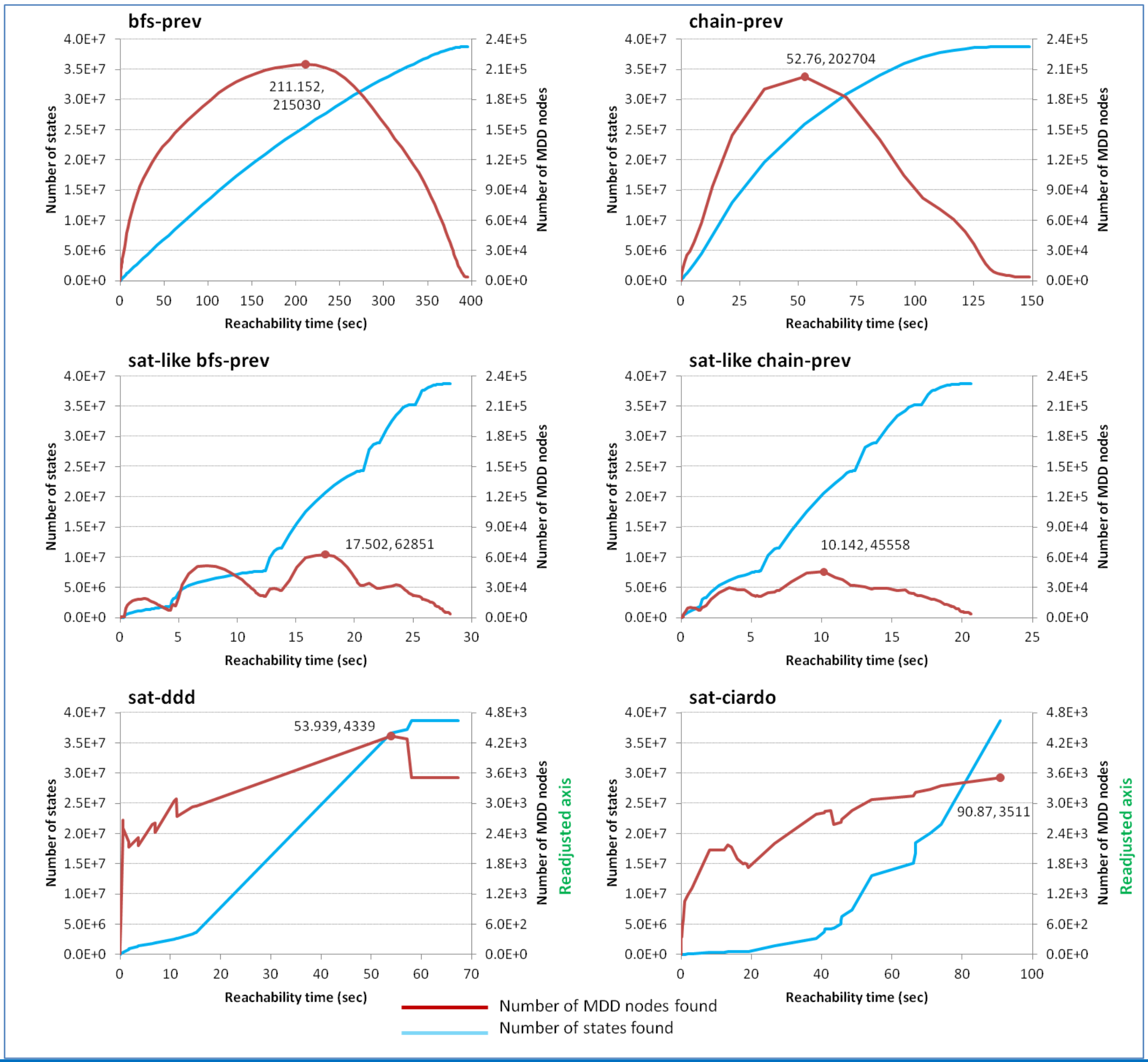
Final state space size	Final MDD node size	Fastest reachability option
18553032	71247	sat-ddd

Figure B. 8: State space evolution plots of DVE model firewire_link.5.



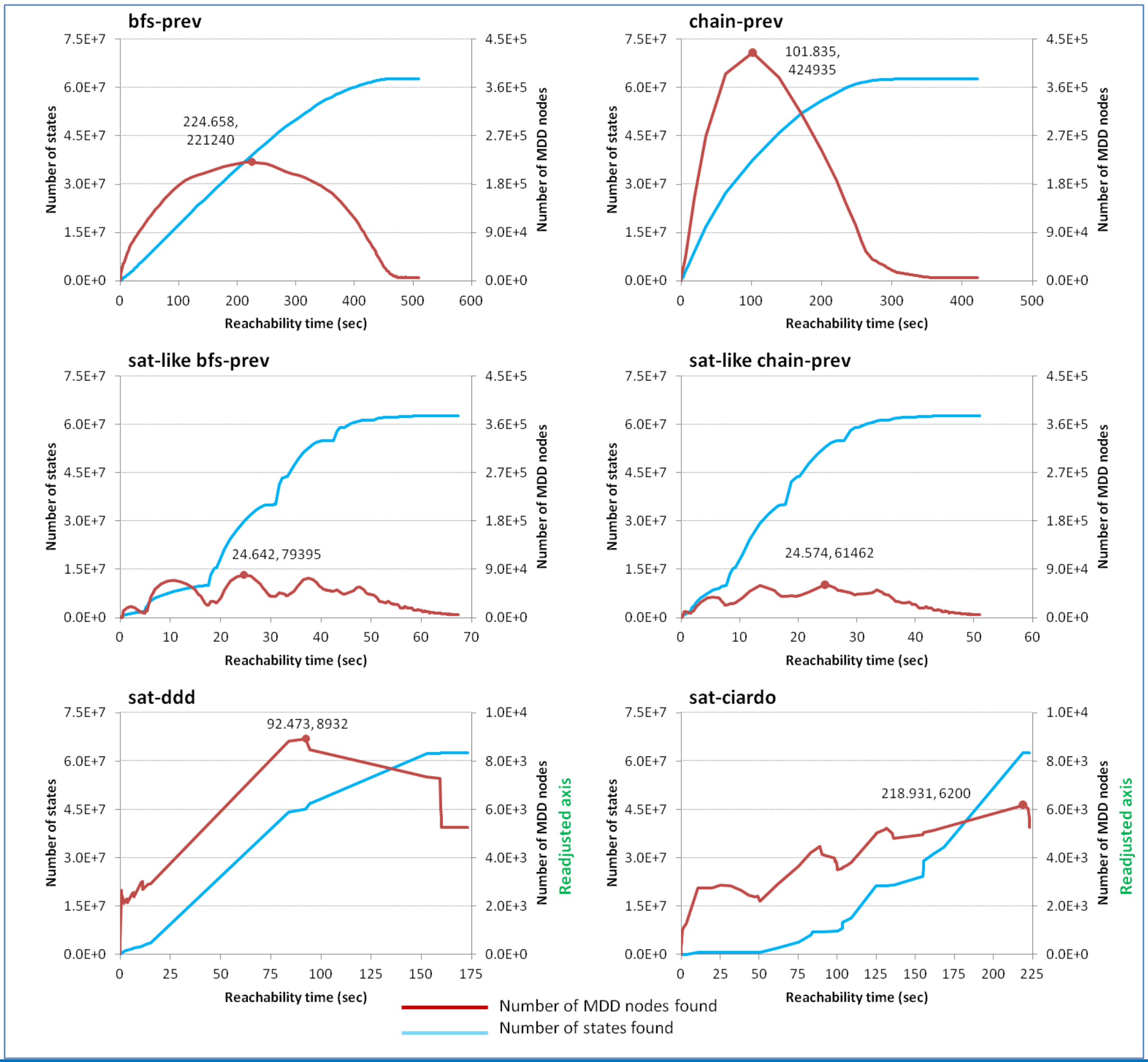
Final state space size	Final MDD node size	Fastest reachability option
41387484	63877	sat-ciardo

Figure B. 9: State space evolution plots of DVE model iprotocol.6.



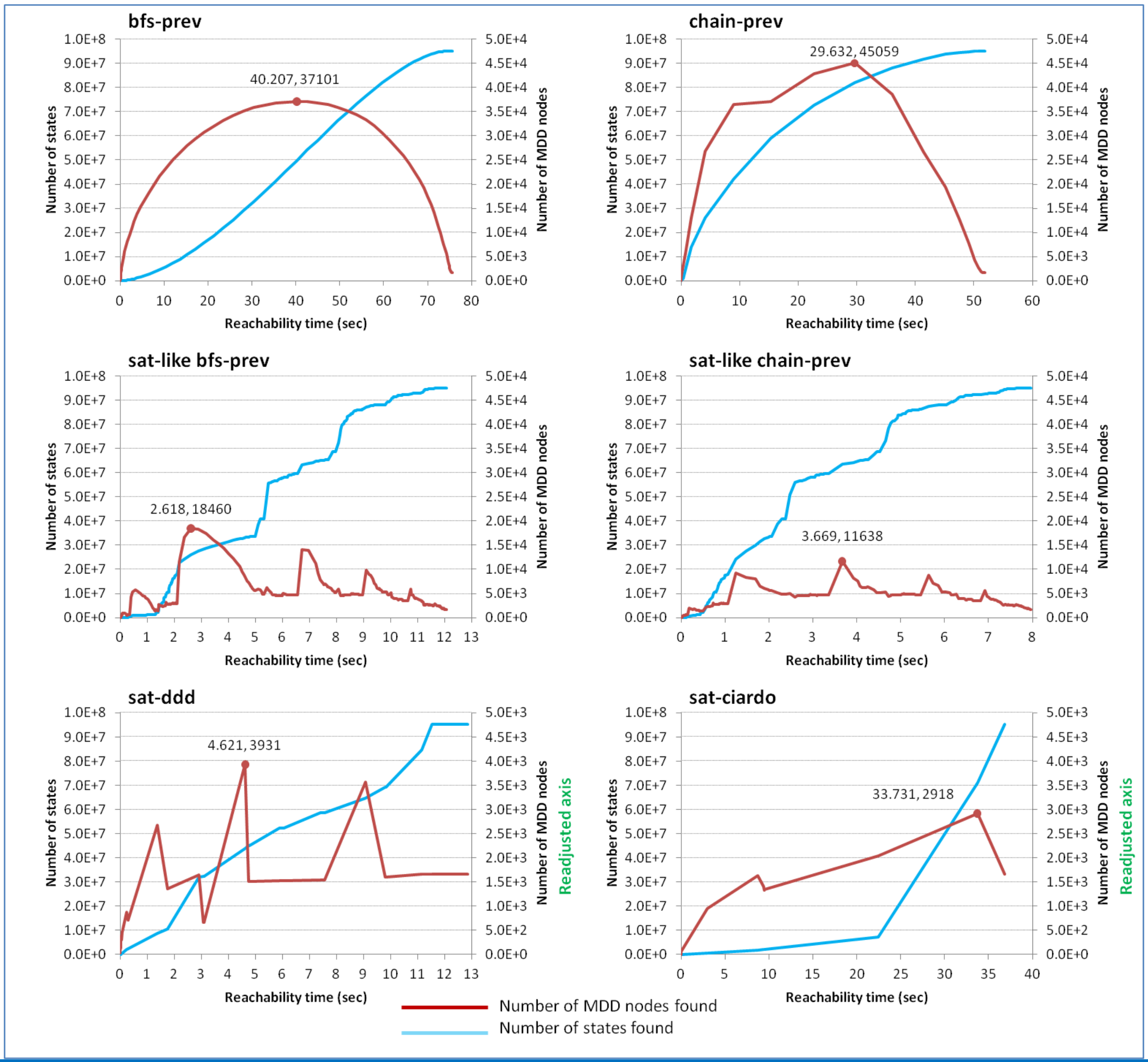
Final state space size	Final MDD node size	Fastest reachability option
38717846	3511	sat-like chain-prev

Figure B. 10: State space evolution plots of DVE model lamport.7.



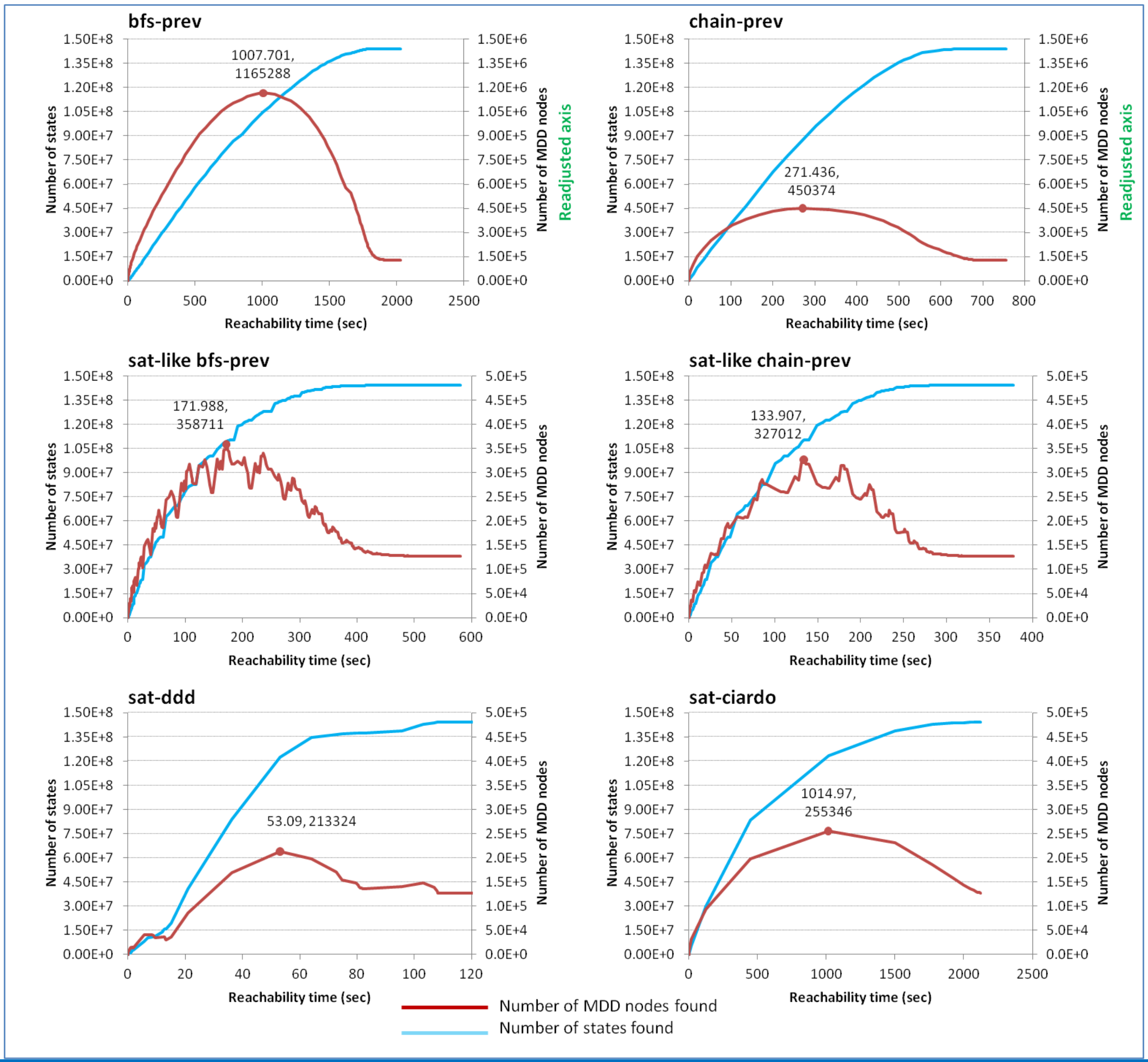
Final state space size	Final MDD node size	Fastest reachability option
62669317	5273	sat-like chain-prev

Figure B. 11: State space evolution plots of DVE model lamport.8.



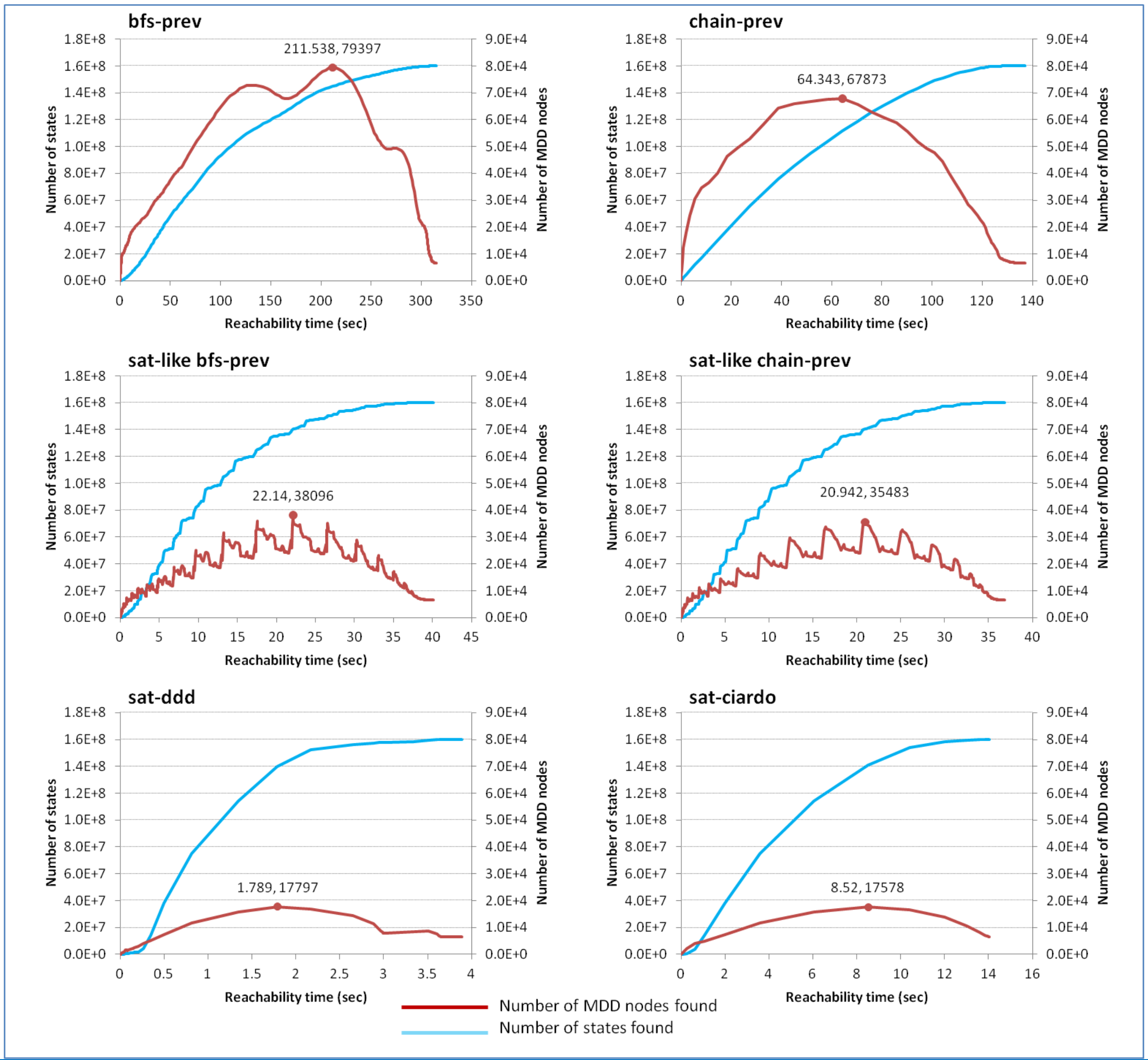
Final state space size	Final MDD node size	Fastest reachability option
95118524	1659	sat-like chain-prev

Figure B. 12: State space evolution plots of DVE model lamport_nonatomic.5.



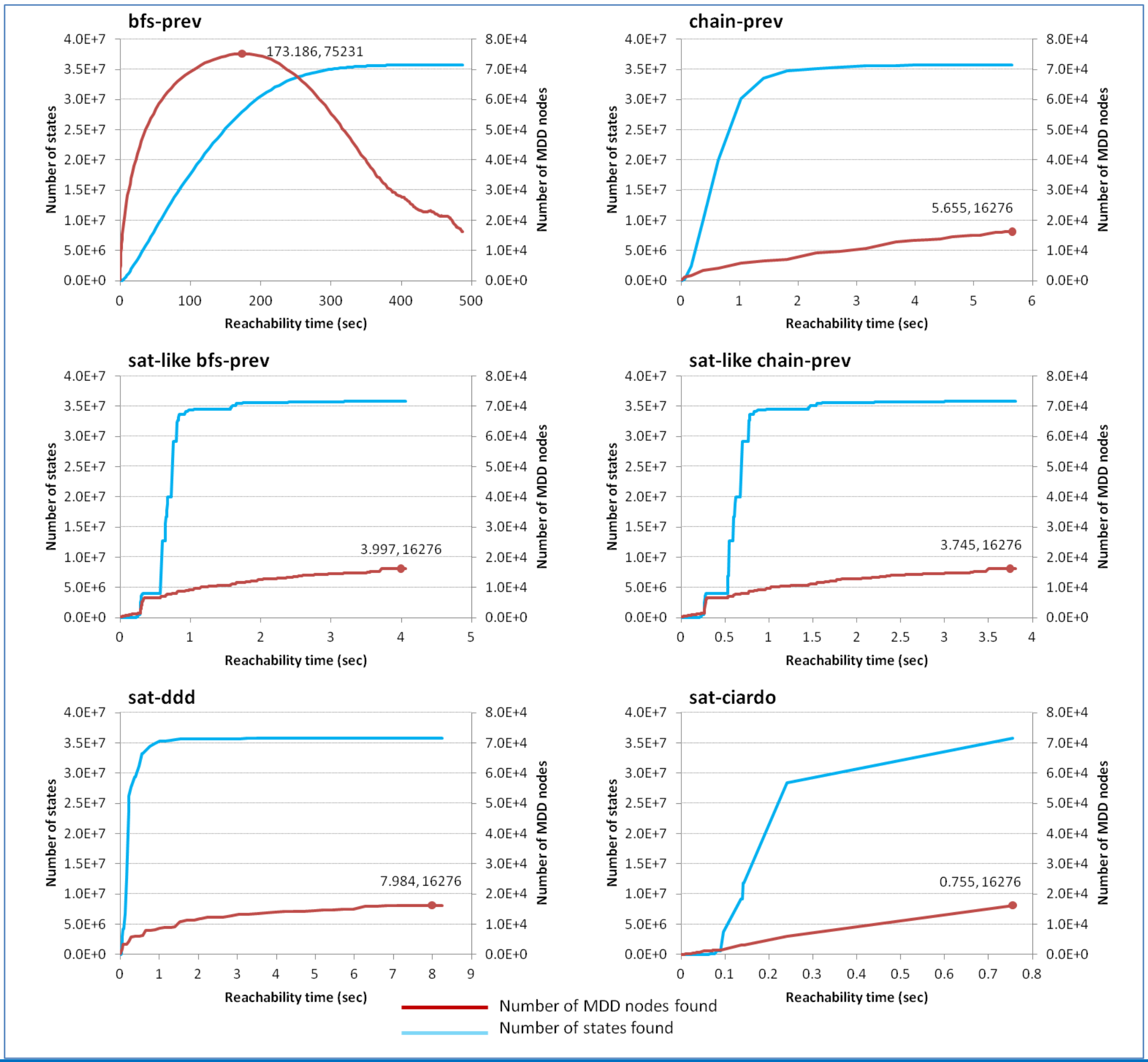
Final state space size	Final MDD node size	Fastest reachability option
144151628	127219	sat-ddd

Figure B. 13: State space evolution plots of DVE model lann.6.



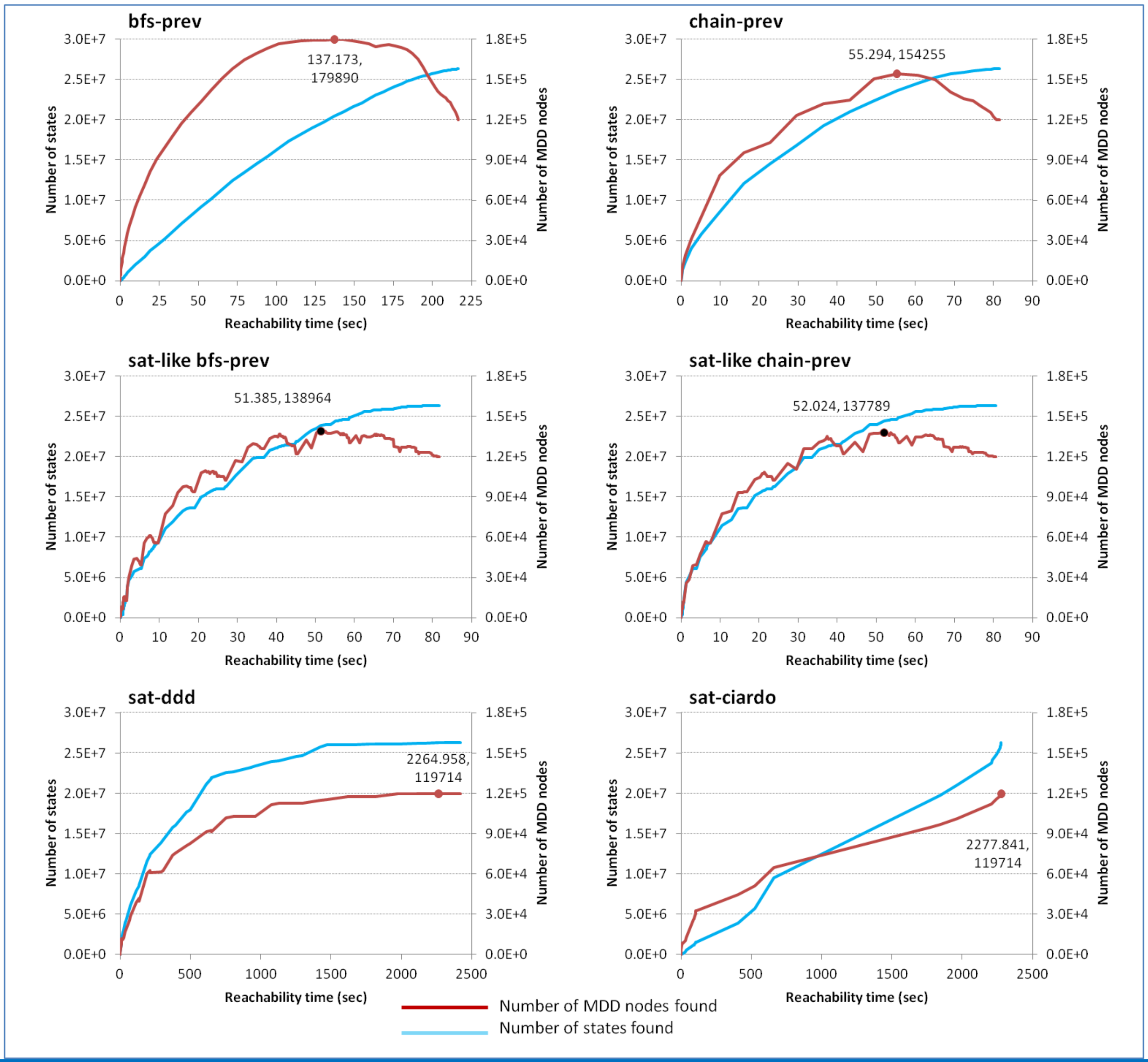
Final state space size	Final MDD node size	Fastest reachability option
160025986	6502	sat-ddd

Figure B. 14: State space evolution plots of DVE model lann.7.



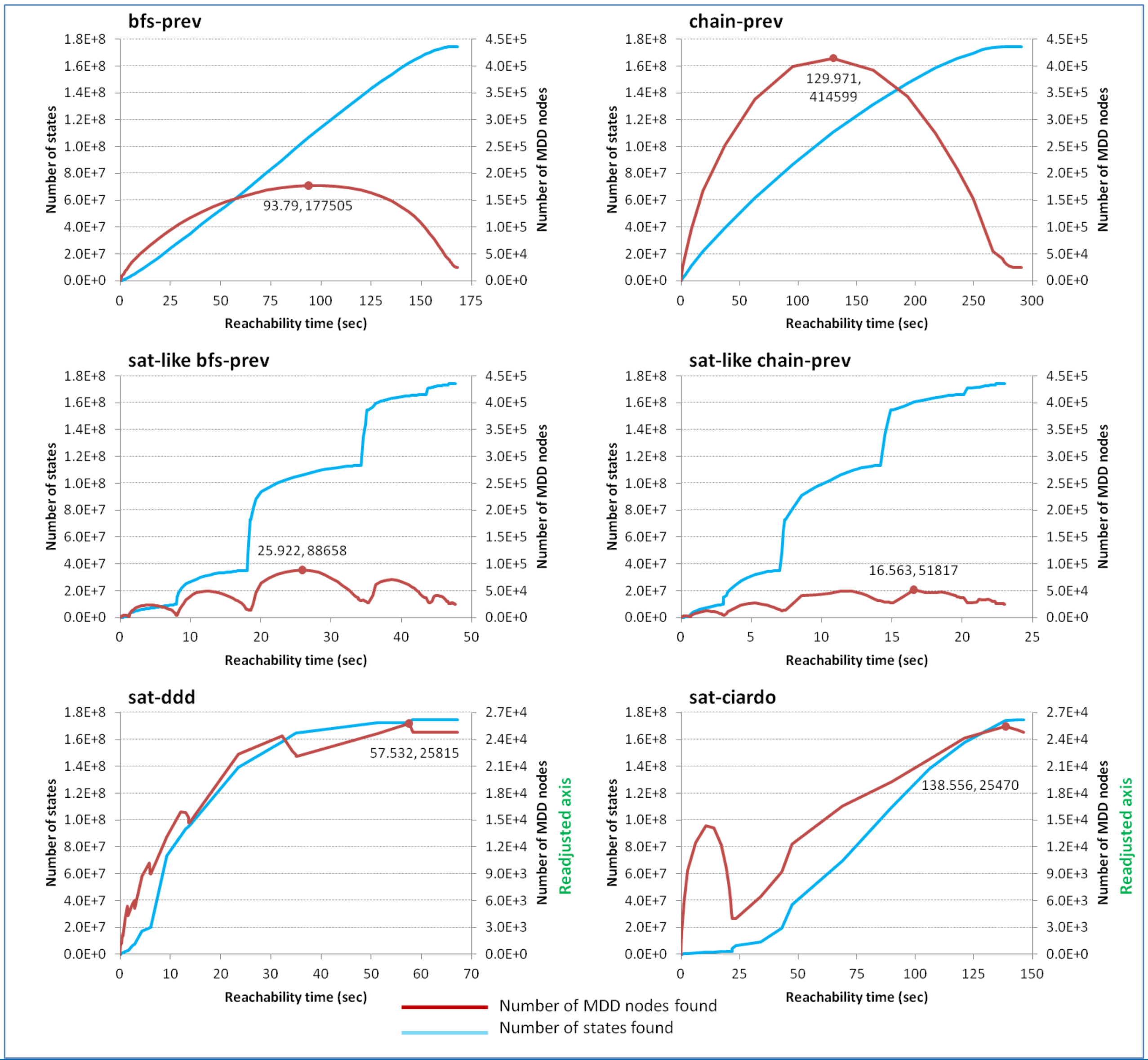
Final state space size	Final MDD node size	Fastest reachability option
35777100	16276	sat-ciardo

Figure B. 15: State space evolution plots of DVE model leader_election.6.



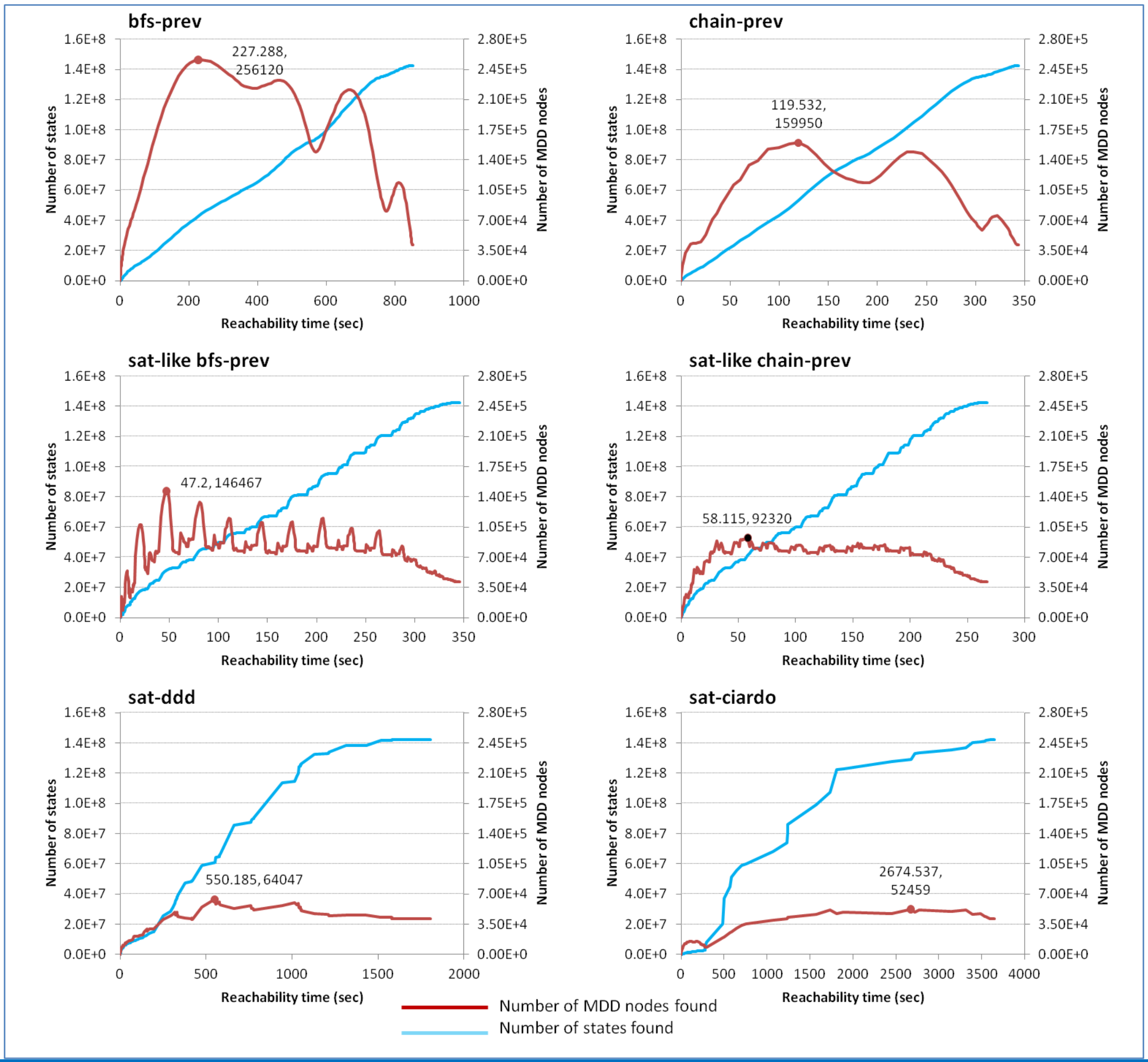
Final state space size	Final MDD node size	Fastest reachability option
26302351	119714	sat-like chain-prev

Figure B. 16: State space evolution plots of DVE model leader_filters.7.



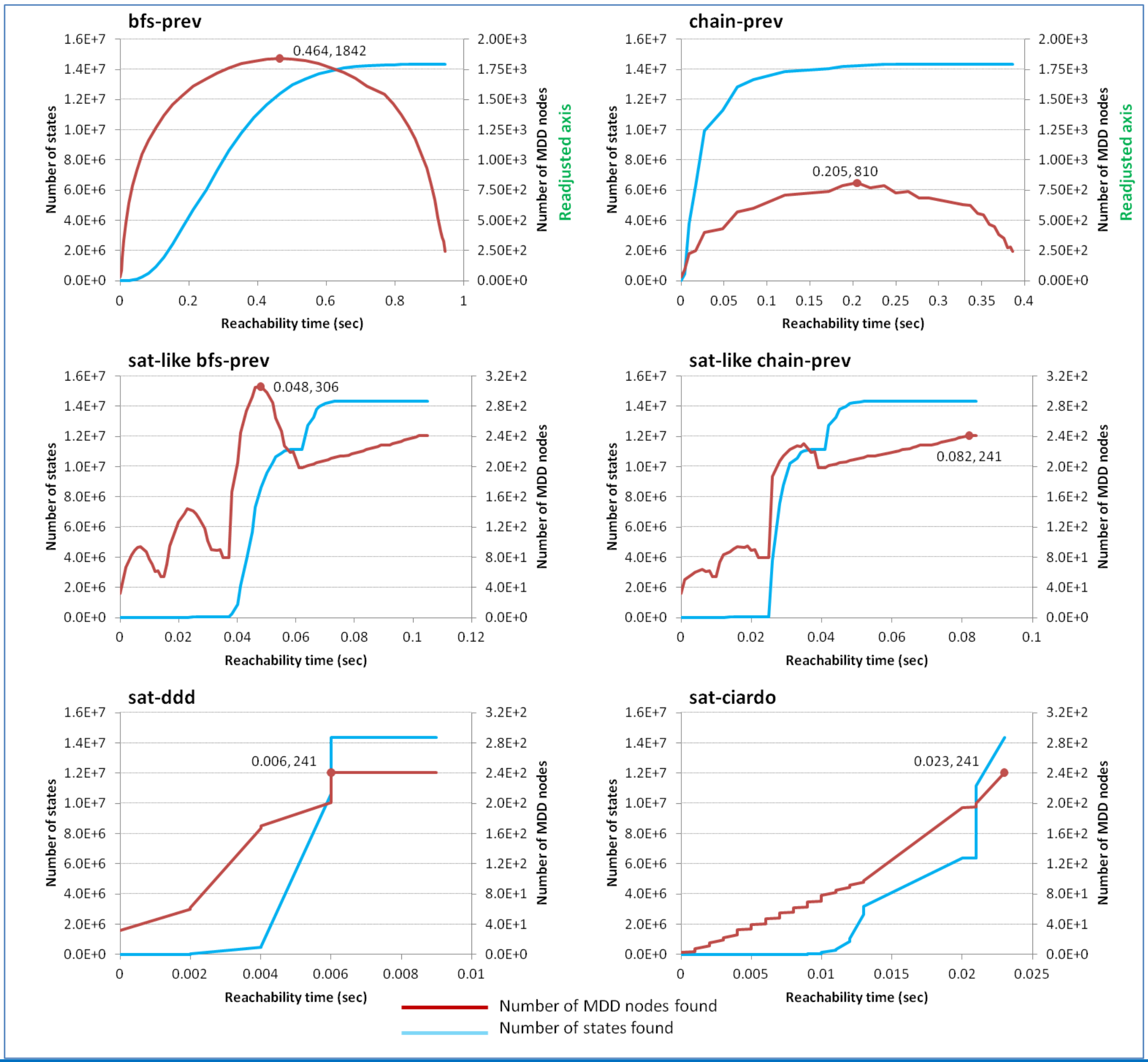
Final state space size	Final MDD node size	Fastest reachability option
174495861	24839	sat-like chain-prev

Figure B. 17: State space evolution plots of DVE model peterson.6.



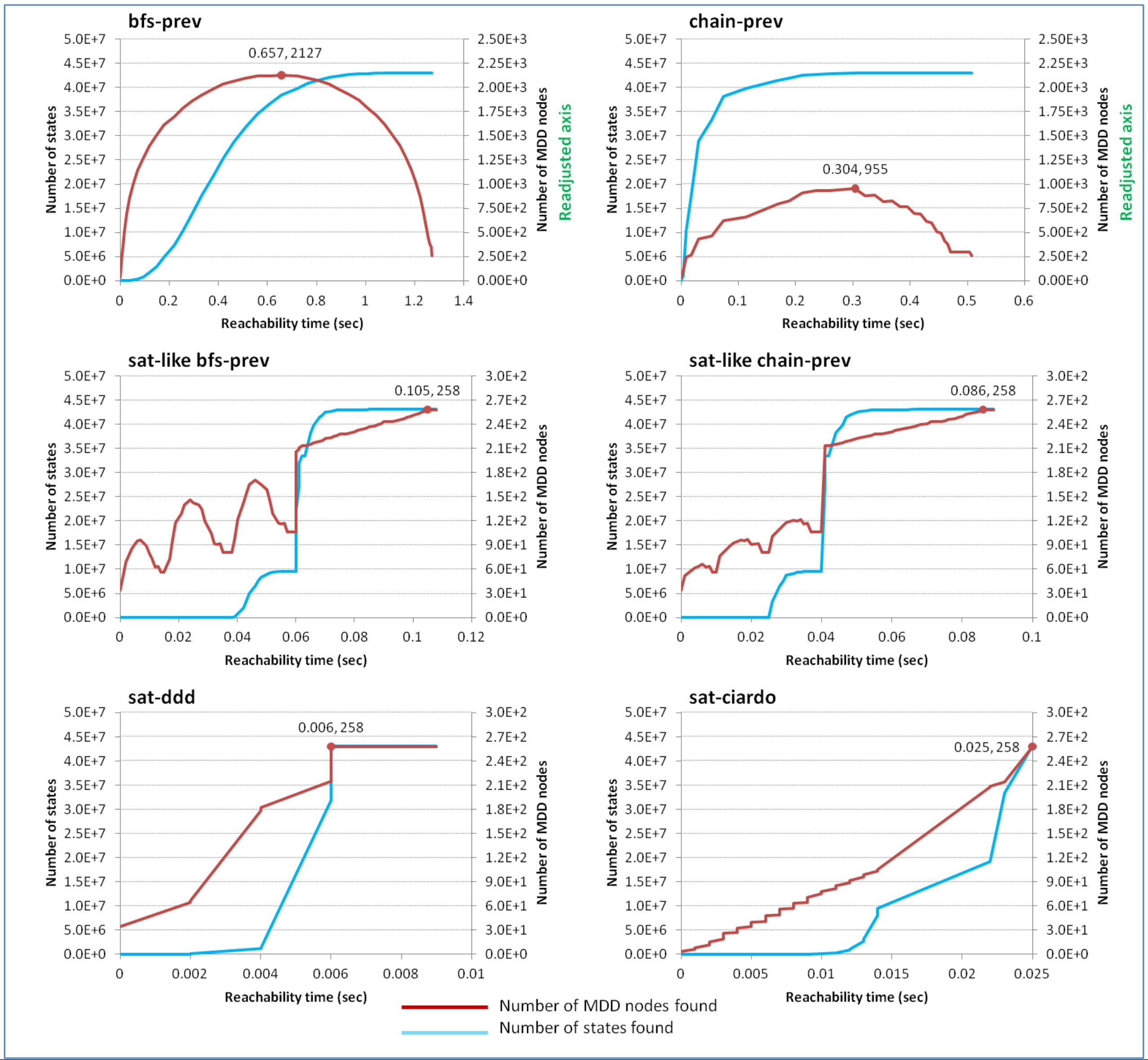
Final state space size	Final MDD node size	Fastest reachability option
142471098	41405	sat-like chain-prev

Figure B. 18: State space evolution plots of DVE model peterson.7.



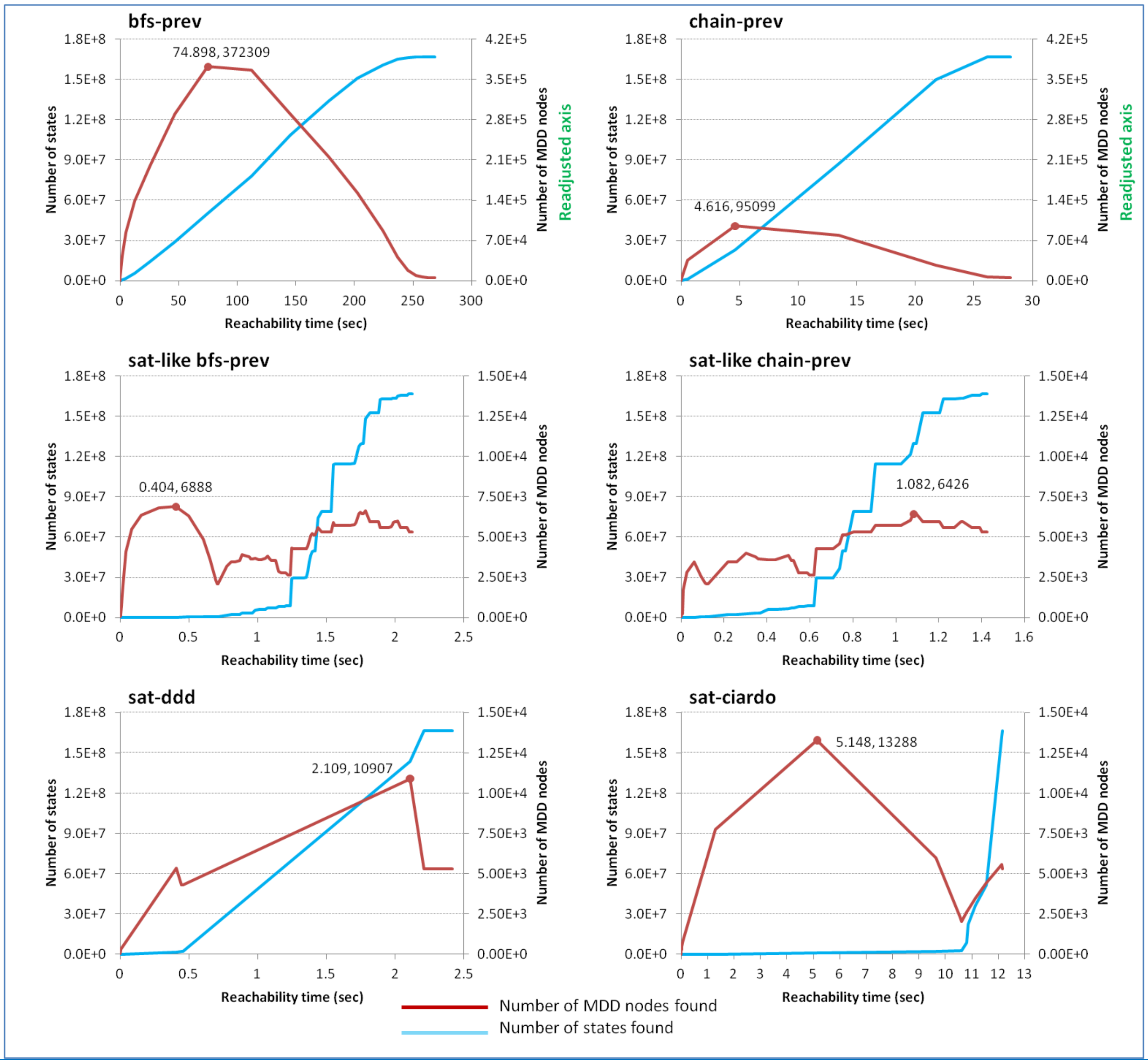
Final state space size	Final MDD node size	Fastest reachability option
14348906	241	sat-ddd

Figure B. 19: State space evolution plots of DVE model phils.6.



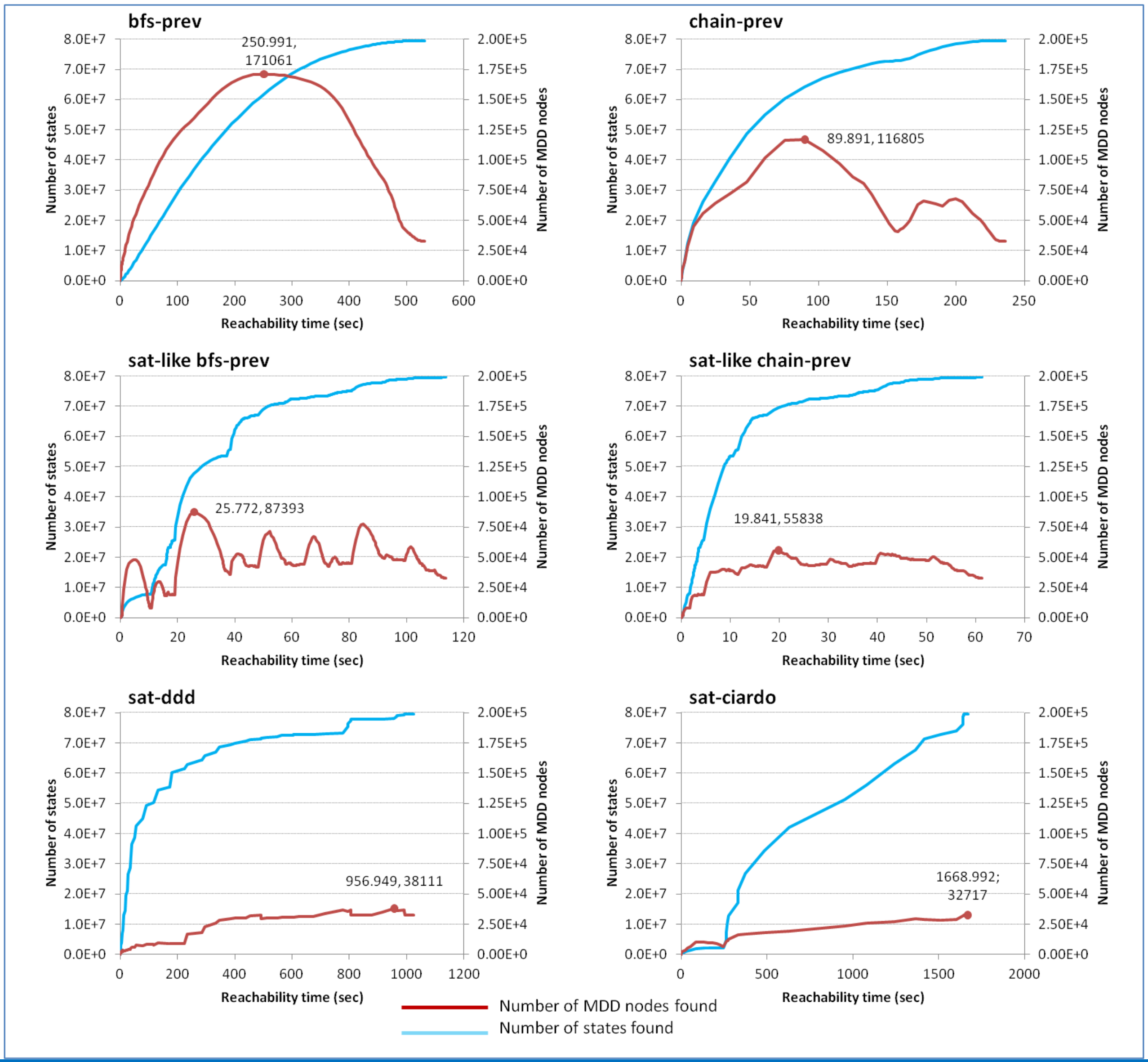
Final state space size	Final MDD node size	Fastest reachability option
43046720	258	sat-ddd

Figure B. 20: State space evolution plots of DVE model phils.8.



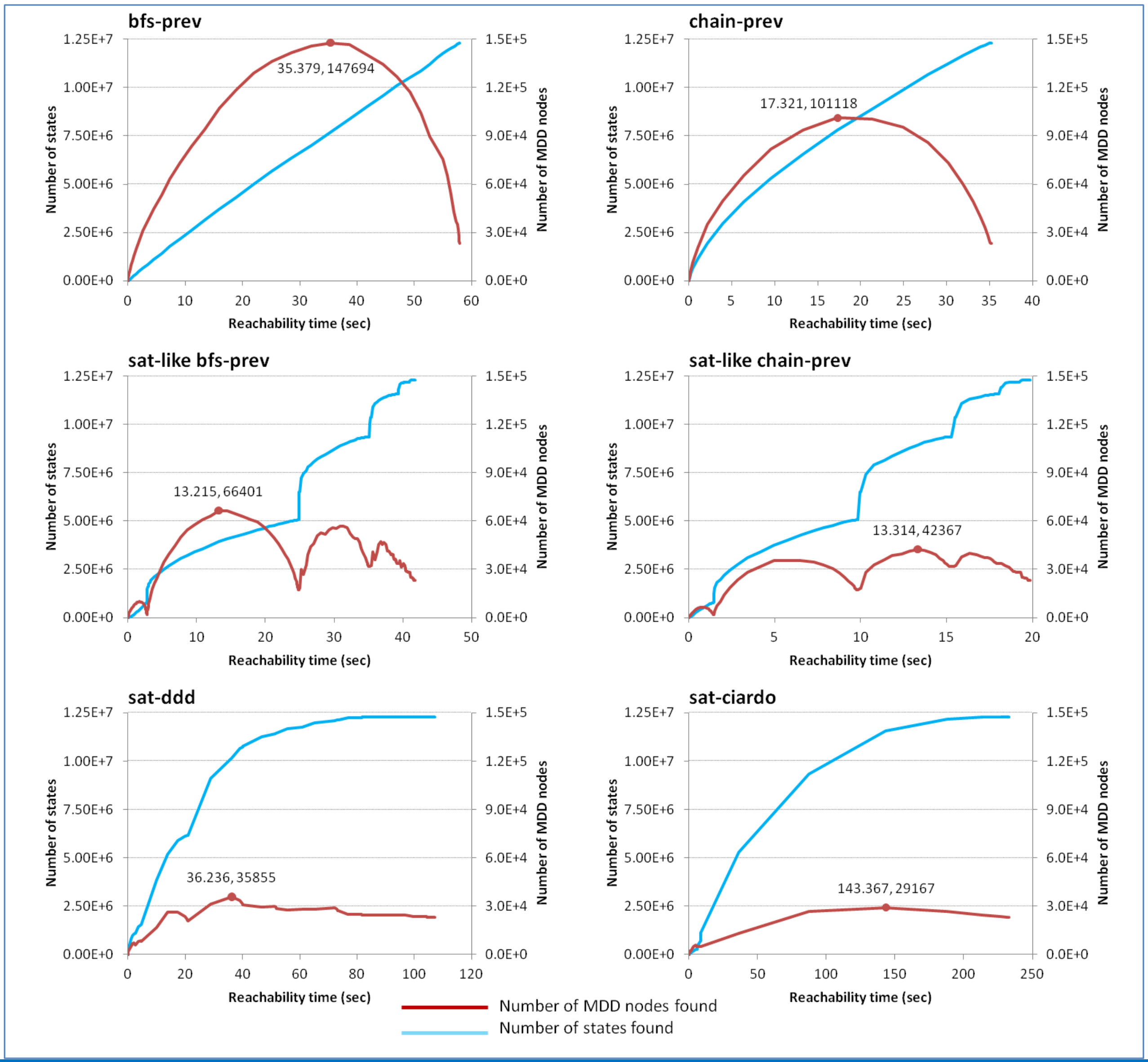
Final state space size	Final MDD node size	Fastest reachability option
166649331	5305	sat-like chain-prev

Figure B. 21: State space evolution plots of DVE model schedule_world.3.



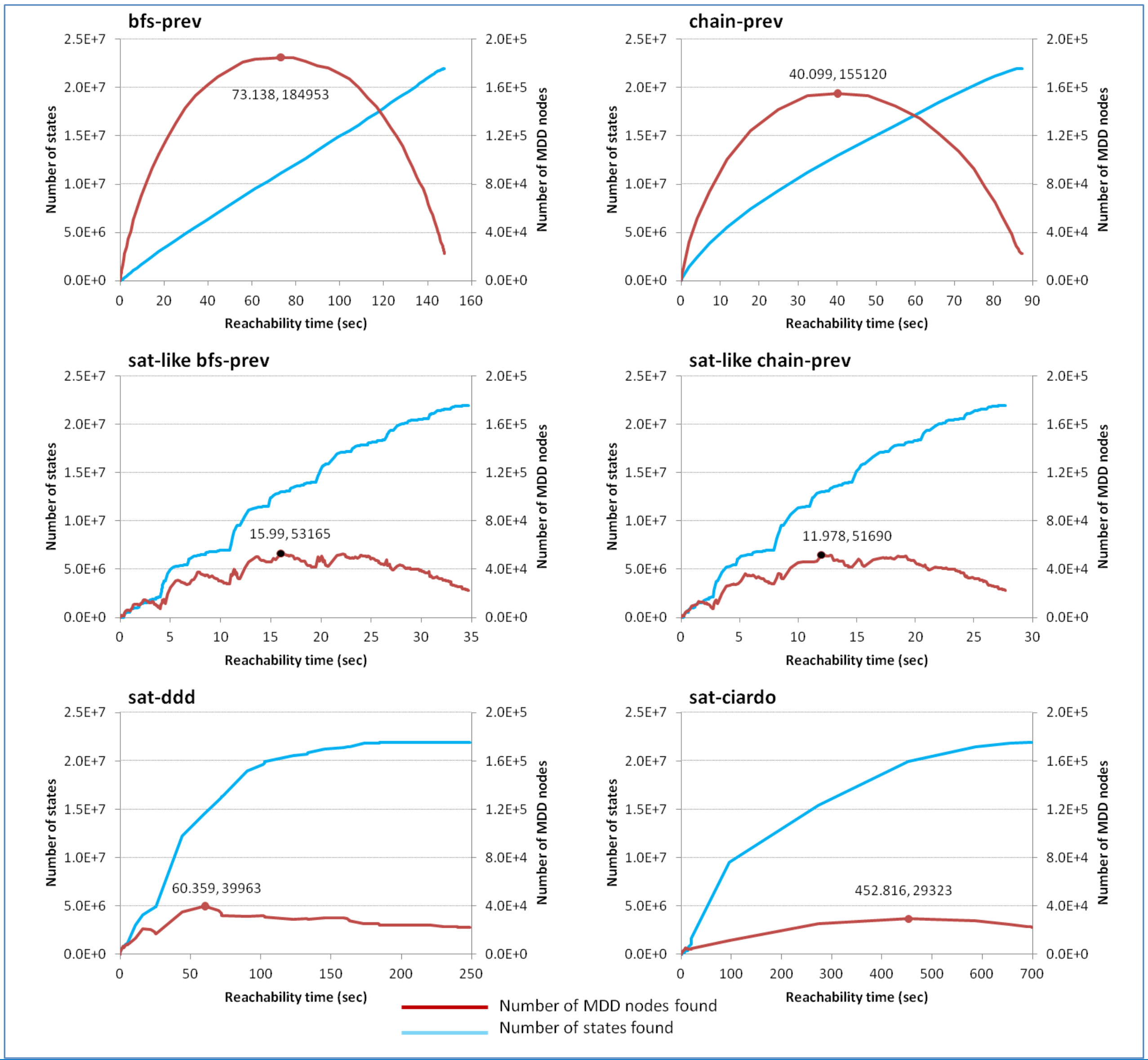
Final state space size	Final MDD node size	Fastest reachability option
79518740	32657	sat-like chain-prev

Figure B. 22: State space evolution plots of DVE model szymanski.5.



Final state space size	Final MDD node size	Fastest reachability option
12291552	23180	sat-like chain-prev

Figure B. 23: State space evolution plots of DVE model telephony.4.

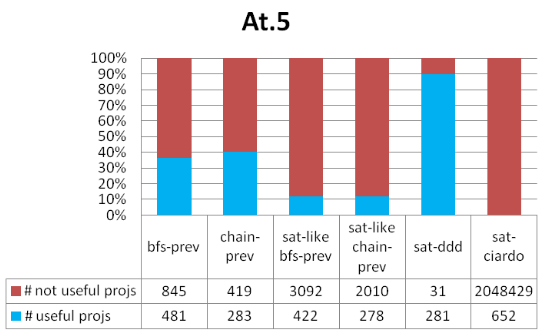


Final state space size	Final MDD node size	Fastest reachability option
21960308	22604	sat-like chain-prev

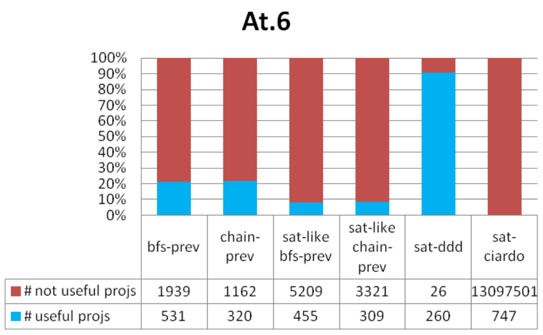
Figure B. 24: State space evolution plots of DVE model telephony.7.

C Results regarding MDD projections made

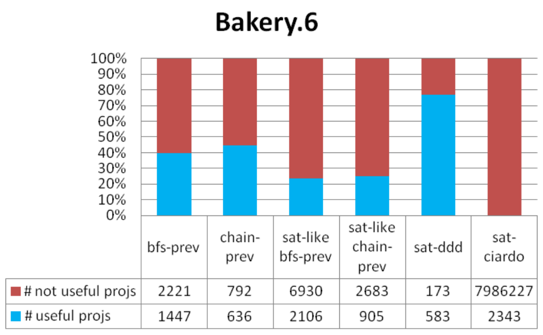
In this appendix the results from Table 5.6 are visualized for each of the remaining 24 DVE models individually, showing the contribution of useful MDD projections made in the total number of MDD projections during the update process of the transition groups for each of the tested reachability options.



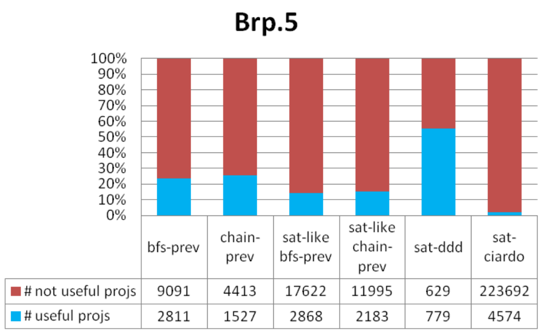
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	845	419	3092	2010	31	2048429
🟦 # useful projs	481	283	422	278	281	652



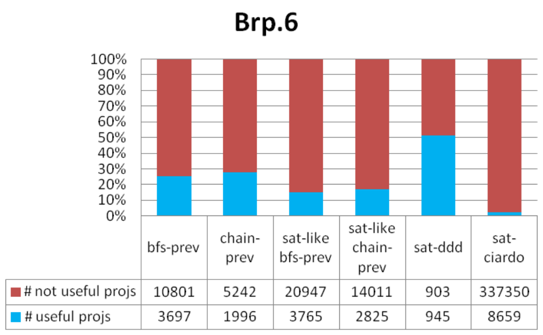
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	1939	1162	5209	3321	26	13097501
🟦 # useful projs	531	320	455	309	260	747



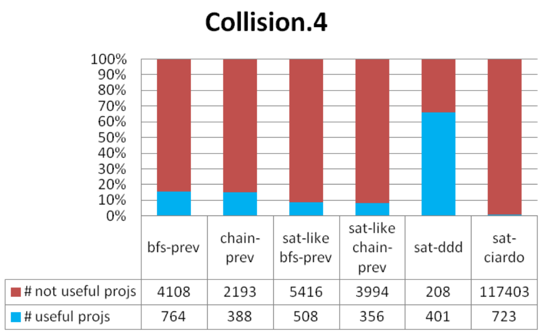
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	2221	792	6930	2683	173	7986227
🟦 # useful projs	1447	636	2106	905	583	2343



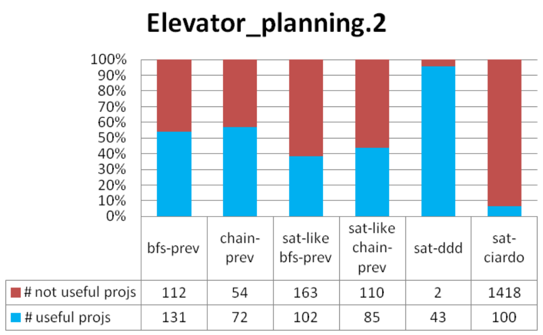
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	9091	4413	17622	11995	629	223692
🟦 # useful projs	2811	1527	2868	2183	779	4574



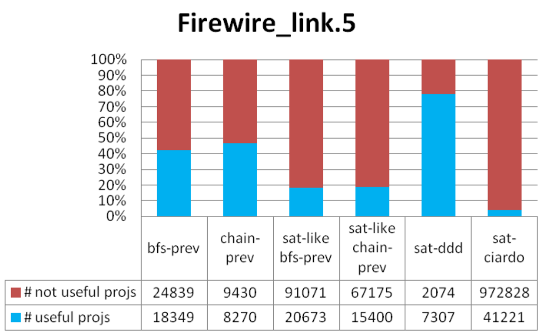
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	10801	5242	20947	14011	903	337350
🟦 # useful projs	3697	1996	3765	2825	945	8659



	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	4108	2193	5416	3994	208	117403
🟦 # useful projs	764	388	508	356	401	723

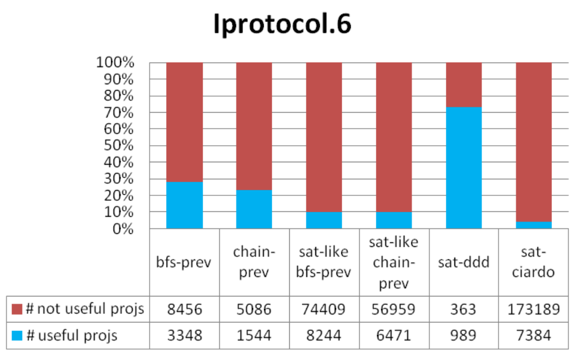


	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	112	54	163	110	2	1418
🟦 # useful projs	131	72	102	85	43	100

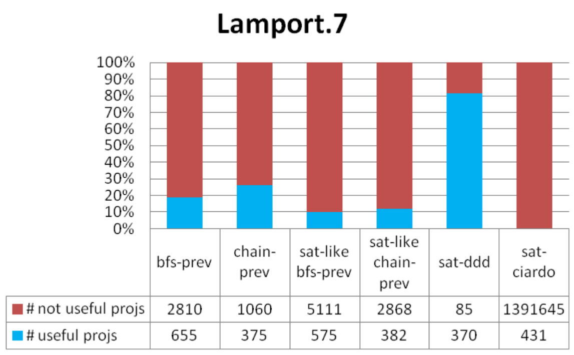


	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	24839	9430	91071	67175	2074	972828
🟦 # useful projs	18349	8270	20673	15400	7307	41221

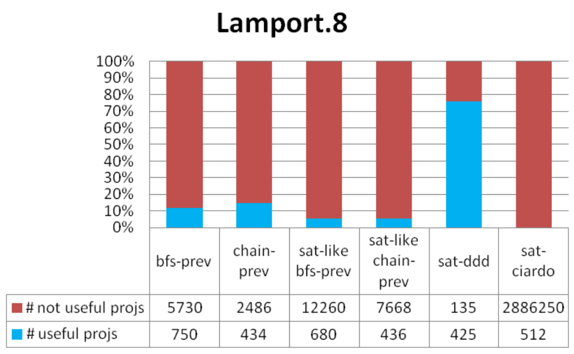
Figure C. 1: Bar charts showing the contribution of useful and useless MDD projections using the reachability tools from LTSmin for 8 DVE model (1).



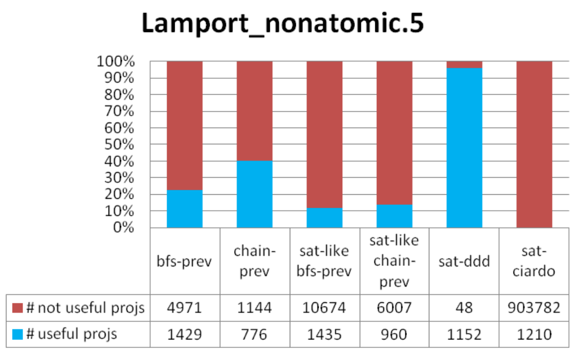
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	8456	5086	74409	56959	363	173189
🟦 # useful projs	3348	1544	8244	6471	989	7384



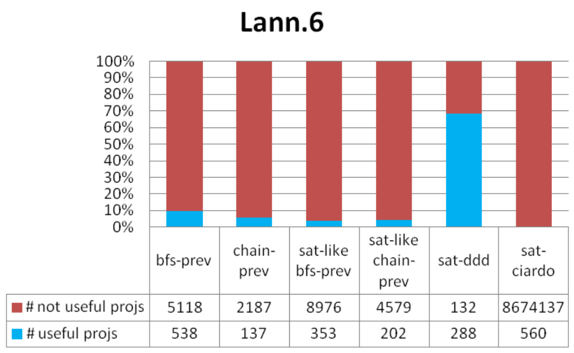
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	2810	1060	5111	2868	85	1391645
🟦 # useful projs	655	375	575	382	370	431



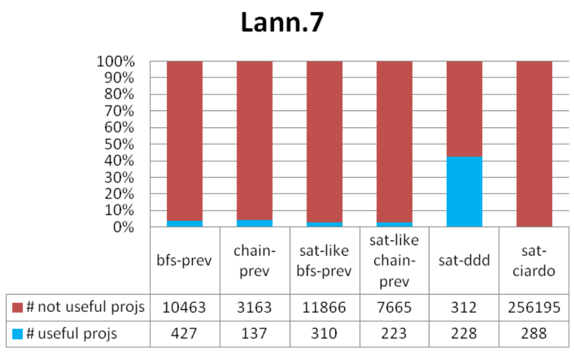
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	5730	2486	12260	7668	135	2886250
🟦 # useful projs	750	434	680	436	425	512



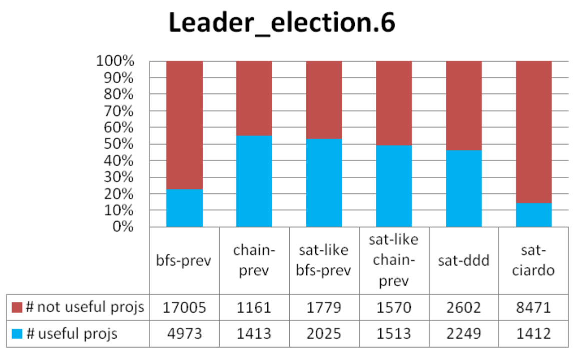
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	4971	1144	10674	6007	48	903782
🟦 # useful projs	1429	776	1435	960	1152	1210



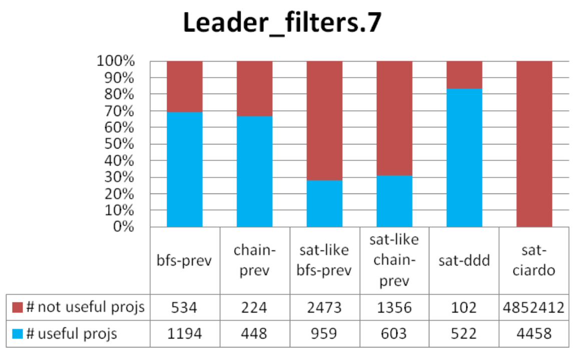
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	5118	2187	8976	4579	132	8674137
🟦 # useful projs	538	137	353	202	288	560



	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	10463	3163	11866	7665	312	256195
🟦 # useful projs	427	137	310	223	228	288

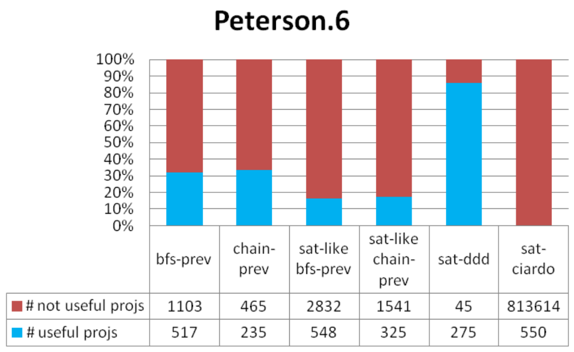


	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	17005	1161	1779	1570	2602	8471
🟦 # useful projs	4973	1413	2025	1513	2249	1412

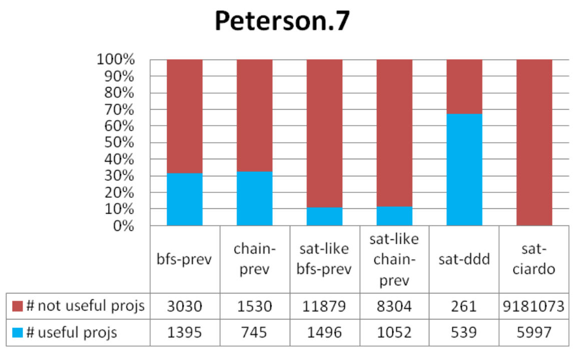


	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	534	224	2473	1356	102	4852412
🟦 # useful projs	1194	448	959	603	522	4458

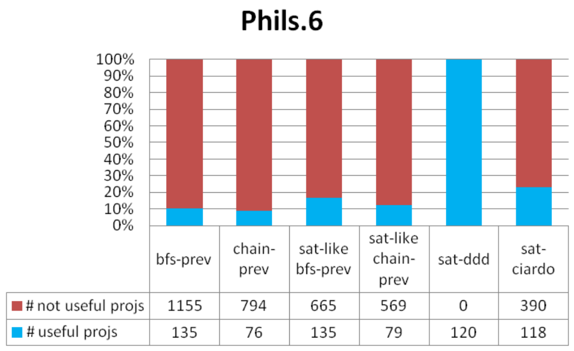
Figure C. 2: Bar charts showing the contribution of useful and useless MDD projections using the reachability tools from LTSmin for 8 DVE model (2).



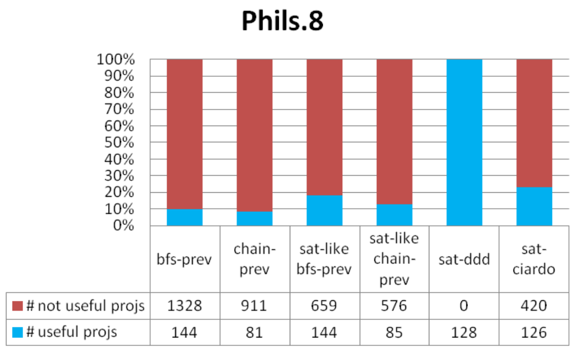
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	1103	465	2832	1541	45	813614
🟦 # useful projs	517	235	548	325	275	550



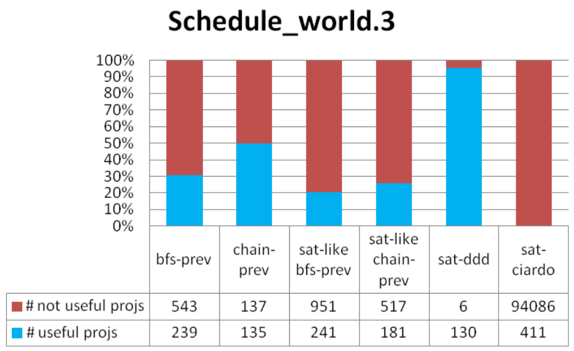
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	3030	1530	11879	8304	261	9181073
🟦 # useful projs	1395	745	1496	1052	539	5997



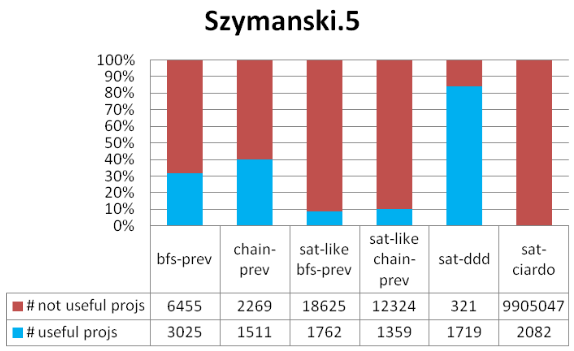
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	1155	794	665	569	0	390
🟦 # useful projs	135	76	135	79	120	118



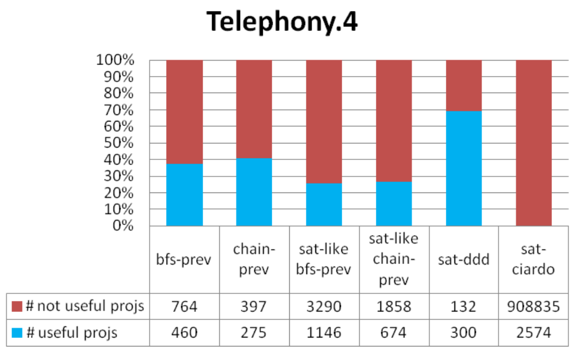
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	1328	911	659	576	0	420
🟦 # useful projs	144	81	144	85	128	126



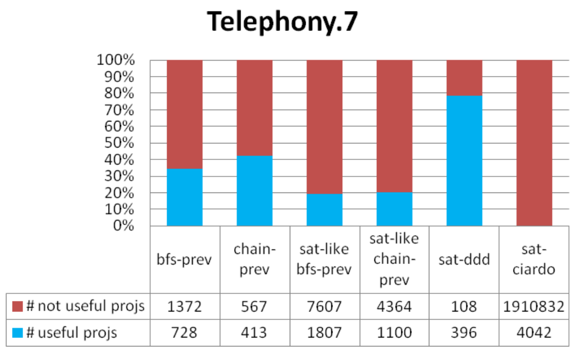
	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	543	137	951	517	6	94086
🟦 # useful projs	239	135	241	181	130	411



	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	6455	2269	18625	12324	321	9905047
🟦 # useful projs	3025	1511	1762	1359	1719	2082



	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	764	397	3290	1858	132	908835
🟦 # useful projs	460	275	1146	674	300	2574



	bfs-prev	chain-prev	sat-like bfs-prev	sat-like chain-prev	sat-ddd	sat-ciardo
🟥 # not useful projs	1372	567	7607	4364	108	1910832
🟦 # useful projs	728	413	1807	1100	396	4042

Figure C. 3: Bar charts showing the contribution of useful and useless MDD projections using the reachability tools from LTSmin for 8 DVE model (3).